The term 'parallel computing' means different things depending on the application area. In this book we focus on parallel computing – and more specifically parallel *programming*; we will not discuss a lot of theory – in the context of scientific computing.

Two of the most common software systems for parallel programming in scientific computing are MPI and OpenMP. They target different types of parallelism, and use very different constructs. Thus, by covering both of them in one book we can offer a treatment of parallelism that spans a large range of possible applications.

Finally, we also discuss the PETSc (Portable Toolkit for Scientific Computing) library, which offers an abstraction level higher than MPI or OpenMP, geared specifically towards parallel linear algebra, and very specifically the sort of linear algebra computations arising from Partial Differential Equation modeling.

The main languages in scientific computing are C/C++ and Fortran. We will discuss both MPI and OpenMP with many examples in these two languages. For MPI and the PETSc library we will also discuss the Python interfaces.

**Comments** This book is in perpetual state of revision and refinement. Please send comments of any kind to eijkhout@tacc.utexas.edu.
Contents

I MPI 15

1 Getting started with MPI 17
  1.1 Distributed memory and message passing 17
  1.2 History 18
  1.3 Basic model 18
  1.4 Making and running an MPI program 19
  1.5 Language bindings 20
  1.6 Review 24
  1.7 Sources used in this chapter 25

2 MPI topic: Functional parallelism 26
  2.1 The SPMD model 26
  2.2 Starting and running MPI processes 28
  2.3 Processor identification 32
  2.4 Functional parallelism 36
  2.5 Distributed computing and distributed data 37
  2.6 Review questions 38
  2.7 Sources used in this chapter 39

3 MPI topic: Collectives 41
  3.1 Working with global information 41
  3.2 Reduction 44
  3.3 Rooted collectives: broadcast, reduce 49
  3.4 Scan operations 57
  3.5 Rooted collectives: gather and scatter 60
  3.6 All-to-all 65
  3.7 Reduce-scatter 67
  3.8 Barrier 71
  3.9 Variable-size-input collectives 72
  3.10 MPI Operators 76
  3.11 Nonblocking collectives 81
  3.12 Performance of collectives 86
  3.13 Collectives and synchronization 87
  3.14 Performance considerations 90

3
### CONTENTS

3.15 **Review questions** ................................................. 93  
3.16 **Sources used in this chapter** .................................. 95  

4 **MPI topic: Point-to-point** ........................................... 118  
4.1 **Blocking point-to-point operations** ............................. 118  
4.2 **Nonblocking point-to-point operations** ......................... 133  
4.3 **More about point-to-point communication** ..................... 147  
4.4 **Review questions** .................................................. 155  
4.5 **Sources used in this chapter** .................................... 160  

5 **MPI topic: Communication modes** .................................. 181  
5.1 **Persistent communication** ....................................... 181  
5.2 **Partitioned communication** ...................................... 185  
5.3 **Synchronous and asynchronous communication** ................ 187  
5.4 **Local and nonlocal operations** ................................... 188  
5.5 **Buffered communication** ......................................... 188  
5.6 **Sources used in this chapter** .................................... 192  

6 **MPI topic: Data types** ................................................ 194  
6.1 **Data type handling** ................................................. 194  
6.2 **Elementary data types** ............................................. 195  
6.3 **Derived datatypes** .................................................. 202  
6.4 **Big data types** ...................................................... 221  
6.5 **Type maps and type matching** ................................... 224  
6.6 **Type extent** .......................................................... 224  
6.7 **Reconstructing types** .............................................. 231  
6.8 **Packing** ............................................................. 232  
6.9 **Review questions** .................................................. 235  
6.10 **Sources used in this chapter** ................................... 236  

7 **MPI topic: Communicators** .......................................... 266  
7.1 **Basic communicators** .............................................. 266  
7.2 **Duplicating communicators** ...................................... 267  
7.3 **Sub-communicators** ............................................... 271  
7.4 **Splitting a communicator** ........................................... 272  
7.5 **Communicators and groups** ...................................... 275  
7.6 **Intercommunicators** ............................................... 277  
7.7 **Review questions** .................................................. 281  
7.8 **Sources used in this chapter** .................................... 282  

8 **MPI topic: Process management** ...................................... 289  
8.1 **Process spawning** .................................................. 289  
8.2 **Socket-style communications** ................................... 293  
8.3 **Sessions** ............................................................ 296
### CONTENTS

8.4 Functionality available outside init/finalize ............................................. 299  
8.5 Sources used in this chapter ........................................................................... 301

9 **MPI topic: One-sided communication** ......................................................... 311  
  9.1 Windows ........................................................................................................ 312  
  9.2 Active target synchronization: epochs ............................................................. 316  
  9.3 Put, get, accumulate ....................................................................................... 319  
  9.4 Passive target synchronization ..................................................................... 330  
  9.5 More about window memory ......................................................................... 333  
  9.6 Assertions ....................................................................................................... 337  
  9.7 Implementation .............................................................................................. 338  
  9.8 Review questions ........................................................................................... 339  
  9.9 Sources used in this chapter ......................................................................... 340

10 **MPI topic: File I/O** .................................................................................... 356  
  10.1 File handling ................................................................................................. 357  
  10.2 File reading and writing ................................................................................ 358  
  10.3 Consistency .................................................................................................... 364  
  10.4 Constants ...................................................................................................... 364  
  10.5 Error handling ............................................................................................... 364  
  10.6 Review questions ......................................................................................... 365  
  10.7 Sources used in this chapter ....................................................................... 366

11 **MPI topic: Topologies** ............................................................................... 367  
  11.1 Cartesian grid topology ................................................................................ 367  
  11.2 Distributed graph topology ......................................................................... 372  
  11.3 Sources used in this chapter ....................................................................... 380

12 **MPI topic: Shared memory** ...................................................................... 384  
  12.1 Recognizing shared memory ....................................................................... 384  
  12.2 Shared memory for windows ...................................................................... 386  
  12.3 Sources used in this chapter ....................................................................... 390

13 **MPI topic: Hybrid computing** .................................................................. 395  
  13.1 MPI support for threading .......................................................................... 395  
  13.2 Sources used in this chapter ....................................................................... 398

14 **MPI topic: Tools interface** ....................................................................... 399  
  14.1 Initializing the tools interface ..................................................................... 399  
  14.2 Control variables ......................................................................................... 399  
  14.3 Performance variables ................................................................................ 401  
  14.4 Categories of variables ................................................................................ 403  
  14.5 Events ........................................................................................................... 404  
  14.6 Sources used in this chapter ....................................................................... 405
## 15 MPI leftover topics

- **15.1 Contextual information, attributes, etc.**
- **15.2 Error handling**
- **15.3 Fortran issues**
- **15.4 Progress**
- **15.5 Fault tolerance**
- **15.6 Performance, tools, and profiling**
- **15.7 Determinism**
- **15.8 Subtleties with processor synchronization**
- **15.9 Shell interaction**
- **15.10 Leftover topics**
- **15.11 Literature**
- **15.12 Sources used in this chapter**

### 16 MPI Examples

- **16.1 Bandwidth and halfbandwidth**
- **16.2 Sources used in this chapter**

## II OpenMP

### 17 Getting started with OpenMP

- **17.1 The OpenMP model**
- **17.2 Compiling and running an OpenMP program**
- **17.3 Your first OpenMP program**
- **17.4 Thread data**
- **17.5 Creating parallelism**
- **17.6 Sources used in this chapter**

### 18 OpenMP topic: Parallel regions

- **18.1 Creating parallelism with parallel regions**
- **18.2 Nested parallelism**
- **18.3 Cancel parallel construct**
- **18.4 Review questions**
- **18.5 Sources used in this chapter**

### 19 OpenMP topic: Loop parallelism

- **19.1 Loop parallelism**
- **19.2 An example**
- **19.3 Loop schedules**
- **19.4 Reductions**
- **19.5 Collapsing nested loops**
- **19.6 Ordered iterations**
- **19.7 nowait**
## Contents

19.8 While loops .................................................. 469  
19.9 Scaling tests .................................................. 470  
19.10 Review questions ......................................... 471  
19.11 Sources used in this chapter .............................. 473  

20 OpenMP topic: Reductions ................................. 476  
    20.1 Reductions: why, what, how? ......................... 476  
    20.2 Built-in reduction ...................................... 479  
    20.3 Initial value for reductions ......................... 479  
    20.4 User-defined reductions .............................. 480  
    20.5 Reductions and floating-point math ............. 483  
    20.6 Sources used in this chapter ....................... 484  

21 OpenMP topic: Work sharing ..................... 486  
    21.1 Sections ............................................... 486  
    21.2 Single/master .......................................... 487  
    21.3 Fortran array syntax parallelization ...... 488  
    21.4 Sources used in this chapter ....................... 489  

22 OpenMP topic: Controlling thread data ........ 490  
    22.1 Shared data ........................................... 490  
    22.2 Private data .......................................... 490  
    22.3 Data in dynamic scope ............................... 491  
    22.4 Temporary variables in a loop .......... 492  
    22.5 Default ................................................. 492  
    22.6 Array data ............................................. 493  
    22.7 First and last private .............................. 494  
    22.8 Persistent data through threadprivate ... 494  
    22.9 Allocators ............................................ 496  
    22.10 Sources used in this chapter ................... 497  

23 OpenMP topic: Synchronization .................. 499  
    23.1 Barrier .................................................. 499  
    23.2 Mutual exclusion .................................... 500  
    23.3 Locks .................................................... 502  
    23.4 Example: Fibonacci computation ............. 505  
    23.5 Review .................................................. 507  
    23.6 Review questions .................................... 507  
    23.7 Sources used in this chapter ................... 509  

24 OpenMP topic: Tasks ................................. 511  
    24.1 Task data .............................................. 512  
    24.2 Task synchronization ............................... 513  
    24.3 Task dependencies .................................. 514
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.4</td>
<td>Task reduction</td>
<td>516</td>
</tr>
<tr>
<td>24.5</td>
<td>More</td>
<td>516</td>
</tr>
<tr>
<td>24.6</td>
<td>Examples</td>
<td>517</td>
</tr>
<tr>
<td>24.7</td>
<td>Sources used in this chapter</td>
<td>519</td>
</tr>
<tr>
<td>25</td>
<td>OpenMP topic: Affinity</td>
<td>522</td>
</tr>
<tr>
<td>25.1</td>
<td>OpenMP thread affinity control</td>
<td>522</td>
</tr>
<tr>
<td>25.2</td>
<td>First-touch</td>
<td>526</td>
</tr>
<tr>
<td>25.3</td>
<td>Affinity control outside OpenMP</td>
<td>528</td>
</tr>
<tr>
<td>25.4</td>
<td>Sources used in this chapter</td>
<td>529</td>
</tr>
<tr>
<td>26</td>
<td>OpenMP topic: Memory model</td>
<td>530</td>
</tr>
<tr>
<td>26.1</td>
<td>Thread synchronization</td>
<td>530</td>
</tr>
<tr>
<td>26.2</td>
<td>Data races</td>
<td>531</td>
</tr>
<tr>
<td>26.3</td>
<td>Relaxed memory model</td>
<td>533</td>
</tr>
<tr>
<td>26.4</td>
<td>Sources used in this chapter</td>
<td>534</td>
</tr>
<tr>
<td>27</td>
<td>OpenMP topic: SIMD processing</td>
<td>535</td>
</tr>
<tr>
<td>27.1</td>
<td>Sources used in this chapter</td>
<td>539</td>
</tr>
<tr>
<td>28</td>
<td>OpenMP topic: Offloading</td>
<td>540</td>
</tr>
<tr>
<td>28.1</td>
<td>Data on the device</td>
<td>540</td>
</tr>
<tr>
<td>28.2</td>
<td>Execution on the device</td>
<td>541</td>
</tr>
<tr>
<td>28.3</td>
<td>Sources used in this chapter</td>
<td>542</td>
</tr>
<tr>
<td>29</td>
<td>OpenMP remaining topics</td>
<td>543</td>
</tr>
<tr>
<td>29.1</td>
<td>Runtime functions and internal control variables</td>
<td>543</td>
</tr>
<tr>
<td>29.2</td>
<td>Timing</td>
<td>545</td>
</tr>
<tr>
<td>29.3</td>
<td>Thread safety</td>
<td>545</td>
</tr>
<tr>
<td>29.4</td>
<td>Performance and tuning</td>
<td>546</td>
</tr>
<tr>
<td>29.5</td>
<td>Accelerators</td>
<td>547</td>
</tr>
<tr>
<td>29.6</td>
<td>Tools interface</td>
<td>547</td>
</tr>
<tr>
<td>29.7</td>
<td>OpenMP standards</td>
<td>548</td>
</tr>
<tr>
<td>29.8</td>
<td>Sources used in this chapter</td>
<td>549</td>
</tr>
<tr>
<td>30</td>
<td>OpenMP Review</td>
<td>550</td>
</tr>
<tr>
<td>30.1</td>
<td>Concepts review</td>
<td>550</td>
</tr>
<tr>
<td>30.2</td>
<td>Review questions</td>
<td>551</td>
</tr>
<tr>
<td>30.3</td>
<td>Sources used in this chapter</td>
<td>559</td>
</tr>
<tr>
<td>31</td>
<td>OpenMP Examples</td>
<td>560</td>
</tr>
<tr>
<td>31.1</td>
<td>N-body problems</td>
<td>560</td>
</tr>
<tr>
<td>31.2</td>
<td>Tree traversal</td>
<td>564</td>
</tr>
<tr>
<td>31.3</td>
<td>Depth-first search</td>
<td>565</td>
</tr>
<tr>
<td>31.4</td>
<td>Sources used in this chapter</td>
<td>570</td>
</tr>
</tbody>
</table>
### III PETSc

#### 32 PETSc basics
- 32.1 What is PETSc and why? ......................................................... 572
- 32.2 Basics of running a PETSc program ........................................... 574
- 32.3 PETSc installation ..................................................................... 578
- 32.4 Sources used in this chapter ..................................................... 580

#### 33 PETSc objects
- 33.1 Distributed objects ................................................................. 581
- 33.2 Scalars ..................................................................................... 582
- 33.3 Vec: Vectors ............................................................................. 584
- 33.4 Mat: Matrices ........................................................................... 595
- 33.5 Index sets and Vector Scatters ................................................... 606
- 33.6 AO: Application Orderings ....................................................... 608
- 33.7 Partitionings ............................................................................ 608
- 33.8 Sources used in this chapter ..................................................... 610

#### 34 Grid support
- 34.1 Grid definition ........................................................................... 617
- 34.2 General Data Management ....................................................... 621
- 34.3 Constructing a vector on a grid .................................................. 624
- 34.4 Vectors of a distributed array .................................................... 624
- 34.5 Matrices of a distributed array ................................................... 625
- 34.6 Sources used in this chapter ..................................................... 627

#### 35 Finite Elements support
- 35.1 Sources used in this chapter ..................................................... 635

#### 36 PETSc solvers
- 36.1 KSP: linear system solvers ....................................................... 636
- 36.2 Direct solvers ............................................................................ 646
- 36.3 Control through command line options ..................................... 646
- 36.4 Sources used in this chapter ..................................................... 648

#### 37 PETSc nonlinear solvers
- 37.1 Nonlinear systems ................................................................... 649
- 37.2 Time-stepping .......................................................................... 651
- 37.3 Sources used in this chapter ..................................................... 653

#### 38 PETSc GPU support
- 38.1 Installation with GPUs ............................................................. 654
- 38.2 Setup for GPU .......................................................................... 654
- 38.3 Distributed objects ................................................................... 654
- 38.4 Other ......................................................................................... 655
38.5 Sources used in this chapter ............................................. 656

39 PETSc tools ........................................................................... 657
  39.1 Error checking and debugging ........................................... 657
  39.2 Program output ............................................................... 659
  39.3 Commandline options ...................................................... 663
  39.4 Timing and profiling ......................................................... 665
  39.5 Memory management ....................................................... 666
  39.6 Sources used in this chapter .............................................. 668

40 PETSc topics ......................................................................... 672
  40.1 Communicators ............................................................... 672
  40.2 Sources used in this chapter .............................................. 673

IV Other programming models .................................................. 675

41 Co-array Fortran ..................................................................... 676
  41.1 History and design ............................................................ 676
  41.2 Compiling and running ..................................................... 676
  41.3 Basics ............................................................................. 676
  41.4 Sources used in this chapter .............................................. 680

42 Sycl, OneAPI, DPC++ .............................................................. 681
  42.1 Logistics ........................................................................ 681
  42.2 Platforms and devices ...................................................... 682
  42.3 Queues .......................................................................... 682
  42.4 Kernels .......................................................................... 683
  42.5 Parallel operations ............................................................ 684
  42.6 Memory access ............................................................... 688
  42.7 Parallel output ............................................................... 691
  42.8 DPCPP extensions ............................................................ 691
  42.9 Intel devcloud notes ........................................................ 691
  42.10 Examples ..................................................................... 692
  42.11 Sources used in this chapter ........................................... 693

43 Python multiprocessing ........................................................ 701
  43.1 Software and hardware .................................................... 701
  43.2 Process ......................................................................... 701
  43.3 Pools and mapping ........................................................ 703
  43.4 Shared data .................................................................... 705
  43.5 Sources used in this chapter ............................................ 705
## V The Rest

### 44 Exploring computer architecture

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>44.1 Tools for discovery</td>
<td>708</td>
</tr>
<tr>
<td>44.2 Sources used in this chapter</td>
<td>709</td>
</tr>
</tbody>
</table>

### 45 Hybrid computing

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>45.1 Affinity</td>
<td>710</td>
</tr>
<tr>
<td>45.2 What does the hardware look like?</td>
<td>711</td>
</tr>
<tr>
<td>45.3 Affinity control</td>
<td>712</td>
</tr>
<tr>
<td>45.4 Discussion</td>
<td>712</td>
</tr>
<tr>
<td>45.5 Processes and cores and affinity</td>
<td>715</td>
</tr>
<tr>
<td>45.6 Practical specification</td>
<td>716</td>
</tr>
<tr>
<td>45.7 Sources used in this chapter</td>
<td>718</td>
</tr>
</tbody>
</table>

### 46 Random number generation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>46.1 Sources used in this chapter</td>
<td>719</td>
</tr>
</tbody>
</table>

### 47 Parallel I/O

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>47.1 Sources used in this chapter</td>
<td>720</td>
</tr>
</tbody>
</table>

### 48 Support libraries

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>48.1 SimGrid</td>
<td>721</td>
</tr>
<tr>
<td>48.2 Other</td>
<td>722</td>
</tr>
<tr>
<td>48.3 Sources used in this chapter</td>
<td>723</td>
</tr>
</tbody>
</table>

## VI Tutorials

### 49 Debugging

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>49.1 Step 0: compiling for debug</td>
<td>727</td>
</tr>
<tr>
<td>49.2 Invoking gdb</td>
<td>728</td>
</tr>
<tr>
<td>49.3 Finding errors</td>
<td>729</td>
</tr>
<tr>
<td>49.4 Memory debugging with Valgrind</td>
<td>730</td>
</tr>
<tr>
<td>49.5 Stepping through a program</td>
<td>731</td>
</tr>
<tr>
<td>49.6 Inspecting values</td>
<td>732</td>
</tr>
<tr>
<td>49.7 Parallel debugging</td>
<td>733</td>
</tr>
<tr>
<td>49.8 Further reading</td>
<td>734</td>
</tr>
</tbody>
</table>

### 50 Tracing, timing, and profiling

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>50.1 Timing</td>
<td>735</td>
</tr>
<tr>
<td>50.2 Tau</td>
<td>736</td>
</tr>
</tbody>
</table>

### 51 SimGrid

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>51 SimGrid</td>
<td>744</td>
</tr>
</tbody>
</table>
## 52 Batch systems

- **Cluster structure** ........................................... 746
- **Queues** ..................................................... 747
- **Job running** ............................................... 748
- **The script file** ............................................. 748
- **Parallelism handling** .................................... 750
- **Job running** ............................................... 751
- **Scheduling strategies** .................................... 751
- **File systems** .............................................. 752
- **Examples** .................................................. 752
- **Review questions** ......................................... 753

## 53 Parallel I/O

- **Use sequential I/O** ....................................... 755
- **MPI I/O** .................................................... 755
- **Higher level libraries** .................................... 755

## VII Class projects

## 54 A Style Guide to Project Submissions

## 55 Warmup Exercises

- **Hello world** ................................................ 761
- **Collectives** ................................................ 761
- **Linear arrays of processors** ............................. 762

## 56 Mandelbrot set

- **Invocation** ................................................ 766
- **Tools** ....................................................... 766
- **Bulk task scheduling** ..................................... 768
- **Collective task scheduling** ............................... 769
- **Asynchronous task scheduling** ......................... 770
- **One-sided solution** ....................................... 770

## 57 Data parallel grids

- **Description of the problem** .............................. 771
- **Code basics** ............................................... 772

## 58 N-body problems

- **Solution methods** ......................................... 774
- **Shared memory approaches** ............................. 774
- **Distributed memory approaches** ....................... 774
## VIII Didactics

### 59 Teaching guide

#### 59.1 Sources used in this chapter

### 60 Teaching from mental models

#### 60.1 Introduction
#### 60.2 Implied mental models
#### 60.3 Teaching MPI, the usual way
#### 60.4 Teaching MPI, our proposal
#### 60.5 ‘Parallel computer games’
#### 60.6 Further course summary
#### 60.7 Prospect for an online course
#### 60.8 Evaluation and discussion
#### 60.9 Summary
#### 60.10 Sources used in this chapter

## IX Bibliography, index, and list of acronyms

### 61 Bibliography

### 62 List of acronyms

### 63 General Index

### 64 Lists of notes

#### 64.1 MPI-4 notes
#### 64.2 Fortran notes
#### 64.3 C++ notes
#### 64.4 The MPL C++ interface
#### 64.5 Python notes

### 65 Index of MPI commands and keywords

#### 65.1 From the standard document
#### 65.2 MPI for Python

### 66 Index of OpenMP commands

### 67 Index of PETSc commands

### 68 Index of SYCL commands
PART I

MPI
This section of the book teaches MPI (‘Message Passing Interface’), the dominant model for distributed memory programming in science and engineering. It will instill the following competencies.

Basic level:
- The student will understand the SPMD model and how it is realized in MPI (chapter 2).
- The student will know the basic collective calls, both with and without a root process, and can use them in applications (chapter 3).
- The student knows the basic blocking and non-blocking point-to-point calls, and how to use them (chapter 4).

Intermediate level:
- The students knows about derived datatypes and can use them in communication routines (chapter 6).
- The student knows about intra-communicators, and some basic calls for creating subcommunicators (chapter 7); also Cartesian process topologies (section 11.1).
- The student understands the basic design of MPI I/O calls and can use them in basic applications (chapter 10).
- The student understands about graph process topologies and neighborhood collectives (section 11.2).

Advanced level:
- The student understands one-sided communication routines, including window creation routines, and synchronization mechanisms (chapter 9).
- The student understands MPI shared memory, its advantages, and ow it is based on windows (chapter 12).
- The student understands MPI process spawning mechanisms and inter-communicators (chapter 8).
Chapter 1

Getting started with MPI

In this chapter you will learn the use of the main tool for distributed memory programming: the Message Passing Interface (MPI) library. The MPI library has about 250 routines, many of which you may never need. Since this is a textbook, not a reference manual, we will focus on the important concepts and give the important routines for each concept. What you learn here should be enough for most common purposes. You are advised to keep a reference document handy, in case there is a specialized routine, or to look up subtleties about the routines you use.

1.1 Distributed memory and message passing

In its simplest form, a distributed memory machine is a collection of single computers hooked up with network cables. In fact, this has a name: a Beowulf cluster. As you recognize from that setup, each processor can run an independent program, and has its own memory without direct access to other processors’ memory. MPI is the magic that makes multiple instantiations of the same executable run so that they know about each other and can exchange data through the network.

One of the reasons that MPI is so successful as a tool for high performance on clusters is that it is very explicit: the programmer controls many details of the data motion between the processors. Consequently, a capable programmer can write very efficient code with MPI. Unfortunately, that programmer will have to spell things out in considerable detail. For this reason, people sometimes call MPI ‘the assembly language of parallel programming’. If that sounds scary, be assured that things are not that bad. You can get started fairly quickly with MPI, using just the basics, and coming to the more sophisticated tools only when necessary.

Another reason that MPI was a big hit with programmers is that it does not ask you to learn a new language: it is a library that can be interfaced to C/C++ or Fortran; there are even bindings to Python and Java (not described in this course). This does not mean, however, that it is simple to ‘add parallelism’ to an existing sequential program. An MPI version of a serial program takes considerable rewriting; certainly more than shared memory parallelism through OpenMP, discussed later in this book.

MPI is also easy to install: there are free implementations that you can download and install on any computer that has a Unix-like operating system, even if that is not a parallel machine. However, if you are working on a supercomputer cluster, likely there will already be an MPI installation, tuned to that machine’s network.
1. Getting started with MPI

1.2 History

Before the MPI standard was developed in 1993-4, there were many libraries for distributed memory computing, often proprietary to a vendor platform. MPI standardized the inter-process communication mechanisms. Other features, such as process management in PVM, or parallel I/O were omitted. Later versions of the standard have included many of these features.

Since MPI was designed by a large number of academic and commercial participants, it quickly became a standard. A few packages from the pre-MPI era, such as Charmpp [16], are still in use since they support mechanisms that do not exist in MPI.

1.3 Basic model

Here we sketch the two most common scenarios for using MPI. In the first, the user is working on an interactive machine, which has network access to a number of hosts, typically a network of workstations; see figure 1.1. The user types the command mpiexec and supplies

- The number of hosts involved,
- their names, possibly in a hostfile,
- and other parameters, such as whether to include the interactive host; followed by
- the name of the program and its parameters.

The mpiexec program then makes an ssh connection to each of the hosts, giving them sufficient information that they can find each other. All the output of the processors is piped through the mpiexec program, and appears on the interactive console.

In the second scenario (figure 1.2) the user prepares a batch job script with commands, and these will be run when the batch scheduler gives a number of hosts to the job. Now the batch script contains the mpiexec command, and the hostfile is dynamically generated when the job starts. Since the job now runs at a time when the user may not be logged in, any screen output goes into an output file.

---

1. A command variant is mpirun; your local cluster may have a different mechanism.
Figure 1.2: Batch MPI setup

You see that in both scenarios the parallel program is started by the `mpiexec` command using an Single Program Multiple Data (SPMD) mode of execution: all hosts execute the same program. It is possible for different hosts to execute different programs, but we will not consider that in this book.

There can be options and environment variables that are specific to some MPI installations, or to the network.

- `mpich` and its derivatives such as Intel MPI or Cray MPI have `mpiexec` options: https://www.mpich.org/static/docs/v3.1/www1/mpiexec.html

1.4 Making and running an MPI program

MPI is a library, called from programs in ordinary programming languages such as C/C++ or Fortran. To compile such a program you use your regular compiler:

```
gcc -c my_mpi_prog.c -I/path/to/mpi.h
gcc -o my_mpiProg my_mpiProg.o -L/path/to/mpi -lmpich
```

However, MPI libraries may have different names between different architectures, making it hard to have a portable makefile. Therefore, MPI typically has shell scripts around your compiler call, called `mpicc`, `mpicxx`, `mpif90` for C/C++/Fortran respectively.

```
mpicc -c my_mpi_prog.c
mpicc -o my_mpi_prog my_mpiProg.o
```

If you want to know what `mpicc` does, there is usually an option that prints out its definition. On a Mac with the `clang` compiler:

```
$ mpicc -show
clang -fPIC -fstack-protector -fno-stack-check -Qunused-arguments -g3 -O0 -Wno-implicit-function-declaration ...
```

**Remark 1** In OpenMPI, these commands are binary executables by default, but you can make it a shell script by passing the `--enable-script-wrapper-compilers` option at configure time.
1. Getting started with MPI

MPI programs can be run on many different architectures. Obviously it is your ambition (or at least your
dream) to run your code on a cluster with a hundred thousand processors and a fast network. But maybe
you only have a small cluster with plain ethernet. Or maybe you’re sitting in a plane, with just your laptop.
An MPI program can be run in all these circumstances – within the limits of your available memory of
course.

The way this works is that you do not start your executable directly, but you use a program, typically
called mpiexec or something similar, which makes a connection to all available processors and starts a
run of your executable there. So if you have a thousand nodes in your cluster, mpiexec can start your
program once on each, and if you only have your laptop it can start a few instances there. In the latter
case you will of course not get great performance, but at least you can test your code for correctness.

*Python note 1: Running mpi4py programs.* Load the TACC-provided python:

```
module load python
    and run it as:
ibrun python-mpi yourprogram.py
```

1.5 Language bindings

1.5.1 C

The MPI library is written in C. However, the standard is careful to distinguish between MPI routines,
versus their C bindings. In fact, as of MPI MPI-4, for a number of routines there are two bindings, depending
on whether you want 4 byte integers, or larger. See section 6.4, in particular 6.4.1.

1.5.2 C++, including MPL

C++ bindings were defined in the standard at one point, but they were declared deprecated, and have been
officially removed in the MPI 3MPI-3 Thus, MPI can be used from C++ by including

```
#include <mpi.h>
```

and using the C API.

The boost library has its own version of MPI, but it seems not to be under further development. A recent
effort at idiomatic C++ support is Message Passing Layer (MPL) [https://github.com/rabauke/mpl](https://github.com/rabauke/mpl).
This book has an index of MPL notes and commands: section 64.4.

*MPL note 1: Notes format.* MPL is a C++ header-only library. Notes on MPI usage from MPL will be indi-
cated like this.

1.5.3 Fortran

*Fortran note 1: Formatting of Fortran notes.* Fortran-specific notes will be indicated with a note like this.

Traditionally, Fortran bindings for MPI look very much like the C ones, except that each routine has a final
error return parameter. You will find that a lot of MPI code in Fortran conforms to this.
However, in the MPI standard it is recommended that an MPI implementation providing a Fortran interface provide a module named \texttt{mpi\_f08} that can be used in a Fortran program. This incorporates the following improvements:

- This defines MPI routines to have an optional final parameter for the error.
- There are some visible implications of using the \texttt{mpi\_f08} module, mostly related to the fact that some of the 'MPI datatypes' such as \texttt{MPI\_Comm}, which were declared as \texttt{Integer} previously, are now a Fortran Type. See the following sections for details: Communicator 7.1, Datatype 6.1, Info 15.1.1, Op 3.10.2, Request 4.2.1, Status 4.3.2, Window 9.1.
- The \texttt{mpi\_f08} module solves a problem with previous Fortran90 bindings: Fortran90 is a strongly typed language, so it is not possible to pass argument by reference to their address, as C/C++ do with the \texttt{void*} type for send and receive buffers. This was solved by having separate routines for each datatype, and providing an \texttt{Interface} block in the MPI module. If you manage to request a version that does not exist, the compiler will display a message like \texttt{There is no matching specific subroutine for this generic subroutine call [MPI\_Send]}

For details see \url{http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node409.htm}.

1.5.4 Python

Python note 2: Python notes. Python-specific notes will be indicated with a note like this.

The \texttt{mpi4py} package \cite{mpi4py} of python bindings is not defined by the MPI standards committee. Instead, it is the work of an individual, Lisandro Dalcin.

In a way, the Python interface is the most elegant. It uses Object-Oriented (OO) techniques such as methods on objects, and many default arguments.

Notable about the Python bindings is that many communication routines exist in two variants:

- a version that can send arbitrary Python objects. These routines have lowercase names such as \texttt{bcast}; and
- a version that sends \texttt{numpy} objects; these routines have names such as \texttt{Bcast}. Their syntax can be slightly different.

The first version looks more 'pythonic', is easier to write, and can do things like sending python objects, but it is also decidedly less efficient since data is packed and unpacked with \texttt{pickle}. As a common sense guideline, use the \texttt{numpy} interface in the performance-critical parts of your code, and the pythonic interface only for complicated actions in a setup phase.

Codes with \texttt{mpi4py} can be interfaced to other languages through Swig or conversion routines.

Data in \texttt{numpy} can be specified as a simple object, or \texttt{[data, (count,displ), datatype]}.

1.5.5 How to read routine signatures

Throughout the MPI part of this book we will give the reference syntax of the routines. This typically comprises:

- The semantics: routine name and list of parameters and what they mean.
- C syntax: the routine definition as it appears in the \texttt{mpi.h} file.
1. Getting started with MPI

- Fortran syntax: routine definition with parameters, giving in/out specification.
- Python syntax: routine name, indicating to what class it applies, and parameter, indicating which ones are optional.

These 'routine signatures' look like code but they are not! Here is how you translate them.

1.5.5.1 C

The typically C routine specification in MPI looks like:

```c
int MPI_Comm_size(MPI_Comm comm, int *nprocs)
```

This means that

- The routine returns an `int` parameter. Strictly speaking you should test against `MPI_SUCCESS` (for all error codes, see section 15.2.1):

  ```c
  MPI_Comm comm = MPI_COMM_WORLD;
  int nprocs;
  int errorcode;
  errorcode = MPI_Comm_size( MPI_COMM_WORLD, &nprocs);
  if (errorcode!=MPI_SUCCESS) {
    printf("Routine MPI_Comm_size failed! code=%d\n", errorcode);
    return 1;
  }
  ```

However, the error codes are hardly ever useful, and there is not much your program can do to recover from an error. Most people call the routine as

```c
MPI_Comm_size( /* parameter ... */);
```

For more on error handling, see section 15.2.

- The first argument is of type `MPI_Comm`. This is not a C built-in datatype, but it behaves like one. There are many of these `MPI_something` datatypes in MPI. So you can write:

  ```c
  MPI_Comm my_comm = MPI_COMM_WORLD; // using a predefined value
  MPI_Comm_size( comm, /* remaining parameters */);
  ```

- Finally, there is a ‘star’ parameter. This means that the routine wants an address, rather than a value. You would typically write:

  ```c
  MPI_Comm my_comm = MPI_COMM_WORLD; // using a predefined value
  int nprocs;
  MPI_Comm_size( comm, &nprocs );
  ```

Seeing a ‘star’ parameter usually means either: the routine has an array argument, or: the routine internally sets the value of a variable. The latter is the case here.

1.5.5.2 Fortran

The Fortran specification looks like:
1.5. Language bindings

```fortran
MPI_Comm_size(comm, size, ierror)
Type(MPI_Comm), Intent(In) :: comm
Integer, Intent(Out) :: size
Integer, Optional, Intent(Out) :: ierror
```

or for the Fortran90 legacy mode:

```fortran
MPI_Comm_size(comm, size, ierror)
INTEGER, INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

The syntax of using this routine is close to this specification: you write

```fortran
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
! Legacy: Integer :: comm = MPI_COMM_WORLD
Integer :: comm = MPI_COMM_WORLD
Integer :: size,ierr
CALL MPI_Comm_size( comm, size ) ! without the optional ierr
```

- Most Fortran routines have the same parameters as the corresponding C routine, except that they all have the error code as final parameter, instead of as a function result. As with C, you can ignore the value of that parameter. Just don’t forget it.
- The types of the parameters are given in the specification.
- Where C routines have `MPI_Comm` and `MPI_Request` and such parameters, Fortran has `INTEGER` parameters, or sometimes arrays of integers.

### 1.5.5.3 Python

The Python interface to MPI uses classes and objects. Thus, a specification like:

```python
MPI.Comm.Send(self, buf, int dest, int tag=0)
```

should be parsed as follows.

- First of all, you need the MPI class:
  ```python
  from mpi4py import MPI
  ```

- Next, you need a `Comm` object. Often you will use the predefined communicator
  ```python
  comm = MPI.COMM_WORLD
  ```

- The keyword `self` indicates that the actual routine `Send` is a method of the `Comm` object, so you call:
  ```python
  comm.Send( .... )
  ```

- Parameters that are listed by themselves, such as `buf`, as positional. Parameters that are listed with a type, such as `int dest` are keyword parameters. Keyword parameters that have a value specified, such as `int tag=0` are optional, with the default value indicated. Thus, the typical call for this routine is:
  ```python
  comm.Send(sendbuf,dest=other)
  ```
1. Getting started with MPI

specifying the send buffer as positional parameter, the destination as keyword parameter, and
using the default value for the optional tag.

Some python routines are 'class methods', and their specification lacks the `self` keyword. For instance:

\[
\text{\texttt{MPI.Request.Waitall}}(\text{\texttt{type cls, requests, statuses=None}})
\]

would be used as

\[
\text{\texttt{MPI.Request.Waitall}}(\text{\texttt{requests}})
\]

1.6 Review

**Review 1.1.** What determines the parallelism of an MPI job?

1. The size of the cluster you run on.
2. The number of cores per cluster node.
3. The parameters of the MPI starter (`mpiexec`, `ibrun`,...)

**Review 1.2.** T/F: the number of cores of your laptop is the limit of how many MPI processes you can start up.

**Review 1.3.** Do the following languages have an object-oriented interface to MPI? In what sense?

1. C
2. C++
3. Fortran2008
4. Python
1.7 Sources used in this chapter

1.7.1 Listing of code header
Chapter 2

MPI topic: Functional parallelism

2.1 The SPMD model

MPI programs conform largely to the Single Program Multiple Data (SPMD) model, where each processor runs the same executable. This running executable we call a *process*.

When MPI was first written, 20 years ago, it was clear what a processor was: it was what was in a computer on someone’s desk, or in a rack. If this computer was part of a networked cluster, you called it a *node*. So if you ran an MPI program, each node would have one MPI process; figure 2.1. You could of course run

![Cluster structure as of the mid 1990s](image)

Figure 2.1: Cluster structure as of the mid 1990s

more than one process, using the *time slicing* of the Operating System (OS), but that would give you no extra performance.

These days the situation is more complicated. You can still talk about a node in a cluster, but now a node can contain more than one processor chip (sometimes called a *socket*), and each processor chip probably has multiple *cores*. Figure 2.2 shows how you could explore this using a mix of MPI between the nodes, and a shared memory programming system on the nodes.

However, since each core can act like an independent processor, you can also have multiple MPI processes per node. To MPI, the cores look like the old completely separate processors. This is the ‘pure MPI’ model.
2.1. The SPMD model

![Hybrid cluster structure](image1)

Figure 2.2: Hybrid cluster structure

![MPI-only cluster structure](image2)

Figure 2.3: MPI-only cluster structure

of figure 2.3, which we will use in most of this part of the book. (Hybrid computing will be discussed in chapter 45.)

This is somewhat confusing: the old processors needed MPI programming, because they were physically separated. The cores on a modern processor, on the other hand, share the same memory, and even some caches. In its basic mode MPI seems to ignore all of this: each core receives an MPI process and the programmer writes the same send/receive call no matter where the other process is located. In fact, you can’t immediately see whether two cores are on the same node or different nodes. Of course, on the implementation level MPI uses a different communication mechanism depending on whether cores are
2. MPI topic: Functional parallelism

on the same socket or on different nodes, so you don’t have to worry about lack of efficiency.

**Remark 2** *In some rare cases you may want to run in an Multiple Program Multiple Data (MPMD) mode, rather than SPMD. This can be achieved either on the OS level (see section 15.9.4), using options of the mpiexec mechanism, or you can use MPI’s built-in process management; chapter 8. Like I said, this concerns only rare cases.*

### 2.2 Starting and running MPI processes

The SPMD model may be initially confusing. Even though there is only a single source, compiled into a single executable, the parallel run comprises a number of independently started MPI processes (see section 1.3 for the mechanism).

The following exercises are designed to give you an intuition for this one-source-many-processes setup. In the first exercise you will see that the mechanism for starting MPI programs starts up independent copies. There is nothing in the source that says ‘and now you become parallel’.

The following exercise demonstrates this point.

**Exercise 2.1.** Write a ‘hello world’ program, without any MPI in it, and run it in parallel with mpiexec or your local equivalent. Explain the output.

(There is a skeleton for this exercise under the name hello.)

This exercise is illustrated in figure 2.4.
2.2. Starting and running MPI processes

**Figure 2.1 MPI_Init**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.2 MPI_Finalize**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Finalize</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.2.1 Headers

If you use MPI commands in a program file, be sure to include the proper header file, `mpi.h` or `mpif.h`.

```c
#include "mpi.h" // for C
#include "mpif.h" ! for Fortran
```

For Fortran90, many MPI installations also have an MPI module, so you can write

```fortran
use mpi ! pre 3.0
use mpi_f08 ! 3.0 standard
```

The internals of these files can be different between MPI installations, so you can not compile one file against one `mpi.h` file and another file, even with the same compiler on the same machine, against a different MPI.

*Fortran note 2: New developments only in f08 module.* New language developments, such as large counts; section 6.4.2 will only be included in the `mpi_f08` module, not in the earlier mechanisms.

*Python note 3: Import mpi module.* It’s easiest to

```python
from mpi4py import MPI
```

*MPL note 2: Header file.* To compile MPL programs, add a line

```cpp
#include <mpl/mpl.hpp>
```

to your file.

### 2.2.2 Initialization / finalization

Every (useful) MPI program has to start with *MPI initialization* through a call to `MPI_Init` (figure 2.1), and have `MPI_Finalize` (figure 2.2) to finish the use of MPI in your program. The init call is different between the various languages.

In C, you can pass `argc` and `argv`, the arguments of a C language main program:

```c
int main(int argc, char **argv) {
    
    return 0;
}
```
2. MPI topic: Functional parallelism

(It is allowed to pass NULL for these arguments.)

Fortran (before 2008) lacks this commandline argument handling, so MPI_Init lacks those arguments.

After MPI_Finalize no MPI routines (with a few exceptions such as MPI_Finalized) can be called. In particular, it is not allowed to call MPI_Init again. If you want to do that, use the sessions model; section 8.3.

Python note 4: Initialize/finalize. In many cases, no initialize and finalize calls are needed: the statement

```python
from mpi4py import MPI

# stuff
MPI.Init()
```

performs the initialization. Likewise, the finalize happens at the end of the program.

However, for special cases, there is an mpi4py.rc object that can be set in between importing mpi4py and importing mpi4py.MPI:

```python
import mpi4py
mpi4py.rc.initialize = False
mpi4py.rc.finalize = False
from mpi4py import MPI
MPI.Init()
# stuff
MPI.Finalize()
```

**MPL note 3: Init, finalize.** There is no initialization or finalize call.

**MPL implementation note:** Initialization is done at the first mpi::environment method call, such as `comm_world`.

This may look a bit like declaring 'this is the parallel part of a program', but that's not true: again, the whole code is executed multiple times in parallel.

**Exercise 2.2.** Add the commands MPI_Init and MPI_Finalize to your code. Put three different print statements in your code: one before the init, one between init and finalize, and one after the finalize. Again explain the output.

Run your program on a large scale, using a batch job. Where does the output go?

Experiment with

```bash
MY_MPIRUN_OPTIONS="-prepend-rank" ibrun yourprogram
```

**Remark 3** For hybrid MPI-plus-threads programming there is also a call MPI_Init_thread. For that, see section 13.1.

2.2.2.1 Aborting an MPI run

Apart from MPI_Finalize, which signals a successful conclusion of the MPI run, an abnormal end to a run can be forced by MPI_Abort (figure 2.3). This stop execution on all processes associated with the communicator, but many implementations will terminate all processes. The value parameter is returned to the environment.
2.2. Starting and running MPI processes

The commandline arguments `argc` and `argv` are only guaranteed to be passed to process zero, so the best way to pass commandline information is by a broadcast (section 3.3.3).

There are a few commands, such as `MPI_Get_processor_name`, that are allowed before `MPI_Init`.

If MPI is used in a library, MPI can have already been initialized in a main program. For this reason, one can test where `MPI_Init` has been called with `MPI_Initialized` (figure 2.4).

You can test whether `MPI_Finalize` has been called with `MPI_Finalized` (figure 2.5).

2.2.2.2 Testing the initialized/finalized status

2.2.2.3 Information about the run

Once MPI has been initialized, the `MPI_INFO_ENV` object contains a number of key/value pairs describing run-specific information; see section 15.1.1.
2. MPI topic: Functional parallelism

Figure 2.5 MPI_Finalized

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Finalized</td>
<td>flag</td>
<td>true if MPI was finalized</td>
<td>int*</td>
<td>LOGICAL</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.6 MPI_Get_processor_name

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Get_processor_name</td>
<td>name</td>
<td>A unique specifier for the actual (as opposed to virtual) node.</td>
<td>char*</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>resultlen</td>
<td>Length (in printable characters) of the result returned in name</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
</tbody>
</table>

Python:

MPI.Get_processor_name()

2.2.2.4 Commandline arguments

The MPI_Init routines takes a reference to argc and argv for the following reason: the MPI_Init calls filters out the arguments to mpirun or mpiexec, thereby lowering the value of argc and eliminating some of the argv arguments.

On the other hand, the commandline arguments that are meant for mpiexec wind up in the MPI_INFO_ENV object as a set of key/value pairs; see section 15.1.1.

2.3 Processor identification

Since all processes in an MPI job are instantiations of the same executable, you’d think that they all execute the exact same instructions, which would not be terribly useful. You will now learn how to distinguish processes from each other, so that together they can start doing useful work.

2.3.1 Processor name

In the following exercise you will print out the hostname of each MPI process with MPI_Get_processor_name (figure 2.6) as a first way of distinguishing between processes. This routine has a character buffer argument, which needs to be allocated by you. The length of the buffer is also passed, and on return that parameter has the actually used length. The maximum needed length is MPI_MAX_PROCESSOR_NAME.
2.3. Processor identification

Code:

```c
// procname.c
int name_length = MPI_MAX_PROCESSOR_NAME;
char proc_name[name_length];
MPI_Get_processor_name(proc_name, &name_length);
printf("This process is running on node \\
\%s\n", proc_name);
```

Output:

make[2]: `procname' is up to date.
TACC: Starting up job 3954439
TACC: Starting parallel tasks...
This process is running on node <<c206-030 frontera.tacc.utexas.edu>>
This process is running on node <<c206-031 frontera.tacc.utexas.edu>>
This process is running on node <<c206-031 frontera.tacc.utexas.edu>>
This process is running on node <<c206-031 frontera.tacc.utexas.edu>>
This process is running on node <<c206-031 frontera.tacc.utexas.edu>>
This process is running on node <<c206-030 frontera.tacc.utexas.edu>>
This process is running on node <<c206-030 frontera.tacc.utexas.edu>>
TACC: Shutdown complete. Exiting.

(Underallocating the buffer will not lead to a runtime error.)

**Exercise 2.3.** Use the command `MPI_Get_processor_name`. Confirm that you are able to run a program that uses two different nodes.

_MPL note 4: Processor name._ The `processor_name` call is an environment method returning a `std::string`:

```c
std::string mpl::environment::processor_name ();
```

2.3.2 Communicators

First we need to introduce the concept of communicator, which is an abstract description of a group of processes. For now you only need to know about the existence of the `MPI_Comm` data type, and that there is a pre-defined communicator `MPI_COMM_WORLD` which describes all the processes involved in your parallel run.

In the procedural languages C, a communicator is a variable that is passed to most routines:

```c
#include <mpi.h>
MPI_Comm comm = MPI_COMM_WORLD;
MPI_Send( /* stuff */ comm );
```

_Fortran note 3: Communicator type._ In Fortran, pre-2008 a communicator was an opaque handle, stored in an `Integer`. With Fortran 2008, communicators are derived types:

```fortran
use mpi_f08
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
call MPI_Send( ... comm )
```

_Python note 5: Communicator objects._ In object-oriented languages, a communicator is an object, and rather than passing it to routines, the routines are often methods of the communicator object:

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD
comm.Send( buffer, target )
```

_MPL note 5: World communicator._ The naive way of declaring a communicator would be:
2. MPI topic: Functional parallelism

For the full source of this example, see section 2.7.2

calling the predefined environment method comm_world.

However, if the variable will always correspond to the world communicator, it is better to make it const and declare it to be a reference:

```c++
const mpl::communicator &comm_world = mpl::environment::comm_world();
```

For the full source of this example, see section 2.7.2

**MPL note 6: Communicator copying.** The communicator class has its copy operator deleted; however, copy initialization exists:

```c++
const mpl::communicator &comm = mpl::environment::comm_world();
cout << "same": " << boolalpha << (comm==comm) << endl;
```

**MPL note 7: Communicator passing.** Pass communicators by reference to avoid communicator duplication:

```c++
void comm_val( const mpl::communicator comm );
```

```c++
void comm_ref( const mpl::communicator &comm );
```

You will learn much more about communicators in chapter 7.

### 2.3.3 Process and communicator properties: rank and size

To distinguish between processes in a communicator, MPI provides two calls

1. **MPI_Comm_size** (figure 2.7) reports how many processes there are in all; and
2. **MPI_Comm_rank** (figure 2.8) states what the number of the process is that calls this routine.
2.3. Processor identification

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_size</td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>number of processes in group</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>of comm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MPL:

```cpp
int mpl::communicator::size ( ) const
```

Python:

```python
MPI.Comm.Get_size(self)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_rank</td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rank</td>
<td>rank of the calling process in</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>group of comm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MPL:

```cpp
int mpl::communicator::rank ( ) const
```

Python:

```python
MPI.Comm.Get_rank(self)
```

If every process executes the `MPI_Comm_size` call, they all get the same result, namely the total number of processes in your run. On the other hand, if every process executes `MPI_Comm_rank`, they all get a different result, namely their own unique number, an integer in the range from zero to the number of processes minus 1. See figure 2.5. In other words, each process can find out 'I am process 5 out of a total of 20'.

**Exercise 2.4.** Write a program where each process prints out a message reporting its number, and how many processes there are:

```
Hello from process 2 out of 5!
```

Write a second version of this program, where each process opens a unique file and writes to it. On some clusters this may not be advisable if you have large numbers of processors, since it can overload the file system.

(There is a skeleton for this exercise under the name commrank.)

**Exercise 2.5.** Write a program where only the process with number zero reports on how many processes there are in total.

In object-oriented approaches to MPI, that is, mpi4py and MPL, the `MPI_Comm_rank` and `MPI_Comm_size` routines are methods of the communicator class:

*Python note 6: Communicator rank and size.* Rank and size are methods of the communicator object. Note that their names are slightly different from the MPI standard names.
2. MPI topic: Functional parallelism

MPL note 8: Rank and size. The rank of a process (by `mpl::communicator::rank`) and the size of a communicator (by `mpl::communicator::size`) are both methods of the `communicator` class:

```cpp
const mpl::communicator &comm_world =
    mpl::environment::comm_world();
int procid = comm_world.rank();
int nprocs = comm_world.size();
```

2.4 Functional parallelism

Now that processes can distinguish themselves from each other, they can decide to engage in different activities. In an extreme case you could have a code that looks like

```cpp
// climate simulation:
if (procid==0)
    earth_model();
else if (procid==1)
    sea_model();
else
    air_model();
```
Practice is a little more complicated than this. But we will start exploring this notion of processes deciding on their activity based on their process number.

Being able to tell processes apart is already enough to write some applications, without knowing any other MPI. We will look at a simple parallel search algorithm: based on its rank, a processor can find its section of a search space. For instance, in Monte Carlo codes a large number of random samples is generated and some computation performed on each. (This particular example requires each MPI process to run an independent random number generator, which is not entirely trivial.)

**Exercise 2.6.** Is the number $N = 2,000,000,111$ prime? Let each process test a disjoint set of integers, and print out any factor they find. You don’t have to test all integers $< N$: any factor is at most $\sqrt{N} \approx 45,200$. (Hint: $i \% 0$ probably gives a runtime error.) Can you find more than one solution? (There is a skeleton for this exercise under the name prime.)

**Remark 4** Normally, we expect parallel algorithms to be faster than sequential. Now consider the above exercise. Suppose the number we are testing is divisible by some small prime number, but every process has a large block of numbers to test. In that case the sequential algorithm would have been faster than the parallel one. Food for thought.

As another example, in Boolean satisfiability problems a number of points in a search space needs to be evaluated. Knowing a process’s rank is enough to let it generate its own portion of the search space. The computation of the Mandelbrot set can also be considered as a case of functional parallelism. However, the image that is constructed is data that needs to be kept on one processor, which breaks the symmetry of the parallel run.

Of course, at the end of a functionally parallel run you need to summarize the results, for instance printing out some total. The mechanisms for that you will learn next.

### 2.5 Distributed computing and distributed data

One reason for using MPI is that sometimes you need to work on a single object, say a vector or a matrix, with a data size larger than can fit in the memory of a single processor. With distributed memory, each processor then gets a part of the whole data structure and only works on that.

So let’s say we have a large array, and we want to distribute the data over the processors. That means that, with $p$ processes and $n$ elements per processor, we have a total of $n \cdot p$ elements.

We sometimes say that data is the local part of a distributed array with a total size of $n \cdot p$ elements. However, this array only exists conceptually: each processor has an array with lowest index zero, and you have to translate that yourself to an index in the global array. In other words, you have to write your code in such a way that it acts like you’re working with a large array that is distributed over the processors, while actually manipulating only the local arrays on the processors.

Your typical code then looks like
Exercise 2.7. Allocate on each process an array:

\[
\text{int my_ints[10];}
\]

and fill it so that process 0 has the integers 0…9, process 1 has 10…19, et cetera. It may be hard to print the output in a non-messy way.

If the array size is not perfectly divisible by the number of processors, we have to come up with a division that is uneven, but not too much. You could for instance, write

\[
\text{int Nglobal; // is something large}
\]
\[
\text{Nlocal = Nglobal/ntids,}
\]
\[
\text{excess = Nglobal\%ntids;}
\]
\[
\text{if (mytid==ntids-1)}
\]
\[
\text{Nlocal += excess;}
\]

Exercise 2.8. Argue that this strategy is not optimal. Can you come up with a better distribution? Load balancing is further discussed in HPSC-2.10.

2.6 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

Exercise 2.9. True or false: mpicc is a compiler.

Exercise 2.10. T/F?

1. In C, the result of \texttt{MPI\_Comm\_rank} is a number from zero to number-of-processes-minus-one, inclusive.

2. In Fortran, the result of \texttt{MPI\_Comm\_rank} is a number from one to number-of-processes, inclusive.

Exercise 2.11. What is the function of a hostfile?
2.7. Sources used in this chapter

2.7.1 Listing of code header

2.7.2 Listing of code examples/mpi/mpl/commrank.cxx

```cpp
#include <cstdlib>
#include <iostream>
#include <mpl/mpl.hpp>

int main() {
#if 1
    mpl::communicator comm_world =
        mpl::environment::comm_world();
#else
    const mpl::communicator &comm_world =
        mpl::environment::comm_world();
#endif
    std::cout << "Hello world! I am running on ""
        << mpl::environment::processor_name() 
        << ". My rank is "
        << comm_world.rank() 
        << " out of "
        << comm_world.size() << " processes.\n" << std::endl;
    return EXIT_SUCCESS;
}
```

2.7.3 Listing of code examples/mpi/mpl/commcompare.cxx

```cpp
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <mpl/mpl.hpp>
using namespace std;

int main() {

    const mpl::communicator &comm =
        mpl::environment::comm_world();
    cout << "same: " << boolalpha << (comm==comm) << endl;

    mpl::communicator copy =
        mpl::environment::comm_world();
    cout << "copy: " << boolalpha << (comm==copy) << endl;

    mpl::communicator init = comm;
    cout << "init: " << boolalpha << (init==comm) << endl;

    // WRONG: copy = comm;
```
2. MPI topic: Functional parallelism

```cpp
// error: overload resolution selected deleted operator '='
{
    mpl::communicator init;
    // WRONG: init = mpl::environment::comm_world();
    // error: overload resolution selected deleted operator '='
}

auto eq = comm.compare(copy);
cout << static_cast<int>(eq) << endl;

cout << "Rawcompare\n";
{
    const mpl::communicator &comm =
    mpl::environment::comm_world();
    MPI_Comm
        world_extract = comm.native_handle(),
        world_given = MPI_COMM_WORLD;
    int result;
    MPI_Comm_compare(world_extract, world_given, &result);
    cout << "Compare raw comms: " << "\n"
<< "identical: " << (result == MPI_IDENT)
<< "\n"
<< "congruent: " << (result == MPI_CONGRUENT)
<< "\n"
<< "unequal : " << (result == MPI_UNEQUAL)
<< "\n";
}
cout << "rawcompare\n";

return EXIT_SUCCESS;
}
Chapter 3

MPI topic: Collectives

A certain class of MPI routines are called ‘collective’, or more correctly: ‘collective on a communicator’. This means that if process one in that communicator calls that routine, they all need to call that routine. In this chapter we will discuss collective routines that are about combining the data on all processes in that communicator, but there are also operations such as opening a shared file that are collective, which will be discussed in a later chapter.

3.1 Working with global information

If all processes have individual data, for instance the result of a local computation, you may want to bring that information together, for instance to find the maximal computed value or the sum of all values. Conversely, sometimes one processor has information that needs to be shared with all. For this sort of operation, MPI has collectives.

There are various cases, illustrated in figure 3.1, which you can (sort of) motivate by considering some classroom activities:

- The teacher tells the class when the exam will be. This is a broadcast: the same item of information goes to everyone.
- After the exam, the teacher performs a gather operation to collect the invidivual exams.
- On the other hand, when the teacher computes the average grade, each student has an individual number, but these are now combined to compute a single number. This is a reduction.
- Now the teacher has a list of grades and gives each student their grade. This is a scatter operation, where one process has multiple data items, and gives a different one to all the other processes.

This story is a little different from what happens with MPI processes, because these are more symmetric; the process doing the reducing and broadcasting is no different from the others. Any process can function as the root process in such a collective.

Exercise 3.1. How would you realize the following scenarios with MPI collectives?

1. Let each process compute a random number. You want to print the maximum of these numbers to your screen.
2. Each process computes a random number again. Now you want to scale these numbers by their maximum.
3. Let each process compute a random number. You want to print on what processor the maximum value is computed.
Think about time and space complexity of your suggestions.

3.1.1 Practical use of collectives

Collectives are quite common in scientific applications. For instance, if one process reads data from disc or the commandline, it can use a broadcast or a gather to get the information to other processes. Likewise, at the end of a program run, a gather or reduction can be used to collect summary information about the program run.

However, a more common scenario is that the result of a collective is needed on all processes.

Consider the computation of the standard deviation:

$$\sigma = \sqrt{\frac{1}{N-1} \sum_{i}^{N} (x_i - \mu)^2}$$

where

$$\mu = \frac{\sum_{i}^{N} x_i}{N}$$

and assume that every process stores just one $x_i$ value.

1. The calculation of the average $\mu$ is a reduction, since all the distributed values need to be added.
2. Now every process needs to compute $x_i - \mu$ for its value $x_i$, so $\mu$ is needed everywhere. You can compute this by doing a reduction followed by a broadcast, but it is better to use a so-called \textit{allreduce} operation, which does the reduction and leaves the result on all processors.

3. The calculation of $\sum (x_i - \mu)$ is another sum of distributed data, so we need another reduction operation. Depending on whether each process needs to know $\sigma$, we can again use an allreduce.

### 3.1.2 Synchronization

Collectives are operations that involve all processes in a communicator. A collective is a single call, and it blocks on all processors, meaning that a process calling a collective cannot proceed until the other processes have similarly called the collective.

That does not mean that all processors exit the call at the same time: because of implementational details and network latency they need not be synchronized in their execution. However, semantically we can say that a process can not finish a collective until every other process has at least started the collective.

In addition to these collective operations, there are operations that are said to be ‘collective on their communicator’, but which do not involve data movement. Collective then means that all processors must call this routine; not to do so is an error that will manifest itself in ‘hanging’ code. One such example is \texttt{MPI\_File\_open}.

### 3.1.3 Collectives in MPI

We will now explain the MPI collectives in the following order.

\textbf{Allreduce} We use the allreduce as an introduction to the concepts behind collectives; section 3.2. As explained above, this routines serves many practical scenarios.

\textbf{Broadcast and reduce} We then introduce the concept of a root in the reduce (section 3.3.1) and broadcast (section 3.3.3) collectives.
- Sometimes you want a reduction with partial results, where each processor computes the sum (or other operation) on the values of lower-numbered processors. For this, you use a \textit{scan} collective (section 3.4).

\textbf{Gather and scatter} The gather/scatter collectives deal with more than a single data item; section 3.5.

There are more collectives or variants on the above.
- If every process needs to broadcast to every other, you use an \textit{all-to-all} operation (section 3.6).
- The reduce-scatter is a lesser known combination of collectives; section 3.7.
- A barrier is an operation that makes all processes wait until every process has reached the barrier (section 3.8).
- If you want to gather or scatter information, but the contribution of each processor is of a different size, there are ‘variable’ collectives; they have a \textit{v} in the name (section 3.9).

Finally, there are some advanced topics in collectives.
- User-defined reduction operators; section 3.10.2.
- Nonblocking collectives; section 3.11.
- We briefly discuss performance aspects of collectives in section 3.12.
- We discuss synchronization aspects in section 3.13.
3. MPI topic: Collectives

Figure 3.1 MPI_Allreduce

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allreduce</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Allreduce_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendbuf</td>
<td></td>
<td>starting address of send buffer</td>
<td>const void*</td>
<td>TYPE(*),</td>
<td>IN</td>
</tr>
<tr>
<td>recvbuf</td>
<td></td>
<td>starting address of receive buffer</td>
<td>void*</td>
<td>TYPE(*),</td>
<td>OUT</td>
</tr>
<tr>
<td>count</td>
<td></td>
<td>number of elements in send buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>datatype</td>
<td></td>
<td>datatype of elements in send buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td>op</td>
<td></td>
<td>operation</td>
<td>MPI_Op</td>
<td>TYPE(MPI_Op)</td>
<td>IN</td>
</tr>
<tr>
<td>comm</td>
<td></td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
</tbody>
</table>

MPL:

```cpp
template<typename T , typename F >
void mpl::communicator::allreduce
  ( F,const T & , T & ) const;
  ( F,const T *, T * ,
    const contiguous_layout< T > & ) const;
  ( F,T & , const contiguous_layout< T > & ) const;
  ( F,T *, const contiguous_layout< T > & ) const;

F : reduction function
T : type
Python:

MPI.Comm.Allreduce(self, sendbuf, recvbuf, Op op=SUM)
```

3.2 Reduction

3.2.1 Reduce to all

Above we saw a couple of scenarios where a quantity is reduced, with all proceses getting the result. The MPI call for this is **MPI_Allreduce** (figure 3.1).

Example: we give each process a random number, and sum these numbers together. The result should be approximately 1/2 times the number of processes.

```c
// allreduce.c
float myrandom, sumrandom;
myrandom = (float) rand()/(float)RAND_MAX;
// add the random variables together
MPI_Allreduce(&myrandom,&sumrandom,
  1,MPI_FLOAT,MPI_SUM,comm);
// the result should be approx nprocs/2:
if (procno==nprocs-1)
  printf("Result %.9f compared to .5\n",sumrandom/nprocs);
```

For the full source of this example, see section 3.16.2
Or:

```c
MPI_Count buffersize = 1000;
double *indata,*outdata;
indata = (double*) malloc( buffersize*sizeof(double) );
outdata = (double*) malloc( buffersize*sizeof(double) );
MPI_Allreduce_c(indata,outdata,buffersize,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
```

### 3.2.1.1 Buffer description

This is the first example in this course that involves MPI data buffers: the `MPI_Allreduce` call contains two buffer arguments. In most MPI calls (with the one-sided ones as big exception) a buffer is described by three parameters:

1. a pointer to the data,
2. the number of items in the buffer, and
3. the datatype of the items in the buffer.

Each of these needs some elucidation.

1. The buffer specification depends on the programming languages. Defaults are in section 3.2.4.
2. The count was a 4-byte integer in MPI standard up to and including MPI-3. In the MPI-4 standard the `MPI_Count` data type become allowed. See section 6.4 for details.
3. Datatypes can be elementary, as in the above example, or user-defined. See chapter 6 for details.

**Remark 5** Routines with both a send and receive buffer should not alias these. Instead, see the discussion of `MPI_IN_PLACE`; section 3.3.2.

### 3.2.1.2 Examples and exercises

**Exercise 3.2.** Let each process compute a random number, and compute the sum of these numbers using the `MPI_Allreduce` routine.

\[ \bar{\xi} = \sum_i x_i \]

Each process then scales its value by this sum.

\[ x'_i \leftarrow \frac{x_i}{\bar{\xi}} \]

Compute the sum of the scaled numbers

\[ \bar{\xi}' = \sum_i x'_i \]

and check that it is 1.

(There is a skeleton for this exercise under the name `randommax`.)
Exercise 3.3. Implement a (very simple-minded) Fourier transform: if $f$ is a function on the interval $[0,1]$, then the $n$-th Fourier coefficient is

$$f_n = \int_0^1 f(t)e^{-2\pi i x} \, dx$$

which we approximate by

$$f_n \approx \frac{1}{N} \sum_{i=0}^{N-1} f(ih)e^{-in\pi/N}$$

- Make one distributed array for the $e^{-in\pi/N}$ coefficients,
- make one distributed array for the $f(ih)$ values
- calculate a couple of coefficients

Exercise 3.4. In the previous exercise you worked with a distributed array, computing a local quantity and combining that into a global quantity. Why is it not a good idea to gather the whole distributed array on a single processor, and do all the computation locally?

MPL note 9: Reduction operator. The usual reduction operators are given as templated operators:

```cpp
float
xrank = static_cast<float>( comm_world.rank() ),
xreduce;
// separate recv buffer
comm_world.allreduce(mpl::plus<float>(), xrank, xreduce);
// in place
comm_world.allreduce(mpl::plus<float>(), xrank);
```

For the full source of this example, see section 3.16.3

Note the parentheses after the operator. Also note that the operator comes first, not last.

MPL implementation note: The reduction operator has to be compatible with $\mathbb{T}(\mathbb{T},\mathbb{T})$

For more about operators, see section 3.10.

3.2.2 Inner product as allreduce

One of the more common applications of the reduction operation is the inner product computation. Typically, you have two vectors $x$, $y$ that have the same distribution, that is, where all processes store equal parts of $x$ and $y$. The computation is then

```cpp
local_inprod = 0;
for (i=0; i<localsize; i++)
    local_inprod += x[i]*y[i];
MPI_Allreduce( &local_inprod, &global_inprod, 1, MPI_DOUBLE ... )
```

Exercise 3.5. The Gram-Schmidt method is a simple way to orthogonalize two vectors:

$$u \leftarrow u - (u'v)/(u'u)$$

Implement this, and check that the result is indeed orthogonal.

Suggestion: fill $v$ with the values $\sin 2nh\pi$ where $n = 2\pi/N$, and $u$ with $\sin 2nh\pi + \sin 4nh\pi$. What does $u$ become after orthogonalization?
3.2.3 Reduction operations

Several MPI_Op values are pre-defined. For the list, see section 3.10.1.

For use in reductions and scans it is possible to define your own operator.

\[
\text{MPI} \text{ _Op}_\text{create}( \text{MPI} \text{ _User} \text{ _function} \ast \text{func}, \text{int} \text{ commute}, \text{MPI} \text{ _Op} \ast \text{op});
\]

For more details, see section 3.10.2.

3.2.4 Data buffers

Collectives are the first example you see of MPI routines that involve transfer of user data. Here, and in every other case, you see that the data description involves:

- A buffer. This can be a scalar or an array.
- A datatype. This describes whether the buffer contains integers, single/double floating point numbers, or more complicated types, to be discussed later.
- A count. This states how many of the elements of the given datatype are in the send buffer, or will be received into the receive buffer.

These three together describe what MPI needs to send through the network.

In the various languages such a buffer/count/datatype triplet is specified in different ways.

First of all, in C the buffer is always an opaque handle, that is, a \texttt{void*} parameter to which you supply an address. This means that an MPI call can take two forms.

For scalars we need to use the ampersand operator to take the address:

\[
\text{float} \ x, y; \\
\text{MPI} \text{ _Allreduce}( \&x, \&y, 1, \text{MPI} \text{ _FLOAT}, \ldots);
\]

But for arrays we use the fact that arrays and addresses are more-or-less equivalent in:

\[
\text{float} \ xx[2], yy[2]; \\
\text{MPI} \text{ _Allreduce}( xx, yy, 2, \text{MPI} \text{ _FLOAT}, \ldots);
\]

You could cast the buffers and write:

\[
\text{MPI} \text{ _Allreduce}( (\text{void}* ) \&x, (\text{void}* ) \&y, 1, \text{MPI} \text{ _FLOAT}, \ldots); \\
\text{MPI} \text{ _Allreduce}( (\text{void}* ) xx, (\text{void}* ) yy, 2, \text{MPI} \text{ _FLOAT}, \ldots);
\]

but that is not necessary. The compiler will not complain if you leave out the cast.

\textit{C++ note 1: Buffer treatment.} Treatment of scalars in C++ is the same as in C. However, for arrays you have the choice between C-style arrays, and \texttt{std::vector} or \texttt{std::array}. For the latter there are two ways of dealing with buffers:

\[
\text{vector<float> } xx(25); \\
\text{MPI} \text{ _Send}( \text{xx} \text{ . data}(), 25, \text{MPI} \text{ _FLOAT}, \ldots); \\
\text{MPI} \text{ _Send}( \&xx[0], 25, \text{MPI} \text{ _FLOAT}, \ldots);
\]
3. MPI topic: Collectives

Fortran note 4: MPI send/recv buffers. In Fortran parameters are always passed by reference, so the buffer is treated the same way:

```fortran
Real*4 :: x
Real*4, dimension(2) :: xx
call MPI_Allreduce( x,1,MPI_REAL4, ... )
call MPI_Allreduce( xx,2,MPI_REAL4, ... )
```

In discussing OO languages, we first note that the official C++ Application Programmer Interface (API) has been removed from the standard.

Specification of the buffer/count/datatype triplet is not needed explicitly in OO languages.

Python note 7: Buffers from numpy. Most MPI routines in Python have both a variant that can send arbitrary Python data, and one that is based on numpy arrays. The former looks the most 'pythonic', and is very flexible, but is usually demonstrably inefficient.

```python
# allreduce.py
random_number = random.randint(1,random_bound)
# native mode send
max_random = comm.allreduce(random_number,op=MPI.MAX)
```

For the full source of this example, see section 3.16.4

In the numpy variant, all buffers are numpy objects, which carry information about their type and size. For scalar reductions this means we have to create an array for the receive buffer, even though only one element is used.

```python
myrandom = np.empty(1,dtype=int)
myrandom[0] = random_number
allrandom = np.empty(nprocs,dtype=int)
# numpy mode send
comm.Allreduce(myrandom,allrandom[:1],op=MPI.MAX)
```

For the full source of this example, see section 3.16.4

Python note 8: Buffers from subarrays. In many examples you will pass a whole Numpy array as send/receive buffer. Should want to use a buffer that corresponds to a subset of an array, you can use the following notation:

```python
MPI_Whatever( buffer[...],5 ) # more stuff
```

for passing the buffer that starts at location 5 of the array.

MPL note 10: Scalar buffers. Buffer type handling is done through polymorphism and templating: no explicit indication of types.

Scalars are handled as such:

```c
float x,y;
comm.bcast( 0,x ); // note: root first
comm.allreduce( mpl::plus<float>(), x,y );
```

where the reduction function needs to be compatible with the type of the buffer.

MPL note 11: Vector buffers. If your buffer is a `std::vector` you need to take the `.data()` component of it:
3.3. Rooted collectives: broadcast, reduce

In some scenarios there is a certain process that has a privileged status.

- One process can generate or read in the initial data for a program run. This then needs to be communicated to all other processes.
- At the end of a program run, often one process needs to output some summary information.

This process is called the root process, and we will now consider routines that have a root.

3.3.1 Reduce to a root

In the broadcast operation a single data item was communicated to all processes. A reduction operation with MPI_Reduce (figure 3.2) goes the other way: each process has a data item, and these are all brought together into a single item.

Here are the essential elements of a reduction operation:

```c++
MPI_Reduce( senddata, recvdata..., operator, root, comm );
```

- There is the original data, and the data resulting from the reduction. It is a design decision of MPI that it will not by default overwrite the original data. The send data and receive data are of the same size and type: if every processor has one real number, the reduced result is again one real number.
- It is possible to indicate explicitly that a single buffer is used, and thereby the original data overwritten; see section 3.3.2 for this 'in place' mode.
- There is a reduction operator. Popular choices are MPI_SUM, MPI_PROD and MPI_MAX, but complicated operators such as finding the location of the maximum value exist. (For the full list, see section 3.10.1.) You can also define your own operators; section 3.10.2.

---

Victor Eijkhout 49
Figure 3.2 MPI_Reduce

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Reduce</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Reduce_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendbuf</td>
<td>address of send buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(1..)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>recvbuf</td>
<td>address of receive buffer</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(1..)</td>
<td>OUT</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>number of elements in send buffer</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td>datatype of elements of send buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td>datatype</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op</td>
<td>reduce operation</td>
<td>MPI_Op</td>
<td>TYPE(MPI_Op)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>root</td>
<td>rank of root process</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
<td></td>
</tr>
</tbody>
</table>

MPL:

```cpp
void mpl::communicator::reduce
    // root, in place
    ( F f, int root_rank, T & sendrecvdata ) const
    ( F f, int root_rank, T * sendrecvdata, const contiguous_layout< T > & l ) const

    // non-root
    ( F f, int root_rank, const T & senddata ) const
    ( F f, int root_rank,
      const T * senddata, const contiguous_layout< T > & l ) const

    // general
    ( F f, int root_rank, const T & senddata, T & recvdata ) const
    ( F f, int root_rank,
      const T * senddata, T * recvdata, const contiguous_layout< T > & l ) const
```

Python:

```python
comm.Reduce(self, sendbuf, recvbuf, Op op=SUM, int root=0)
native:
comm.reduce(self, sendobj=None, recvobj=None, op=SUM, int root=0)
```

- There is a root process that receives the result of the reduction. Since the nonroot processes do not receive the reduced data, they can actually leave the receive buffer undefined.

```c
// reduce.c
float myrandom = (float) rand()/(float)RAND_MAX,
    result;
int target_proc = nprocs-1;
// add all the random variables together
MPI_Reduce(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,
    target_proc,comm);
// the result should be approx nprocs/2:
if (procno==target_proc)
   printf("Result %6.3f compared to nprocs/%2.1f\n",
         result,nprocs/2.);
```
3.3. Rooted collectives: broadcast, reduce

For the full source of this example, see section 3.16.6

Exercise 3.6. Write a program where each process computes a random number, and process 0 finds and prints the maximum generated value. Let each process print its value, just to check the correctness of your program.

Collective operations can also take an array argument, instead of just a scalar. In that case, the operation is applied pointwise to each location in the array.

Exercise 3.7. Create on each process an array of length 2 integers, and put the values 1, 2 in it on each process. Do a sum reduction on that array. Can you predict what the result should be? Code it. Was your prediction right?

3.3.2 Reduce in place

By default MPI will not overwrite the original data with the reduction result, but you can tell it to do so using the MPI_IN_PLACE specifier:

```c
// allreduceinplace.c
for (int irand=0; irand<nrandoms; irand++)
    myrandoms[irand] = (float) rand()/(float)RAND_MAX;
// add all the random variables together
MPI_Allreduce(MPI_IN_PLACE, myrandoms,
              nrandoms, MPI_FLOAT, MPI_SUM, comm);
```

For the full source of this example, see section 3.16.7

Now every process only has a receive buffer, so this has the advantage of saving half the memory. Each process puts its input values in the receive buffer, and these are overwritten by the reduced result.

The above example used MPI_IN_PLACE in MPI_Allreduce; in MPI_Reduce it’s little tricky. The reasoning is as follows:

- In MPI_Reduce every process has a buffer to contribute, but only the root needs a receive buffer. Therefore, MPI_IN_PLACE takes the place of the receive buffer on any processor except for the root ...
- ... while the root, which needs a receive buffer, has MPI_IN_PLACE takes the place of the send buffer. In order to contribute its value, the root needs to put this in the receive buffer.

Here is one way you could write the in-place version of MPI_Reduce:

```c
if (procno==root)
    MPI_Reduce(MPI_IN_PLACE, myrandoms,
               nrandoms, MPI_FLOAT, MPI_SUM, root, comm);
else
    MPI_Reduce(myrandoms, MPI_IN_PLACE,
               nrandoms, MPI_FLOAT, MPI_SUM, root, comm);
```

For the full source of this example, see section 3.16.7

However, as a point of style, having different versions of a a collective in different branches of a condition is infelicitous. The following may be preferable:
3. MPI topic: Collectives

```c
float *sendbuf,*recvbuf;
if (procno==root) {
    sendbuf = MPI_IN_PLACE; recvbuf = myrands;
} else {
    sendbuf = myrands; recvbuf = MPI_IN_PLACE;
}
MPI_Reduce(sendbuf,recvbuf,
nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
```

For the full source of this example, see section 3.16.7

In Fortran you can not do these address calculations. You can use the solution with a conditional:

```fortran
!! reduceinplace.F90
call random_number(mynumber)
target_proc = ntids-1;
! add all the random variables together
if (mytid.eq.target_proc) then
    result = mytid
    call MPI_Reduce(MPI_IN_PLACE,result,1,MPI_REAL,MPI_SUM,&
                    target_proc,comm)
else
    mynumber = mytid
    call MPI_Reduce(mynumber,result,1,MPI_REAL,MPI_SUM,&
                    target_proc,comm)
end if
```

For the full source of this example, see section 3.16.8

but you can also solve this with pointers:

```fortran
!! reduceinplaceptr.F90
in_place_val = MPI_IN_PLACE
if (mytid.eq.target_proc) then
    ! set pointers
    result_ptr => result
    mynumber_ptr => in_place_val
    ! target sets value in receive buffer
    result_ptr = mytid
else
    ! set pointers
    mynumber_ptr => mynumber
    result_ptr => in_place_val
    ! non-targets set value in send buffer
    mynumber_ptr = mytid
end if
call MPI_Reduce(mynumber_ptr,result_ptr,1,MPI_REAL,MPI_SUM,&
                target_proc,comm,err)
```

For the full source of this example, see section 3.16.9

**Python note 9: In-place collectives.** The value `MPI.IN_PLACE` can be used for the send buffer:

```python
# allreduceinplace.py
myrandom = np.empty(1,dtype=int)
myrandom[0] = random_number
```
3.3. Rooted collectives: broadcast, reduce

For the full source of this example, see section 3.16.10

MPL note 13: Reduce in place. The in-place variant is activated by specifying only one instead of two buffer arguments.

For the full source of this example, see section 3.16.3

Reducing a buffer requires specification of a contiguous_layout:

For the full source of this example, see section 3.16.11

Note that the buffers are of type T*, so it is necessary to take the data() of any std::vector and such.

MPL note 14: Reduce on non-root. There is a separate variant for non-root usage of rooted collectives:

3.3.3 Broadcast

A broadcast models the scenario where one process, the ‘root’ process, owns some data, and it communicates it to all other processes.

The broadcast routine(MPI_Bcast (figure 3.3) has the following structure:
3. MPI topic: Collectives

Figure 3.3 MPI_Bcast

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Bcast</td>
<td>buffer</td>
<td>starting address of buffer</td>
<td>void*</td>
<td>TYPE(*),</td>
<td>INOUT</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>number of entries in buffer</td>
<td>int</td>
<td>DIMENSION(..)</td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>datatype</td>
<td>datatype of buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>root</td>
<td>rank of broadcast root</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
</tbody>
</table>

MPL:

```cpp
template<typename T >
void mpl::communicator::bcast
(int root, T & data ) const
(int root, T * data, const layout< T > & l ) const
```

Python:

```python
MPI.Comm.Bcast(self, buf, int root=0)
```

Here:

- There is only one buffer, the send buffer. Before the call, the root process has data in this buffer; the other processes allocate a same sized buffer, but for them the contents are irrelevant.
- The root is the process that is sending its data. Typically, it will be the root of a broadcast tree.

Example: in general we cannot assume that all processes get the command line arguments, so we broadcast them from process 0.

```c
// init.c
if ( procs ==0 ) {
    if ( argc==1 ||  // the program is called without parameter
        ( argc>1 && strcmp(argv[1],"-h") ) // user asked for help
    ) {
        printf("\nUsage: init [0-9]+\n");
        MPI_Abort(comm,1);
    }
    input_argument = atoi(argv[1]);
}
MPI_Bcast(&input_argument,1,MPI_INT,0,comm);
```

For the full source of this example, see section 3.16.12

**Python note 10: Sending objects.** In python it is both possible to send objects, and to send more C-like buffers. The two possibilities correspond (see section 1.5.4) to different routine names; the buffers have to be created as numpy objects.

We illustrate both the general Python and numpy variants. In the former variant the result is given as a function return; in the numpy variant the send buffer is reused.
3.3. Rooted collectives: broadcast, reduce

```python
# first native
if procid==root:
    buffer = [ 5.0 ] * dsize
else:
    buffer = [ 0.0 ] * dsize
buffer = comm.bcast(obj=buffer, root=root)
if not reduce( lambda x,y:x and y, [ buffer[i]==5.0 for i in range(len(buffer)) ] ):
    print( "Something wrong on proc %d: native buffer <<%s>>" \
        % (procid,str(buffer)) )

# then with NumPy
buffer = np.arange(dsize, dtype=np.float64)
if procid==root:
    for i in range(dsize):
        buffer[i] = 5.0
    comm.Bcast( buffer, root=root )
if not all( buffer==5.0 ):
    print( "Something wrong on proc %d: numpy buffer <<%s>>" \
        % (procid,str(buffer)) )
else:
    if procid==root:
        print("Success.")
```

For the full source of this example, see section 3.16.13

MPL note 15: Broadcast. The broadcast call comes in two variants, with scalar argument and general layout:

```cpp
template<typename T >
void mpl::communicator::bcast( int root_rank, T &data ) const;
void mpl::communicator::bcast( int root_rank, T *data, const layout< T > &l ) const;
```

Note that the root argument comes first.

For the following exercise, study figure 3.2.

**Exercise 3.8.** The Gauss-Jordan algorithm for solving a linear system with a matrix $A$ (or computing its inverse) runs as follows:

for pivot $k = 1,...,n$

let the vector of scalings $t^{(k)}_i = A_{ik}/A_{kk}$

for row $r \neq k$

for column $c = 1,...,n$

$A_{rc} \leftarrow A_{rc} - t^{(k)}_r A_{kc}$

where we ignore the update of the righthand side, or the formation of the inverse. Let a matrix be distributed with each process storing one column. Implement the Gauss-Jordan algorithm as a series of broadcasts: in iteration $k$ process $k$ computes and broadcasts the scaling vector $\{ t^{(k)}_i \}_i$. Replicate the right-hand side on all processors.

Victor Eijkhout 55
3. MPI topic: Collectives

![Gauss-Jordan elimination example](image)

**Figure 3.2: Gauss-Jordan elimination example**
3.4. Scan operations

The MPI_Scan operation also performs a reduction, but it keeps the partial results. That is, if processor \( i \) contains a number \( x_i \), and \( \oplus \) is an operator, then the scan operation leaves \( x_0 \oplus \cdots \oplus x_i \) on processor \( i \). This type of operation is often called a prefix operation; see HPSC-27.

The MPI_Scan (figure 3.4) routine is an inclusive scan operation, meaning that it includes the data on the process itself; MPI_Exscan (see section 3.4.1) is exclusive, and does not include the data on the calling process.
3. MPI topic: Collectives

process: 0 1 2 ... p - 1  
data: x₀ x₁ x₂ ... xₚ₋₁  
inclusive: x₀ x₀ ⊕ x₁ x₀ ⊕ x₁ ⊕ x₂ ... ⊕ₗ=₀ xₗ  
exclusive: unchanged x₀ x₀ ⊕ x₁ ... ⊕ₗ=₀ xₗ

// scan.c  
// add all the random variables together  
MPI_Scan(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,comm);  
// the result should be approaching nprocs/2:  
if (procn==nprocs-1)  
printf("Result %6.3f compared to nprocs/2=%5.2fn",  
       result,nprocs/2.);

For the full source of this example, see section 3.16.14

In python mode the result is a function return value, with numpy the result is passed as the second parameter.

```
# scan.py  
mycontrib = 10+random.randint(1,nprocs)  
myfirst = 0  
mypartial = comm.scan(mycontrib)  
sbuf = np.empty(1,dtype=int)  
rbuf = np.empty(1,dtype=int)  
sbuf[0] = mycontrib  
comm.Scan(sbuf,rbuf)
```

For the full source of this example, see section 3.16.15

You can use any of the given reduction operators, (for the list, see section 3.10.1), or a user-defined one. In the latter case, the MPI_Op operations do not return an error code.

*MPL note 16: Scan operations. As in the C/F interfaces, MPL interfaces to the scan routines have the same calling sequences as the `Allreduce` routine.*

### 3.4.1 Exclusive scan

Often, the more useful variant is the exclusive scan MPI_Exscan (figure 3.5) with the same signature.

The result of the exclusive scan is undefined on processor 0 (None in python), and on processor 1 it is a copy of the send value of processor 1. In particular, the MPI_Op need not be called on these two processors.

**Exercise 3.10.** The exclusive definition, which computes x₀ ⊕ xₗ₋₁ on processor i, can be derived from the inclusive operation for operations such as MPI_SUM or MPI_PROD. Are there operators where that is not the case?

### 3.4.2 Use of scan operations

The MPI_Scan operation is often useful with indexing data. Suppose that every processor p has a local vector where the number of elements np is dynamically determined. In order to translate the local numbering
3.4. Scan operations

Figure 3.5 MPI_Exscan

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Exscan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Exscan_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendbuf</td>
<td>starting address</td>
<td>const void*</td>
<td>TYPE(*),</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>of send buffer</td>
<td>DIMENSION(..)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>recvbuf</td>
<td>starting address</td>
<td>void*</td>
<td>TYPE(*),</td>
<td>OUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>of receive buffer</td>
<td>DIMENSION(..)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>number of elements</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in input buffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td>datatype of elements</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>datatype</td>
<td></td>
<td>in input buffer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op</td>
<td>operation</td>
<td>MPI_Op</td>
<td>TYPE(MPI_Op)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td>intra-communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
<td></td>
</tr>
</tbody>
</table>

MPL:

```cpp
template<typename T , typename F >
void mpl::communicator::exscan
( F,const T &, T & ) const;
( F,const T *, T *,
  const contiguous_layout< T > & ) const;
( F,T & ) const;
( F,T *, const contiguous_layout< T > & ) const;
F : reduction function
T : type
```

Python:

```python
res = Intracomm.exscan( sendobj=None,recvobj=None,op=MPI.SUM)
```

0 \ldots n_p - 1 to a global numbering one does a scan with the number of local elements as input. The output is then the global number of the first local variable.

As an example, setting *Fast Fourier Transform (FFT)* coefficients requires this translation. If the local sizes are all equal, determining the global index of the first element is an easy calculation. For the irregular case, we first do a scan:

```c
// fft.c
MPI_Allreduce( &localsize,&globalsize,1,MPI_INT,MPI_SUM, comm );
globalsize += 1;
int myfirst=0;
MPI_Exscan( &localsize,&myfirst,1,MPI_INT,MPI_SUM, comm );

for ( int i=0; i<localsize; i++ )
  vector[i] = sin( pi*freq* (i+1+myfirst) / globalsize );
```

**Exercise 3.11.**

- Let each process compute a random value \( n_{\text{local}} \), and allocate an array of that length. Define

\[
N = \sum n_{\text{local}}
\]
3. MPI topic: Collectives

• Fill the array with consecutive integers, so that all local arrays, laid end-to-end, contain the numbers $0 \ldots N - 1$. (See figure 3.3.)

(There is a skeleton for this exercise under the name `scangather`.)

Exercise 3.12. Did you use `MPI_Scan` or `MPI_Exscan` for the previous exercise? How would you describe the result of the other scan operation, given the same input?

It is possible to do a segmented scan. Let $x_i$ be a series of numbers that we want to sum to $X_i$ as follows. Let $y_i$ be a series of booleans such that

\[
\begin{cases}
  X_i = x_i & \text{if } y_i = 0 \\
  X_i = X_{i-1} + x_i & \text{if } y_i = 1
\end{cases}
\]

(This is the basis for the implementation of the sparse matrix vector product as prefix operation; see HPSC-27.2.)

This means that $X_i$ sums the segments between locations where $y_i = 0$ and the first subsequent place where $y_i = 1$. To implement this, you need a user-defined operator

\[
\begin{pmatrix} X \\ x \\ y \end{pmatrix} = \begin{pmatrix} X_1 \\ x_1 \\ y_1 \end{pmatrix} \oplus \begin{pmatrix} X_2 \\ x_2 \\ y_2 \end{pmatrix} : \begin{cases}
  X = x_1 + x_2 & \text{if } y_2 == 1 \\
  X = x_2 & \text{if } y_2 == 0
\end{cases}
\]

This operator is not communitative, and it needs to be declared as such with `MPI_Op_create`; see section 3.10.2

3.5 Rooted collectives: gather and scatter

Figure 3.4: Gather collects all data onto a root
In the MPI_Scatter operation, the root spreads information to all other processes. The difference with a broadcast is that it involves individual information from/to every process. Thus, the gather operation typically has an array of items, one coming from each sending process, and scatter has an array, with an individual item for each receiving process; see figure 3.5.

These gather and scatter collectives have a different parameter list from the broadcast/reduce. The broadcast/reduce involves the same amount of data on each process, so it was enough to have a single datatype/size specification; for one buffer in the broadcast, and for both buffers in the reduce call. In the gather/scatter calls you have

- a large buffer on the root, with a datatype and size specification, and
- a smaller buffer on each process, with its own type and size specification.

In the gather and scatter calls, each processor has $n$ elements of individual data. There is also a root processor that has an array of length $np$, where $p$ is the number of processors. The gather call collects all this data from the processors to the root; the scatter call assumes that the information is initially on the root and it is spread to the individual processors.

Here is a small example:

```c
// gather.c
// we assume that each process has a value "localsize"
// the root process collects these values

if (procno==root)
    localsizes = (int*) malloc( nprocs*sizeof(int) );

// everyone contributes their info
MPI_Gather(&localsize,1,MPI_INT,
            localsizes,1,MPI_INT,root,comm);
```

*For the full source of this example, see section 3.16.16*

This will also be the basis of a more elaborate example in section 3.9.

**Exercise 3.13.** Let each process compute a random number. You want to print the maximum value and on what processor it is computed. What collective(s) do you
3. MPI topic: Collectives

Figure 3.6 MPI_Gather

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Gather</td>
<td>sendbuf</td>
<td>starting address of send buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>sendcount</td>
<td>number of elements in send buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>sendtype</td>
<td>datatype of send buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>recvbuf</td>
<td>address of receive buffer</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>recvcount</td>
<td>number of elements for any single receive</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>recvtype</td>
<td>datatype of recv buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>root</td>
<td>rank of receiving process</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
</tbody>
</table>

MPL:

```c
void mpl::communicator::gather
    ( int root_rank, const T & senddata ) const
    ( int root_rank, const T & senddata, T * recvdata ) const
    ( int root_rank, const T * senddata, const layout< T > & sendl ) const
    ( int root_rank, const T * senddata, const layout< T > & sendl, T * recvdata, const layout< T > & recvl ) const
```

Python:

```python
MPI.Comm.Gather
    (self, sendbuf, recvbuf, int root=0)
```

use? Write a short program.

The MPI_Scatter operation is in some sense the inverse of the gather: the root process has an array of length \( np \) where \( p \) is the number of processors and \( n \) the number of elements each processor will receive.

```c
int MPI_Scatter
    (void* sendbuf, int sendcount, MPI_Datatype sendtype,
     void* recvbuf, int recvcount, MPI_Datatype recvtype,
     int root, MPI_Comm comm)
```

Two things to note about these routines:

- The signature for MPI_Gather (figure 3.6) has two ‘count’ parameters, one for the length of the individual send buffers, and one for the receive buffer. However, confusingly, the second parameter (which is only relevant on the root) does not indicate the total amount of information coming in, but rather the size of each contribution. Thus, the two count parameters will usually be the same (at least on the root); they can differ if you use different MPI_Datatype values for the sending and receiving processors.
• While every process has a send buffer in the gather, and a receive buffer in the scatter call, only the root process needs the long array in which to gather, or from which to scatter. However, because in SPMD mode all processes need to use the same routine, a parameter for this long array is always present. Nonroot processes can use a null pointer here.
• More elegantly, the MPI_IN_PLACE option can be used for buffers that are not applicable, such as the receive buffer on a sending process. See section 3.3.2.

MPL note 17: Gather/scatter. Gathering (by communicator::gather) or scattering (by communicator::scatter) a single scalar takes a scalar argument and a raw array:

```cpp
vector<float> v;
float x;
comm_world.scatter(0, v.data(), x);
```

For the full source of this example, see section 3.16.11

If more than a single scalar is gathered, or scattered into, it becomes necessary to specify a layout:

```cpp
vector<float> vrecv(2), vsend(2*nprocs);
mpi::contiguous_layout<float> twonums(2);
comm_world.scatter(0, vsend.data(), twonums, vrecv.data(), twonums);
```

For the full source of this example, see section 3.16.11

MPL note 18: Gather on nonroot. Logically speaking, on every nonroot process, the gather call only has a send buffer. MPL supports this by having two variants that only specify the send data.

```cpp
if (procno==0) {
    vector<int> size_buffer(nprocs);
    comm_world.gather(0, my_number_of_elements, size_buffer.data());
} else {
    /*
    * If you are not the root, do versions with only send buffers
    */
    comm_world.gather(0, my_number_of_elements);
}
```

3.5.1 Examples

In some applications, each process computes a row or column of a matrix, but for some calculation (such as the determinant) it is more efficient to have the whole matrix on one process. You should of course only do this if this matrix is essentially smaller than the full problem, such as an interface system or the last coarsening level in multigrid.

Figure 3.6 pictures this. Note that conceptually we are gathering a two-dimensional object, but the buffer is of course one-dimensional. You will later see how this can be done more elegantly with the ‘subarray’ datatype; section 6.3.4.
Another thing you can do with a distributed matrix is to transpose it.

```c
// itransposeblock.c
for (int iproc=0; iproc<nprocs; iproc++) {
    MPI_Scatter( regular,1,MPI_DOUBLE,
                &transpose[iproc],1,MPI_DOUBLE,
                iproc,comm);
}
```

For the full source of this example, see section 3.16.17

In this example, each process scatters its column. This needs to be done only once, yet the scatter happens in a loop. The trick here is that a process only originates the scatter when it is the root, which happens only once. Why do we need a loop? That is because each element of a process’ row originates from a different scatter operation.

**Exercise 3.14.** Can you rewrite this code so that it uses a gather rather than a scatter? Does that change anything essential about structure of the code?

**Exercise 3.15.** Take the code from exercise 3.11 and extend it to gather all local buffers onto rank zero. Since the local arrays are of differing lengths, this requires MPI_Gatherv. How do you construct the lengths and displacements arrays? (There is a skeleton for this exercise under the name scangather.)

### 3.5.2 Allgather

The **MPI_Allgather** (figure 3.7) routine does the same gather onto every process: each process winds up with the totality of all data; figure 3.7.

This routine can be used in the simplest implementation of the dense matrix-vector product to give each processor the full input; see HPSC-6.2.2.

Some cases look like an all-gather but can be implemented more efficiently. Suppose you have two distributed vectors, and you want to create a new vector that contains those elements of the one that do not appear in the other. You could implement this by gathering the second vector on each processor, but this may be prohibitive in memory usage.
3.6. All-to-all

The all-to-all operation MPI_Alltoall (figure 3.8) can be seen as a collection of simultaneous broadcasts or simultaneous gathers. The parameter specification is much like an allgather, with a separate send and receive buffer, and no root specified. As with the gather call, the receive count corresponds to an individual receive, not the total amount.

Exercise 3.16. Can you think of another algorithm for taking the set difference of two distributed vectors. Hint: look up bucket brigade algorithm; section 4.1.5. What is the time and space complexity of this algorithm? Can you think of other advantages beside a reduction in workspace?

3.6 All-to-all

Victor Eijkhout
### 3. MPI topic: Collectives

#### Figure 3.8 MPI_Alltoall

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Alltoall</td>
<td></td>
<td>starting address of send buffer</td>
<td>const void*</td>
<td>TYPE(*),</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>MPI_Alltoall_c</td>
<td>number of elements sent to</td>
<td>int</td>
<td>DIMENSION(..)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sendbuf</td>
<td>each process</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendcount</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td>datatype of send buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td></td>
</tr>
<tr>
<td>sendtype</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>recvbuf</td>
<td></td>
<td>address of receive buffer</td>
<td>void*</td>
<td>TYPE(*),</td>
<td>OUT</td>
</tr>
<tr>
<td>recvcount</td>
<td>INTEGER</td>
<td>number of elements received</td>
<td>int</td>
<td>DIMENSION(..)</td>
<td></td>
</tr>
<tr>
<td>recvtype</td>
<td></td>
<td>from any process</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td>INTEGER</td>
<td>datatype of receive buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unlike the gather call, the send buffer now obeys the same principle: with a send count of 1, the buffer has a length of the number of processes.

### 3.6.1 All-to-all as data transpose

The all-to-all operation can be considered as a data transpose. For instance, assume that each process knows how much data to send to every other process. If you draw a connectivity matrix of size $P \times P$, the...
denoting who-sends-to-who, then the send information can be put in rows:

\[ \forall i : C[i,j] > 0 \quad \text{if process } i \text{ sends to process } j. \]

Conversely, the columns then denote the receive information:

\[ \forall j : C[i,j] > 0 \quad \text{if process } j \text{ receives from process } i. \]

The typical application for such data transposition is in the FFT algorithm, where it can take tens of percents of the running time on large clusters.

We will consider another application of data transposition, namely *radix sort*, but we will do that in a couple of steps. First of all:

**Exercise 3.17.** In the initial stage of a *radix sort*, each process considers how many elements to send to every other process. Use *MPI_Alltoall* to derive from this how many elements they will receive from every other process.

### 3.6.2 All-to-all-v

The major part of the *radix sort* algorithm consist of every process sending some of its elements to each of the other processes. The routine *MPI_Alltoallv* (figure 3.9) is used for this pattern:

- Every process scatters its data to all others,
- but the amount of data is different per process.

**Exercise 3.18.** The actual data shuffle of a *radix sort* can be done with *MPI_Alltoallv*. Finish the code of exercise 3.17.

### 3.7 Reduce-scatter

There are several MPI collectives that are functionally equivalent to a combination of others. You have already seen *MPI_Allreduce* which is equivalent to a reduction followed by a broadcast. Often such combinations can be more efficient than using the individual calls; see HPSC-6.1.

Here is another example: *MPI_Reduce_scatter* is equivalent to a reduction on an array of data (meaning a pointwise reduction on each array location) followed by a scatter of this array to the individual processes.

We will discuss this routine, or rather its variant *MPI_Reduce_scatter_block* (figure 3.10), using an important example: the *sparse matrix-vector product* (see HPSC-6.5.1 for background information). Each process contains one or more matrix rows, so by looking at indices the process can decide what other processes it needs to receive data from, that is, each process knows how many messages it will receive, and from which processes. The problem is for a process to find out what other processes it needs to send data to.

Let’s set up the data:
3. MPI topic: Collectives

Figure 3.9 MPI_Alltoallv

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Alltoallv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendbuf</td>
<td></td>
<td>starting address of send buffer</td>
<td>const void*</td>
<td>TYPE(*),</td>
<td>IN</td>
</tr>
<tr>
<td>sendcounts</td>
<td></td>
<td>non-negative integer array (of length group size) specifying the number of</td>
<td>const int[]</td>
<td>DIMENSION(..)</td>
<td></td>
</tr>
<tr>
<td>sdispls</td>
<td></td>
<td>integer array (of length group size). Entry j specifies the displacement</td>
<td>const int[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendtype</td>
<td></td>
<td>datatye of send buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td>recvbuf</td>
<td></td>
<td>address of receive buffer</td>
<td>void*</td>
<td>TYPE(*),</td>
<td>OUT</td>
</tr>
<tr>
<td>recvcounts</td>
<td></td>
<td>non-negative integer array (of length group size) specifying the number of</td>
<td>const int[]</td>
<td>DIMENSION(..)</td>
<td></td>
</tr>
<tr>
<td>rdispls</td>
<td></td>
<td>integer array (of length group size). Entry i specifies the displacement</td>
<td>const int[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>recvtype</td>
<td></td>
<td>datatye of receive buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td>comm</td>
<td></td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
</tbody>
</table>

// reducscatter.c
int
// data that we know:
*i_recv_from_proc = (int*) malloc(nprocs*sizeof(int)),
*procs_to_recv_from, nprocs_to_recv_from=0,
// data we are going to determin:
*procs_to_send_to,nprocs_to_send_to;

For the full source of this example, see section 3.16.18

Each process creates an array of ones and zeros, describing who it needs data from. Ideally, we only need the array procs_to_recv_from but initially we need the (possibly much larger) array i_recv_from_proc.
Next, the \texttt{MPI\_Reduce\_scatter\_block} call then computes, on each process, how many messages it needs to send.

\begin{verbatim}
MPI\_Reduce\_scatter\_block
(i\_recv\_from\_proc,&nprocs\_to\_send\_to,1,MPI\_INT,
MPI\_SUM,comm);
\end{verbatim}

\textit{For the full source of this example, see section 3.16.18}

We do not yet have the information to which processes to send. For that, each process sends a zero-size message to each of its senders. Conversely, it then does a receive to with \texttt{MPI\_ANY\_SOURCE} to discover who is requesting data from it. The crucial point to the \texttt{MPI\_Reduce\_scatter\_block} call is that, without it, a process would not know how many of these zero-size messages to expect.

\begin{verbatim}
/*
 * Send a zero-size msg to everyone that you receive from,
 * just to let them know that they need to send to you.
 */
MPI\_Request send\_requests[nprocs\_to\_recv\_from];
for (int iproc=0; iproc<nprocs\_to\_recv\_from; iproc++) {
    int proc=procs\_to\_recv\_from[iproc];
    double send\_buffer=0.0;
    MPI\_Isend(&send\_buffer,0,MPI\_DOUBLE, /*to:/*/ proc,0,comm,
                   &send\_requests[iproc]);
}

/*
 * Do as many receives as you know are coming in;
 * use wildcards since you don't know where they are coming from.
 * The source is a process you need to send to.
 */
procs\_to\_send\_to = (int*)malloc( nprocs\_to\_send\_to * sizeof(int) );
for (int iproc=0; iproc<nprocs\_to\_send\_to; iproc++) {
    double recv\_buffer;
    MPI\_Status status;
    MPI\_Recv(&recv\_buffer,0,MPI\_DOUBLE,MPI\_ANY\_SOURCE,MPI\_ANY\_TAG,comm,
                   &status);
}
\end{verbatim}
The **MPI_Reduce_scatter** (figure 3.10) call is more general: instead of indicating the mere presence of a message between two processes, by having individual receive counts one can, for instance, indicate the size of the messages.

We can look at reduce-scatter as a limited form of the all-to-all data transposition discussed above (section 3.6.1). Suppose that the matrix \( C \) contains only 0/1, indicating whether or not a messages is send, rather than the actual amounts. If a receiving process only needs to know how many messages to receive, rather than where they come from, it is enough to know the column sum, rather than the full column (see figure 3.9).

Another application of the reduce-scatter mechanism is in the dense matrix-vector product, if a two-dimensional data distribution is used.

### 3.7.1 Examples

An important application of this is establishing an irregular communication pattern. Assume that each process knows which other processes it wants to communicate with; the problem is to let the other processes know about this. The solution is to use **MPI_Reduce_scatter** to find out how many processes want to communicate with you:

```c
MPI_Reduce_scatter_block(i_recv_from_proc,&nprocs_to_send_to,1,MPI_INT,
```
For the full source of this example, see section 3.16.18

and then wait for precisely that many messages with a source value of \texttt{MPI\_ANY\_SOURCE}.

\begin{verbatim}
/*
 * Send a zero-size msg to everyone that you receive from,
 * just to let them know that they need to send to you.
 */
MPI_Request send_requests[nprocs_to_recv_from];
for (int iproc=0; iproc<nprocs_to_recv_from; iproc++) {
    int proc=procs_to_recv_from[iproc];
    double send_buffer=0.;
    MPI_Isend(&send_buffer,0,MPI\_DOUBLE,/*to:*/ proc,0,comm, &send_requests[iproc]);
}

/*
 * Do as many receives as you know are coming in;
 * use wildcards since you don't know where they are coming from.
 * The source is a process you need to send to.
 */
procs_to_send_to = (int*)malloc( nprocs_to_send_to * sizeof(int) );
for (int iproc=0; iproc<nprocs_to_send_to; iproc++) {
    double recv_buffer;
    MPI\_Status status;
    MPI\_Recv(&recv_buffer,0,MPI\_DOUBLE,MPI\_ANY\_SOURCE,MPI\_ANY\_TAG,comm, &status);
    procs_to_send_to[iproc] = status\_MPI\_SOURCE;
}
MPI\_Waitall(nprocs_to_recv_from,send_requests,MPI\_STATUSES\_IGNORE);
\end{verbatim}

For the full source of this example, see section 3.16.18

Use of \texttt{MPI\_Reduce\_scatter} to implement the two-dimensional matrix-vector product. Set up separate row and column communicators with \texttt{MPI\_Comm\_split}, use \texttt{MPI\_Reduce\_scatter} to combine local products.

\begin{verbatim}
MPI\_Allgather(&my\_x,1,MPI\_DOUBLE,
    local\_x,1,MPI\_DOUBLE,environ.col\_comm);
MPI\_Reduce\_scatter(local\_y,&my\_y,&ione,MPI\_DOUBLE,
    MPI\_SUM,environ.row\_comm);
\end{verbatim}

3.8 Barrier

A barrier call, \texttt{MPI\_Barrier} (figure 3.11) is a routine that blocks all processes until they have all reached the barrier call. Thus it achieves time synchronization of the processes.

This call’s simplicity is contrasted with its usefulness, which is very limited. It is almost never necessary to synchronize processes through a barrier: for most purposes it does not matter if processors are out of sync. Conversely, collectives (except the new nonblocking ones; section 3.11) introduce a barrier of sorts themselves.
3. MPI topic: Collectives

3.9 Variable-size-input collectives

In the gather and scatter call above each processor received or sent an identical number of items. In many cases this is appropriate, but sometimes each processor wants or contributes an individual number of items.

Let’s take the gather calls as an example. Assume that each processor does a local computation that produces a number of data elements, and this number is different for each processor (or at least not the same for all). In the regular MPI_Gather call the root processor had a buffer of size \( nP \), where \( n \) is the number of elements produced on each processor, and \( P \) the number of processors. The contribution from processor \( p \) would go into locations \( pn, \ldots, (p+1)n-1 \).

For the variable case, we first need to compute the total required buffer size. This can be done through a simple MPI_Reduce with MPI_SUM as reduction operator: the buffer size is \( \sum_p n_p \) where \( n_p \) is the number of elements on processor \( p \). But you can also postpone this calculation for a minute.

The next question is where the contributions of the processor will go into this buffer. For the contribution from processor \( p \) that is \( \sum_{q<p} n_{p}, \ldots, \sum_{q\leq p} n_{p} - 1 \). To compute this, the root processor needs to have all the \( n_p \) numbers, and it can collect them with an MPI_Gather call.

We now have all the ingredients. All the processors specify a send buffer just as with MPI_Gather. However, the receive buffer specification on the root is more complicated. It now consists of:

- outbuffer
- array-of-outcounts
- array-of-displacements
- outtype

and you have just seen how to construct that information.

For example, in an MPI_Gatherv (figure 3.12) call each process has an individual number of items to contribute. To gather this, the root process needs to find these individual amounts with an MPI_Gather call, and locally construct the offsets array. Note how the offsets array has size ntids+1: the final offset value is automatically the total size of all incoming data. See the example below.

There are various calls where processors can have buffers of differing sizes.

- In MPI_Scatterv (figure 3.13) the root process has a different amount of data for each recipient.
- In MPI_Gatherv, conversely, each process contributes a different sized send buffer to the received result; MPI_Allgatherv (figure 3.14) does the same, but leaves its result on all processes; MPI_Alltoallv does a different variable-sized gather on each process.

3.9.1 Example of Gatherv

We use MPI_Gatherv to do an irregular gather onto a root. We first need an MPI_Gather to determine offsets.
### 3.9. Variable-size-input collectives

**Figure 3.12 MPI_Gatherv**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Gatherv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Gatherv_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendbuf</td>
<td></td>
<td>starting address of send buffer</td>
<td>const void*</td>
<td>TYPE(*),</td>
<td>IN</td>
</tr>
<tr>
<td>sendcount</td>
<td></td>
<td>number of elements in send buffer</td>
<td>int</td>
<td>DIMENSION(..)</td>
<td></td>
</tr>
<tr>
<td>sendtype</td>
<td>INTEGER</td>
<td>datatype of send buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td></td>
</tr>
<tr>
<td>recvbuf</td>
<td></td>
<td>address of receive buffer</td>
<td>void*</td>
<td>TYPE(*),</td>
<td>OUT</td>
</tr>
<tr>
<td>recvcounts</td>
<td>INTEGER(*)</td>
<td>non-negative integer array (of length group size) containing the number of</td>
<td>const int[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>elements that are received from each process</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>displs</td>
<td>INTEGER(*)</td>
<td>integer array (of length group size). Entry i specifies the displacement</td>
<td>const int[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>relative to recvbuf at which to place the incoming data from process i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>recvtype</td>
<td>INTEGER(*)</td>
<td>datatype of recv buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td></td>
</tr>
<tr>
<td>root</td>
<td>int</td>
<td>rank of receiving process</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td>comm</td>
<td>MPI_Comm</td>
<td>communicator</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
<td>IN</td>
</tr>
</tbody>
</table>

**MPL:**

```cpp
template<typename T>
void gatherv
    (int root_rank, const T *senddata, const layout<T> &sendl,
     T *recvdata, const layouts<T> &recvls, const displacements &recvdispls) const
    (int root_rank, const T *senddata, const layout<T> &sendl,
     T *recvdata, const layouts<T> &recvls) const
    (int root_rank, const T *senddata, const layout<T> &sendl ) const
```

**Python:**

```python
Gatherv(self, sendbuf, [recvbuf,counts], int root=0)
```
3. MPI topic: Collectives

Figure 3.13 MPI_Scatterv

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Scatterv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Scatterv_c</td>
<td>address of send buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sendbuf</td>
<td>non-negative integer array (of length group size) specifying the number of elements to send to each rank</td>
<td>const int[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sendcounts</td>
<td>integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i</td>
<td>const int[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>displs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sendtype</td>
<td>datatype of send buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>recvbuf</td>
<td>address of receive buffer</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>recvcount</td>
<td>number of elements in receive buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>recvtype</td>
<td>datatype of receive buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>root</td>
<td>rank of sending process</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
</tbody>
</table>

Code:

```c
// gatherv.c
// we assume that each process has an array "localdata"
// of size "localsize"

// the root process decides how much data will be coming:
// allocate arrays to contain size and offset information
if (procno==root) {
  localsizes = (int*) malloc( nprocs*sizeof(int) );
  offsets = (int*) malloc( nprocs*sizeof(int) );
}
// everyone contributes their local size info
MPI_Gather(&localsize,1,MPI_INT,
            localsizes,1,MPI_INT,root,comm);
// the root constructs the offsets array
if (procno==root) {
  int total_data = 0;
  for (int i=0; i<nprocs; i++) {
    offsets[i] = total_data;
    total_data += localsizes[i];
  }
  alldata = (int*) malloc( total_data*sizeof(int) );
}
// everyone contributes their data
MPI_Gatherv(localdata,localsize,MPI_INT,
            &alldata,localsizes,offsets,MPI_INT,root,comm);
```

Output:

make[2]: 'gatherv' is up to date.
TACC: Starting up job 3954519
TACC: Starting parallel tasks...
Local sizes: 13, 9, 7, 10, 12, 5, 13, 6,
Collected:
0:1,1,1,1,1,1,1,1,1,1; 1:2,2,2,2,2,2,2,2; 2:3,3,3,3,3,3; 3:4,4,4,4,4,4,4,4,4,4; 4:5,5,5,5,5,5,5,5,5,5,5; 5:6,6,6,6; 6:7,7,7,7,7,7,7,7,7,7,7; 7:8,8,8,8,8;
TACC: Shutdown complete. Exiting.
3.9. Variable-size-input collectives

### Figure 3.14 MPI_Allgatherv

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allgatherv</td>
<td>sendbuf</td>
<td>starting address of send buffer</td>
<td>const void*</td>
<td>TYPE(*),</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DIMENSION(...)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sendcount</td>
<td>number of elements in send buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Allgatherv_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sendbuf</td>
<td></td>
<td>const void*</td>
<td>TYPE(*),</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DIMENSION(...)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sendcount</td>
<td></td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sendtype</td>
<td>datatype of send buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>recvbuf</td>
<td>address of receive buffer</td>
<td>void*</td>
<td>TYPE(*),</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DIMENSION(...)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>recvcounts</td>
<td>non-negative integer array (of length group size) containing the number of elements that are received from each process</td>
<td>const int[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Count</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>displs</td>
<td>integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i</td>
<td>const int[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Aint</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>recvtype</td>
<td>datatype of receive buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Comm.Allgatherv(self, sendbuf, recvbuf)
where recvbuf = "[ array, counts, displs, type]"
```

```
# gatherv.py
# implicitly using root=0
globalsize = comm.reduce(localsize)
if procid==0:
    print("Global size=\%d \% globalsize")
collecteddata = np.empty(globalsize,dtype=int)
counts = comm.gather(localsize)
comm.Gatherv(localdata, [collecteddata, counts])
```

For the full source of this example, see section 3.16.19

### 3.9.2 Example of Allgatherv

Prior to the actual gatherv call, we need to construct the count and displacement arrays. The easiest way is to use a reduction.

```
|| // allgatherv.c
```


3. MPI topic: Collectives

MPI_Allgather

( &my_count,1,MPI_INT,
  recv_counts,1,MPI_INT, comm );

int accumulate = 0;
for (int i=0; i<nprocs; i++) {
  recv_displs[i] = accumulate; accumulate += recv_counts[i]; }

int *global_array = (int*) malloc(accumulate*sizeof(int));

MPI_Allgatherv

( my_array,procno+1,MPI_INT,
  global_array,recv_counts,recv_displs,MPI_INT, comm );

For the full source of this example, see section 3.16.20

In python the receive buffer has to contain the counts and displacements arrays.

```python
# allgatherv.py
mycount = procid+1
my_array = np.empty(mycount,dtype=np.float64)
```

For the full source of this example, see section 3.16.21

```python
my_count = np.empty(1,dtype=int)
my_count[0] = mycount
comm.Allgather( my_count,recv_counts )

accumulate = 0
for p in range(nprocs):
  recv_displs[p] = accumulate; accumulate += recv_counts[p]

global_array = np.empty(accumulate,dtype=np.float64)

comm.Allgatherv( my_array, [global_array,recv_counts,recv_displs,MPI.DOUBLE] )
```

For the full source of this example, see section 3.16.21

3.9.3 Variable all-to-all

The variable all-to-all routine MPI_Alltoallv is discussed in section 3.6.2.

3.10 MPI Operators

MPI operators, that is, objects of type MPI_Op, are used in reduction operators. Most common operators, such as sum or maximum, have been built into the MPI library; see section 3.10.1. It is also possible to define new operators; see section 3.10.2.

3.10.1 Pre-defined operators

The following is the list of pre-defined operators MPI_Op values.
### 3.10. MPI Operators

<table>
<thead>
<tr>
<th>MPI type</th>
<th>meaning</th>
<th>applies to</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>maximum</td>
<td>integer, floating point</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>minimum</td>
<td></td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>sum</td>
<td>integer, floating point, complex, multilanguage types</td>
</tr>
<tr>
<td>MPI_REPLACE</td>
<td>overwrite</td>
<td></td>
</tr>
<tr>
<td>MPI_NO_OP</td>
<td>no change</td>
<td></td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>product</td>
<td></td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>logical and</td>
<td>C integer, logical</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>logical xor</td>
<td></td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>bitwise and</td>
<td>integer, byte, multilanguage types</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>bitwise or</td>
<td></td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>bitwise xor</td>
<td></td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max value and location</td>
<td>MPI_DOUBLE_INT and such</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>min value and location</td>
<td></td>
</tr>
</tbody>
</table>

#### 3.10.1.1 Minloc and maxloc

The MPI_MAXLOC and MPI_MINLOC operations yield both the maximum and the rank on which it occurs. Their result is a struct of the data over which the reduction happens, and an int.

In C, the types to use in the reduction call are: MPI_FLOAT_INT, MPI_LONG_INT, MPI_DOUBLE_INT, MPI_SHORT_INT, MPI_2INT, MPI_LONG_DOUBLE_INT. Likewise, the input needs to consist of such structures: the input should be an array of such struct types, where the int is the rank of the number.

These types may have some unusual size properties:

**Code:**

```c
// longint.c

MPI_Type_size( MPI_LONG_INT,&s );
printf("MPI_LONG_INT size=%d\n",s);
MPI_Aint ss;
MPI_Type_extent( MPI_LONG_INT,&ss );
printf("MPI_LONG_INT extent=%ld\n",ss);
```

**Output:**

```
MPI_LONG_INT size=12
MPI_LONG_INT extent=16
```

**Fortran note 5: Min/maxloc types.** The original Fortran interface to MPI was designed around Fortran77 features, so it is not using Fortran derived types (Type keyword). Instead, all integer indices are stored in whatever the type is that is being reduced. The available result types are then MPI_2REAL, MPI_2DOUBLE_PRECISION, MPI_2INTEGER.

Likewise, the input needs to be arrays of such type. Consider this example:

```fortran
Real*8, dimension(2,N) :: input, output
call MPI_Reduce( input, output, N, MPI_2DOUBLE_PRECISION, &
                 MPI_MAXLOC, root, comm )
```

**MPL note 19: Operators.** Arithmetic: plus, multiplies, max, min.
3. MPI topic: Collectives

Figure 3.15 MPI_Op_create

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Op_create</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Op_create_c</td>
<td>user_fn</td>
<td>user defined function</td>
<td>MPI_User_function</td>
<td></td>
</tr>
<tr>
<td>MPI_User_function_c</td>
<td>PROCEDURE(MPI_User_function)</td>
<td>IN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>commute</td>
<td>true if commutative; false otherwise.</td>
<td>int</td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>op</td>
<td>operation</td>
<td>MPI_Op*</td>
<td>TYPE(MPI_Op)</td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Op.create(cls, function, bool commute=False)
```

Logic: logical_and, logical_or, logical_xor.

Bitwise: bit_and, bit_or, bit_xor.

3.10.2 User-defined operators

In addition to predefined operators, MPI has the possibility of user-defined operators to use in a reduction or scan operation.

The routine for this is MPI_Op_create (figure 3.15), which takes a user function and turns it into an object of type MPI_Op, which can then be used in any reduction:

```c
MPI_Op rwz;
MPI_Op_create(reduce_without_zero, 1, &rwz);
MPI_Allreduce(data+procno, &positive_minimum, 1, MPI_INT, rwz, comm);
```

*For the full source of this example, see section 3.16.22*

Python note 11: Define reduction operator. In python, Op.Create is a class method for the MPI class.

```python
rwz = MPI.Op.Create(reduceWithoutZero)
positive_minimum = np.zeros(1, dtype=intc)
comm.Allreduce(data[procid], positive_minimum, rwz);
```

*For the full source of this example, see section 3.16.23*

The user function needs to have the following signature:

```c
typedef void MPI_User_function
           ( void *invec, void *inoutvec, int *len,
             MPI_Datatype *datatype);
```

```c
FUNCTION USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE)
            <type> INVEC(LEN), INOUTVEC(LEN)
            INTEGER LEN, TYPE
```

For example, here is an operator for finding the smallest nonzero number in an array of nonnegative integers:
3.10. MPI Operators

void reduce_without_zero(void *in, void *inout, int *len, MPI_Datatype *type) {
   // r is the already reduced value, n is the new value
   int n = *(int*)in, r = *(int*)inout;
   int m;
   if (n==0) { // new value is zero: keep r
      m = r;
   } else if (r==0) {
      m = n;
   } else if (n<r) { // new value is less but not zero: use n
      m = n;
   } else { // new value is more: use r
      m = r;
   }
   *(int*)inout = m;
}

For the full source of this example, see section 3.16.22

Python note 12: Reduction function. The python equivalent of such a function receives bare buffers as arguments. Therefore, it is best to turn them first into NumPy arrays using np.frombuffer:

```python
# redctpositive.py
def reduceWithoutZero(in_buf, inout_buf, datatype):
    typecode = MPI._typecode(datatype)
    assert typecode is not None # check MPI datatype is built-in
dtype = np.dtype(typecode)

    in_array = np.frombuffer(in_buf, dtype)
    inout_array = np.frombuffer(inout_buf, dtype)

    n = in_array[0]; r = inout_array[0]
    if n==0:
        m = r
    elif r==0:
        m = n
    elif n<r:
        m = n
    else:
        m = r
    inout_array[0] = m
```

For the full source of this example, see section 3.16.23

The assert statement accounts for the fact that this mapping of MPI datatype to NumPy dtype only works for built-in MPI datatypes.

MPL note 20: User-defined operators. A user-defined operator can be a templated class with an `operator()`.

Example:

```cpp
// reduceuser.cxx
template<typename T>
class lcm {
    public:
        T operator()(T a, T b) {
```
3. MPI topic: Collectives

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Op_commutative</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>op</td>
<td></td>
<td>operation</td>
<td>MPI_Op</td>
<td>TYPE(MPI_Op)</td>
<td></td>
</tr>
<tr>
<td>commute</td>
<td></td>
<td>true if op is commutative,</td>
<td>int*</td>
<td>LOGICAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>false otherwise</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T \text{zero} = \text{T}();
T t((a/gcd(a, b))*b);
if (t<zero)
  return -t;
return t;
}

MPL note 21: Lambda operator. You can also do the reduction by lambda:

```c
comm_world.reduce
  ( [] (int i, int j) -> int
    { return i+j; },
    0, data );
```

The function has an array length argument \text{len}, to allow for pointwise reduction on a whole array at once. The \text{inoutvec} array contains partially reduced results, and is typically overwritten by the function.

There are some restrictions on the user function:

- It may not call MPI functions, except for \text{MPI Abort}.
- It must be associative; it can be optionally commutative, which fact is passed to the \text{MPI Op create} call.

**Exercise 3.19.** Write the reduction function to implement the one-norm of a vector:

\[
\|x\|_1 = \sum_i |x_i|.
\]

(There is a skeleton for this exercise under the name onenorm.)

The operator can be destroyed with a corresponding \text{MPI Op free}.

\| int MPI_Op_free(MPI_Op *op) \|

This sets the operator to \text{MPI_OP NULL}. This is not necessary in OO languages, where the destructor takes care of it.

You can query the commutativity of an operator with \text{MPI Op commutative} (figure 3.16).

3.10.3 Local reduction

The application of an MPI Op can be performed with the routine \text{MPI Reduce local} (figure 3.17). Using this routine and some send/receive scheme you can build your own global reductions. Note that this routine does not take a communicator because it is purely local.
3.11. Nonblocking collectives

Above you have seen how the ‘Isend’ and ‘Irecv’ routines can overlap communication with computation. This is not possible with the collectives you have seen so far: they act like blocking sends or receives. However, there are also nonblocking collectives, introduced in MPI-3.

Such operations can be used to increase efficiency. For instance, computing

\[ y \leftarrow Ax + (x^t x)y \]

involves a matrix-vector product, which is dominated by computation in the sparse matrix case, and an inner product which is typically dominated by the communication cost. You would code this as

```c
MPI_Iallreduce( .... x ...., &request);
// compute the matrix vector product
MPI_Wait(request);
// do the addition
```

This can also be used for 3D FFT operations [13]. Occasionally, a nonblocking collective can be used for nonobvious purposes, such as the MPI_Ibarrier in [14].

These have roughly the same calling sequence as their blocking counterparts, except that they output an MPI_Request. You can then use an MPI_Wait call to make sure the collective has completed.

Nonblocking collectives offer a number of performance advantages:

- Do two reductions (on the same communicator) with different operators simultaneously:

\[ \alpha \leftarrow x^t y \]
\[ \beta \leftarrow \|z\|_\infty \]

which translates to:

```c
MPI_Allreduce( &local_xy, &global_xy, 1, MPI_DOUBLE, MPI_SUM, comm);
```

```c
MPI_Allreduce( &local_xinf, &global_xin, 1, MPI_DOUBLE, MPI_MAX, comm);
```
3. MPI topic: Collectives

- do collectives on overlapping communicators simultaneously;
- overlap a nonblocking collective with a blocking one.

**Exercise 3.20.** Revisit exercise 7.1. Let only the first row and first column have certain data, which they broadcast through columns and rows respectively. Each process is now involved in two simultaneous collectives. Implement this with nonblocking broadcasts, and time the difference between a blocking and a nonblocking solution. (There is a skeleton for this exercise under the name procgridnonblock.)

**Remark 6** Blocking and nonblocking don’t match: either all processes call the nonblocking or all call the blocking one. Thus the following code is incorrect:

```cpp
if (rank==root)
    MPI_Reduce( &x /* ... */ root,comm );
else
    MPI_Ireduce( &x /* ... */ );
```

This is unlike the point-to-point behavior of nonblocking calls: you can catch a message with `MPI_Irecv` that was sent with `MPI_Send`.

**Remark 7** Unlike sends and received, collectives have no identifying tag. With blocking collectives that does not lead to ambiguity problems. With nonblocking collectives it means that all processes need to issue them in identical order.

List of nonblocking collectives:

- `MPI_Igather`, `MPI_Igatherv`, `MPI_Iallgather` (figure 3.18), `MPI_Iallgatherv`,
- `MPI_Iscatter`, `MPI_Iscatterv`,
- `MPI_Ireduce`, `MPI_Iallreduce` (figure 3.19), `MPI_Ireduce_scatter`, `MPI_Ireduce_scatter_block`.
- `MPI_Ialltoall`, `MPI_Ialltoallv`, `MPI_Ialltoallw`,
- `MPI_Ibarrier`, section 3.11.2,
- `MPI_Ibcast`,
- `MPI_Iexscan`, `MPI_Iscan`.

**MPL note 22: Nonblocking collectives.** Nonblocking collectives have the same argument list as the corresponding blocking variant, except that instead of a void result, they return an `irequest`. (See 29)

```cpp
// ireducescalar.cpp
float x[1.], sum;
auto reduce_request = 
    comm_world.ireduce(mpl::plus<float>(), 0, x, sum);
reduce_request.wait();
if (comm_world.rank()==0) {
    std::cout << "sum = " << sum << "\n";
}
```
### 3.11. Nonblocking collectives

#### Figure 3.18 MPI_Iallgather

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Iallgather</td>
<td>sendbuf</td>
<td>starting address of send buffer</td>
<td>const void* TYPE(*), DIMENSION(..)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sendcount</td>
<td>number of elements in send buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Count</td>
<td>IN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sendtype</td>
<td>datatype of send buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>recvbuf</td>
<td>address of receive buffer</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>recvcount</td>
<td>number of elements received from any process</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Count</td>
<td>IN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>recvtype</td>
<td>datatype of receive buffer elements</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>request</td>
<td>communication request</td>
<td>MPI_Request*</td>
<td>TYPE(MPI_Request)</td>
<td>OUT</td>
</tr>
</tbody>
</table>

#### Figure 3.19 MPI_Iallreduce

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Iallreduce</td>
<td>sendbuf</td>
<td>starting address of send buffer</td>
<td>const void* TYPE(*), DIMENSION(..)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>recvbuf</td>
<td>starting address of receive buffer</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>number of elements in send buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Count</td>
<td>IN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>datatype</td>
<td>datatype of elements of send buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>op</td>
<td>operation</td>
<td>MPI_Op</td>
<td>TYPE(MPI_Op)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>request</td>
<td>communication request</td>
<td>MPI_Request*</td>
<td>TYPE(MPI_Request)</td>
<td>OUT</td>
</tr>
</tbody>
</table>

Victor Eijkhout 83
3. MPI topic: Collectives

3.11.1 Examples

3.11.1.1 Array transpose

To illustrate the overlapping of multiple nonblocking collectives, consider transposing a data matrix. Initially, each process has one row of the matrix; after transposition each process has a column. Since each row needs to be distributed to all processes, algorithmically this corresponds to a series of scatter calls, one originating from each process.

```c
// itransposeblock.c
for (int iproc=0; iproc<nprocs; iproc++) {
    MPI_Scatter( regular, 1, MPI_DOUBLE,
                 &transpose[iproc], 1, MPI_DOUBLE,
                 iproc, comm);
}
```

For the full source of this example, see section 3.16.17

Introducing the nonblocking MPI_Iscatter call, this becomes:

```c
MPI_Request scatter_requests[nprocs];
for (int iproc=0; iproc<nprocs; iproc++) {
    MPI_Iscatter( regular, 1, MPI_DOUBLE,
                  &transpose[iproc], 1, MPI_DOUBLE,
                  iproc, comm, scatter_requests+iproc);
}
MPI_Waitall(nprocs, scatter_requests, MPI_STATUSES_IGNORE);
```

For the full source of this example, see section 3.16.17

Exercise 3.21. Can you implement the same algorithm with MPI_Igather?

3.11.1.2 Stencils

![Five-point stencil gather](image)

Figure 3.10: Illustration of five-point stencil gather
3.11. Nonblocking collectives

The ever-popular five-point stencil evaluation does not look like a collective operation, and indeed, it is usually evaluated with (nonblocking) send/recv operations. However, if we create a subcommunicator on each subdomain that contains precisely that domain and its neighbors, (see figure 3.10) we can formulate the communication pattern as a gather on each of these. With ordinary collectives this can not be formulated in a deadlock-free manner, but nonblocking collectives make this feasible.

We will see an even more elegant formulation of this operation in section 11.2.

3.11.2 Nonblocking barrier

Probably the most surprising nonblocking collective is the nonblocking barrier MPI_Ibarrier (figure 3.20). The way to understand this is to think of a barrier not in terms of temporal synchronization, but state agreement: reaching a barrier is a sign that a process has attained a certain state, and leaving a barrier means that all processes are in the same state. The ordinary barrier is then a blocking wait for agreement, while with a nonblocking barrier:

• Posting the barrier means that a process has reached a certain state; and
• the request being fulfilled means that all processes have reached the barrier.

One scenario would be local refinement, where some processes decide to refine their subdomain, which fact they need to communicate to their neighbors. The problem here is that most processes are not among these neighbors, so they should not post a receive of any type. Instead, any refining process sends to its neighbors, and every process posts a barrier.

```c
// ibarrierprobe.c
if (i_do_send) {
   /*
   * Pick a random process to send to,
   * not yourself.
   */
   int receiver = rand()%nprocs;
   MPI_Ssend(&data,1,MPI_FLOAT,receiver,0,comm);
}
/*
* Everyone posts the non-blocking barrier
* and gets a request to test/wait for
*/
MPI_Request barrier_request;
MPI_Ibarrier(comm,&barrier_request);
```

For the full source of this example, see section 3.16.24
3. MPI topic: Collectives

Now every process alternately probes for messages and tests for completion of the barrier. Probing is done through the nonblocking `MPI_Iprobe` call, while testing completion of the barrier is done through `MPI_Test`.

```c
for ( ; ; step++) {
    int barrier_done_flag=0;
    MPI_Test(&barrier_request,&barrier_done_flag,
             MPI_STATUS_IGNORE);
    //stop if you're done!
    if (barrier_done_flag) {
        break;
    } else {
        // if you're not done with the barrier:
        int flag; MPI_Status status;
        MPI_Iprobe(
            MPI_ANY_SOURCE,MPI_ANY_TAG,
            comm, &flag, &status );
        if (flag) {
            // absorb message!
        }
    }
}
```

For the full source of this example, see section 3.16.24

We can use a nonblocking barrier to good effect, utilizing the idle time that would result from a blocking barrier. In the following code fragment processes test for completion of the barrier, and failing to detect such completion, perform some local work.

```c
// findbarrier.c
MPI_Request final_barrier;
MPI_Ibarrier(comm,&final_barrier);

int global_finish=mysleep;
do {
    int all_done_flag=0;
    MPI_Test(&final_barrier,&all_done_flag,MPI_STATUS_IGNORE);
    if (all_done_flag) {
        break;
    } else {
        int flag; MPI_Status status;
        // force progress
        MPI_Iprobe(
            MPI_ANY_SOURCE,MPI_ANY_TAG,
            comm, &flag, MPI_STATUS_IGNORE );
        printf("[%d] going to work for another second\n",procid);
        sleep(1);
        global_finish++;
    }
} while (1);
```

For the full source of this example, see section 3.16.25

3.12 Performance of collectives

It is easy to visualize a broadcast as in figure 3.11: see figure 3.11. The root sends all of its data directly to
3.13 Collectives and synchronization

The time that a message takes can simply be modeled as

$$\alpha + \beta n,$$

where $\alpha$ is the latency, a one time delay from establishing the communication between two processes, and $\beta$ is the time-per-byte, or the inverse of the bandwidth, and $n$ the number of bytes sent.

Under the assumption that a processor can only send one message at a time, the broadcast in figure 3.11 would take a time proportional to the number of processors.

**Exercise 3.22.** What is the total time required for a broadcast involving $p$ processes? Give $\alpha$ and $\beta$ terms separately.

One way to ameliorate that is to structure the broadcast in a tree-like fashion. This is depicted in figure 3.12.

**Exercise 3.23.** How does the communication time now depend on the number of processors, again $\alpha$ and $\beta$ terms separately.

What would be a lower bound on the $\alpha, \beta$ terms? The theory of the complexity of collectives is described in more detail in HPSC-6.1; see also [3].

3.13 Collectives and synchronization

Collectives, other than a barrier, have a synchronizing effect between processors. For instance, in...
3. MPI topic: Collectives

```c
MPI_Bcast(....data....root);
MPI_Send(.....);
```

the send operations on all processors will occur after the root executes the broadcast. Conversely, in a

Figure 3.13: Trace of a reduction operation between two dual-socket 12-core nodes

reduce operation the root may have to wait for other processes. This is illustrated in figure 3.13, which gives a TAU trace of a reduction operation on two nodes, with two six-core sockets (processors) each. We see that:

- In each socket, the reduction is a linear accumulation;
- on each node, cores zero and six then combine their result;
- after which the final accumulation is done through the network.

We also see that the two nodes are not perfectly in sync, which is normal for MPI applications. As a result, core 0 on the first node will sit idle until it receives the partial result from core 12, which is on the second node.

While collectives synchronize in a loose sense, it is not possible to make any statements about events before and after the collectives between processors:

```c
...event 1...
MPI_Bcast(....);
...event 2....
```

1. This uses mvapich version 1.6; in version 1.9 the implementation of an on-node reduction has changed to simulate shared memory.
3.13. Collectives and synchronization

The most logical execution is:

\[ P_0 \quad P_1 \quad P_2 \]

However, this ordering is allowed too:

\[ P_0 \quad P_1 \quad P_2 \]

Which looks from a distance like:

\[ P_0 \quad P_1 \quad P_2 \]

In other words, one of the messages seems to go 'back in time'.

Figure 3.14: Possible temporal orderings of send and collective calls
Consider a specific scenario:

```c
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, &status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, &status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

Note the `MPI_ANY_SOURCE` parameter in the receive calls on processor 1. One obvious execution of this would be:

1. The send from 2 is caught by processor 1;
2. Everyone executes the broadcast;
3. The send from 0 is caught by processor 1.

However, it is equally possible to have this execution:

1. Processor 0 starts its broadcast, then executes the send;
2. Processor 1’s receive catches the data from 0, then it executes its part of the broadcast;
3. Processor 1 catches the data sent by 2, and finally processor 2 does its part of the broadcast.

This is illustrated in figure 3.14.

### 3.14 Performance considerations

In this section we will consider how collectives can be implemented in multiple ways, and the performance implications of such decisions. You can test the algorithms described here using SimGrid (section 51).

#### 3.14.1 Scalability

We are motivated to write parallel software from two considerations. First of all, if we have a certain problem to solve which normally takes time $T$, then we hope that with $p$ processors it will take time $T/p$. If this is true, we call our parallelization scheme *scalable in time*. In practice, we often accept small extra terms: as you will see below, parallelization often adds a term $\log_2 p$ to the running time.

**Exercise 3.24.** Discuss scalability of the following algorithms:

- You have an array of floating point numbers. You need to compute the sine of each
- You a two-dimensional array, denoting the interval $[-2,2]^2$. You want to make a picture of the *Mandelbrot set*, so you need to compute the color of each point.
• The primality test of exercise 2.6.

There is also the notion that a parallel algorithm can be scalable in space: more processors gives you more memory so that you can run a larger problem.

**Exercise 3.25.** Discuss space scalability in the context of modern processor design.

### 3.14.2 Complexity and scalability of collectives

#### 3.14.2.1 Broadcast

**Naive broadcast** Write a broadcast operation where the root does an `MPI_Send` to each other process.

What is the expected performance of this in terms of $\alpha, \beta$?

Run some tests and confirm.

**Simple ring** Let the root only send to the next process, and that one send to its neighbor. This scheme is known as a bucket brigade; see also section 4.1.5.

What is the expected performance of this in terms of $\alpha, \beta$?

Run some tests and confirm.

![Figure 3.15: A pipelined bucket brigade](image)

**Pipelined ring** In a ring broadcast, each process needs to receive the whole message before it can pass it on. We can increase the efficiency by breaking up the message and sending it in multiple parts.
3. MPI topic: Collectives

(See figure 3.15.) This will be advantageous for messages that are long enough that the bandwidth cost dominates the latency.

Assume a send buffer of length more than 1. Divide the send buffer into a number of chunks. The root sends the chunks successively to the next process, and each process sends on whatever chunks it receives.

What is the expected performance of this in terms of $\alpha, \beta$? Why is this better than the simple ring?

Run some tests and confirm.

**Recursive doubling**  Collectives such as broadcast can be implemented through recursive doubling, where the root sends to another process, then the root and the other process send to two more, those four send to four more, et cetera. However, in an actual physical architecture this scheme can be realized in multiple ways that have drastically different performance.

First consider the implementation where process 0 is the root, and it starts by sending to process 1; then they send to 2 and 3; these four send to 4–7, et cetera. If the architecture is a linear array of processors, this will lead to contention: multiple messages wanting to go through the same wire. (This is also related to the concept of bisection bandwidth.)

In the following analyses we will assume wormhole routing: a message sets up a path through the network, reserving the necessary wires, and performing a send in time independent of the distance through the network. That is, the send time for any message can be modeled as

$$T(n) = \alpha + \beta n$$

regardless source and destination, as long as the necessary connections are available.

**Exercise 3.26.** Analyze the running time of a recursive doubling broadcast as just described, with wormhole routing.

Implement this broadcast in terms of blocking MPI send and receive calls. If you have SimGrid available, run tests with a number of parameters.

The alternative, that avoids contention, is to let each doubling stage divide the network into separate halves. That is, process 0 sends to $P/2$, after which these two repeat the algorithm in the two halves of the network, sending to $P/4$ and $3P/4$ respectively.

**Exercise 3.27.** Analyze this variant of recursive doubling. Code it and measure runtimes on SimGrid.

**Exercise 3.28.** Revisit exercise 3.26 and replace the blocking calls by nonblocking `MPI_Isend`/`MPI_Irecv` calls.

Make sure to test that the data is correctly propagated.

MPI implementations often have multiple algorithms, which they dynamically switch between. Sometimes you can determine the choice yourself through environment variables.

3.15 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

Review 3.29. How would you realize the following scenarios with MPI collectives?
- Let each process compute a random number. You want to print the maximum of these numbers to your screen.
- Each process computes a random number again. Now you want to scale these numbers by their maximum.
- Let each process compute a random number. You want to print on what processor the maximum value is computed.

Review 3.30. MPI collectives can be sorted in at least the following categories
1. rooted vs rootless
2. using uniform buffer lengths vs variable length buffers
3. blocking vs nonblocking.
Give examples of each type.

Review 3.31. True or false: collective routines are all about communicating user data between the processes.

Review 3.32. True or false: an MPI_Scatter call puts the same data on each process.

Review 3.33. True or false: using the option MPI_IN_PLACE you only need space for a send buffer in MPI_Reduce.

Review 3.34. True or false: using the option MPI_IN_PLACE you only need space for a send buffer in MPI_Gather.

Review 3.35. Given a distributed array, with every processor storing

```plaintext
double x[N]; // N can vary per processor
```

give the approximate MPI-based code that computes the maximum value in the array, and leaves the result on every processor.

Review 3.36.

```plaintext
double data[Nglobal];
int myfirst = /* something */ , mylast = /* something */;
for (int i=myfirst; i<mylast; i++) {
    if (i>0 && i<N-1) {
        process_point( data,i,Nglobal );
    }
}
void process_point( double *data, int i, int N ) {
    data[i-1] = g(i-1); data[i] = g(i); data[i+1] = g(i+1);
    data[i] = f(data[i-1],data[i],data[i+1]);
}
```

Is this scalable in time? Is this scalable in space? What is the missing MPI call?

Review 3.37.

```plaintext
double data[Nlocal+2]; // include left and right neighbor
int myfirst = /* something */ , mylast = myfirst+Nlocal;
```
for (int i=0; i<Nlocal; i++) {
    if (i>0 && i<N-1) {
        process_point( data,i,Nlocal );
    }
}

void process_point( double *data, int i0, int n ) {
    int i = i0+1;
    data[i-1] = g(i-1); data[i] = g(i); data[i+1] = g(i+1);
    data[i] = f(data[i-1],data[i],data[i+1]);
}

Is this scalable in time? Is this scalable in space? What is the missing MPI call?

Review 3.38. With data as in the previous question, given the code for normalizing the array, that is, scaling each element so that $\|x\|_2 = 1$.

Review 3.39. Just like MPI_Allreduce is equivalent to MPI_Reduce followed by MPI_Bcast, MPI_Reduce_scatter is equivalent to at least one of the following combinations. Select those that are equivalent, and discuss differences in time or space complexity:

1. MPI_Reduce followed by MPI_Scatter;
2. MPI_Gather followed by MPI_Scatter;
3. MPI_Allreduce followed by MPI_Scatter;
4. MPI_Allreduce followed by a local operation (which?);
5. MPI_Allgather followed by a local operation (which?).

Review 3.40. Think of at least two algorithms for doing a broadcast. Compare them with regards to asymptotic behavior.
3.16. Sources used in this chapter

3.16.1 Listing of code header

3.16.2 Listing of code examples/mpi/c/allreduce.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
    #include "globalinit.c"
    float myrandom,sumrandom;
    myrandom = (float) rand()/(float)RAND_MAX;
    // add the random variables together
    MPI_Allreduce(&myrandom,&sumrandom,
                  1,MPI_FLOAT,MPI_SUM,comm);
    // the result should be approx nprocs/2:
    if (procno==nprocs-1)
        printf("Result %6.9f compared to .5\n",sumrandom/nprocs);
    MPI_Finalize();
    return 0;
}
```

3.16.3 Listing of code examples/mpi/mpl/collectscalar.cxx

```c
#include <cstdlib>
#include <complex>
#include <iostream>
#include <vector>
#include <mpl/mpl.hpp>

int main() {
    // MPI Comm world
    const mpl::communicator &comm_world=mpl::environment::comm_world();

    // vector of consecutive floats
    std::vector<float> v;
    if (comm_world.rank()==0)
        for (int i=0; i<comm_world.size(); ++i)
            v.push_back(i);

    // if you scatter, everyone gets a number equal to their rank.
    // rank 0 scatters data to all processes
    float x;
    comm_world.scatter(0, v.data(), x);
}
```
std::cout << "rank " << comm_world.rank() << " got " << x << '\n';

// wait until all processes have reached this point
comm_world.barrier();

// multiply that number, giving twice your rank
x*=2;

// rank 0 gathers data from all processes
comm_world.gather(0, x, v.data());
if (comm_world.rank()==0) {
    std::cout << "got";
    for (int i=0; i<comm_world.size(); ++i)
        std::cout << " " << i << ":" << v[i];
    std::cout << std::endl;
}

// wait until all processes have reached this point
comm_world.barrier();

// calculate global sum and pass result to rank 0
if (comm_world.rank()==0) {
    float sum;
    comm_world.reduce(mpl::plus<float>(), 0, x, sum);
    std::cout << "sum = " << sum << '\n';
} else
    comm_world.reduce(mpl::plus<float>(), 0, x);

// wait until all processes have reached this point
comm_world.barrier();

// calculate global sum and pass result to all
{
    float
        xrank = static_cast<float>( comm_world.rank() ),
        xreduce;
    // separate recv buffer
    comm_world.allreduce(mpl::plus<float>(), xrank,xreduce);
    // in place
    comm_world.allreduce(mpl::plus<float>(), xrank);
    if ( comm_world.rank()==comm_world.size()-1 )
        std::cout << "Allreduce got: separate=" << xreduce
                  << "", inplace=" << xrank << std::endl;
}

// calculate global sum and pass result to root
{
    int root = 1;
    float
        xrank = static_cast<float>( comm_world.rank() ),
        xreduce;
    // separate receive buffer
    comm_world.reduce(mpl::plus<float>(), root, xrank,xreduce);
3.16. Sources used in this chapter

```c++
// in place
comm_world.reduce(mpl::plus<float>(), root, xrank);
if ( comm_world.rank()==root )
    std::cout << "Allreduce got: separate=" << xreduce
    << ", inplace=" << xrank << std::endl;

return EXIT_SUCCESS;
}

3.16.4 Listing of code examples/mpi/p/allreduce.py

import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)
random.seed(procid)
random_bound = nprocs*nprocs
random_number = random.randint(1,random_bound)
# print("[%d] random=%d" % (procid,random_number))

# native mode send
max_random = comm.allreduce(random_number,op=MPI.MAX)
if procid==0:
    print("Python native:
 max=%d" % max_random)

myrandom = np.empty(1,dtype=int)
myrandom[0] = random_number
allrandom = np.empty(nprocs,dtype=int)
# numpy mode send
comm.Allreduce(myrandom,allrandom[:1],op=MPI.MAX)
if procid==0:
    print("Python numpy:
 max=%d" % allrandom[0])

sumrandom = np.zeros(1,dtype=int)
sumrandom[0] = myrandom[0]
#### WRONG polymorphic use does not work
#comm.Allreduce(sumrandom[:1])
comm.Allreduce(MPI.IN_PLACE,sumrandom[:1],op=MPI.SUM)
if procid==0:
    print("Sum of randoms: %d, compare %d" % (sumrandom[0],nprocs*random_bound/2) )
```

Victor Eijkhout
3. MPI topic: Collectives

3.16.5 Listing of code examples/mpi/mpl/sendrange.cxx

```cpp
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

#include <mpl/mpl.hpp>

int main() {
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    if (comm_world.size()<2)
        return EXIT_FAILURE;

    vector<double> v(15);
    if (comm_world.rank()==0) {
        // initialize
        for ( auto &x : v ) x = 1.41;
        
        /*
        * Send and report
        */
        comm_world.send(v.begin(), v.end(), 1); // send to rank 1
    } else if (comm_world.rank()==1) {
        /*
        * Receive data and report
        */
        comm_world.recv(v.begin(), v.end(), 0); // receive from rank 0

        cout << "Got:"
        for ( auto x : v )
            cout << " " << x;
        cout << endl;
    }
    return EXIT_SUCCESS;
}
```

3.16.6 Listing of code examples/mpi/c/reduce.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Initialize
    int array[5] = {1, 2, 3, 4, 5};
    float array2[5] = {1.0, 2.0, 3.0, 4.0, 5.0};

    // Reduce operations
    MPI_Reduce_scatter(&array, 5, MPI_INT, &array2[0], 5, MPI_FLOAT, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // Check result
    if (world_rank == 0) {
        printf("Result: ");
        for (int i = 0; i < 5; i++)
            printf("%.2f ", array2[i]);
        printf("\n");
    }
    MPI_Finalize();
    return 0;
}
```
3.16. Sources used in this chapter

```c
int main(int argc, char **argv) {
#include "globalinit.c"

    float myrandom = (float) rand()/(float)RAND_MAX,
    result;
    int target_proc = nprocs-1;
    // add all the random variables together
    MPI_Reduce(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,
                target_proc,comm);
    // the result should be approx nprocs/2:
    if (procno==target_proc)
        printf("Result %6.3f compared to nprocs/2=%5.2f\n",
               result,nprocs/2.);

    MPI_Finalize();
    return 0;
}
```

3.16.7 Listing of code examples/mpi/c/allreduceinplace.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char **argv) {
    #include "globalinit.c"

    int nrandoms = 500000;
    float *myrandoms;
    myrandoms = (float*) malloc(nrandoms*sizeof(float));
    for (int iter=1; iter<=3; iter++) {
        /*
        * We show three different ways of doing the same reduction;
        * this illustrates syntax more than semantics
        */
        if (iter==1) {
            for (int irand=0; irand<nrandoms; irand++)
                myrandoms[irand] = (float) rand()/(float)RAND_MAX;
            // add all the random variables together
            MPI_Allreduce(MPI_IN_PLACE,myrandoms,
                          nrandoms,MPI_FLOAT,MPI_SUM,comm);
        } else if (iter==2) {
            for (int irand=0; irand<nrandoms; irand++)
                myrandoms[irand] = (float) rand()/(float)RAND_MAX;
            int root=nprocs-1;
            if (procno==root)
                MPI_Reduce(MPI_IN_PLACE,myrandoms,
                           nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
            else
                MPI_Reduce(myrandoms,MPI_IN_PLACE,
```
3. MPI topic: Collectives

```fortran
3.16.8 Listing of code examples/mpi/f/reduceinplace.F90

Program ReduceInPlace

use mpi
implicit none
real :: mynumber, result
integer :: target_proc
#include "globalinit.F90"
call random_number(mynumber)
target_proc = ntids-1;
! add all the random variables together
if (mytid.eq.target_proc) then
   result = mytid
   call MPI_Reduce(MPI_IN_PLACE,result,1,MPI_REAL,MPI_SUM,&
       target_proc,comm,err)
else
   mynumber = mytid
   call MPI_Reduce(mynumber,result,1,MPI_REAL,MPI_SUM,&
       target_proc,comm,err)
end if
! the result should be ntids*(ntids-1)/2:
```

---

```
3.16.8 Listing of code examples/mpi/f/reduceinplace.F90

Program ReduceInPlace

use mpi
implicit none
real :: mynumber, result
integer :: target_proc
#include "globalinit.F90"
call random_number(mynumber)
target_proc = ntids-1;
! add all the random variables together
if (mytid.eq.target_proc) then
   result = mytid
   call MPI_Reduce(MPI_IN_PLACE,result,1,MPI_REAL,MPI_SUM,&
       target_proc,comm,err)
else
   mynumber = mytid
   call MPI_Reduce(mynumber,result,1,MPI_REAL,MPI_SUM,&
       target_proc,comm,err)
end if
! the result should be ntids*(ntids-1)/2:
```
3.16.9 Listing of code examples/mpi/f08/reduceinplaceptr.F90

Program ReduceInPlace

```
use mpi_f08

real,target :: mynumber,result,in_place_val
real,pointer :: mynumber_ptr,result_ptr
integer :: target_proc

#include "globalinit.F90"

call random_number(mynumber)
target_proc = ntids-1;
in_place_val = MPI_IN_PLACE
if (mytid.eq.target_proc) then
    ! set pointers
    result_ptr => result
    mynumber_ptr => in_place_val
    ! target sets value in receive buffer
    result_ptr = mytid
else
    ! set pointers
    mynumber_ptr => mynumber
    result_ptr => in_place_val
    ! non-targets set value in send buffer
    mynumber_ptr = mytid
end if

call MPI_Reduce(mynumber_ptr,result_ptr,1,MPI_REAL,MPI_SUM,&
                target_proc,comm,err)
! the result should be ntids*(ntids-1)/2:
if (mytid.eq.target_proc) then
    write(*,'("Result ",f5.2," compared to n(n-1)/2="f5.2") &
            result,ntids*(ntids-1)/2.
else
    write(*,'("Result ",f5.2," compared to n(n-1)/2="f5.2") &
            result,ntids*(ntids-1)/2.
end if

call MPI_Finalize(err)
end Program ReduceInPlace
```

3.16.10 Listing of code examples/mpi/p/allreduceinplace.py

```
import numpy as np
```

Victor Eijkhout
3. MPI topic: Collectives

```python
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

random_number = random.randint(1,nprocs*nprocs)
print("[%d] random=%d" % (procid,random_number))

myrandom = np.empty(1,dtype=int)
myrandom[0] = random_number
comm.Allreduce(MPI.IN_PLACE,myrandom,op=MPI.MAX)

if procid==0:
    print("Python numpy:
    max=%d" % myrandom[0])
```

3.16.11 Listing of code examples/mpi/ml/collectbuffer.cxx

```c
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <vector>
using std::vector;
#include <mpl/mpl.hpp>

int main() {

    // MPI Comm world
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    int nprocs = comm_world.size(), procno = comm_world.rank();
    int iprint = procno==nprocs-1;

    /* Reduce a 2 int buffer */
    float
    xrank = static_cast<float>( comm_world.rank() );
    vector<float> rank2p2p1{ 2*xrank,2*xrank+1 },reduce2p2p1{0,0};
    mpl::contiguous_layout<float> two_floats(rank2p2p1.size());
    comm_world.allreduce
    (mpl::plus<float>(), rank2p2p1.data(),reduce2p2p1.data(),two_floats);
```
if ( iprint )
    cout << "Got: " << reduce2p2p1.at(0) << "," << reduce2p2p1.at(1) << endl;

/*
  * Scatter one number to each proc
  */
if (iprint) cout << "Scattering 0--p" << endl;

vector<float> v;

if (comm_world.rank()==0)
    for (int i=0; i<comm_world.size(); ++i)
        v.push_back(i);
// if you scatter, everyone gets a number equal to their rank.
// rank 0 scatters data to all processes
float x;
comm_world.scatter(0, v.data(), x);

if (iprint)
    cout << "rank " << procno << " got " << x << '\n';

/*
  * Scatter two numbers to each proc
  */
if (iprint) cout << "Scatter 0--2p" << endl;

vector<float> vrecv(2),vsend(2*nprocs);

if (comm_world.rank()==0)
    for (int i=0; i<2*nprocs; ++i)
        vsend.at(i) = i;
// rank 0 scatters data to all processes
// if you scatter, everyone gets 2p,2p+1
mpl::contiguous_layout<float> twonums(2);
comm_world.scatter
(0, vsend.data(), twonums, vrecv.data(), twonums );

if (iprint)
    cout << "rank " << procno << " got "
        << vrecv[0] << "," << vrecv[1] << '\n';

return 0;
// multiply that number, giving twice your rank
x*=2;

// rank 0 gathers data from all processes
comm_world.gather(0, x, v.data());
if (comm_world.rank()==0) {
    cout << "got";
3. MPI topic: Collectives

for (int i=0; i<comm_world.size(); ++i)
    cout << " " << i << ":" << v[i];
    cout << endl;
}

// wait until all processes have reached this point
comm_world.barrier();

// calculate global sum and pass result to rank 0
if (comm_world.rank()==0) {
    float sum;
    comm_world.reduce(mpl::plus<float>(), 0, x, sum);
    cout << "sum = " << sum << '
';
} else
    comm_world.reduce(mpl::plus<float>(), 0, x);

// wait until all processes have reached this point
comm_world.barrier();

return EXIT_SUCCESS;

3.16.12 Listing of code examples/mpi/c/init.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
    int input_argument;
    #include "globalinit.c"
    if (procno==0) {
        if ( argc==1 || // the program is called without parameter
            ( argc>1 && !strcmp(argv[1],"-h") ) // user asked for help
        ) {
            printf("\nUsage: init [0-9]+\n");
            MPI_Abort(comm,1);
        }
        input_argument = atoi(argv[1]);
    }
    MPI_Bcast(&input_argument,1,MPI_INT,0,comm);
    printf("Processor %d gets %d\n",procno,input_argument);
    MPI_Finalize();
    return 0;
}
3.16.13  Listing of code examples/mpi/p/bcast.py

from mpi4py import MPI
import numpy as np

from functools import reduce
import sys

comm = MPI.COMM_WORLD

procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

root = 1
dsize = 10

# first native
if procid==root:
    buffer = [ 5.0 ] * dsize
else:
    buffer = [ 0.0 ] * dsize
buffer = comm.bcast(obj=buffer,root=root)
if not reduce( lambda x,y:x and y,
               [ buffer[i]==5.0 for i in range(len(buffer)) ] ):
    print("Something wrong on proc %d: native buffer <<%s>>" \
          % (procid,str(buffer)) )

# then with Numpy
buffer = np.arange(dsize, dtype=np.float64)
if procid==root:
    for i in range(dsize):
        buffer[i] = 5.0
comm.Bcast( buffer,root=root )
if not all( buffer==5.0 ):
    print("Something wrong on proc %d: numpy buffer <<%s>>" \
          % (procid,str(buffer)) )
else:
    if procid==root:
        print("Success.")

3.16.14  Listing of code examples/mpi/c/scan.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char **argv) {

    #include "globalinit.c"

Victor Eijkhout
3. MPI topic: Collectives

```c
float myrandom = (float) rand()/(float)RAND_MAX,
    result;
// add all the random variables together
MPI_Scan(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,comm);
// the result should be approaching nprocs/2:
if (procno==nprocs-1)
    printf("Result %6.3f compared to nprocs/2=%5.2f\n",
            result,nprocs/2.);

MPI_Finalize();
return 0;
}
```

3.16.15 Listing of code examples/mpi/p/scan.py

```python
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)
mycontrib = 10+random.randint(1,nprocs)
myfirst = 0
mypartial = comm.scan(mycontrib)
print("[%d] local: %d, partial: %d" % (procid,mycontrib,mypartial))
sbuf = np.empty(1,dtype=int)
rbuf = np.empty(1,dtype=int)
sbuf[0] = mycontrib
comm.Scan(sbuf,rbuf)
print("[%d] numpy local: %d, partial: %d" % (procid,mycontrib,rbuf[0]))
```

3.16.16 Listing of code examples/mpi/c/gather.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

    int localsize = 10+10*( (float) rand()/(float)RAND_MAX -.5),
    root = nprocs-1;
```
3.16. Sources used in this chapter

```c
int *localsizes=NULL;
// create local data
int *localdata = (int*) malloc( localsize*sizeof(int) );
for (int i=0; i<localsize; i++)
    localdata[i] = procno+1;
// we assume that each process has a value "localsize"
// the root process collects these values

if (procno==root)
    localsizes = (int*) malloc( nprocs*sizeof(int) );

// everyone contributes their info
MPI_Gather(&localsize,1,MPI_INT,
    localsizes,1,MPI_INT,root,comm);
if (procno==root) {
    printf("Local sizes: ");
    for (int i=0; i<nprocs; i++)
        printf("%d, ",localsizes[i]);
    printf("\n");
}

MPI_Finalize();
return 0;
```

3.16.17 Listing of code examples/mpi/c/itransposeblock.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {

    #include "globalinit.c"

    /*
     * Allocate matrix and transpose:
     * - one column per rank for regular
     * - one row per rank for transpose
     */
    double *regular,*transpose;
    regular = (double*) malloc( nprocs*sizeof(double) );
    transpose = (double*) malloc( nprocs*sizeof(double) );
    // each process has columns m*nprocs -- m*(nprocs+1)
    for (int ip=0; ip<nprocs; ip++)
        regular[ip] = procno*nprocs + ip;

    /*
     * Each proc does a scatter
     */
    #if 0
Victor Eijkhout
107
```
3. MPI topic: Collectives

// reference code:
for (int iproc=0; iproc<nprocs; iproc++) {
    MPI_Scatter(regular,1,MPI_DOUBLE,
        &(transpose[iproc]),1,MPI_DOUBLE,
        iproc,comm);
}
#else
    MPI_Request scatter_requests[nprocs];
    for (int iproc=0; iproc<nprocs; iproc++) {
        MPI_Iscatter(regular,1,MPI_DOUBLE,
            &(transpose[iproc]),1,MPI_DOUBLE,
            iproc,comm,scatter_requests+iproc);
    }
    MPI_Waitall(nprocs,scatter_requests,MPI_STATUSES_IGNORE);
#endif

/*
 * Check the result
 */
printf("[%d] ",procno);
for (int ip=0; ip<nprocs; ip++)
    printf(" %5.2f",transpose[ip]);
printf("\n");
MPI_Finalize();
return 0;
}

3.16.18 Listing of code examples/mpi/c/reducescatter.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
    #include "globalinit.c"

    /*
     * Set up an array of which processes you will receive from
     */
    int
    // data that we know:
    *i_recv_from_proc = (int*) malloc(nprocs*sizeof(int)),
    *procs_to_recv_from, nprocs_to_recv_from=0,
    // data we are going to determin:
    *procs_to_send_to,nprocs_to_send_to;
    //
    /*
     * Initialize
     */
for (int i=0; i<nprocs; i++) {
    i_recv_from_proc[i] = 0;
}

/ *
* Generate array of "yes/no I recv from proc p",
* and condensed array of procs I receive from.
*/
nprocs_to_recv_from = 0;
for (int iproc=0; iproc<nprocs; iproc++)
    // pick random procs to receive from, not yourself.
    if ( (float) rand()/(float)RAND_MAX < 2./nprocs && iproc!=procno ) {
        i_recv_from_proc[iproc] = 1;
        nprocs_to_recv_from++;
    }
procs_to_recv_from = (int*) malloc(nprocs_to_recv_from*sizeof(int));
int count_procs_to_recv_from = 0;
for (int iproc=0; iproc<nprocs; iproc++)
    if ( i_recv_from_proc[iproc] )
        procs_to_recv_from[count_procs_to_recv_from++] = iproc;
ASSERT( count_procs_to_recv_from==nprocs_to_recv_from );

/ *
*/
printf("[%d] receiving from:",procno);
for (int iproc=0; iproc<nprocs_to_recv_from; iproc++)
    printf(" %3d",procs_to_recv_from[iproc]);
printf(".\n");

/ *
* Now find how many procs will send to you
*/
MPI_Reduce_scatter_block
    (i_recv_from_proc,&nprocs_to_send_to,1,MPI_INT,
     MPI_SUM,comm);

/ *
* Send a zero-size msg to everyone that you receive from,
* just to let them know that they need to send to you.
*/
MPI_Request send_requests[nprocs_to_recv_from];
for (int iproc=0; iproc<nprocs_to_recv_from; iproc++) {
    int proc=procs_to_recv_from[iproc];
    double send_buffer=0.;
    MPI_Isend(&send_buffer,0,MPI_DOUBLE, /*to:*/ proc,0,comm,
                &send_requests[iproc]);
}

/ *
* Do as many receives as you know are coming in;
* use wildcards since you don't know where they are coming from.
* The source is a process you need to send to.
*/
3. MPI topic: Collectives

procs_to_send_to = (int*)malloc( nprocs_to_send_to * sizeof(int) );
for (int iproc=0; iproc<nprocs_to_send_to; iproc++) {
    double recv_buffer;
    MPI_Status status;
    MPI_Recv(&recv_buffer,0,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,comm,
    &status);
    procs_to_send_to[iproc] = status.MPI_SOURCE;
}
MPI_Waitall(nprocs_to_recv_from,send_requests,MPI_STATUSES_IGNORE);

printf("[%d] sending to:",procno);
for (int iproc=0; iproc<nprocs_to_send_to; iproc++)
    printf(" %3d",procs_to_send_to[iproc]);
printf(".
");
MPI_Finalize();
return 0;

3.16.19 Listing of code examples/mpi/p/gatherv.py

import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
sys.exit(1)

localsize = random.randint(2,10)
print("[%d] local size=%d" % (procid,localsize))
localdata = np.empty(localsize,dtype=int)
for i in range(localsize):
    localdata[i] = procid

# implicitly using root=0
globalsize = comm.reduce(localsize)
if procid==0:
    print("Global size=%d" % globalsize)
collecteddata = np.empty(globalsize,dtype=int)
counts = comm.gather(localsize)
comm.Gatherv(localdata, [collecteddata, counts])
if procid==0:
    print("Collected",str(collecteddata))

3.16.20 Listing of code examples/mpi/c/allgatherv.c

#include <stdlib.h>
#include <stdio.h>

110 Parallel Computing – r428
3.16. Sources used in this chapter

#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

int my_count = procno+1;
int *my_array = (int*) malloc(my_count*sizeof(int));
for (int i=0; i<my_count; i++)
    my_array[i] = procno;
int *recv_counts = (int*) malloc(nprocs*sizeof(int));
int *recv_displs = (int*) malloc(nprocs*sizeof(int));

MPI_Allgather
    ( &my_count,1,MPI_INT,
        recv_counts,1,MPI_INT, comm );
int accumulate = 0;
for (int i=0; i<nprocs; i++) {
    recv_displs[i] = accumulate; accumulate += recv_counts[i];
} int *global_array = (int*) malloc(accumulate*sizeof(int));
MPI_Allgatherv
    ( my_array,procno+1,MPI_INT,
        global_array,recv_counts,recv_displs,MPI_INT, comm );

if (procno==0) {
    for (int p=0; p<nprocs; p++)
        if (recv_counts[p]!=p+1)
            printf("count[%d] should be %d, not %d\n",
                p,p+1,recv_counts[p]);
    int c = 0;
    for (int p=0; p<nprocs; p++)
        for (int q=0; q<p; q++)
            if (global_array[c++]!=p)
                printf("p=%d, q=%d should be %d, not %d\n",
                    p,q,p,global_array[c-1]);
}

MPI_Finalize();
return 0;
}

3.16.21 Listing of code examples/mpi/p/allgatherv.py

import numpy as np
import random # random.randint(1,N), random.random()
import sys
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:

Victor Eijkhout 111
3. MPI topic: Collectives

    print(“C’mon, get real....”)  
    sys.exit(1)

mycount = procid+1
my_array = np.empty(mycount,dtype=np.float64)
for i in range(mycount):
    my_array[i] = procid
recv_counts = np.empty(nprocs,dtype=int)
recv_displs = np.empty(nprocs,dtype=int)

my_count = np.empty(1,dtype=int)
my_count[0] = mycount
comm.Allgather( my_count,recv_counts )

accumulate = 0
for p in range(nprocs):
    recv_displs[p] = accumulate; accumulate += recv_counts[p]
global_array = np.empty(accumulate,dtype=np.float64)
comm.Allgatherv( my_array, [global_array,recv_counts,recv_displs,MPI.DOUBLE] )

# other syntax:
# comm.Allgatherv( [my_array,mycount,0,MPI.DOUBLE], [global_array,recv_counts,recv_displs,MPI.DOUBLE] )

if procid==0:
    #print(pro cid,global_array)
    for p in range(nprocs):
        if recv_counts[p]!=p+1:
            print("recv count[\%d] should be \%d, not \%d" \\
              \% (p,p+1,recv_counts[p])
        c = 0
    for p in range(nprocs):
        for q in range(p+1):
            if global_array[c]!=p:
                print("p=\%d, q=\%d should be \%d, not \%d" \\
                   \% (p,q,p,global_array[c])
        c += 1
    print "finished"

3.16.22 Listing of code examples/mpi/c/reductpositive.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

//int reduce_without_zero(int r,int n);
void reduce_without_zero(void *in, void *inout, int *len, MPI_Datatype *type) { 
    // r is the already reduced value, n is the new value
    int n = *(int*)in, r = *(int*)inout;
    int m;
    if (n==0) { // new value is zero: keep r
3.16. Sources used in this chapter

```c
m = r;
} else if (r==0) {
    m = n;
} else if (n<r) { // new value is less but not zero: use n
    m = n;
} else { // new value is more: use r
    m = r;
};
#ifdef DEBUG
    printf("combine %d %d : %d\n",r,n,m);
#endif
// return the new value
*(int*)inout = m;
}

int main(int argc, char **argv) {
    #include "globalinit.c"

    int m,mreduct=2000000000,ndata = 10, data[10] = {2,3,0,5,0,1,8,12,4,0},
        positive_minimum;
    if (nprocs>ndata) {
        printf("Too many procs for this example: at most %d\n",ndata);
        return 1;
    }
    for (int i=0; i<nprocs; i++)
        if (data[i]<mreduct & data[i]>0)
            mreduct = data[i];

    MPI_Op rwz;
    MPI_Op_create(reduce_without_zero,1,&rwz);
    MPI_Allreduce(data+procno,&positive_minimum,1,MPI_INT,rwz,comm);
    // check that the distributed result is the same as sequential
    if (mreduct!=positive_minimum)
        printf("[%d] Result %d should be %d\n",procno,positive_minimum,mreduct);
    else if (procno==0)
        printf("User-defined reduction successful: %d\n",positive_minimum);

    MPI_Finalize();
    return 0;
}
```

3.16.23 Listing of code examples/mpi/p/reductpositive.py

```python
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
```
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

def reduceWithoutZero(in_buf, inout_buf, datatype):
    typecode = MPI._typecode(datatype)
    assert typecode is not None ## check MPI datatype is built-in
dtype = np.dtype(typecode)

    in_array = np.frombuffer(in_buf, dtype)
inout_array = np.frombuffer(inout_buf, dtype)

    n = in_array[0]; r = inout_array[0]
    if n==0:
        m = r
    elif r==0:
        m = n
    elif n<r:
        m = n
    else:
        m = r
    inout_array[0] = m

ndata = 10
data = np.zeros(10,dtype=intc)
data[:] = [2,3,0,5,0,1,8,12,4,0]

    if nprocs>ndata:
        print("Too many procs for this example: at most %d\n" %ndata)
        sys.exit(1)

    # compute reduction by hand
    #
    mreduct=2000000000
    for i in range(nprocs):
        if data[i]<mreduct and data[i]>0:
            mreduct = data[i]

rwz = MPI.Op.Create(reduceWithoutZero)
positive_minimum = np.zeros(1,dtype=intc)
comm.Allreduce(data[procid],positive_minimum,rwz);

    # check that the distributed result is the same as sequential
    #
    if mreduct!=positive_minimum:
        print("[X%d] Result %d should be %d\n" % \
            procid,positive_minimum,mreduct)
    elif procid==0:
        print("User-defined reduction successful: %d\n" % positive_minimum)
3.16.24 Listing of code examples/mpi/c/ibarrierprobe.c

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {

#include "globalinit.c"

/*
 * Pick one random process
 * that will do a send
 */
int sender,receiver;
if (procno==0)
 sender = rand()%nprocs;
MPI_Bcast(&sender,1,MPI_INT,0,comm);
int i_do_send = sender==procno;

float data=1.;
MPI_Request send_request;
if (i_do_send) {
 /*
 * Pick a random process to send to,
 * not yourself.
 */
 int receiver = rand()%nprocs;
 while (receiver==procno) receiver = rand()%nprocs;
 printf("[%d] random send performed to %d\n",procno,receiver);
 /*MPI_Isend(&data,1,MPI_FLOAT,receiver,0,comm,&send_request);*/
 MPI_Ssend(&data,1,MPI_FLOAT,receiver,0,comm);
}
/*
 * Everyone posts the non-blocking barrier
 * and gets a request to test/wait for
 */
MPI_Request barrier_request;
MPI_Ibarrier(comm,&barrier_request);

int step=0;
/*
 * Now everyone repeatedly tests the barrier
 * and probes for incoming message.
 * If the barrier completes, there are no
 * incoming message.
 */
MPI_BARRIER(comm);
double tstart = MPI_Wtime();

for ( ; ; step++) {
 int barrier_done_flag=0;

3. MPI topic: Collectives

```c
MPI_Test(&barrier_request,&barrier_done_flag,
    MPI_STATUS_IGNORE);
//stop if you're done!
if (barrier_done_flag) {
    break;
} else {
    // if you're not done with the barrier:
    int flag; MPI_Status status;
    MPI_Iprobe
        ( MPI_ANY_SOURCE,MPI_ANY_TAG,
        comm, &flag, &status );
    if (flag) {
        // absorb message!
        int sender = status.MPI_SOURCE;
        MPI_Recv(&data,1,MPI_FLOAT,sender,0,comm,MPI_STATUS_IGNORE);
        printf("[%d] random receive from %d\n",procno,sender);
    }
}

MPI_Barrier(comm);
double duration = MPI_Wtime()-tstart;
if (procno==0) printf("Probe loop: %e\n",duration);
printf("[%d] concluded after %d steps\n",procno,step);
MPI_Wait(&barrier_request,MPI_STATUS_IGNORE);
/* if (i_do_send) */
/* MPI_Wait(&send_request,MPI_STATUS_IGNORE); */
MPI_Finalize();
return 0;
```

3.16.25 Listing of code examples/mpi/c/findbarrier.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {
    MPI_Comm comm;
    int nprocs,procid;
    
    MPI_Init(&argc,&argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procid);
    
    int mysleep;
    srand(procid*time(NULL));
    mysleep = nprocs * (rand()/(double)RAND_MAX);
    
```
printf("[%d] working for %d seconds\n", procid, sleep(mysleep);

printf("[%d] finished, now posting barrier\n", procid);
MPI_Request final_barrier;
MPI_Ibarrier(comm, &final_barrier);

int global_finish = mysleep;
do {
    int all_done_flag = 0;
    MPI_Test(&final_barrier, &all_done_flag, MPI_STATUS_IGNORE);
    if (all_done_flag) {
        break;
    } else {
        int flag; MPI_Status status;
        // force progress
        MPI_Iprobe
            ( MPI_ANY_SOURCE, MPI_ANY_TAG,
             comm, &flag, MPI_STATUS_IGNORE );
        printf("[%d] going to work for another second\n", procid);
        sleep(1);
        global_finish++;
    }
} while (1);

MPI_Wait(&final_barrier, MPI_STATUS_IGNORE);
printf("[%d] concluded %d work, total time %d\n",
       procid, mysleep, global_finish);
MPI_Finalize();
return 0;
}
Chapter 4

MPI topic: Point-to-point

4.1 Blocking point-to-point operations

Suppose you have an array of numbers $x_i : i = 0, \ldots, N$ and you want to compute

$$y_i = (x_{i-1} + x_i + x_{i+1})/3 : i = 1, \ldots, N - 1.$$ 

As before (see figure 2.6), we give each processor a subset of the $x_i$s and $y_i$s. Let’s define $i_p$ as the first index of $y$ that is computed by processor $p$. (What is the last index computed by processor $p$? How many indices are computed on that processor?)

We often talk about the owner computes model of parallel computing: each processor ‘owns’ certain data items, and it computes their value.

Now let’s investigate how processor $p$ goes about computing $y_i$ for the $i$-values it owns. Let’s assume that process $p$ also stores the values $x_i$ for these same indices. Now, for many values $i$ it can evaluate the computation

$$y_i = (x_{i-1} + x_i + x_{i+1})/3$$

locally (figure 4.1).

However, there is a problem with computing $y$ in the first index $i_p$ on processor $p$:

$$y_{i_p} = (x_{i_p-1} + x_{i_p} + x_{i_p+1})/3$$

The point to the left, $x_{i_p-1}$, is not stored on process $p$ (it is stored on $p-1$), so it is not immediately available for use by process $p$. (figure 4.2). There is a similar story with the last index that $p$ tries to compute: that involves a value that is only present on $p + 1$.

You see that there is a need for processor-to-processor, or technically point-to-point, information exchange. MPI realizes this through matched send and receive calls:

• One process does a send to a specific other process;
• the other process does a specific receive from that source.

We will now discuss the send and receive routines in detail.
4.1. Blocking point-to-point operations

4.1.1 Example: ping-pong

A simple scenario for information exchange between just two processes is the ping-pong: process A sends data to process B, which sends data back to A. This means that process A executes the code

```c
MPI_Send(/* to: */ B .....);
MPI_Recv(/* from: */ B ...);
```

while process B executes

```c
MPI_Recv(/* from: */ A ...);
MPI_Send(/* to: */ A .....);
```

Since we are programming in SPMD mode, this means our program looks like:

```c
if ( /* I am process A */ ) {
    MPI_Send(/* to: */ B .....);
    MPI_Recv(/* from: */ B ...);
} else if ( /* I am process B */ ) {
    MPI_Recv(/* from: */ A ...);
    MPI_Send(/* to: */ A .....);
}
```

**Remark 8** The structure of the send and receive calls shows the symmetric nature of MPI: every target process is reached with the same send call, no matter whether it’s running on the same multicore chip as the sender, or on a computational node halfway across the machine room, taking several network hops to reach. Of course,
4. MPI topic: Point-to-point

Figure 4.1 MPI_Send

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Send_c (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(buf)</td>
<td>initial address of send buffer</td>
<td>const void*</td>
<td>TYPE(*),</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>number of elements in send buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td>datatype of each send buffer element</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td></td>
</tr>
<tr>
<td>datatype</td>
<td>IN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dest</td>
<td>rank of destination</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>tag</td>
<td>message tag</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
<td></td>
</tr>
</tbody>
</table>

MPL:

```
template<typename T >
void mpl::communicator::send
    ( const T scalar&,int dest,tag = tag(0) ) const
    ( const T *buffer,const layout< T > &,int dest,tag = tag(0) ) const
    ( iterT begin,iterT end,int dest,tag = tag(0) ) const
T : scalar type
begin : begin iterator
end : end iterator
Python:

any self-respecting MPI implementation optimizes for the case where sender and receiver have access to the same shared memory. This means that a send/recv pair is realized as a copy operation from the sender buffer to the receiver buffer, rather than a network transfer.

4.1.2 Send call

The blocking send command is MPI_Send (figure 4.1). Example:

```c
#include <sendandrecv.c>

double send_data = 1.;

MPI_Send
    ( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
    /* to: */ /* receiver, */ tag: */ 0,
    /* communicator: */ /* comm); */
```

For the full source of this example, see section 4.5.2

The send call has the following elements.
4.1. Blocking point-to-point operations

**Buffer**  The *send buffer* is described by a trio of buffer/count/datatype. See section 3.2.4 for discussion.

**Target**  The *message target* is an explicit process rank to send to. This rank is a number from zero up to the result of `MPI_Comm_size`. It is allowed for a process to send to itself, but this may lead to a runtime deadlock; see section 4.1.4 for discussion.

**Tag**  Next, a message can have a *tag*. Many applications have each sender send only one message at a time to a given receiver. For the case where there are multiple simultaneous messages between the same sender/receiver pair, the tag can be used to disambiguate between the messages.

Often, a tag value of zero is safe to use. Indeed, OO interfaces to MPI typically have the tag as an optional parameter with value zero. If you do use tag values, you can use the key `MPI_TAG_UB` to query what the maximum value is that can be used; see section 15.1.2.

**Communicator**  Finally, in common with the vast majority of MPI calls, there is a communicator argument that provides a context for the send transaction.

*MPL note 23: Blocking send and receive.* MPL uses a default value for the tag, and it can deduce the type of the buffer. Sending a scalar becomes:

```cpp
// sendscalar.cxx
if (comm_world.rank()==0) {
    double pi=3.14;
    comm_world.send(pi, 1); // send to rank 1
    cout << "sent: " << pi << '\n';
} else if (comm_world.rank()==1) {
    double pi=0;
    comm_world.recv(pi, 0); // receive from rank 0
    cout << "got : " << pi << '\n';
}
```

*For the full source of this example, see section 4.5.3*

*MPL note 24: Sending arrays.* MPL can send *static arrays* without further layout specification:

```cpp
// sendarray.cxx
double v[2][2][2];
comm_world.send(v, 1); // send to rank 1
comm_world.recv(v, 0); // receive from rank 0
```

*For the full source of this example, see section 4.5.4*

Sending vectors uses a general mechanism:

```cpp
// sendbuffer.cxx
std::vector<double> v(8);
mpl::contiguous_layout<double> v_layout(v.size());
comm_world.send(v.data(), v_layout, 1); // send to rank 1
comm_world.recv(v.data(), v_layout, 0); // receive from rank 0
```
For the full source of this example, see section 4.5.5

MPL note 25: Iterator buffers. It is possible to send containers by iterators

```c++
// sendrange.cxx
vector<double> v(15);
    comm_world.send(v.begin(), v.end(), 1); // send to rank 1
    comm_world.recv(v.begin(), v.end(), 0); // receive from rank 0
```

For the full source of this example, see section 4.5.6

MPL note 26: Iterator layout. Noncontiguous iterable objects can be send with a iterator_layout:

```cpp
std::list<int> v(20, 0);
mpl::iterator_layout<int> l(v.begin(), v.end());
    comm_world.recv(&(*v.begin()), l, 0);
```

4.1.3 Receive call

The basic blocking receive command is MPI_Recv (figure 4.2).

An example:

```c
double recv_data;
MPI_Recv
( /* recv buffer/count/type: */ &recv_data, 1, MPI_DOUBLE,
    /* from: */ sender, /* tag: */ 0,
    /* communicator: */ comm,
    /* recv status: */ MPI_STATUS_IGNORE);
```

For the full source of this example, see section 4.5.2

This is similar in structure to the send call, with some exceptions.

**Buffer** The receive buffer has the same buffer/count/data parameters as the send call. However, the count argument here indicates the size of the buffer, rather than the actual length of a message. This sets an upper bound on the length of the incoming message.

- For receiving messages with unknown length, use MPI_Probe; section 4.3.1.
- A message longer than the buffer size will give an overflow error, either returning an error, or ending your program; see section 15.2.2.

The length of the received message can be determined from the status object; see section 4.3.2 for more detail.

**Source** Mirroring the target argument of the MPI_Send call, MPI_Recv has a message source argument. This can be either a specific rank, or it can be the MPI_ANY_SOURCE wildcard. In the latter case, the actual source can be determined after the message has been received; see section 4.3.2. A source value of MPI_PROC_NULL is also allowed, which makes the receive succeed immediately with no data received.

MPL note 27: Any source. The constant mpl::any_source equals MPI_ANY_SOURCE (by constexpr).
4.1. Blocking point-to-point operations

Figure 4.2 MPI_Recv

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Recv</td>
<td>buf</td>
<td>initial address of receive buffer</td>
<td>void*</td>
<td>TYPE(*),</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>number of elements in receive buffer</td>
<td>int</td>
<td>DIMENSION(..)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Count</td>
<td>IN</td>
<td>INTEGER IN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>datatype</td>
<td>datatype of each receive buffer element</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>source</td>
<td>rank of source or MPI_ANY_SOURCE</td>
<td>int</td>
<td>INTEGER IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tag</td>
<td>message tag or MPI_ANY_TAG</td>
<td>int</td>
<td>INTEGER IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm) IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>status</td>
<td>status object</td>
<td>MPI_Status*</td>
<td>TYPE(MPI_Status) OUT</td>
<td></td>
</tr>
</tbody>
</table>

MPL:

template<typename T >
status mpl::communicator::recv
  ( T &,int,tag = tag(0) ) const inline
  ( T *,const layout< T > &,int,tag = tag(0) ) const
  ( iterT begin,iterT end,int source, tag t = tag(0) ) const

Python:
Comm.Recv(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG,
  Status status=None)
Python native:
recvbuf = Comm.recv(self, buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,
  Status status=None)

Tag  Similar to the message source, the message tag of a receive call can be a specific value or a wildcard, in this case MPI_ANY_TAG. Again, see below.

Communicator  The communicator argument almost goes without remarking.

Status  The MPI_Recv command has one parameter that the send call lacks: the MPI_Status object, describing the message status. This gives information about the message received, for instance if you used wildcards for source or tag. See section 4.3.2 for more about the status object.

Remark 9  If you’re not interested in the status, as is the case in many examples in this book, specify the constant MPI_STATUS_IGNORE. Note that the signature of MPI_Recv lists the status parameter as ‘output’; this ‘direction’ of the parameter of course only applies if you do not specify this constant. (See also section 15.10.1.)

Exercise 4.1.  Implement the ping-pong program. Add a timer using MPI_Wtime. For the status argument of the receive call, use MPI_STATUS_IGNORE.
4. MPI topic: Point-to-point

- Run multiple ping-pongs (say a thousand) and put the timer around the loop. The first run may take longer; try to discard it.
- Run your code with the two communicating processes first on the same node, then on different nodes. Do you see a difference?
- Then modify the program to use longer messages. How does the timing increase with message size?

For bonus points, can you do a regression to determine $\alpha, \beta$?
(There is a skeleton for this exercise under the name pingpong.)

**Exercise 4.2.** Take your pingpong program and modify it to let half the processors be source and the other half the targets. Does the pingpong time increase?

### 4.1.4 Problems with blocking communication

You may be tempted to think that the send call puts the data somewhere in the network, and the sending code can progress, as in figure 4.3, left. But this ideal scenario is not realistic: it assumes that somewhere in the network there is buffer capacity for all messages that are in transit. This is not the case: data resides on the sender, and the sending call blocks, until the receiver has received all of it. (There is a exception for small messages, as explained in the next section.)

The use of `MPI_Send` and `MPI_Recv` is known as *blocking communication*: when your code reaches a send or receive call, it blocks until the call is successfully completed.

Technically, blocking operations are called *non-local* since their execution depends on factors that are not local to the process. See section 5.4.

For a receive call it is clear that the receiving code will wait until the data has actually come in, but for a send call this is more subtle.

#### 4.1.4.1 Deadlock

Suppose two process need to exchange data, and consider the following pseudo-code, which purports to exchange data between processes 0 and 1:

```c
other = 1-mytid; /* if I am 0, other is 1; and vice versa */
receive(source=other);
send(target=other);
```

![Figure 4.3: Illustration of an ideal (left) and actual (right) send-receive interaction](image-url)
Imagine that the two processes execute this code. They both issue the send call... and then can’t go on, because they are both waiting for the other to issue the send call corresponding to their receive call. This is known as deadlock.

4.1.4.2 Eager vs rendezvous protocol

Messages can be sent using (at least) two different protocols:

1. Rendezvous protocol, and
2. Eager protocol.

The rendezvous protocol is the most general. Sending a message takes several steps:

1. the sender sends a header, typically containing the message envelope;
2. the receiver returns a ‘ready-to-send’ message;
3. the sender sends the actual data.

The purpose of this is to prepare the receiver buffer space for large messages. However, it implies that the sender has to wait for some return message from the receiver, making the behavior a synchronous message.

For the eager protocol, consider the example:

```c
other = 1-mytid; /* if I am 0, other is 1; and vice versa */
send(target=other);
receive(source=other);
```

With a synchronous protocol you should get deadlock, since the send calls will be waiting for the receive operation to be posted.

In practice, however, this code will often work. The reason is that MPI implementations sometimes send small messages regardless of whether the receive has been posted. This relies on the availability of some amount of available buffer space. The size under which this behavior is used is sometimes referred to as the eager limit.

To illustrate eager and blocking behavior in MPI_Send, consider an example where we send gradually larger messages. From the screen output you can see what the largest message was that fell under the eager limit; after that the code hangs because of a deadlock.

```c
// sendblock.c
other = 1-procno;
/* loop over increasingly large messages */
for (int size=1; size<2000000000; size*=10) {
    sendbuf = (int*) malloc(size*sizeof(int));
    recvbuf = (int*) malloc(size*sizeof(int));
    if (!sendbuf || !recvbuf) {
        printf("Out of memory\n"); MPI_Abort(comm,1);
    }
    MPI_Send(sendbuf,size,MPI_INT,other,0,comm);
    MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
    /* If control reaches this point, the send call
did not block. If the send call blocks,
we do not reach this point, and the program will hang.
```
For the full source of this example, see section 4.5.7

```c
!! sendblock.F90
other = 1 - mytid
size = 1
do
    allocate(sendbuf(size)); allocate(recvbuf(size))
    print *, size
    call MPI_Send(sendbuf, size, MPI_INTEGER, other, 0, comm, err)
    call MPI_Recv(recvbuf, size, MPI_INTEGER, other, 0, comm, status, err)
    if (mytid==0) then
        print *, "MPI_Send did not block for size", size
    end if
    deallocate(sendbuf); deallocate(recvbuf)
    size = size*10
    if (size>2000000000) goto 20
end do
20 continue
```

For the full source of this example, see section 4.5.8

```python
# sendblock.py
size = 1
while size<2000000000:
    sendbuf = np.empty(size, dtype=int)
    recvbuf = np.empty(size, dtype=int)
    comm.Send(sendbuf, dest=other)
    comm.Recv(recvbuf, source=other)
    if procid<other:
        print("Send did not block for", size)
    size *= 10
```

For the full source of this example, see section 4.5.9

If you want a code to exhibit the same blocking behavior for all message sizes, you force the send call to be blocking by using `MPI_Ssend`, which has the same calling sequence as `MPI_Send`, but which does not allow eager sends.

```c
// ssendblock.c
other = 1 - procnos;
sendbuf = (int*) malloc(sizeof(int));
recvbuf = (int*) malloc(sizeof(int));
size = 1;
MPI_Ssend(sendbuf, size, MPI_INT, other, 0, comm);
MPI_Recv(recvbuf, size, MPI_INT, other, 0, comm, &status);
printf("This statement is not reached\n");
```

For the full source of this example, see section 4.5.10

Formally you can describe deadlock as follows. Draw up a graph where every process is a node, and draw a directed arc from process A to B if A is waiting for B. There is deadlock if this directed graph has a loop.
4.1. Blocking point-to-point operations

The solution to the deadlock in the above example is to first do the send from 0 to 1, and then from 1 to 0
(or the other way around). So the code would look like:

```c
if ( /* I am processor 0 */ ) {
    send(target=other);
    receive(source=other);
} else {
    receive(source=other);
    send(target=other);
}
```

Eager sends also influences non-blocking sends. The wait call after a non-blocking send:

**Code:**

```c
// eager1.c
printf("Sending %lu elements\n",n);
MPI_Request request;
MPI_Isend(buffer,n,MPI_DOUBLE,processB,0,comm,&request);
MPI_Wait(&request,MPI_STATUS_IGNORE);
printf(".. concluded\n");
```

**Output:**

Setting eager limit to 5000 bytes
TACC: Starting up job 4049189
TACC: Starting parallel tasks...
Sending 1 elements
.. concluded
Sending 10 elements
.. concluded
Sending 100 elements
.. concluded
Sending 1000 elements
.. concluded
^C
```

will return immediately, regardless any receive call, if the message is under the eager limit.

The eager limit is implementation-specific. For instance, for Intel MPI there is a variable `I_MPI_EAGER_THRESHOLD` (old versions) or `I_MPI_SHM_EAGER_THRESHOLD`; for mvapich2 it is `MV2_IBA_EAGER_THRESHOLD`, and for OpenMPI the `--mca` options `btl_openib_eager_limit` and `btl_openib_rndv_eager_limit`.

4.1.4.3 Serialization

There is a second, even more subtle problem with blocking communication. Consider the scenario where every processor needs to pass data to its successor, that is, the processor with the next higher rank. The basic idea would be to first send to your successor, then receive from your predecessor. Since the last processor does not have a successor it skips the send, and likewise the first processor skips the receive. The pseudo-code looks like:

```c
successor = mytid+1; predecessor = mytid-1;
if ( /* I am not the last processor */ )
    send(target=successor);
if ( /* I am not the first processor */ )
    receive(source=predecessor)
```

**Exercise 4.3.** (Classroom exercise) Each student holds a piece of paper in the right hand – keep your left hand behind your back – and we want to execute:

1. Give the paper to your right neighbor;
2. Accept the paper from your left neighbor.
4. MPI topic: Point-to-point

Including boundary conditions for first and last process, that becomes the following program:

1. If you are not the rightmost student, turn to the right and give the paper to your right neighbor.
2. If you are not the leftmost student, turn to your left and accept the paper from your left neighbor.

This code does not deadlock. All processors but the last one block on the send call, but the last processor executes the receive call. Thus, the processor before the last one can do its send, and subsequently continue to its receive, which enables another send, et cetera.

In one way this code does what you intended to do: it will terminate (instead of hanging forever on a deadlock) and exchange data the right way. However, the execution now suffers from unexpected serialization: only one processor is active at any time, so what should have been a parallel operation becomes a sequential one. This is illustrated in figure 4.4.

**Exercise 4.4.** Implement the above algorithm using `MPI_Send` and `MPI_Recv` calls. Run the code, and use TAU to reproduce the trace output of figure 4.4. If you don’t have TAU, can you show this serialization behavior using timings, for instance running it on an increasing number of processes? (There is a skeleton for this exercise under the name `rightsend`.)

It is possible to orchestrate your processes to get an efficient and deadlock-free execution, but doing so is a bit cumbersome.

**Exercise 4.5.** The above solution treated every processor equally. Can you come up with a solution that uses blocking sends and receives, but does not suffer from the serialization behavior?

There are better solutions which we will explore in the next section.
4.1. Blocking point-to-point operations

4.1.5 Bucket brigade

The problem with the previous exercise was that an operation that was conceptually parallel, became serial in execution. On the other hand, sometimes the operation is actually serial in nature. One example is the bucket brigade operation, where a piece of data is successively passed down a sequence of processors.

**Exercise 4.6.** Take the code of exercise 4.4 and modify it so that the data from process zero gets propagated to every process. Specifically, compute all partial sums $\sum_{i=0}^{p} i^2$:

\[
\begin{cases}
  x_0 = 1 & \text{on process zero} \\
  x_p = x_{p-1} + (p + 1)^2 & \text{on process } p
\end{cases}
\]

Use MPI_Send and MPI_Recv; make sure to get the order right.

Food for thought: all quantities involved here are integers. Is it a good idea to use the integer datatype here?

Question: could you have done this with a collective call? (There is a skeleton for this exercise under the name bucketblock.)

**Remark 10** There is an MPI_Scan routine (section 3.4) that performs the same computation, but computationally more efficiently. Thus, this exercise only serves to illustrate the principle.

4.1.6 Pairwise exchange

Above you saw that with blocking sends the precise ordering of the send and receive calls is crucial. Use the wrong ordering and you get either deadlock, or something that is not efficient at all in parallel. MPI has a way out of this problem that is sufficient for many purposes: the combined send/recv call MPI_Sendrecv (figure 4.3).

The sendrecv call works great if every process is paired up. You would then write

\[
\begin{align*}
\| & \text{sendrecv( ....from... ...to... )}; \\
\text{MPI_Comm_rank}(comm,&\text{procno}); \\
\text{MPI_Sendrecv}( .... \\
  /* from: */ procno-1 \\
  ... ... \\
  /* to: */ procno+1 \\
  ... );
\end{align*}
\]

with the right choice of source and destination. For instance, to send data to your right neighbor:

\[
\begin{align*}
\text{MPI_Comm_rank}(comm,&\text{procno}); \\
\text{MPI_Sendrecv}( .... \\
  /* from: */ procno-1 \\
  ... ... \\
  /* to: */ procno+1 \\
  ... );
\end{align*}
\]

This scheme is correct for all processes but the first and last. In order to use the sendrecv call on these processes, we use MPI_PROC_NULL for the non-existing processes that the endpoints communicate with.

\[
\begin{align*}
\text{MPI_Comm_rank}( .... &\text{mytid } ); \\
\text{if ( /* I am not the first processor */ )} \\
  \text{predecessor = mytid-1}; \\
\text{else} \\
  \text{predecessor = MPI_PROC_NULL};
\end{align*}
\]
### MPI topic: Point-to-point

#### Figure 4.3 MPI_Sendrecv

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Sendrecv</td>
<td>sendbuf</td>
<td>initial address of send buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(../)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>sendcount</td>
<td>number of elements in send buffer</td>
<td>int</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sendtype</td>
<td>type of elements in send buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dest</td>
<td>rank of destination</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>sendtag</td>
<td>send tag</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>recvbuf</td>
<td>initial address of receive buffer</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(../)</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>recvcount</td>
<td>number of elements in receive buffer</td>
<td>int</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>recvtype</td>
<td>type of elements receive buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>source</td>
<td>rank of source or MPI_ANY_SOURCE</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>recvtag</td>
<td>receive tag or MPI_ANY_TAG</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>status</td>
<td>status object</td>
<td>MPI_Status*</td>
<td>TYPE(MPI_Status)</td>
<td>OUT</td>
</tr>
</tbody>
</table>

**MPL:**

```cpp
template<typename T >
status mpl::communicator::sendrecv
( const T & senddata, int dest, tag sendtag,
  T & recvdata, int source, tag recvtag )

( const T * senddata, const layout< T > & sendl, int dest, tag sendtag,
  T * recvdata, const layout< T > & recvl, int source, tag recvtag )

( iterT1 begin1, iterT1 end1, int dest, tag sendtag,
  iterT2 begin2, iterT2 end2, int source, tag recvtag )
```

**Python:**

```python
Sendrecv(self,
         sendbuf, int dest, int sendtag=0,
         recvbuf=None, int source=ANY_SOURCE, int recvtag=ANY_TAG,
         Status status=None)
```

---

130  
Parallel Computing – r428
4.1. Blocking point-to-point operations

```c
if ( /* I am not the last processor */ )
    successor = mytid+1;
else
    successor = MPI_PROC_NULL;
sendrecv(from=predecessor,to=successor);
```

where the sendrecv call is executed by all processors.

All processors but the last one send to their neighbor; the target value of MPI_PROC_NULL for the last processor means a 'send to the null processor': no actual send is done. The null processor value is also of use with the MPI_Sendrecv call; section 4.1.6.

Likewise, receive from MPI_PROC_NULL succeeds without altering the receive buffer. The corresponding MPI_Status object has source MPI_PROC_NULL, tag MPI_ANY_TAG, and count zero.

**Remark 11** The MPI_Sendrecv can inter-operate with the normal send and receive calls, both blocking and non-blocking. Thus it would also be possible to replace the MPI_Sendrecv calls at the end points by simple sends or receives.

**MPL note 28: Send-rev call.** The send-rev call in MPL has the same possibilities for specifying the send and receive buffer as the separate send and recv calls: scalar, layout, iterator. However, out of the nine conceivably possible routine signatures, only the versions are available where the send and receive buffer are specified the same way. Also, the send and receive tag need to be specified; they do not have default values.

```cpp
// sendrecv.cxx
mpl::tag t0(0);
comm_world.sendrecv(  
    { mydata,sendto,t0,  
        leftdata,recvfrom,t0  
    );
```

**Exercise 4.7.** Revisit exercise 4.3 and solve it using MPI_Sendrecv.

If you have TAU installed, make a trace. Does it look different from the serialized send/recv code? If you don’t have TAU, run your code with different numbers of processes and show that the runtime is essentially constant.

This call makes it easy to exchange data between two processors: both specify the other as both target and source. However, there need not be any such relation between target and source: it is possible to receive from a predecessor in some ordering, and send to a successor in that ordering; see figure 4.5.

For the above three-point combination scheme you need to move data both left right, so you need two MPI_Sendrecv calls; see figure 4.6.

**Exercise 4.8.** Implement the above three-point combination scheme using MPI_Sendrecv; every processor only has a single number to send to its neighbor.

(There is a skeleton for this exercise under the name sendrecv.)

Hints for this exercise:

- Each process does one send and one receive; if a process needs to skip one or the other, you can specify MPI_PROC_NULL as the other process in the send or receive specification. In that case the corresponding action is not taken.
4. MPI topic: Point-to-point

Figure 4.5: An MPI Sendrecv call

Figure 4.6: Two steps of send/recv to do a three-point combination

- As with the simple send/recv calls, processes have to match up: if process $p$ specifies $p'$ as the destination of the send part of the call, $p'$ needs to specify $p$ as the source of the recv part.

The following exercise lets you implement a sorting algorithm with the send-receive call.

**Exercise 4.9.** A very simple sorting algorithm is swap sort or odd-even transposition sort: pairs of processors compare data, and if necessary exchange. The elementary step is called a compare-and-swap: in a pair of processors each sends their data to the other; one keeps the minimum values, and the other the maximum. For simplicity, in this exercise we give each processor just a single number.

The exchange sort algorithm is split in even and odd stages, where in the even stage, processors $2i$ and $2i + 1$ compare and swap data, and in the odd stage, processors $2i + 1$ and $2i + 2$ compare and swap. You need to repeat this $P/2$ times, where $P$ is the number of processors; see figure 4.7.

Implement this algorithm using MPI_Sendrecv. (Use MPI_PROC_NULL for the edge cases if needed.) Use a gather call to print the global state of the distributed array at the beginning and end of the sorting process.

**Remark 12** It is not possible to use MPI_IN_PLACE for the buffers. Instead, the routine MPI_Sendrecv_replace (figure 4.4) has only one buffer, used as both send and receive buffer. Of course, this requires the send and receive messages to fit in that one buffer.

---

1. There is an MPI_Compare_and_swap call. Do not use that.
4.2 Nonblocking point-to-point operations

The structure of communication is often a reflection of the structure of the operation. With some regular applications we also get a regular communication pattern. Consider again the above operation:

\[ y_i = x_{i-1} + x_i + x_{i+1} : i = 1, \ldots, N - 1 \]

**Exercise 4.10.** Extend this exercise to the case where each process hold an equal number of elements, more than 1. Consider figure 4.8 for inspiration. Is it coincidence that the algorithm takes the same number of steps as in the single scalar case?

The following material is for the recently released MPI-4 standard and may not be supported yet.

There are non-blocking and persistent versions of this routine: MPI_Isendrecv, MPI_Sendrecv_init, MPI_Isendrecv_replace, MPI_Sendrecv_replace_init.

End of MPI-4 material
4. MPI topic: Point-to-point

Figure 4.4 MPI_Sendrecv_replace

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Sendrecv_replace (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>buf</td>
<td>initial address of send and receive buffer</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>INOUT</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>number of elements in send and receive buffer</td>
<td>int</td>
<td></td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>MPI_Count INTEGER IN</td>
<td>type of elements in send and receive buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>dest</td>
<td>rank of destination</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>sendtag</td>
<td>send message tag</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>source</td>
<td>rank of source or MPI_ANY_SOURCE</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>recvtag</td>
<td>receive message tag or MPI_ANY_TAG</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>status</td>
<td>status object</td>
<td>MPI_Status*</td>
<td>TYPE(MPI_Status)</td>
<td>OUT</td>
<td></td>
</tr>
</tbody>
</table>

Doing this in parallel induces communication, as pictured in figure 4.1.

We note:
- The data is one-dimensional, and we have a linear ordering of the processors.
- The operation involves neighboring data points, and we communicate with neighboring processors.

Above you saw how you can use information exchange between pairs of processors
- using MPI_Send and MPI_Recv, if you are careful; or
- using MPI_Sendrecv, as long as there is indeed some sort of pairing of processors.

However, there are circumstances where it is not possible, not efficient, or simply not convenient, to have such a deterministic setup of the send and receive calls. Figure 4.9 illustrates such a case, where processors are organized in a general graph pattern. Here, the numbers of sends and receive of a processor do not

![Figure 4.9: Processors with unbalanced send/receive patterns](image-url)
need to match.

In such cases, one wants a possibility to state 'these are the expected incoming messages', without having to wait for them in sequence. Likewise, one wants to declare the outgoing messages without having to do them in any particular sequence. Imposing any sequence on the sends and receives is likely to run into the serialization behavior observed above, or at least be inefficient since processors will be waiting for messages.

### 4.2.1 Nonblocking send and receive calls

In the previous section you saw that blocking communication makes programming tricky if you want to avoid deadlock and performance problems. The main advantage of these routines is that you have full control about where the data is: if the send call returns the data has been successfully received, and the send buffer can be used for other purposes or de-allocated.

By contrast, the nonblocking calls MPI_Isend (figure 4.5) and MPI_Irecv (figure 4.6) (where the ‘I’ stands for ‘immediate’ or ‘incomplete’) do not wait for their counterpart: in effect they tell the runtime system ‘here is some data and please send it as follows’ or ‘here is some buffer space, and expect such-and-such data to come’. This is illustrated in figure 4.10.

```c
// isendandirecv.c
double send_data = 1.;
MPI_Request request;
MPI_Isend
    ( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
      /* to: */ receiver, /* tag: */ 0,
      /* communicator: */ comm,
      /* request: */ &request);
MPI_Wait(&request,MPI_STATUS_IGNORE);
```

For the full source of this example, see section 4.5.11
Figure 4.5 MPI_Isend

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Isend</td>
<td></td>
<td>initial address of send buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>IN</td>
</tr>
<tr>
<td>MPI_Isend_c</td>
<td>buf</td>
<td></td>
<td>count</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>number of elements in send buffer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td>datatype of each send buffer element</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype) IN</td>
<td></td>
</tr>
<tr>
<td>datatype</td>
<td></td>
<td></td>
<td>dest</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rank of destination</td>
<td></td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td>tag</td>
<td></td>
<td>message tag</td>
<td>tag</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td></td>
<td>communicator</td>
<td>comm</td>
<td>TYPE(MPI_Comm) IN</td>
<td>IN</td>
</tr>
<tr>
<td>request</td>
<td></td>
<td>communication request</td>
<td>request</td>
<td>TYPE(MPI_Request) OUT</td>
<td>OUT</td>
</tr>
</tbody>
</table>

MPL:

```cpp
template<typename T>
irequest mpl::communicator::isend
  ( const T & data, int dest, tag t = tag(0) ) const;
  ( const T * data, const layout<T> & l, int dest, tag t = tag(0) ) const;
  ( iterT begin, iterT end, int dest, tag t = tag(0) ) const;
```  

Python:

```python
request = MPI.Comm.Isend(self, buf, int dest, int tag=0)
```

/* from: */ sender, /* tag: */ 0,
/* communicator: */ comm,
/* request: */ &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);

For the full source of this example, see section 4.5.11

Issuing the MPI_Isend / MPI_Irecv call is sometimes referred to as posting a send/receive.

4.2.2 Request completion: wait calls

From the definition of MPI_Isend / MPI_Irecv, you seen that nonblocking routine yields an MPI_Request object. This request can then be used to query whether the operation has concluded. You may also notice that the MPI_Irecv routine does not yield an MPI_Status object. This makes sense: the status object describes the actually received data, and at the completion of the MPI_Irecv call there is no received data yet.

Waiting for the request is done with a number of routines. We first consider MPI_Wait (figure 4.7). It takes the request as input, and gives an MPI_Status as output. If you don’t need the status object, you can pass MPI_STATUS_IGNORE.

```c++
// hangwait.c
if (procno==sender) {
  for (int p=0; p<nprocs-1; p++) {
    double send = 1.;
  
```
4.2. Nonblocking point-to-point operations

### Figure 4.6 MPI_Irecv

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Irecv</td>
<td>buf</td>
<td>initial address of receive buffer</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(...)</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>number of elements in receive buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>datatype</td>
<td>datatype of each receive buffer element</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>source</td>
<td>rank of source or MPI_ANY_SOURCE</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>tag</td>
<td>message tag or MPI_ANY_TAG</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>request</td>
<td>communication request</td>
<td>MPI_Request*</td>
<td>TYPE(MPI_Request)OUT</td>
<td></td>
</tr>
</tbody>
</table>

MPL:

```cpp
template<typename T >
irequest mpl::communicator::irecv
    ( const T & data, int src, tag t = tag(0) ) const;
    ( const T * data, const layout< T > & l, int src, tag t = tag(0) ) const;
    ( iterT begin, iterT end, int src, tag t = tag(0) ) const;
```

Python:

```python
recvbuf = Comm.irecv(self, buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,
                     Request request=None)
```

Python numpy:

```python
Comm.Irecv(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG,
           Request status=None)
```

### Figure 4.7 MPI_Wait

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Wait</td>
<td>request</td>
<td>request</td>
<td>MPI_Request*</td>
<td>TYPE(MPI_Request)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>status</td>
<td>status object</td>
<td>MPI_Status*</td>
<td>TYPE(MPI_Status)</td>
<td></td>
</tr>
</tbody>
</table>

Python:

```
MPI.Request.Wait(type cls, request, status=None)
```
4. MPI topic: Point-to-point

```c
MPI_Send( &send,1,MPI_DOUBLE,p,0,comm);

else {
    double recv=0.;
    MPI_Request request;
    MPI_Irecv( &recv,1,MPI_DOUBLE,sender,0,comm,&request);
    MPI_Wait(&request,MPI_STATUS_IGNORE);
}
```

For the full source of this example, see section 4.5.12

The request is passed by reference, so that the wait routine can free it:

- The wait call deallocates the request object, and
- sets the value of the variable to `MPI_REQUEST_NULL`.

(See section 4.2.4 for details.)

*MPL note 29: Requests from nonblocking calls.* Nonblocking routines have an `irequest` as function result. Note: not a parameter passed by reference, as in the C interface. The various wait calls are methods of the `irequest` class.

```c
double recv_data;
mpl::irequest recv_request =
    comm_world.irecv( recv_data,sender );
recv_request.wait();
```

For the full source of this example, see section 4.5.13

You can not default-construct the request variable:

```c
// DOES NOT COMPIL:
mpl::irequest recv_request;
recv_request = comm.irecv( ... );
```

This means that the normal sequence of first declaring, and then filling in, the request variable is not possible.

*MPL implementation note:* The wait call always returns a `status` object; not assigning it means that the destructor is called on it.

Here we discuss in some detail the various wait calls. These are blocking; for the nonblocking versions see section 4.2.3.

### 4.2.2.1 Wait for one request

`MPI_Wait` waits for a a single request. If you are indeed waiting for a single nonblocking communication to complete, this is the right routine. If you are waiting for multiple requests you could call this routine in a loop.

```c
for (p=0; p<nrequests ; p++) // Not efficient!
    MPI_Wait(&request[p],&status[p]);
```

However, this would be inefficient if the first request is fulfilled much later than the others: your waiting process would have lots of idle time. In that case, use one of the following routines.
4.2. Nonblocking point-to-point operations

Figure 4.8 MPI_Waitall

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Waitall</td>
<td>count</td>
<td>list length</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>array_of_requests</td>
<td>array of requests</td>
<td>MPI_Request[]</td>
<td>TYPE(MPI_Request)(count)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_of_statuses</td>
<td>array of status objects</td>
<td>MPI_Status[]</td>
<td>TYPE(MPI_Status)(*)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Request.Waitall(type cls, requests, statuses=None)
```

4.2.2.2 Wait for all requests

**MPI_Waitall** allows you to wait for a number of requests, and it does not matter in what sequence they are satisfied. Using this routine is easier to code than the loop above, and it could be more efficient.

```c
// irecvloop.c
MPI_Request requests =
    (MPI_Request*) malloc( 2*nprocs*sizeof(MPI_Request) );
recv_buffers = (int*) malloc( nprocs*sizeof(int) );
send_buffers = (int*) malloc( nprocs*sizeof(int) );
for (int p=0; p<nprocs; p++) {
    int
        left_p = (p-1+nprocs) % nprocs,
        right_p = (p+1) % nprocs;
    send_buffer[p] = nprocs-p;
    MPI_Isend(sendbuffer+p,1,MPI_INT, right_p,0, requests+2*p);
    MPI_Irecv(recvbuffer+p,1,MPI_INT, left_p,0, requests+2*p+1);
}
/* your useful code here */
MPI_Waitall(2*nprocs,requests,MPI_STATUSES_IGNORE);
```

The output argument is an array or **MPI_Status** object. If you don’t need the status objects, you can pass **MPI_STATUSES_IGNORE**.

**Exercise 4.11.** Revisit exercise 4.6 and consider replacing the blocking calls by nonblocking ones. How far apart can you put the **MPI_Isend**/**MPI_Irecv** calls and the corresponding **MPI_Wait**s?

(There is a skeleton for this exercise under the name bucketpipenonblock.)

**Exercise 4.12.** Create two distributed arrays of positive integers. Take the set difference of the two: the first array needs to be transformed to remove from it those numbers that are in the second array.

How could you solve this with an **MPI_Allgather** call? Why is it not a good idea to do so? Solve this exercise instead with a circular bucket brigade algorithm.

(There is a skeleton for this exercise under the name setdiff.)

**Python note 13: Handling a single request.** Non-blocking routines such as **MPI_Isend** return a request object. The **MPI_Wait** is a class method, not a method of the request object:

```python
Victor Eijkhout
```
4. MPI topic: Point-to-point

```python
## irecvsingle.py
sendbuffer = np.empty( nprocs, dtype=int )
recvbuffer = np.empty( nprocs, dtype=int )

left_p = (procid-1) % nprocs
right_p = (procid+1) % nprocs
send_request = comm.Isend\
   ( sendbuffer[procid:procid+1], dest=left_p)
recv_request = comm.Irecv\
   ( sendbuffer[procid:procid+1], source=right_p)
MPI.Request.Wait(send_request)
MPI.Request.Wait(recv_request)
```

For the full source of this example, see section 4.5.14

**Python note 14: Request arrays.** An array of requests (for the waitall/some/any calls) is an ordinary Python list:

```python
## irecvloop.py
requests = [] # [ None ] * (2*nprocs)
sendbuffer = np.empty( nprocs, dtype=int )
recvbuffer = np.empty( nprocs, dtype=int )

for p in range(nprocs):
    left_p = (p-1) % nprocs
    right_p = (p+1) % nprocs
    requests.append( comm.Isend\n       ( sendbuffer[p:p+1], dest=left_p )
    requests.append( comm.Irecv\n       ( sendbuffer[p:p+1], source=right_p )
    )
MPI.Request.Waitall(requests)
```

For the full source of this example, see section 4.5.14

The **MPI.Waitall** method is again a class method.

### 4.2.2.3 Wait for any requests

The ‘waitall’ routine is good if you need all nonblocking communications to be finished before you can proceed with the rest of the program. However, sometimes it is possible to take action as each request is satisfied. In that case you could use **MPI.Waitany** (figure 4.9) and write:

```c
for (p=0; p<nrequests; p++) {
    MPI_Irecv(buffer+index, /* ... */ , requests+index);
}
for (p=0; p<nrequests; p++) {
    MPI_Waitany(nrequests, request_array, &index, &status);
    // operate on buffer[index]
}
```

Note that this routine takes a single status argument, passed by reference, and not an array of statuses!

**Fortran note 6: Index of requests.** The index parameter is the index in the array of requests, so it uses 1-based indexing.
4.2. Nonblocking point-to-point operations

Figure 4.9 MPI_Waitany

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Waitany</td>
<td>count</td>
<td>list length</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>array_of_requests</td>
<td>array of requests</td>
<td>MPI_Request[]</td>
<td>TYPE(MPI_Request)(count)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>index</td>
<td>index of handle for operation</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>status</td>
<td>status object</td>
<td>MPI_Status*</td>
<td>TYPE(MPI_Status)</td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Request.Waitany( requests, status=None )
```
class method, returns index

For the full source of this example, see section 4.5.15

```fortran
!! irecvsource.F90
  if (mytid==ntids-1) then
    do p=1,ntids-1
      print *,"post"
      call MPI_Irecv(recv_buffer(p),1,MPI_INTEGER,p-1,0,comm,&
                      requests(p),err)
    end do
    do p=1,ntids-1
      call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE,err)
      write(*,'("Message from",i3,"":",i5)') index,recv_buffer(index)
    end do
  end if
```

For the full source of this example, see section 4.5.16

```fortran
!! waitnull.F90
  Type(MPI_Request),dimension(:),allocatable :: requests
  allocate(requests(ntids-1))
  call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE)
  if (.not. requests(index)==MPI_REQUEST_NULL) then
    print *,"This request should be null:",index
  end if
```

For the full source of this example, see section 4.5.16

**MPL note 30: Request pools.** Instead of an array of requests, use an *irequest_pool* object, which acts like a vector of requests, meaning that you can *push* onto it.

```c++
// irecvsource.hxx
mpl::irequest_pool recv_requests;
for (int p=0; p<nprocs-1; p++) {
  recv_requests.push( comm_world.irecv( recv_buffer[p] , p ) );
}
```

Victor Eijkhout
For the full source of this example, see section 4.5.17

You can not declare a pool of a fixed size and assign elements.

MPL note 31: Wait any. The `irequest_pool` class has methods `waitany`, `waitall`, `testany`, `testall`, `waitsome`, `testsome`.

The 'any' methods return a `std::pair<bool, size_t>`, with `false` meaning `index==MPI_UNDEFINED` meaning no more requests to be satisfied.

```cpp
auto [success, index] = recv_requests.waitany();
if (success) {
    auto recv_status = recv_requests.get_status(index);
}
```

For the full source of this example, see section 4.5.17

Same for `testany`, then false means no requests test true.

### Polling with MPI Wait any

The `MPI_Waitany` routine can be used to implement polling: occasionally check for incoming messages while other work is going on.

Code:

```c
// irecvsource.c
if (procno==nprocs-1) {
    int *recv_buffer;
    MPI_Request *request; MPI_Status status;
    recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));
    request = (MPI_Request*) malloc((nprocs-1)*sizeof(MPI_Request));

    for (int p=0; p<nprocs-1; p++) {
        ierr = MPI_Irecv(recv_buffer+p,1,MPI_INT, p,0,comm,
                          request+p); CHK(ierr);
    }
    for (int p=0; p<nprocs-1; p++) {
        int index,sender;
        MPI_Waitany(nprocs-1,request,&index,&status);
        if (index==status(MPI_SOURCE))
            printf("Mismatch index %d vs source %d\n",
                    index,status(MPI_SOURCE));
        printf("Message from %d: %d\n",
                index,recv_buffer[index]);
    }
} else {
    ierr = MPI_Send(&procno,1,MPI_INT, nprocs-1,0,comm);
}
```

Output:

```
make[3]: `irecvsource' is up to date.
process 1 waits 6s before sending
process 2 waits 3s before sending
process 0 waits 13s before sending
process 3 waits 8s before sending
process 5 waits 1s before sending
process 6 waits 14s before sending
process 4 waits 12s before sending
Message from 5: 5
Message from 2: 2
Message from 1: 1
Message from 3: 3
Message from 4: 4
Message from 0: 0
Message from 6: 6
```

```
## irecvsource.py
if procid==nprocs-1:
    receive_buffer = np.empty(nprocs-1,dtype=int)
    requests = [ None ] * (nprocs-1)
```
4.2. Nonblocking point-to-point operations

```python
def example_code():
    for sender in range(nprocs-1):
        requests[sender] = comm.Irecv(receive_buffer[sender:sender+1], source=sender)
    # alternatively: requests = [ comm.Irecv(s) for s in ... ]
    status = MPI.Status()
    for sender in range(nprocs-1):
        ind = MPI.Request.Waitany(requests, status=status)
        if ind != status.Get_source():
            print(f"sender mismatch: %d vs %d" % (ind, status.Get_source()))
            print(f"received from", ind)
        else:
            mywait = random.randint(1,2*nprocs)
            print(f"[%d] wait for %d seconds" % (procid, mywait))
            time.sleep(mywait)
            mydata = np.empty(1, dtype=int)
            mydata[0] = procid
            comm.Send([mydata, MPI.INT], dest=nprocs-1)

        mywait = random.randint(1,2*nprocs)
        print(f"[%d] wait for %d seconds" % (procid, mywait))
        time.sleep(mywait)
        mydata = np.empty(1, dtype=int)
        mydata[0] = procid
        comm.Send([mydata, MPI.INT], dest=nprocs-1)
```

For the full source of this example, see section 4.5.18

Each process except for the root does a blocking send; the root posts MPI_Irecv from all other processors, then loops with MPI_Waitany until all requests have come in. Use MPI_SOURCE to test the index parameter of the wait call.

Note the MPI_STATUS_IGNORE parameter: we know everything about the incoming message, so we do not need to query a status object. Contrast this with the example in section 4.3.2.1.

4.2.2.5 Wait for some requests

Finally, MPI_Waitsome is very much like MPI_Waitany, except that it returns multiple numbers, if multiple requests are satisfied. Now the status argument is an array of MPI_Status objects.

Figure 4.11 shows the trace of a nonblocking execution using MPI_Waitall.

4.2.2.6 Receive status of the wait calls

The MPI_Wait... routines have the MPI_Status objects as output. If you are not interested in the status information, you can use the values MPI_STATUS_IGNORE for MPI_Wait and MPI_Waitany, or MPI_STATUSES_IGNORE for MPI_Waitall, MPI_Waitsome, MPI_Testall, MPI_Testsome.

**Remark 13** The routines that can return multiple statuses, can return the error condition MPI_ERR_IN_STATUS, indicating that one of the statuses was in error. See section 4.3.2.3.

**Exercise 4.13.**

(There is a skeleton for this exercise under the name isendirecv.) Now use nonblocking send/receive routines to implement the three-point averaging operation

\[ y_i = \frac{x_{i-1} + x_i + x_{i+1}}{3} : i = 1, ..., N - 1 \]

on a distributed array. (Hint: use MPI_PROC_NULL at the ends.)
4. MPI topic: Point-to-point

4.2.2.7 Latency hiding / overlapping communication and computation

There is a second motivation for the $\text{Isend}/\text{Irecv}$ calls: if your hardware supports it, the communication can happen while your program can continue to do useful work:

```c
// start nonblocking communication
MPI_Isend( ... ); MPI_Irecv( ... );
// do work that does not depend on incoming data
....
// wait for the Isend/Irecv calls to finish
MPI_Wait( ... );
// now do the work that absolutely needs the incoming data
....
```

This is known as overlapping computation and communication, or latency hiding. See also asynchronous progress; section 15.4.

Unfortunately, a lot of this communication involves activity in user space, so the solution would have been to let it be handled by a separate thread. Until recently, processors were not efficient at doing such multi-threading, so true overlap stayed a promise for the future. Some network cards have support for this overlap, but it requires a nontrivial combination of hardware, firmware, and MPI implementation.

**Exercise 4.14.**

(There is a skeleton for this exercise under the name `isendirecvarray`.) Take your code of exercise 4.13 and modify it to use latency hiding. Operations that can be performed without needing data from neighbors should be performed in between the $\text{MPI_Isend} / \text{MPI_Irecv}$ calls and the corresponding $\text{MPI_Wait}$ calls.

**Remark 14** You have now seen various send types: blocking, nonblocking, synchronous. Can a receiver see what kind of message was sent? Are different receive routines needed? The answer is that, on the receiving
end, there is nothing to distinguish a nonblocking or synchronous message. The `MPI_Recv` call can match any of the send routines you have seen so far (but not `MPI_Sendrecv`), and conversely a message sent with `MPI_Send` can be received by `MPI_Irecv`.

### 4.2.2.8 Buffer issues in nonblocking communication

While the use of nonblocking routines prevents deadlock, it introduces problems of its own.

- With a blocking send call, you could repeatedly fill the send buffer and send it off.

```c
double *buffer;
for ( ... p ... ) {
    buffer = // fill in the data
    MPI_Send( buffer, ... /* to: */ p );
}
```

- On the other hand, when a nonblocking send call returns, the actual send may not have been executed, so the send buffer may not be safe to overwrite. Similarly, when the recv call returns, you do not know for sure that the expected data is in it. Only after the corresponding wait call are you use that the buffer has been sent, or has received its contents.

- To send multiple messages with nonblocking calls you therefore have to allocate multiple buffers.

```c
double **buffers;
for ( ... p ... ) {
    buffers[p] = // fill in the data
    MPI_Send( buffers[p], ... /* to: */ p );
}
MPI_Wait( /* the requests */);
```

```c
// irecvloop.c
MPI_Request requests =
    (MPI_Request*) malloc( 2*nprocs*sizeof(MPI_Request) );
recv_buffers = (int*) malloc( nprocs*sizeof(int) );
send_buffers = (int*) malloc( nprocs*sizeof(int) );
for (int p=0; p<nprocs; p++) {
    int
        left_p = (p-1+nprocs) % nprocs,
        right_p = (p+1) % nprocs;
    send_buffer[p] = nprocs-p;
    MPI_Isend(sendbuffer+p,1,MPI_INT, right_p,0, requests+2*p);
    MPI_Irecv(recvbuffer+p,1,MPI_INT, left_p,0, requests+2*p+1);
}
/* your useful code here */
MPI_Waitall(2*nprocs,requests,MPI_STATUSES_IGNORE);
```

For the full source of this example, see section 4.5.19

The last example we explicitly noted the possibility of overlapping computation and communication.

### 4.2.3 Wait and test calls

The `MPI_Wait`... routines are blocking. Thus, they are a good solution if the receiving process cannot do anything until the data (or at least some data) is actually received. The `MPI_Test`... calls are themselves
nonblocking: they test for whether one or more requests have been fullfilled, but otherwise immediately return. It is also a local operation: it does not force progress.

The MPI_Test call can be used in the manager-worker model: the manager process creates tasks, and sends them to whichever worker process has finished its work. (This uses a receive from MPI_ANY_SOURCE, and a subsequent test on the MPI_SOURCE field of the receive status.) While waiting for the workers, the manager can do useful work too, which requires a periodic check on incoming message.

Pseudo-code:

```c
while (not done) {
    // create new inputs for a while
    ....
    // see if anyone has finished
    MPI_Test( .... &index, &flag );
    if (flag) {
        // receive processed data and send new
    }
}
```

If the test is true, the request is deallocated and set to MPI_REQUEST_NULL, or, in the case of an active persistent request, set to inactive.

Analogous to MPI_Wait, MPI_Waitany, MPI_Waitall, MPI_Waitsome, there are MPI_Test (figure 4.10), MPI_Testany, MPI_Testall, MPI_Testsome.

**Exercise 4.15.** Read section HPSC-6.5 and give pseudo-code for the distributed sparse matrix-vector product using the above idiom for using MPI_Test... calls. Discuss the advantages and disadvantages of this approach. The answer is not going to be black and white: discuss when you expect which approach to be preferable.

### 4.2.4 More about requests

Every nonblocking call allocates an MPI_Request object. Unlike MPI_Status, an MPI_Request variable is not actually an object, but instead it is an (opaque) pointer. This means that when you call, for instance, MPI_Irecv, MPI will allocate an actual request object, and return its address in the MPI_Request variable.

Correspondingly, calls to MPI_Wait or MPI_Test free this object, setting the handle to MPI_REQUEST_NULL. (There is an exception for persistent communications where the request is only set to ‘inactive’; sec-
4.3 More about point-to-point communication

4.3.1 Message probing

MPI receive calls specify a receive buffer, and its size has to be enough for any data sent. In case you really have no idea how much data is being sent, and you don’t want to overallocate the receive buffer, you can use a ‘probe’ call.

The routines MPI_Probe (figure 4.13) and MPI_Iprobe (figure 4.14) (for which see also section 15.4) accept a message but do not copy the data. Instead, when probing tells you that there is a message, you can use MPI_Get_count (section 4.3.2.4) to determine its size, allocate a large enough receive buffer, and do a regular receive to have the data copied.

```c
// probe.c
if (procno==receiver) {
    MPI_Status status;
    MPI_Probe(sender,0,comm,&status);
}
```
4. MPI topic: Point-to-point

Figure 4.13 MPI_Probe

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Probe (</td>
<td>source</td>
<td>rank of source or MPI_ANY_SOURCE</td>
<td>int INT</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tag</td>
<td>message tag or MPI_ANY_TAG</td>
<td>int INT</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>status</td>
<td>status object</td>
<td>MPI_Status*</td>
<td>TYPE(MPI_Status)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.14 MPI_Iprobe

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Iprobe (</td>
<td>source</td>
<td>rank of source or MPI_ANY_SOURCE</td>
<td>int INT</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tag</td>
<td>message tag or MPI_ANY_TAG</td>
<td>int INT</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flag</td>
<td>true if there is a matching message that can be received</td>
<td>int* LOGICAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>status</td>
<td>status object</td>
<td>MPI_Status*</td>
<td>TYPE(MPI_Status)</td>
<td></td>
</tr>
</tbody>
</table>

```c
int count;
MPI_Get_count(&status,MPI_FLOAT,&count);
float recv_buffer[count];
MPI_Recv(recv_buffer,count,MPI_FLOAT, sender,0,comm,MPI_STATUS_IGNORE);
} else if (procno==sender) {
    float buffer[buffer_size];
    ierr = MPI_Send(buffer,buffer_size,MPI_FLOAT, receiver,0,comm); CHK(ierr);
} else {
```

For the full source of this example, see section 4.5.20

There is a problem with the MPI_Probe call in a multithreaded environment: the following scenario can happen.

1. A thread determines by probing that a certain message has come in.
2. It issues a blocking receive call for that message...
3. But in between the probe and the receive call another thread has already received the message.
4. …Leaving the first thread in a blocked state with not message to receive.

This is solved by MPI_Mprobe (figure 4.15), which after a successful probe removes the message from the matching queue: the list of messages that can be matched by a receive call. The thread that matched the probe now issues an MPI_Mrecv (figure 4.16) call on that message through an object of type MPI_Message.
4.3. More about point-to-point communication

**Figure 4.15 MPI_Mprobe**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Mprobe (</td>
<td>source</td>
<td>rank of source or MPI_ANY_SOURCE</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tag</td>
<td>message tag or MPI_ANY_TAG</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>message</td>
<td>returned message</td>
<td>MPI_Message*</td>
<td>TYPE(MPI_Message)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>status</td>
<td>status object</td>
<td>MPI_Status*</td>
<td>TYPE(MPI_Status)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.16 MPI_Mrecv**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Mrecv (</td>
<td>buf</td>
<td>initial address of receive buffer</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>number of elements in receive buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>datatype</td>
<td>datatype of each receive buffer element</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>message</td>
<td>message</td>
<td>MPI_Message*</td>
<td>TYPE(MPI_Message)</td>
<td>INOUT</td>
</tr>
<tr>
<td></td>
<td>status</td>
<td>status object</td>
<td>MPI_Status*</td>
<td>TYPE(MPI_Status)</td>
<td>OUT</td>
</tr>
</tbody>
</table>

4.3.2 The Status object and wildcards

In section 4.1.1 you saw that MPI_Recv has a ‘status’ argument of type MPI_Status that MPI_Send lacks. (The various MPI_Wait... routines also have a status argument; see section 4.2.1.) Often you specify MPI_STATUS_IGNORE for this argument: commonly you know what data is coming in and where it is coming from.

However, in some circumstances the recipient may not know all details of a message when you make the receive call, so MPI has a way of querying the status of the message:

- If you are expecting multiple incoming messages, it may be most efficient to deal with them in the order in which they arrive. So, instead of waiting for specific message, you would specify MPI_ANY_SOURCE or MPI_ANY_TAG in the description of the receive message. Now you have to be able to ask ‘who did this message come from, and what is in it’.
- Maybe you know the sender of a message, but the amount of data is unknown. In that case you can overallocate your receive buffer, and after the message is received ask how big it was, or you can ‘probe’ an incoming message and allocate enough data when you find out how much data is being sent.

To do this, the receive call has a MPI_Status parameter. The MPI_Status object is a structure (in C a struct, in F90 an array, in F2008 a derived type) with freely accessible members:
• MPI_ERROR gives the error status of the receive call; see section 4.3.2.3.
• MPI_SOURCE gives the source of the message; see section 4.3.2.1.
• MPI_TAG gives the tag with which the message was received; see section 4.3.2.2.
• The number if items in the message can be deduced from the status object, but through a function call to MPI_Get_count, not as a structure member; see section 4.3.2.4.

Fortran note 7: Status object in f08. The mpi_f08 module turns many handles (such as communicators) from Fortran Integers into Types. Retrieving the integer from the type is usually done through the %val member, but for the status object this is more difficult. The routines MPI_Status_f2f08 and MPI_Status_f082f convert between these. (Remarkably, these routines are even available in C, where they operate on MPI_Fint, MPI_F08_status arguments.)

Python note 15: Status object. The status object is explicitly created before being passed to the receive routine. It has the usual query methods:

```python
# pingpongbig.py
status = MPI.Status()  
comm.Recv( rdata, source=0, status=status)  
count = status.Get_count(MPI.DOUBLE)
```

(The count function without argument returns a result in bytes.)

MPL note 32: Status object. The mpl::status_t object is created by the receive (or wait) call:

```c
mpl::contiguous_layout<double> target_layout(count);
mpl::status_t recv_status =  
comm_world.recv(target.data(), target_layout, the_other);  
recv_count = recv_status.get_count<double>();
```

For the full source of this example, see section 6.10.6

4.3.2.1 Source

In some applications it makes sense that a message can come from one of a number of processes. In this case, it is possible to specify MPI_ANY_SOURCE as the source. To find out the source where the message actually came from, you would use the MPI_SOURCE field of the status object that is delivered by MPI_Recv or the MPI_Wait... call after an MPI_Irecv.

```c
MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,  
        &status);  
sender = status.MPI_SOURCE;
```

There are various scenarios where receiving from ‘any source’ makes sense. One is that of the manager-worker model. The manager task would first send data to the worker tasks, then issues a blocking wait for the data of whichever process finishes first.

This code snippet is a simple model for this: all workers processes wait a random amount of time. For efficiency, the manager process accepts message from any source.

```c
// anysource.c
if (procmo==nprocs-1) {
```
4.3. More about point-to-point communication

```c
/*
 * The last process receives from every other process
 */
int *recv_buffer;
recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));

/*
 * Messages can come in any order, so use MPI_ANY_SOURCE
 */
MPI_Status status;
for (int p=0; p<nprocs-1; p++) {
    err = MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
                   &status); CHK(err);
    int sender = status.MPI_SOURCE;
    printf("Message from sender=\%d: \%d\n", 
           sender,recv_buffer[p]);
}
free(recv_buffer);
} else {
    /*
     * Each rank waits an unpredictable amount of time,
     * then sends to the last process in line.
     */
    float randomfraction = (rand() / (double)RAND_MAX);
    int randomwait = (int) ( nprocs * randomfraction );
    printf("process \%d waits for \%e/\%d=\%d\n", 
            procno,randomfraction,nprocs,randomwait);
    sleep(randomwait);
    err = MPI_Send(&randomwait,1,MPI_INT, nprocs-1,0,comm); CHK(err);
}
```

For the full source of this example, see section 4.5.22

In Fortran2003 style, the source is a member of the Status type.

```fortran
!! anysource.F90
type(MPI_Status) :: status
allocate(recv_buffer(ntids-1))
do p=0,ntids-2
   call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER, 
                 MPI_ANY_SOURCE,0,comm,status)
   sender = status%MPI_SOURCE
end do
```

For the full source of this example, see section 4.5.23

In Fortran90 style, the source is an index in the Status array.

```fortran
!! anysource.F90
integer :: status(MPI_STATUS_SIZE)
alocate(recv_buffer(ntids-1))
do p=0,ntids-2
   call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER, 
                 MPI_ANY_SOURCE,0,comm,status,err)
   sender = status(MPI_SOURCE)
end do
```

For the full source of this example, see section 4.5.24
4. MPI topic: Point-to-point

*MPL note 33: Status source querying.* The `status` object can be queried:

```c
int source = recv_status.source();
```

### 4.3.2.2 Tag

If a processor is expecting more than one message from a single other processor, message tags are used to distinguish between them. In that case, a value of `MPI_ANY_TAG` can be used, and the actual `tag` of a message can be retrieved as the `MPI_TAG` member in the status structure. See section 4.3.2.1 about `MPI_SOURCE` for how to use this.

*MPL note 34: Message tag.* MPL differs from other APIs in its treatment of tags: a tag is not directly an integer, but an object of class `tag`.

```c
// sendrecv.cxx
mpl::tag t0(0);
comm_world.sendrecv(
  mydata,sendto,t0,
  leftdata,recvfrom,t0 );
```

The `tag` class has a couple of methods such as `mpl::tag::any()` (for the `MPI_ANY_TAG` wildcard in receive calls) and `mpl::tag::up()` (maximal tag, found from the `MPI_TAG_UB` attribute).

### 4.3.2.3 Error

Any errors during the receive operation can be found as the `MPI_ERROR` member of the status structure. This field is only set by functions that return multiple statuses, such as `MPI_Waitall`. For functions that return a single status, any error is returned as the function result. For a function returning multiple statuses, the presence of any error is indicated by a result of `MPI_ERR_IN_STATUS`; section 4.2.2.6.

### 4.3.2.4 Count

If the amount of data received is not known a priori, the `count` of elements received can be found by `MPI_Get_count` (figure 4.17):

```c
// count.c
if (procid==0) {
  int sendcount = (rand()>.5) ? N : N-1;
  MPI_Send( buffer,sendcount,MPI_FLOAT,target,0,comm );
} else if (procid==target) {
  MPI_Status status;
  int recvcount;
  MPI_Recv( buffer,N,MPI_FLOAT,0,0,comm,&status );
  MPI_Get_count(&status,MPI_FLOAT,&recvcount);
  printf("Received %d elements\n",recvcount);
}
```

*For the full source of this example, see section 4.5.25*
4.3. More about point-to-point communication

Figure 4.17 $\text{MPI\_Get\_count}$

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Get_count</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Get_count_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>status</td>
<td></td>
<td>return status of receive operation</td>
<td>const</td>
<td>TYPE(MPI_Status)</td>
<td>IN</td>
</tr>
<tr>
<td>datatype</td>
<td></td>
<td>datatype of each receive buffer entry</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td>count</td>
<td></td>
<td>number of received entries</td>
<td>int*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MPL:

template<typename T>
int mpl::status::get_count () const

template<typename T>
int mpl::status::get_count (const layout<T> &l) const

Python:

status.Get_count( Datatype datatype=BYTE )

Output:

make[3]: `count' is up to date.
TACC: Starting up job 4051425
TACC: Setting up parallel environment for MVAPICH2
TACC: Starting parallel tasks...
One less
Received 9 elements
TACC: Shutdown complete. Exiting.

This may be necessary since the `count' argument to $\text{MPI\_Recv}$ is the buffer size, not an indication of the actually received number of data items.

Remarks.

- Unlike the source and tag, the message count is not directly a member of the status structure.
- The `count' returned is the number of elements of the specified datatype. If this is a derived type (section 6.3) this is not the same as the number of elementary datatype elements. For that, use $\text{MPI\_Get\_elements}$ (figure 4.18) or $\text{MPI\_Get\_elements\_x}$ which returns the number of basic elements.

$\text{MPL\ note 35: Receive count.}$ The $\text{get\_count}$ function is a method of the status object. The argument type is handled through templating:

```plaintext
!! count.F90
if (procid==0) then
  sendcount = N
  call random_number(fraction)
  if (fraction>.5) then
    print *,"One less" ; sendcount = N-1
  end if
  call MPI\_Send( buffer,sendcount,MPI\_REAL,target,0,comm )
else if (procid==target) then
  call MPI\_Recv( buffer,N,MPI\_REAL,0,0,comm,status )
  call MPI\_Get\_count(status,MPI\_FLOAT,recvcount)
  print *,"Received",recvcount,"elements"
end if
```

Victor Eijkhout 153
4. MPI topic: Point-to-point

Figure 4.18 MPI_Get_elements

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Get_elements</td>
<td>status</td>
<td>return status of receive operation</td>
<td>const</td>
<td>TYPE(MPI_Status) IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>datatype</td>
<td>datatype used by receive operation</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>number of received basic elements</td>
<td>int*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
// recvstatus.cxx
double pi=0;
auto s = comm_world.recv(pi, 0); // receive from rank 0
int c = s.get_count<double>();
std::cout << "got : " << c << ", scalar(s): " << pi << '\n';
```

For the full source of this example, see section 4.5.26

4.3.2.5 Example: receiving from any source

Consider an example where the last process receives from every other process. We could implement this as a loop

```c
for (int p=0; p<nprocs-1; p++)
    MPI_Recv(/* from source= */ p);
```

but this may incur idle time if the messages arrive out of order.

Instead, we use the MPI_ANY_SOURCE specifier to give a wildcard behavior to the receive call: using this value for the 'source' value means that we accept messages from any source within the communicator, and messages are only matched by tag value. (Note that size and type of the receive buffer are not used for message matching!)

We then retrieve the actual source from the MPI_Status object through the MPI_SOURCE field.

```c
// anysource.c
if (procno==nprocs-1) {
    /*
    * The last process receives from every other process
    */
    int *recv_buffer;
    recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));

    /*
    * Messages can come in any order, so use MPI_ANY_SOURCE
    */
    MPI_Status status;
```
4.4. Review questions

```c
for (int p=0; p<nprocs-1; p++) {
    err = MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm, &status); CHK(err);
    int sender = status.MPI_SOURCE;
    printf("Message from sender=\%d: \%d\n", sender,recv_buffer[p]);
} free(recv_buffer);
else {
    /* Each rank waits an unpredictable amount of time,
     * then sends to the last process in line.
     */
    float randomfraction = (rand() / (double)RAND_MAX);
    int randomwait = (int) ( nprocs * randomfraction );
    printf("process %d waits for %e/%d=\%d\n", procno,randomfraction,nprocs,randomwait);
    sleep(randomwait);
    err = MPI_Send(&randomwait,1,MPI_INT, nprocs-1,0,comm); CHK(err);
}
```

For the full source of this example, see section 4.5.22
```
## anysource.py
rstatus = MPI.Status()
comm.Recv(rbuf,source=MPI.ANY_SOURCE,status=rstatus)
print("Message came from \%d " % rstatus.Get_source())
```

For the full source of this example, see section 4.5.27

In sections and 4.2.3 we explained the manager-worker model (and in chapter 56 you can do programming project with it). This design patterns offers an opportunity for inspecting the `MPI_SOURCE` field of the `MPI_Status` object describing the data that was received.

4.3.3 Errors

MPI routines return `MPI_SUCCESS` upon succesful completion. The following error codes can be returned (see section 15.2.1 for details) for completion with error by both send and receive operations: `MPI_ERR_COMM`, `MPI_ERR_COUNT`, `MPI_ERR_TYPE`, `MPI_ERR_TAG`, `MPI_ERR_RANK`.

4.3.4 Message envelope

Apart from its bare data, each message has a message envelope. This has enough information to distinguish messages from each other: the source, destination, tag, communicator.

4.4 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

**Review 4.16.** Describe a deadlock scenario involving three processors.
Review 4.17. True or false: a message sent with `MPI_Isend` from one processor can be received with an `MPI_Recv` call on another processor.

Review 4.18. True or false: a message sent with `MPI_Send` from one processor can be received with an `MPI_Irecv` on another processor.

Review 4.19. Why does the `MPI_Irecv` call not have an `MPI_Status` argument?

Review 4.20. Suppose you are testing ping-pong timings. Why is it generally not a good idea to use processes 0 and 1 for the source and target processor? Can you come up with a better guess?


Review 4.22. What are the three routines for one-sided data transfer?

Review 4.23. In the following fragments assume that all buffers have been allocated with sufficient size. For each fragment note whether it deadlocks or not. Discuss performance issues.

```c
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);

int ireq = 0;
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Isend(.sbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Irecv(rbuffers[p],buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);
MPI_Waitall(nprocs-1,requests,MPI_STATUSUSES_IGNORE);
```
```c
int ireq = 0;
for (int p=0; p<nprocs; p++)
    if (p!=procid)
        MPI_Irecv(rbuffers[p], buflen, MPI_INT, p, 0, comm, &(requests[ireq++]));
MPI_Waitall(nprocs-1, requests, MPI_STATUSES_IGNORE);
for (int p=0; p<nprocs; p++)
    if (p!=procid)
        MPI_Send(sbuffer, buflen, MPI_INT, p, 0, comm);
```
Fortran codes:

```fortran
    do p=0,nprocs-1
      if (p/=procid) then
        call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)
      end if
    end do
    do p=0,nprocs-1
      if (p/=procid) then
        call MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE,ierr)
      end if
    end do
    
    ireq = 0
    do p=0,nprocs-1
      if (p/=procid) then
        call MPI_Isend(sbuffers(1,p+1),buflen,MPI_INT,p,0,comm,&
                        requests(ireq+1),ierr)
        ireq = ireq+1
      end if
    end do
    do p=0,nprocs-1
      if (p/=procid) then
        call MPI_Recv(rbuffers(1,p+1),buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE,ierr)
      end if
    end do
    call MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE,ierr)
```

4. MPI topic: Point-to-point
4.4. Review questions

```fortran
// block5.F90
ireq = 0
do p=0,nprocs-1
  if (p/=procid) then
    call MPI_Irecv(rbuffers(1,p+1),buflen,MPI_INT,p,0,comm,&
    requests(ireq+1),ierr)
    ireq = ireq+1
  end if
end do
end do
call MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE,ierr)
```

Review 4.24. Consider a ring-wise communication where

```fortran
int
  next = (mytid+1) % ntids,
  prev = (mytid+ntids-1) % ntids;
```

and each process sends to next, and receives from prev.

The normal solution for preventing deadlock is to use both MPI_Isend and MPI_Irecv.
The send and receive complete at the wait call. But does it matter in what sequence
you do the wait calls?

```fortran
// ring3.c
MPI_Request req1,req2;
MPI_Irecv(&y,1,MPI_DOUBLE,prev,0,comm,&req1);
MPI_Isend(&x,1,MPI_DOUBLE,next,0,comm,&req2);
MPI_Wait(&req1,MPI_STATUS_IGNORE);
MPI_Wait(&req2,MPI_STATUS_IGNORE);
```

Can we have one nonblocking and one blocking call? Do these scenarios block?

```fortran
// ring1.c
MPI_Request req;
MPI_Isend(&x,1,MPI_DOUBLE,next,0,comm,&req);
MPI_Recv(&y,1,MPI_DOUBLE,prev,0,comm,
         MPI_STATUS_IGNORE);
MPI_Wait(&req,MPI_STATUS_IGNORE);
```

```fortran
// ring2.c
MPI_Request req;
MPI_Irecv(&y,1,MPI_DOUBLE,prev,0,comm,&req);
MPI_Ssend(&x,1,MPI_DOUBLE,next,0,comm);
MPI_Wait(&req,MPI_STATUS_IGNORE);
```
4. MPI topic: Point-to-point

4.5 Sources used in this chapter

4.5.1 Listing of code header

4.5.2 Listing of code examples/mpi/c/sendandrecv.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

    MPI_Comm comm = MPI_COMM_WORLD;
    int nprocs, procno;

    MPI_Init(&argc,&argv);

    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procno);

    /*
    * We set up a single communication between
    * the first and last process
    */
    int sender,receiver;
    sender = 0; receiver = nprocs-1;

    if (procno==sender) {
        double send_data = 1.;
        MPI_Send
        ( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
        /* to: */ receiver, /* tag: */ 0,
        /* communicator: */ comm);
        printf("[%d] Send successfully concluded\n",procno);
    } else if (procno==receiver) {
        double recv_data;
        MPI_Recv
        ( /* recv buffer/count/type: */ &recv_data,1,MPI_DOUBLE,
        /* from: */ sender, /* tag: */ 0,
        /* communicator: */ comm,
        /* recv status: */ MPI_STATUS_IGNORE);
        printf("[%d] Receive successfully concluded\n",procno);
    }

    MPI_Finalize();
    return 0;
}
```

4.5.3 Listing of code examples/mpi/mpl/sendscalar.cxx

```c
#include <cstdlib>
```

160 Parallel Computing – r428
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <mpl/mpl.hpp>

int main() {

    const mpl::communicator &comm_world = mpl::environment::comm_world();
    if (comm_world.size() < 2)
        return EXIT_FAILURE;

    // send and receive a single floating point number
    if (comm_world.rank() == 0) {
        double pi = 3.14;
        comm_world.send(pi, 1); // send to rank 1
        cout << "sent: " << pi << '\n';
    } else if (comm_world.rank() == 1) {
        double pi = 0;
        comm_world.recv(pi, 0); // receive from rank 0
        cout << "got : " << pi << '\n';
    }

    return EXIT_SUCCESS;
}

4.5.4 Listing of code examples/mpi/mpl/sendarray.cxx

#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <sstream>
using std::stringstream;
#include <mpl/mpl.hpp>

int main() {

    const mpl::communicator &comm_world = mpl::environment::comm_world();
    if (comm_world.size() < 2)
        return EXIT_FAILURE;

    /*
    * The compiler knows about arrays so we can send them `as is'
    */
    double v[2][2][2];

    // Initialize the data
    if (comm_world.rank() == 0) {
        double *vt = &v[0][0][0];
        for (int i = 0; i < 8; i++)
            *vt++ = i;
    }
4. MPI topic: Point-to-point

        /*
         * Send and report
         */
        comm_world.send(v, 1); // send to rank 1

        stringstream s;
        s << "sent: ";
        vt = &(v[0][0][0]);
        for (int i=0; i<8; i++)
            s << " " << *(vt+i);
        cout << s.str() << '\n';

        // std::cout << "sent: ";
        // for (double &x : v)
        // std::cout << x << ' ';
        // std::cout << '\n';

    } else if (comm_world.rank()==1) {

        /*
         * Receive data and report
         */
        comm_world.recv(v, 0); // receive from rank 0

        stringstream s;
        s << "got : ";
        double *vt = &(v[0][0][0]);
        for (int i=0; i<8; i++)
            s << " " << *(vt+i);
        cout << s.str() << '\n';

    }
    return EXIT_SUCCESS;
}

4.5.5  Listing of code examples/mpi/mlp/sendbuffer.cxx

#include <cstdlib>
#include <iostream>
#include <vector>
#include <mpl/mlp.hpp>

int main() {
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    if (comm_world.size()<2)
        return EXIT_FAILURE;

        /*
         * To send a std::vector we declare a contiguous layout
         */
        std::vector<double> v(8);
        mpl::contiguous_layout<double> v_layout(v.size());
4.5. Sources used in this chapter

---

4.5.6 Listing of code examples/mpi/mpl/sendrange.cxx

```c++
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <vector>
using std::vector;
#include <mpl/mpl.hpp>

int main() {
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    if (comm_world.size()<2)
        return EXIT_FAILURE;

    vector<double> v(15);
    // Initialize the data
    if (comm_world.rank()==0) {
        double init=0;
        for (double &x : v) {
            x=init;
            ++init;
        }
        /*
         * Send and report
         */
        comm_world.send(v.data(), v_layout, 1); // send to rank 1
        std::cout << "sent: ";
        for (double &x : v)
            std::cout << x << ' ';
        std::cout << '
';
    } else if (comm_world.rank()==1) {
        /*
         * Receive data and report
         */
        comm_world.recv(v.data(), v_layout, 0); // receive from rank 0
        std::cout << "got : ";
        for (double &x : v)
            std::cout << x << ' ';
        std::cout << '
';
        return EXIT_SUCCESS;
    }
}
```
if (comm_world.rank()==0) {
    // initialize
    for ( auto &x : v ) x = 1.41;

    /*
     * Send and report
     */
    comm_world.send(v.begin(), v.end(), 1); // send to rank 1
}
else if (comm_world.rank()==1) {
    /*
     * Receive data and report
     */
    comm_world.recv(v.begin(), v.end(), 0); // receive from rank 0
    cout << "Got:";
    for ( auto x : v )
        cout << " " << x;
    cout << endl;
}
return EXIT_SUCCESS;

4.5.7 Listing of code examples/mpi/c/sendblock.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

/** This program shows deadlocking behaviour
    when two processes exchange data with blocking
    send and receive calls.
    Under a certain limit MPI_Send may actually not be
    blocking; we loop, increasing the message size,
    to find roughly the crossover point.
*/
int main(int argc,char **argv) {
    int *recvbuf, *sendbuf;
    MPI_Status status;

    #include "globalinit.c"

    /* we only use processors 0 and 1 */
    int other;
    if (procno>1) goto skip;
    other = 1-procno;
    /* loop over increasingly large messages */
    for (int size=1; size<2000000000; size*=10) {

4.5. Sources used in this chapter

sendbuf = (int*) malloc(size*sizeof(int));
recvbuf = (int*) malloc(size*sizeof(int));
if (!sendbuf || !recvbuf) {
    printf("Out of memory\n"); MPI_Abort(comm,1);
}
MPI_Send(sendbuf,size,MPI_INT,other,0,comm);
MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
/* If control reaches this point, the send call
did not block. If the send call blocks,
we do not reach this point, and the program will hang.
*/
if (procno==0)
    printf("Send did not block for size %d\n",size);
free(sendbuf); free(recvbuf);
}
skip:
MPI_Finalize();
return 0;
}

4.5.8 Listing of code examples/mpi/f/sendblock.F90

Program SendBlock

implicit none
#include "mpif.h"

integer :: other,size,status(MPI_STATUS_SIZE)
integer,dimension(:),allocatable :: sendbuf,recvbuf
#include "globalinit.F90"
if (mytid>1) goto 10
other = 1-mytid
size = 1
do
   allocate(sendbuf(size)); allocate(recvbuf(size))
   print *,size
   call MPI_Send(sendbuf,size,MPI_INTEGER,other,0,comm,err)
   call MPI_Recv(recvbuf,size,MPI_INTEGER,other,0,comm,status,err)
   if (mytid==0) then
       print *,"MPI_Send did not block for size",size
   end if
   deallocate(sendbuf); deallocate(recvbuf)
   size = size*10
   if (size>2000000000) goto 20
end do
10 call MPI_Finalize(err)
end program SendBlock

Victor Eijkhout
4. MPI topic: Point-to-point

4.5.9  Listing of code examples/mpi/p/sendblock.py

```python
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

if procid in [0,nprocs-1]:
    other = nprocs-1-procid
    size = 1
    while size<2000000000:
        sendbuf = np.empty(size, dtype=int)
        recvbuf = np.empty(size, dtype=int)
        comm.Send(sendbuf,dest=other)
        comm.Recv(sendbuf,source=other)
        if procid<other:
            print("Send did not block for",size)
        size *= 10
```

4.5.10  Listing of code examples/mpi/c/ssendblock.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
    int other, size, *recvbuf, *sendbuf;
    MPI_Status status;

    #include "globalinit.c"
    if (procno>1) goto skip;
    other = 1-procno;
    sendbuf = (int*) malloc(sizeof(int));
    recvbuf = (int*) malloc(sizeof(int));
    size = 1;
    MPI_Ssend(sendbuf,size,MPI_INT,other,0,comm);
    MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
    printf("This statement is not reached\n");
    free(sendbuf); free(recvbuf);

skip:    
    MPI_Finalize();
    return 0;
}
```
4.5. Sources used in this chapter

4.5.11 Listing of code examples/mpi/c/isendandirecv.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
    #include "globalinit.c"

    /*
     * We set up a single communication between
     * the first and last process
     */
    int sender,receiver;
    sender = 0; receiver = nprocs-1;

    if (procno==sender) {
        double send_data = 1.;
        MPI_Request request;
        MPI_Isend
            ( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
              /* to: */ receiver, /* tag: */ 0,
              /* communicator: */ comm,
              /* request: */ &request);
        MPI_Wait(&request,MPI_STATUS_IGNORE);
        printf("[%d] Isend successfully concluded\n",procno);
    } else if (procno==receiver) {
        double recv_data;
        MPI_Request request;
        MPI_Irecv
            ( /* recv buffer/count/type: */ &recv_data,1,MPI_DOUBLE,
              /* from: */ sender, /* tag: */ 0,
              /* communicator: */ comm,
              /* request: */ &request);
        MPI_Wait(&request,MPI_STATUS_IGNORE);
        printf("[%d] Ireceive successfully concluded\n",procno);
    }
    MPI_Finalize();
    return 0;
}
```

4.5.12 Listing of code examples/mpi/c/hangwait.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

Victor Eijkhout

#include "mpi.h"
```
int main(int argc, char **argv) {
    include "globalinit.c"

    double
    mydata=procno;
    int sender = nprocs-1;

    if (procno==sender) {
        for (int p=0; p<nprocs-1; p++) {
            double send = 1.;
            MPI_Send( &send,1,MPI_DOUBLE,p,0,comm);
        }
    } else {
        double recv=0.;
        MPI_Request request;
        MPI_Irecv( &recv,1,MPI_DOUBLE,sender,0,comm,&request);
        MPI_Wait(&request,MPI_STATUS_IGNORE);
    }

    MPI_Finalize();
    return 0;
}

4.5.13 Listing of code examples/mpi/mlp/isendandirecv.cxx
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <mpl/mlp.hpp>

int main(int argc, char **argv) {
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    int
    nprocs = comm_world.size(),
    procono = comm_world.rank();
    if (comm_world.size()<2)
        return EXIT_FAILURE;

    int sender = 0, receiver = nprocs-1;

    if (procno==sender) {
        double send_data = 1.;
        mpl::irequest send_request
            ( comm_world.isend( send_data, receiver ) );
        send_request.wait();
        printf("[%d] Isend successfully concluded\n",procno);
    } else if (procno==receiver) {
        double recv_data;
        mpl::irequest recv_request =
4.5. Sources used in this chapter

4.5.14 Listing of code examples/mpi/p/irecvloop.py

```python
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

requests = [] # [ None ] * (2*nprocs)
sendbuffer = np.empty( nprocs, dtype=int )
recvbuffer = np.empty( nprocs, dtype=int )

for p in range(nprocs):
    left_p = (p-1) % nprocs
    right_p = (p+1) % nprocs
    requests.append( comm.Isend( sendbuffer[p:p+1],dest=left_p ) )
    requests.append( comm.Irecv( sendbuffer[p:p+1],source=right_p ) )
MPI.Request.Waitall(requests)

if procid==0:
    print("All messages received")
```

4.5.15 Listing of code examples/mpi/f/irecvsource.F90

```fortran
Program Irecv_source

use mpi
implicit none

integer,dimension(:), allocatable :: recv_buffer,requests
integer :: index,randomint
real :: randomvalue
#include "globalinit.F90"
allocate(recv_buffer(ntids-1))
allocate(requests(ntids-1))
```

Victor Eijkhout
if (mytid==ntids-1) then
   do p=1,ntids-1
      print *, "post"
      call MPI_Irecv(recv_buffer(p),1,MPI_INTEGER,p-1,0,comm,&
         requests(p),err)
   end do
   do p=1,ntids-1
      call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE,err)
      write(*,'("Message from",i3,:",",i5)') index,recv_buffer(index)
   end do
else
   call sleep(6)
   call random_number(randomvalue)
   randomint = randomvalue
   randomint = 30+mytid
   call MPI_Send(randomint,1,MPI_INTEGER, ntids-1,0,comm,err)
end if
end Program Irecv_source

4.5.16 Listing of code examples/mpi/f08/waitnull.F90

Program Waitnull

use mpi_f08

implicit none

!!
!! General stuff
!!
Type(MPI_Comm) :: comm;
integer :: mytid,ntids,i,p,err;

!!
!! random number generatoe
!!
integer :: randsize
integer,allocatable,dimension(:) :: randseed

!!
!! data for this program
!!
Type(MPI_Request),dimension(:),allocatable :: requests
integer,dimension(:),allocatable :: recv_buffer
integer :: index,randomint,success = 1
real :: randomvalue

call MPI_Init()
comm = MPI_COMM_WORLD
call MPI_Comm_rank(comm,mytid)
call MPI_Comm_size(comm,ntids)
call MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN)

!!
!! seed the random number generator
!!
call random_seed(size=randsize)
allocate(randseed(randsize))
do i=1[randsize]
   randseed(i) = 1023*mytid
end do

call random_seed(put=randseed)

allocate(recv_buffer(ntids-1))
allocate(requests(ntids-1))

if (mytid==ntids-1) then
   
   ! the last process posts a receive 
   ! from every other process
   
do p=0,ntids-2
      call MPI_Irecv(recv_buffer(p+1),1,MPI_INTEGER,p,0,comm,&
                     requests(p+1))
   end do

   ! then wait to see what comes in
   
do p=0,ntids-2
      call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE)
      if ( .not. requests(index)==MPI_REQUEST_NULL) then
         print *,"This request should be null:",index
         success = 0
      end if
      !write(*,'("Message from",i3,":",i5)') index,recv_buffer(index)
   end do
else
   
   ! everyone else sends one number to the last 
   ! after some random wait
   
call sleep(6)
call random_number(randomvalue)
   randomint = randomvalue = 30+mytid
   call MPI_Send(randomint,1,MPI_INTEGER, ntids-1,0,comm)
end if

call MPI_Allreduce(MPI_IN_PLACE,success,1,MPI_INTEGER,MPI_SUM,comm)
if (mytid==0) then
   if (success==ntids) then
      print *,"All processes successfully concluded"
   else

Victor Eijkhout

171
4. MPI topic: Point-to-point

        print *,"Failure on",ntids-success,"processes"
        end if
    end if

    call MPI_Finalize()

end Program Waitnull

4.5.17 Listing of code examples/mpi/mpl/irecvsource.cxx

#include <cstdlib>
#include <unistd.h>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <vector>
using std::vector;
#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

    const mpl::communicator &comm_world=mpl::environment::comm_world();
    int
        nprocs = comm_world.size(),
        procno = comm_world.rank();
    if (comm_world.size()<2)
        return EXIT_FAILURE;
    // Initialize the random number generator
    srand((int)(procno*(double)RAND_MAX/nprocs));

    // Initialize the random number generator
    srand((int)(procno*(double)RAND_MAX/nprocs));

    if (procno==nprocs-1) {
        mpl::irequest_pool recv_requests;
        vector<int> recv_buffer(nprocs-1);
        for (int p=0; p<nprocs-1; p++) {
            recv_requests.push( comm_world.irecv( recv_buffer[p], p ) );
        }
        printf("Outstanding request #=%lu\n",recv_requests.size());
        for (int p=0; p<nprocs-1; p++) {
            auto [success,index] = recv_requests.waitany();
            if (success) {
                auto recv_status = recv_requests.get_status(index);
                int source = recv_status.source();
                if (index!=source)
                    printf("Mismatch index %lu vs source %d\n",index,source);
                printf("Message from %lu: %d\n",index,recv_buffer[index]);
            } else
                break;
        }
    } else {

}
4.5. Sources used in this chapter

```c
float randomfraction = (rand() / (double)RAND_MAX);
int randomwait = (int) ( 2* nprocs * randomfraction );
printf("process %d waits for %d\n",procno,randomwait);
sleep(randomwait);
comm_world.send( procno, nprocs-1 );
}
return 0;
```

4.5.18 Listing of code examples/mpi/p/irecvsource.py

```python
import numpy as np
import random
import time
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()

if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

if procid==nprocs-1:
    receive_buffer = np.empty(nprocs-1,dtype=int)
    requests = [ None ] * (nprocs-1)
    for sender in range(nprocs-1):
        requests[sender] = comm.Irecv(receive_buffer[sender:sender+1],source=sender)
        # alternatively: requests = [ comm.Irecv(s) for s in .... ]
    status = MPI.Status()
    for sender in range(nprocs-1):
        ind = MPI.Request.Waitany(requests,status=status)
        if ind!=status.Get_source():
            print("sender mismatch: %d vs %d" % (ind,status.Get_source()))
            print("received from",ind)
else:
    mywait = random.randint(1,2*nprocs)
    print("[%d] wait for %d seconds" % (procid,mywait))
    time.sleep(mywait)
    mydata = np.empty(1,dtype=int)
    mydata[0] = procid
    comm.Send([mydata,MPI.INT],dest=nprocs-1)
```

4.5.19 Listing of code examples/mpi/c/irecvloop.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
```
4. MPI topic: Point-to-point

#include "globalinit.c"

int recvbuf=2, sendbuf=3, other;
MPI_Request requests =
  (MPI_Request*) malloc( 2*nprocs*sizeof(MPI_Request) );
recv_buffers = (int*) malloc( nprocs*sizeof(int) );
send_buffers = (int*) malloc( nprocs*sizeof(int) );
for (int p=0; p<nprocs; p++) {
  int
    left_p = (p-1+nprocs) % nprocs,
    right_p = (p+1) % nprocs;
  send_buffer[p] = nprocs-p;
  MPI_Isend(sendbuffer+p,1,MPI_INT, right_p,0, requests+2*p);
  MPI_Irecv(recvbuffer+p,1,MPI_INT, left_p,0, requests+2*p+1);
}

/* your useful code here */

MPI_Waitall(2*nprocs,requests,MPI_STATUSES_IGNORE);

skip:
  MPI_Finalize();
  return 0;
}

4.5.20 Listing of code examples/mpi/c/probe.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

int main(int argc,char **argv) {

  #include "globalinit.c"

  // Initialize the random number generator
  srand((int)(procno*(double)RAND_MAX/nprocs));

  int sender = 0, receiver = nprocs-1;
  if (procno==receiver) {
    MPI_Status status;
    MPI_Probe(sender,0,comm,&status);
    int count;
    MPI_Get_count(&status,MPI_FLOAT,&count);
    printf("Receiving %d floats\n",count);
    float recv_buffer[count];
    MPI_Recv(recv_buffer,count,MPI_FLOAT, sender,0,comm,MPI_STATUS_IGNORE);
  } else if (procno==sender) {
    float randomfraction = (rand() / (double)RAND_MAX);
    int buffer_size = (int) ( 10 * nprocs * randomfraction );
    printf("Sending %d floats\n",buffer_size);
    float buffer[buffer_size];
4.5. Sources used in this chapter

4.5.21 Listing of code examples/mpi/mpl/vector.cxx

```cpp
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <vector>
using std::vector;
#include <cassert>
#include <mpl/mpl.hpp>

int main(int argc, char **argv) {

    const mpl::communicator &comm_world = mpl::environment::comm_world();
    int nprocs, procno;
    // compute communicator rank and size
    nprocs = comm_world.size();
    procno = comm_world.rank();

    if (nprocs<2) {
        printf("This program needs at least two processes\n");
        return -1;
    }

    int sender = 0, receiver = 1, the_other = 1-procno,
        count = 5, stride=2;
    vector<double> source(stride*count);
    vector<double> target(count);
    for (int i=0; i<stride*count; i++)
        source[i] = i+.5;

    if (procno==sender) {
        mpl::strided_vector_layout<double>
            newvectortype(count,1,stride);
        comm_world.send
            (source.data(),newvectortype,the_other);
    }
    else if (procno==receiver) {
        int recv_count;
        mpl::contiguous_layout<double> target_layout(count);
        mpl::status_t recv_status =
            comm_world.recv(target.data(),target_layout, the_other);
        recv_count = recv_status.get_count<double>();
        assert(recv_count==count);
    }

Victor Eijkhout
175
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

int main(int argc, char **argv) {

    if (procno==nprocs-1) {
        /*
        * The last process receives from every other process
        */
        int *recv_buffer;
        recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));

        /*
        * Messages can come in in any order, so use MPI_ANY_SOURCE
        */
        MPI_Status status;
        for (int p=0; p<nprocs-1; p++) {
            err = MPI_Recv(recv_buffer+p, 1, MPI_INT, MPI_ANY_SOURCE, 0, comm, &status); CHK(err);
            int sender = status.MPI_SOURCE;
            printf("Message from sender=%d: %d\n", sender, recv_buffer[p]);
        }
        free(recv_buffer);
    } else {
        /*
        * Each rank waits an unpredictable amount of time,
        * then sends to the last process in line.
        */
        float randomfraction = (rand() / (double)RAND_MAX);
        int randomwait = (int) (nprocs * randomfraction);
        printf("process %d waits for %e/\%d\n",
```
procno, randomfraction, nprocs, randomwait);
sleep(randomwait);
err = MPI_Send(&randomwait, 1, MPI_INT, nprocs-1, 0, comm); CHK(err);
}

MPI_Finalize();
return 0;

}  

4.5.23 Listing of code examples/mpi/f08/anysource.F90

Program AnySource
	none

integer, dimension(:), allocatable :: recv_buffer
Type(MPI_Status) :: status
real :: randomvalue
integer :: randomint, sender

#include "globalinit.F90"

if (mytid.eq.ntids-1) then
allocate(recv_buffer(ntids-1))
do p=0,ntids-2
   call MPI_Recv(recv_buffer(p+1), 1, MPI_INTEGER, &
                  MPI_ANY_SOURCE, 0, comm, status)
   sender = status%MPI_SOURCE
   print *, "Message from", sender
end do
else
   call random_number(randomvalue)
   randomint = randomvalue*ntids
   call sleep(randomint)
   print *, mytid, "waits for", randomint
   call MPI_Send(p, 1, MPI_INTEGER, ntids-1, 0, comm)
end if

call MPI_Finalize(err)

end program AnySource

4.5.24 Listing of code examples/mpi/f/anysource.F90

Program AnySource

implicit none

#include "mpif.h"
4. MPI topic: Point-to-point

```fortran
integer,dimension(:),allocatable :: recv_buffer
integer :: status(MPI_STATUS_SIZE)
real :: randomvalue
integer :: randomint,sender

#include "globalinit.F90"

if (mytid.eq.ntids-1) then
   allocate(recv_buffer(ntids-1))
   do p=0,ntids-2
      call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER,&
                        MPI_ANY_SOURCE,0,comm,status,err)
      sender = status(MPI_SOURCE)
      print *,"Message from",sender
   end do
else
   call random_number(randomvalue)
   randomint = randomvalue*ntids
   call sleep(randomint)
   print *,mytid,"waits for",randomint
   call MPI_Send(p,1,MPI_INTEGER,ntids-1,0,comm,err)
end if

call MPI_Finalize(err)
end program AnySource
```

4.5.25 Listing of code examples/mpi/c/count.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc,char **argv) {
    MPI_Comm comm;
    int nprocs,procid;

    MPI_Init(&argc,&argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procid);

    #define N 10
    float buffer[N];
    int target = nprocs-1;
    if (procid==0) {
        int sendcount = (rand()>.5) ? N : N-1;
        MPI_Send( buffer,sendcount,MPI_FLOAT,target,0,comm );
    } else if (procid==target) {
        MPI_Status status;
        int recvcount;
        MPI_Recv( buffer,N,MPI_FLOAT,0,0, comm, &status );
    }

    MPI_Finalize(err);
}
```
4.5. Sources used in this chapter

MPI_Get_count(&status, MPI_FLOAT, &recvcount);
printf("Received %d elements\n", recvcount);
}

MPI_Finalize();
return 0;
}

4.5.26 Listing of code examples/mpi/ml/recvstatus.cxx
#include <cstdlib>
#include <complex>
#include <iostream>
#include <mpl/mpl.hpp>

int main() {
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    if (comm_world.size()<2)
        return EXIT_FAILURE;
    // send and recieve a single floating point number
    if (comm_world.rank()==0) {
        double pi=3.14;
        comm_world.send(pi, 1); // send to rank 1
        std::cout << "sent: " << pi << 'n';
    } else if (comm_world.rank()==1) {
        double pi=0;
        auto s = comm_world.recv(pi, 0); // receive from rank 0
        int c = s.get_count<double>();
        std::cout << "got : " << c << " scalar(s): " << pi << 'n';
    }
    return EXIT_SUCCESS;
}

4.5.27 Listing of code examples/mpi/p/anysource.py
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
sys.exit(1)

if procid==nprocs-1:
    rbuf = np.empty(1,dtype=np.float64)
for p in range(procid):
    rstatus = MPI.Status()
    comm.Recv(rbuf,source=MPI.ANY_SOURCE,status=rstatus)
4. MPI topic: Point-to-point

```python
print("Message came from \%d" % rstatus.Get_source())
else:
sbuf = np.empty(1,dtype=np.float64)
sbuf = np.empty(1,dtype=np.float64)
sbuf[0] = 1.
comm.Send(sbuf,dest=procs-1)
```
Chapter 5

MPI topic: Communication modes

5.1 Persistent communication

Persistent communication is a mechanism for dealing with a repeating communication transaction, where the parameters of the transaction, such as sender, receiver, tag, root, and buffer type and size, stay the same. Only the contents of the buffers involved changes between the transactions.

You can imagine that setting up a communication carries some overhead, and if the same communication structure is repeated many times, this overhead may be avoided.

1. For nonblocking communications MPI_Ixxx (both point-to-point and collective) there is a persistent variant MPI_Xxx_init with the same calling sequence. The ‘init’ call produces an MPI_Request output parameter, which can be used to test for completion of the communication.

2. The ‘init’ routine does not start the actual communication: that is done in MPI_Start, or MPI_Startall for multiple requests.

3. Any of the MPI ‘wait’ calls can then be used to conclude the communication.

4. The communication can then be restarted with another ‘start’ call.

5. The wait call does not release the request object, since it can be used for repeat occurrences of this transaction. The request object is only freed with MPI_Request_free.

```
MPI_Send_init( /* ... */ &request);
while ( /* ... */ ) {
    MPI_Start( request );
    MPI_Wait( request, &status );
}
MPI_Request_free( & request );
```

**MPL note 36: Persistent requests.** MPL returns a prequest from persistent ‘init’ routines, rather than an irequest (MPL note 29):

```
template<typename T >
prequest send_init ( const T &data, int dest, tag t=tag(0)) const;
```

Likewise, there is a prequest_pool instead of an irequest_pool (note 30).
5. MPI topic: Communication modes

Figure 5.1 MPI_Send_init

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send_init</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Send_init_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>buf</td>
<td>initial address of send buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>number of elements sent</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>datatype</td>
<td>type of each element</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>dest</td>
<td>rank of destination</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>tag</td>
<td>message tag</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>request</td>
<td>communication request</td>
<td>MPI_Request*</td>
<td>TYPE(MPI_Request)</td>
<td>OUT</td>
<td></td>
</tr>
</tbody>
</table>

Python:

```
MPI.Comm.Send_init(self, buf, int dest, int tag=0)
```

Figure 5.2 MPI_Startall

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Startall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>list length</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>array_of_requests</td>
<td>array of requests</td>
<td>MPI_Request[]</td>
<td>TYPE(MPI_Request)(count)</td>
<td></td>
</tr>
</tbody>
</table>

Python:

```
MPI.Prequest.Startall(type cls, requests)
```

5.1.1 Persistent point-to-point communication

The main persistent point-to-point routines are MPI_Send_init (figure 5.1), which has the same calling sequence as MPI_Isend, and MPI_Recv_init, which has the same calling sequence as MPI_Irecv.

In the following example a ping-pong is implemented with persistent communication. Since we use persistent operations for both send and receive on the ‘ping’ process, we use MPI_Startall (figure 5.2) to start both at the same time, and MPI_Waitall to test their completion. (There is MPI_Start for starting a single persistent transfer.)
5.1. Persistent communication

Code:
```c
// persist.c
if (procno==src) {
    MPI_Send_init
    ⇐(send,s,MPI_DOUBLE,tgt,0,comm,requests+0);
    MPI_Recv_init
    ⇐(recv,s,MPI_DOUBLE,tgt,0,comm,requests+1);
    for (int n=0; n<NEXPERIMENTS; n++) {
        fill_buffer(send,s,n);
        MPI_Startall(2,requests);
        ⇐MPI_Waitall(2,requests,MPI_STATUSES_IGNORE);
        int r = check_buffer(send,s,n);
        if (!r) printf("buffer problem %d\n",s);
    }
    MPI_Request_free(requests+0);
    MPI_Request_free(requests+1);
} else if (procno==tgt) {
    for (int n=0; n<NEXPERIMENTS; n++) {
        MPI_Recv(recv,s,MPI_DOUBLE,src,0,
                 comm,MPI_STATUS_IGNORE);
        MPI_Send(recv,s,MPI_DOUBLE,src,0,
                 comm);
    }
}
```

```python
# persist.py
requests = [ None ] * 2
sendbuf = np.ones(size,dtype=int)
recvbuf = np.ones(size,dtype=int)
if procid==src:
    print("Size:",size)
    times[isize] = MPI.Wtime()
for n in range(nexperiments):
    requests[0] = comm.Isend(sendbuf[0:size],dest=tgt)
    requests[1] = comm.Irecv(recvbuf[0:size],source=tgt)
MPI.Request.Waitall(requests)
sendbuf[0] = sendbuf[0]+1
    times[isize] = MPI.Wtime()-times[isize]
elif procid==tgt:
    for n in range(nexperiments):
        comm.Recv(recvbuf[0:size],source=src)
        comm.Send(recvbuf[0:size],dest=src)
```

Output:
```
make[2]: `persist' is up to date.
TACC: Starting up job 3954679
TACC: Starting parallel tasks...
Pingpong size=1: t=1.9422e-04
Pingpong size=10: t=3.0837e-06
Pingpong size=100: t=5.1647e-06
Pingpong size=1000: t=1.2052e-05
Pingpong size=10000: t=1.1727e-04
Pingpong size=100000: t=3.7169e-04
Persistent size=1: t=3.8853e-06
Persistent size=10: t=2.9610e-06
Persistent size=100: t=3.8563e-06
Persistent size=1000: t=1.0976e-05
Persistent size=10000: t=6.3316e-05
Persistent size=100000: t=4.1875e-04
TACC: Shutdown complete. Exiting.
```

For the full source of this example, see section 5.6.2

As with ordinary send commands, there are persistent variants of the other send modes:

- **MPI_Bsend_init** for buffered communication, section 5.5;
- **MPI_Ssend_init** for synchronous communication, section 5.3.1;
- **MPI_Rsend_init** for ready sends, section 15.8.
5. MPI topic: Communication modes

Figure 5.3 MPI_Allreduce_init

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allreduce_init</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Allreduce_init_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendbuf</td>
<td>starting address of send buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(())</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>recvbuf</td>
<td>starting address of receive buffer</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(())</td>
<td>OUT</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>number of elements in send buffer</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>datatype</td>
<td>datatype of elements in send buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>op</td>
<td>operation</td>
<td>MPI_Op</td>
<td>TYPE(MPI_Op)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>info</td>
<td>info argument</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>request</td>
<td>communication request</td>
<td>MPI_Request*</td>
<td>TYPE(MPI_Request)</td>
<td>OUT</td>
<td></td>
</tr>
</tbody>
</table>

5.1.2 Persistent collectives

The following material is for the recently released MPI-4 standard and may not be supported yet.

For each collective call, there is a persistent variant. As with persistent point-to-point calls (section 5.1.1), these have largely the same calling sequence as the nonpersistent variants, except for:

- an MPI_Info parameter that can be used to pass system-dependent hints; and
- an added final MPI_Request parameter.

(See for instance MPI_Allreduce_init (figure 5.3).) This request (or an array of requests from multiple calls) can then be used by MPI_Start (or MPI_Startall) to initiate the actual communication.

```c
// powerpersist.c
double localnorm, globalnorm = 1.;
MPI_Request reduce_request;
MPI_Allreduce_init
  ( &localnorm, &globalnorm, 1, MPI_DOUBLE, MPI_SUM, 
    comm, MPI_INFO_NULL, &reduce_request);
for (int it=0; it<10; it++) {
    matmult(indata, outdata, buffersize);
    localnorm = localsum(outdata, buffersize);
    MPI_Start( &reduce_request );
    MPI.Wait( &reduce_request, MPI_STATUS_IGNORE );
    scale(outdata, indata, buffersize, 1./sqrt(globalnorm));
}
MPI_Request_free( &reduce_request );
```

Some points.

- Metadata arrays, such as of counts and datatypes, must not be altered until the MPI_Request_free call.
- The initialization call is nonlocal, so it can block until all processes have performed it.
5.2. Partitioned communication

- Multiple persistent collective can be initialized, in which case they satisfy the same restrictions as ordinary collectives, in particular on ordering. Thus, the following code is incorrect:

```c
// WRONG
if (procid==0) {
    MPI_Reduce_init( /* ... */ &req1);
    MPI_Bcast_init( /* ... */ &req2);
} else {
    MPI_Bcast_init( /* ... */ &req2);
    MPI_Reduce_init( /* ... */ &req1);
}
```

However, after initialization the start calls can be in arbitrary order, and in different order among the processes.

Available persistent collectives are:

- `MPI_Barrier_init`
- `MPI_Bcast_init`
- `MPI_Reduce_init`
- `MPI_Allreduce_init`
- `MPI_Reduce_scatter_init`
- `MPI_Reduce_scatter_block_init`
- `MPI_Gather_init`
- `MPI_Gatherv_init`
- `MPI_Allgatherv_init`

End of MPI-4 material

5.1.3 Persistent neighbor communications

The following material is for the recently released MPI-4 standard and may not be supported yet.

There are persistent version of the neighborhood collectives; section 11.2.2.

- `MPI_Neighbor_allgather_init`
- `MPI_Neighbor_allgatherv_init`
- `MPI_Neighbor_alltoall_init`
- `MPI_Neighbor_alltoallv_init`
- `MPI_Neighbor_alltoallw_init`

End of MPI-4 material

5.2 Partitioned communication

The following material is for the recently released MPI-4 standard and may not be supported yet.

Partitioned communication is a variant on persistent communication, where a message is constructed in partitions.

- The normal `MPI_Send_init` is replaced by `MPI_Psend_init` (figure 5.4). Note the presence of an `MPI_Info` argument, as in persistent collectives, but unlike in persistent sends and receives.
- After this, the `MPI_Start` does not actually start the transfer; instead:
  - Each partition of the message is separately declared as read-to-be-sent with `MPI_Pready`.
  - An `MPI_Wait` call completes the operation, indicating that all partitions have been sent.

A common scenario for this is in multi-threaded environments, where each thread can construct its own part of a message. Having partitioned messages means that partially constructed message buffers can be sent off without having to wait for all threads to finish.

Indicating that parts of a message are ready for sending is done by one of the following calls:

- `MPI_Pready` (figure 5.5) for a single partition;
- `MPI_Pready_range` for a range of partitions; and
Figure 5.4 MPI_Psend_init

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MPI_Psend_init (</td>
<td>buf</td>
<td>initial address of send buffer</td>
<td>const void*, TYPE(*), DIMENSION(..)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>partitions</td>
<td>number of partitions</td>
<td>int INTEGER</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>number of elements sent per</td>
<td>MPI_Count INTEGER(KIND=MPI_COUNT_KIND)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>datatype</td>
<td>type of each element</td>
<td>MPI_Datatype TYPE(MPI_Datatype)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dest</td>
<td>rank of destination</td>
<td>int INTEGER</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tag</td>
<td>message tag</td>
<td>int INTEGER</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm TYPE(MPI_Comm)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>info</td>
<td>info argument</td>
<td>MPI_Info TYPE(MPI_Info)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>request</td>
<td>communication request</td>
<td>MPI_Request* TYPE(MPI_Request)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.5 MPI_Pready

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MPI_Pready (</td>
<td>partition</td>
<td>partition to mark ready for</td>
<td>int INTEGER</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>request</td>
<td>transfer partioned communication request</td>
<td>MPI_Request TYPE(MPI_Request)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• MPI_Pready_list for an explicitly enumerated list of partitions.

The MPI_Psend_init call yields an MPI_Request object that can be used to test for completion (see sections 4.2.2 and 4.2.3) of the full operation.

```c
MPI_Request send_request;
MPI_Psend_init(
    sendbuffer, nparts, SIZE, MPI_DOUBLE, tgt, 0,
    comm, MPI_INFO_NULL, &send_request);
for (int it=0; it<ITERATIONS; it++) {
    MPI_Start(&send_request);
    for (int ip=0; ip<nparts; ip++)
        fill_buffer(sendbuffer, partitions[ip], partitions[ip+1], ip);
    MPI_Pready(ip, send_request);
    MPI_Wait(&send_request, MPI_STATUS_IGNORE);
}
MPI_Request_free(&send_request);
```

The receiving side is largely the mirror image of the sending side:

```c
double *recvbuffer = (double*)malloc(bufsize*sizeof(double));
MPI_Request recv_request;
MPI_Precv_init(
    recvbuffer, nparts, SIZE, MPI_DOUBLE, src, 0,
    comm, MPI_INFO_NULL, &recv_request);
```
5.3. Synchronous and asynchronous communication

### MPI_Parrived

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Parrived</td>
<td>request</td>
<td>partitioned communication</td>
<td>MPI_Request</td>
<td>TYPE(MPI_Request)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>partition</td>
<td>partition to be tested</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flag</td>
<td>true if operation completed on the specified partition, false if not</td>
<td>int*</td>
<td>LOGICAL</td>
<td></td>
</tr>
</tbody>
</table>

```c
for (int it=0; it<ITERATIONS; it++) {
    MPI_Start(&recv_request);
    MPI_Wait(&recv_request, MPI_STATUS_IGNORE);
    int r = 1;
    for (ip=0; ip<nparts; ip++)
        r *= check_buffer(recvbuffer, partitions[ip], partitions[ip+1], ip);
}
MPI_Request_free(&recv_request);
```

- a partitioned send can only be matched with a partitioned receive, so we start with an **MPI_Precv_init**.
- Arrival of a partition can be tested with **MPI_Parrived** (figure 5.6).
- A call to **MPI_Wait** completes the operation, indicating that all partitions have arrived.

Again, the **MPI_Request** object from the receive-init call can be used to test for completion of the full receive operation.

*End of MPI-4 material*

---

5.3 Synchronous and asynchronous communication

It is easiest to think of blocking as a form of synchronization with the other process, but that is not quite true. Synchronization is a concept in itself, and we talk about *synchronous* communication if there is actual coordination going on with the other process, and *asynchronous* communication if there is not. Blocking then only refers to the program waiting until the user data is safe to reuse; in the synchronous case a blocking call means that the data is indeed transferred, in the asynchronous case it only means that the data has been transferred to some system buffer. The four possible cases are illustrated in figure 5.1.

5.3.1 Synchronous send operations

MPI has a number of routines for synchronous communication, such as **MPI_Ssend**. Driving home the point that nonblocking and asynchronous are different concepts, there is a routine **MPI_Isend**, which is synchronous but nonblocking. These routines have the same calling sequence as their not-explicitly synchronous variants, and only differ in their semantics.

See section 4.1.4.2 for examples.
5. MPI topic: Communication modes

5.4 Local and nonlocal operations

The MPI standard does not dictate whether communication is buffered. If a message is buffered, a send call can complete, even if no corresponding send has been posted yet. See section 4.1.4.2. Thus, in the standard communication, a send operation is nonlocal: its completion may be depend on whether the corresponding receive has been posted. A local operation is one that is not nonlocal.

On the other hand, buffered communication (routines MPI_Bsend, MPI_Ibsend, MPI_Bsend_init; section 5.5) is local: the presence of an explicit buffer means that a send operation can complete no matter whether the receive has been posted.

The synchronous send (routines MPI_Ssend, MPI_Issend, MPI_Ssend_init; section 15.8) is again nonlocal (even in the nonblocking variant) since it will only complete when the receive call has completed.

Finally, the ready mode send (MPI_Rsend, MPI_Irsend) is nonlocal in the sense that its only correct use is when the corresponding receive has been issued.

5.5 Buffered communication

By now you have probably got the notion that managing buffer space in MPI is important: data has to be somewhere, either in user-allocated arrays or in system buffers. Using buffered communication is yet another way of managing buffer space.

1. You allocate your own buffer space, and you attach it to your process. This buffer is not a send buffer: it is a replacement for buffer space used inside the MPI library or on the network card; figure 5.2. If high-bandwidth memory is available, you could create your buffer there.
5.5. Buffered communication

![Diagram of user data and buffer communication](image)

Figure 5.2: User communication routed through an attached buffer

### Figure 5.7 MPI_Bsend

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Bsend</td>
<td>buf</td>
<td>initial address of send buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>IN</td>
</tr>
<tr>
<td>MPI_Bsend_c</td>
<td>count</td>
<td>number of elements in send buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>datatype</td>
<td>datatype of each send buffer element</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>dest</td>
<td>rank of destination</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>tag</td>
<td>message tag</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
</tbody>
</table>

2. You use the MPI_Bsend (figure 5.7) (or its local variant MPI_Ibsend) call for sending, using otherwise normal send and receive buffers;

3. You detach the buffer when you’re done with the buffered sends.

One advantage of buffered sends is that they are nonblocking: since there is a guaranteed buffer long enough to contain the message, it is not necessary to wait for the receiving process.

We illustrate the use of buffered sends:

```c
// bufring.c
int bsize = BUFLEN*sizeof(float);
float
*sbuf = (float*) malloc( bsize ),
*rbuf = (float*) malloc( bsize );
MPI_Pack_size( BUFLEN,MPI_FLOAT,comm,&bsize);
bsize += MPI_BSEND_OVERHEAD;
float
*buffer = (float*) malloc( bsize );
MPI_Buffer_attach( buffer,bsize );
```
### 5. MPI topic: Communication modes

#### Figure 5.8 MPI_Buffer_attach

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Buffer_attach</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Buffer_attach_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>buffer</td>
<td></td>
<td>initial buffer address</td>
<td>void*</td>
<td>TYPE(*), IN</td>
</tr>
<tr>
<td>size</td>
<td></td>
<td>buffer size, in bytes</td>
<td>int</td>
<td>DIMENSION(..)</td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
err = MPI_Bsend(sbuf, BUFLEN, MPI_FLOAT, next, 0, comm);
MPI_Recv (rbuf, BUFLEN, MPI_FLOAT, prev, 0, comm, MPI_STATUS_IGNORE);
MPI_Buffer_detach( &buffer, &bsize );
```

### 5.5.1 Buffer treatment

There can be only one buffer per process, attached with `MPI_Buffer_attach` (figure 5.8). Its size should be enough for all `MPI_Bsend` calls that are simultaneously outstanding. You can compute the needed size of the buffer with `MPI_Pack_size`; see section 6.8. Additionally, a term of `MPI_BSEND_OVERHEAD` is needed. See the above code fragment.

The buffer is detached with `MPI_Buffer_detach`:

```c
int MPI_Buffer_detach(
    void *buffer, int *size);
```

This returns the address and size of the buffer; the call blocks until all buffered messages have been delivered.

Note that both `MPI_Buffer_attach` and `MPI_Buffer_detach` have a `void*` argument for the buffer, but
- in the attach routine this is the address of the buffer,
- while the detach routine it is the address of the buffer pointer.

This is done so that the detach routine can zero the buffer pointer.

While the buffered send is nonblocking like an `MPI_Isend`, there is no corresponding wait call. You can force delivery by

```c
MPI_Buffer_detach( &b, &n );
MPI_Buffer_attach( b, n );
```

**MPL note 37: Buffered send.** Creating and attaching a buffer is done through `bsend_buffer` and a support routine `bsend_size` helps in calculating the buffer size:

```c
// bufring.cxx
vector<float> sbuf(BUFLEN), rbuf(BUFLEN);
int size{ comm_world.bsend_size<float>(mpl::contiguous_layout<float>(BUFLEN)) };
mpl::bsend_buffer buff(size);
comm_world.bsend(sbuf.data(), mpl::contiguous_layout<float>(BUFLEN), next);
```
5.5. Buffered communication

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Bsend_init</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>buf</td>
<td>initial address of send buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>number of elements sent</td>
<td>int</td>
<td></td>
<td>IN</td>
</tr>
<tr>
<td>MPI_Count</td>
<td>datatype</td>
<td>type of each element</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>dest</td>
<td>rank of destination</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>tag</td>
<td>message tag</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)IN</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>request</td>
<td>communication request</td>
<td>MPI_Request*</td>
<td>TYPE(MPI_Request) OUT</td>
<td>OUT</td>
</tr>
</tbody>
</table>

Constant: `mpl::bsend_overhead` is `constexpr`d to the MPI constant `MPI_BSEND_OVERHEAD`.

**MPL note 38: Buffer attach and detach.** There is a separate attach routine, but normally this is called by the constructor of the `bsend_buffer`. Likewise, the detach routine is called in the buffer destructor.

```cpp
void mpi::environment::buffer_attach (void *buff, int size);
std::pair< void *, int > mpi::environment::buffer_detach ()
```

5.5.2 Bufferend send calls

The possible error codes are

- `MPI_SUCCESS` the routine completed successfully.
- `MPI_ERR_BUFFER` The buffer pointer is invalid; this typically means that you have supplied a null pointer.
- `MPI_ERR_INTERN` An internal error in MPI has been detected.

The asynchronous version is `MPI_Ibsend`, the persistent (see section 5.1) call is `MPI_Bsend_init`.

5.5.3 Persistent buffered communication

There is a persistent variant `MPI_Bsend_init` (figure 5.9) of buffered sends, as with regular sends (section 5.1).
5. MPI topic: Communication modes

5.6 Sources used in this chapter

5.6.1 Listing of code header

5.6.2 Listing of code examples/mpi/p/persist.py

```python
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

nexperiments = 10
nsizes = 6
times = np.empty(nsizes,dtype=np.float64)
src = 0; tgt = nprocs-1

# # ordinary communication
#
size = 1
if procid==src:
    print("Ordinary send/recv")
    for isize in range(nsizes):
        requests = [ None ] * 2
        sendbuf = np.ones(size,dtype=int)
        recvbuf = np.ones(size,dtype=int)
        if procid==src:
            print("Size:",size)
            times[isize] = MPI.Wtime()
            for n in range(nexperiments):
                requests[0] = comm.Isend(sendbuf[0:size],dest=tgt)
                requests[1] = comm.Irecv(recvbuf[0:size],source=tgt)
                MPI.Request.Waitall(requests)
                sendbuf[0] = sendbuf[0]+1
                times[isize] = MPI.Wtime()-times[isize]
        elif procid==tgt:
            for n in range(nexperiments):
                comm.Recv(recvbuf[0:size],source=src)
                comm.Send(recvbuf[0:size],dest=src)
                size *= 10
        if procid==src:
            print("Timings:",times)
    
# # ordinary communication
#
```
```python
size = 1
requests = [None] * 2
if procid==src:
    print("Persistent send/recv")
for isize in range(nsizes):
    sendbuf = np.ones(size,dtype=int)
    recvbuf = np.ones(size,dtype=int)
    if procid==src:
        print("Size:",size)
        requests[0] = comm.Send_init(sendbuf[0:size],dest=tgt)
        requests[1] = comm.Recv_init(recvbuf[0:size],source=tgt)
        times[isize] = MPI.Wtime()
        for n in range(nexperiments):
            MPI.Prequest.Startall(requests)
            MPI.Prequest.Waitall(requests)
            sendbuf[0] = sendbuf[0]+1
            times[isize] = MPI.Wtime()-times[isize]
    elif procid==tgt:
        for n in range(nexperiments):
            comm.Recv(recvbuf[0:size],source=src)
            comm.Send(recvbuf[0:size],dest=src)
    size *= 10
if procid==src:
    print("Timings:",times)
```
Chapter 6

MPI topic: Data types

In the examples you have seen so far, every time data was sent, it was as a contiguous buffer with elements of a single type. In practice you may want to send heterogeneous data, or noncontiguous data.

- Communicating the real parts of an array of complex numbers means specifying every other number.
- Communicating a C structure or Fortran type with more than one type of element is not equivalent to sending an array of elements of a single type.

The datatypes you have dealt with so far are known as *elementary datatypes*; irregular objects are known as *derived datatypes*.

6.1 Data type handling

Datatypes such as MPI_INT are values of the type MPI_Datatype. This type is handled differently in different languages.

Fortran note 8: Derived types for handles. In Fortran before 2008, datatypes variables are stored in Integer variables. With the 2008 standard, datatypes are Fortran derived types:

```fortran
!! vector.F90
Type(MPI_Datatype) :: newvectortype
if (myid==sender) then
    call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
                       newvectortype)
    call MPI_Type_commit(newvectortype)
    call MPI_Send(source,1,newvectortype,receiver,0,comm)
    call MPI_Type_free(newvectortype)
    if (.not. newvectortype==MPI_DATATYPE_NULL) then
        print *,"Trouble freeing datatype"
    else
        print *,"Datatype successfully freed"
    end if
else if (myid==receiver) then
    call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,sender,0,comm,&
                  recv_status)
    call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count)
end if
```
6.2 Elementary data types

Python note 16: Data types. There is a class

```python
>>> mpi4py.MPI.Datatype
```

with predefined values such as

```python
>>> mpi4py.MPI.Datatype.DOUBLE
```

which are themselves objects with methods for creating derived types; see section 6.3.1.

MPL note 39: Other types.

```cpp
// sendlong.cxx
mpi::contiguous_layout<long long> v_layout(v.size());
comm.send(v.data(), v_layout, 1); // send to rank 1
```

Also works with complex of float and double.

MPL note 40: Data types. MPL routines are templated over the data type. The data types, where MPL can infer their internal representation, are enumeration types, C arrays of constant size and the template classes `std::array`, `std::pair` and `std::tuple` of the C++ Standard Template Library. The only limitation is, that the C array and the mentioned template classes hold data elements of types that can be sent or received by MPL.

6.2 Elementary data types

MPI has a number of elementary data types, corresponding to the simple data types of the host languages. The names are made to resemble the types of C and Fortran, for instance `MPI_FLOAT` and `MPI_DOUBLE` corresponding to `float` and `double` in C, versus `MPI_REAL` and `MPI_DOUBLE_PRECISION` corresponding to `Real` and `Double precision` in Fortran.

6.2.1 C/C++

Here we illustrate the correspondence between a type used to declare a variable, and how this type appears in MPI communication routines:

```c
long int i;
MPI_Send(&i,1,MPI_LONG,target,tag,comm);
```

- There is some, but not complete, support for C99 types.
- The `MPI_LONG_INT` type is not an integer type, but rather a `long` and an `int` packed together; see section 3.10.1.1.
- See section 6.2.4 for `MPI_Aint` and more about byte counting.

6.2.2 Fortran

Not all these types need be supported, for instance `MPI_INTEGER16` may not exist, in which case it will be equivalent to `MPI_Datatype_NULL`.

The default integer type `MPI_INTEGER` is equivalent to `INTEGER(KIND=MPI_INTEGER_KIND)`. 

**Victor Eijkhout**
### 6. MPI topic: Data types

<table>
<thead>
<tr>
<th>C type</th>
<th>MPI type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>unsigned char</td>
<td>MPI_UNSIGNED_CHAR</td>
</tr>
<tr>
<td>char</td>
<td>MPI_SIGNED_CHAR</td>
</tr>
<tr>
<td>short</td>
<td>MPI_SHORT</td>
</tr>
<tr>
<td>unsigned short</td>
<td>MPI_UNSIGNED_SHORT</td>
</tr>
<tr>
<td>int</td>
<td>MPI_INT</td>
</tr>
<tr>
<td>unsigned int</td>
<td>MPI_UNSIGNED</td>
</tr>
<tr>
<td>long int</td>
<td>MPI_LONG</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>MPI_UNSIGNED_LONG</td>
</tr>
<tr>
<td>long long int</td>
<td>MPI_LONG_LONG_INT</td>
</tr>
<tr>
<td>float</td>
<td>MPI_FLOAT</td>
</tr>
<tr>
<td>double</td>
<td>MPI_DOUBLE</td>
</tr>
<tr>
<td>long double</td>
<td>MPI_LONG_DOUBLE</td>
</tr>
<tr>
<td>unsigned char</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>(does not correspond to a C type)</td>
<td>MPI_PACKED</td>
</tr>
</tbody>
</table>

Table 6.1: Elementary datatypes in C

<table>
<thead>
<tr>
<th>C type</th>
<th>MPI type</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Bool</td>
<td>MPI_C_BOOL</td>
</tr>
<tr>
<td>float _Complex</td>
<td>MPI_C_COMPLEX</td>
</tr>
<tr>
<td></td>
<td>MPI_C_FLOAT_COMPLEX</td>
</tr>
<tr>
<td>double _Complex</td>
<td>MPI_C_DOUBLE_COMPLEX</td>
</tr>
<tr>
<td>long double_ Complex</td>
<td>MPI_C_LONG_DOUBLE_COMPLEX</td>
</tr>
</tbody>
</table>

Table 6.2: C99 synonym types.

#### 6.2.2.1 Big data types

The C type `MPI_Count` corresponds to an integer of type `MPI_COUNT_KIND`, used most prominently in ‘big data’ routines such as `MPI_Type_size_x` (section 6.6):

```
  Integer(kind=MPI_COUNT_KIND) :: count
  call MPI_Type_size_x(my_type, count)
```

The following material is for the recently released MPI-4 standard and may not be supported yet.

For every routine `MPI_Something` with an `int` count parameter, there is a corresponding routine `MPI_Something_c` with an `MPI_Count` parameter.

The above `MPI_Something_x` routines will probably be deprecated in the MPI-4.1 standard.

End of MPI-4 material

#### 6.2.2.2 Byte counting types

Kind `MPI_ADDRESS_KIND` is used for `MPI_Aint` quantities, used in Remote Memory Access (RMA) windows; see section 9.3.1.
6.2. Elementary data types

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8_t</td>
<td>MPI_INT8_T</td>
</tr>
<tr>
<td>int16_t</td>
<td>MPI_INT16_T</td>
</tr>
<tr>
<td>int32_t</td>
<td>MPI_INT32_T</td>
</tr>
<tr>
<td>int64_t</td>
<td>MPI_INT64_T</td>
</tr>
<tr>
<td>uint8_t</td>
<td>MPI_UINT8_T</td>
</tr>
<tr>
<td>uint16_t</td>
<td>MPI_UINT16_T</td>
</tr>
<tr>
<td>uint32_t</td>
<td>MPI_UINT32_T</td>
</tr>
<tr>
<td>uint64_t</td>
<td>MPI_UINT64_T</td>
</tr>
</tbody>
</table>

Table 6.3: C11 fixed width integer types.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHARACTER</td>
<td>Character(Len=1)</td>
</tr>
<tr>
<td>MPI_INTEGER</td>
<td></td>
</tr>
<tr>
<td>MPI_REAL</td>
<td></td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td></td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td></td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
<tr>
<td>MPI_INTEGER1</td>
<td></td>
</tr>
<tr>
<td>MPI_INTEGER2</td>
<td></td>
</tr>
<tr>
<td>MPI_INTEGER4</td>
<td></td>
</tr>
<tr>
<td>MPI_INTEGER8</td>
<td></td>
</tr>
<tr>
<td>MPI_INTEGER16</td>
<td></td>
</tr>
<tr>
<td>MPI_REAL2</td>
<td></td>
</tr>
<tr>
<td>MPI_REAL4</td>
<td></td>
</tr>
<tr>
<td>MPI_REAL8</td>
<td></td>
</tr>
<tr>
<td>MPI_DOUBLE_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>Complex(Kind=Kind(0.d0))</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: Standard Fortran types (left) and common extension (right)

The MPI_OFFSET_KIND is used to define MPI_Offset quantities, used in file I/O; section 10.2.2.

6.2.2.3 Fortran90 kind-defined types

If your Fortran90 code uses KIND to define scalar types with specified precision, these do not in general correspond to any predefined MPI datatypes. Hence the following routines exist to make MPI equivalences of Fortran scalar types:

- MPI_Type_create_f90_integer (figure 6.1)
- MPI_Type_create_f90_real (figure 6.2)
- MPI_Type_create_f90_complex (figure 6.3).

Examples:

```fortran
INTEGER ( KIND = SELECTED_INT_KIND(15) ) , &
DIMENSION(100) :: array INTEGER :: root , integertype , error

CALL MPI_Type_create_f90_integer( 15 , integertype , error )
CALL MPI_Bcast ( array , 100 , &
            integertype , root , MPI_COMM_WORLD , error )

REAL ( KIND = SELECTED_REAL_KIND(15 ,300) ) , &
DIMENSION(100) :: array
```
6. MPI topic: Data types

Figure 6.1 MPI_Type_create_f90_integer

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_create_f90_integer</td>
<td>r</td>
<td>decimal exponent range, i.e., number of decimal digits</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>newtype</td>
<td>the requested MPI datatype</td>
<td>MPI_Datatype*</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.2 MPI_Type_create_f90_real

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_create_f90_real</td>
<td>p</td>
<td>precision, in decimal digits</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>decimal exponent range</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>newtype</td>
<td>the requested MPI datatype</td>
<td>MPI_Datatype*</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
</tbody>
</table>

CALL MPI_Type_create_f90_real( 15 , 300 , realtype , error )

COMPLEX ( KIND = SELECTED_REAL_KIND(15,300) ), &
DIMENSION(100) :: array
CALL MPI_Type_create_f90_complex( 15 , 300 , complextype , error )

6.2.3 Python

In python, all buffer data comes from Numpy.

<table>
<thead>
<tr>
<th>mpi4py type</th>
<th>NumPy type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI.INT</td>
<td>np.intc</td>
</tr>
<tr>
<td></td>
<td>np.int32</td>
</tr>
<tr>
<td>MPI.LONG</td>
<td>np.int64</td>
</tr>
<tr>
<td>MPI.FLOAT</td>
<td>np.float32</td>
</tr>
<tr>
<td>MPI.DOUBLE</td>
<td>np.float64</td>
</tr>
</tbody>
</table>

In this table we see that Numpy has three integer types, one corresponding to C int, and two with the number of bits explicitly indicated. There used to be a np.int type, but this is deprecated as of Numpy 1.20

Examples:

```
# inttype.py
sizeofint = np.dtype('int32').itemsize
print("Size of numpy int32: {}*.format(sizeofint))
sizeofint = np.dtype('int').itemsize
print("Size of C int: {}*.format(sizeofint))
```

For the full source of this example, see section 6.10.2

```
# allgatherv.py
mycount = procid+1
```
6.2. Elementary data types

Figure 6.3 MPI_Type_create_f90_complex

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_create_f90_complex()</td>
<td>p</td>
<td>precision, in decimal digits</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>decimal exponent range</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>newtype</td>
<td>the requested MPI datatype</td>
<td>MPI_Datatype*</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
</tbody>
</table>

```python
my_array = np.empty(mycount, dtype=np.float64)
```

6.2.3.1 Type correspondences MPI / Python

Above we saw that the number of bytes of a Numpy type can be deduced from

```python
sizeofint = np.dtype('intc').itemsize
```

It is possible to derive the Numpy type corresponding to an MPI type:

```python
## typesize.py
datatype = MPI.FLOAT
typecode = MPI._typecode(datatype)
assert typecode is not None # check MPI datatype is built-in
dtype = np.dtype(typecode)
```

6.2.4 Byte addressing types

So far we have mostly been taking about datatypes in the context of sending them. The MPI_Aint type is not so much for sending, as it is for describing the size of objects, such as the size of an MPI_Win object (section 9.1) or byte displacements in MPI_Type_create_hindexed.

Addresses have type MPI_Aint. The start of the address range is given in MPI_BOTTOM. See also the MPI_Sizeof (section 6.2.5) and MPI_Get_address routines.

Variables of type MPI_Aint can be sent as MPI_AINT:

```python
MPI_Aint address;
MPI_Send( address,1,MPI_AINT, ... );
```

See section 9.5.2 for an example.

In order to prevent overflow errors in byte calculations there are support routines MPI_Aint_add

```python
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)
```

and similarly MPI_Aint_diff.
6. MPI topic: Data types

6.2.4.1 Fortran

The equivalent of `MPI_Aint` in Fortran is an integer of kind `MPI_ADDRESS_KIND`:

```fortran
integer(kind=MPI_ADDRESS_KIND) :: winsize
```

Fortran lacks a `sizeof` operator to query the sizes of datatypes. Since sometimes exact byte counts are necessary, for instance in one-sided communication, Fortran can use the (deprecated) `MPI_Sizeof` routine. See section 6.2.5 for details.

Example usage in `MPI_Win_create`:

```fortran
call MPI_Sizeof(windowdata,window_element_size,ierr)
window_size = window_element_size=500
call MPI_Win_create( windowdata,window_size,window_element_size,... )
```

6.2.4.2 Python

Here is a good way for finding the size of `numpy` datatypes in bytes:

```python
# putfence.py
intsize = np.dtype('int').itemsize
window_data = np.zeros(2,dtype=int)
win = MPI.Win.Create(window_data,intsize,comm=comm)
```

6.2.5 Matching language type to MPI type

In some circumstances you may want to find the MPI type that corresponds to a type in your programming language.

- In C++ functions and classes can be templated, meaning that the type is not fully known:

  ```cpp
template<typename T> {
  class something<T> {
  public:
    void dosend(T input) {
      MPI_Send( &input,1,/* ??? */ );
    };
  }
  }
```

  (Note that in MPI this is hardly ever needed because MPI calls are templated there.)

- Petsc installations use a generic identifier `PetscScalar` (or `PetscReal`) with a configuration-dependent realization.

- The size of a datatype is not always statically known, for instance if the Fortran `KIND` keyword is used.

Here are some MPI mechanisms that address this problem.
### 6.2. Elementary data types

#### Figure 6.4 MPI_Type_match_size

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_match_size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>typeclass</td>
<td>generic type specifier</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>size, in bytes, of representation</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>datatype</td>
<td>datatype with correct type, size</td>
<td>MPI_Datatype*</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Figure 6.5 MPI_Type_size

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Type_size_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>datatype</td>
<td>datatype to get information on datatype size</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>int*</td>
<td>OUT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 6.2.5.1 Type matching in C

Datatypes in C can be translated to MPI types with `MPI_Type_match_size` (figure 6.4) where the `typeclass` argument is one of `MPI_TYPECLASS_REAL`, `MPI_TYPECLASS_INTEGER`, `MPI_TYPECLASS_COMPLEX`.

```c
// typematch.c
float x5;
double x10;
int s5, s10;
MPI_Datatype mpi_x5, mpi_x10;

MPI_Type_match_size(MPI_TYPECLASS_REAL, sizeof(x5), &mpi_x5);
MPI_Type_match_size(MPI_TYPECLASS_REAL, sizeof(x10), &mpi_x10);
MPI_Type_match_size(MPI_TYPECLASS_REAL, sizeof(x10), &mpi_x10);
MPI_Type_size(mpi_x5, &s5);
MPI_Type_size(mpi_x10, &s10);
```

For the full source of this example, see section 6.10.3

The space that MPI takes for a structure type can be queried in a variety of ways. First of all `MPI_Type_size` (figure 6.5) counts the `datatype size` as the number of bytes occupied by the data in a type. That means that in an `MPI vector datatype` it does not count the gaps.

```c
// typesize.c
MPI_Type_vector(count, bs, stride, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
MPI_Type_size(newtype, &size);
ASSERT(sizeof(count*bs)*sizeof(double));
```

For the full source of this example, see section 6.10.4
6. MPI topic: Data types

6.2.5.2 Type matching in Fortran

In Fortran, the size of the datatype in the language can be obtained with `MPI_Sizeof` (note the nonoptional error parameter!). This routine is deprecated in MPI-4: use of `storage_size` and/or `c_sizeof` is recommended.

```f90
!! matchkind.F90
call MPI_Sizeof(x10,s10,ierr)
call MPI_Type_match_size(MPI_TYPECLASS_REAL,s10,mpi_x10)
call MPI_Type_size(mpi_x10,s10)
print *,"10 positions supported, MPI type size is",s10
```

For the full source of this example, see section 6.10.3

Petsc has its own translation mechanism; see section 33.2.

6.3 Derived datatypes

MPI allows you to create your own data types, somewhat (but not completely…) analogous to defining structures in a programming language. MPI data types are mostly of use if you want to send multiple items in one message.

There are two problems with using only elementary datatypes as you have seen so far.

- MPI communication routines can only send multiples of a single data type: it is not possible to send items of different types, even if they are contiguous in memory. It would be possible to use the `MPI_BYTE` data type, but this is not advisable.
- It is also ordinarily not possible to send items of one type if they are not contiguous in memory. You could of course send a contiguous memory area that contains the items you want to send, but that is wasteful of bandwidth.

With MPI data types you can solve these problems in several ways.

- You can create a new contiguous data type consisting of an array of elements of another data type. There is no essential difference between sending one element of such a type and multiple elements of the component type.
- You can create a vector data type consisting of regularly spaced blocks of elements of a component type. This is a first solution to the problem of sending noncontiguous data.
- For not regularly spaced data, there is the indexed data type, where you specify an array of index locations for blocks of elements of a component type. The blocks can each be of a different size.
- The struct data type can accommodate multiple data types.

And you can combine these mechanisms to get irregularly spaced heterogeneous data, et cetera.

6.3.1 Basic calls

The typical sequence of calls for creating a new datatype is as follows:

- You need a variable for the datatype;
• There is a create call, followed by a 'commit' call where MPI performs internal bookkeeping and optimizations;
• The datatype is used, possibly multiple times;
• When the datatype is no longer needed, it must be freed to prevent memory leaks.

In code:

```c
MPI_Datatype newtype;
MPI_Type_something( < oldtype specifications >, &newtype );
MPI_Type_commit( &newtype );
/* code that uses your new type */
MPI_Type_free( &newtype );
```

In Fortran2008:

```fortran
Type(MPI_Datatype) :: newvectortype
call MPI_Type_something( <oldtype specification>, & newvectortype)
call MPI_Type_commit(newvectortype)
!! code that uses your type
call MPI_Type_free(newvectortype)
```

Python note 17: Derived type handling. The various type creation routines are methods of the datatype classes, after which commit and free are methods on the new type.

```python
# vector.py
source = np.empty(stride*count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newvectortype = MPI.DOUBLE.Create_vector(count,1,stride)
    newvectortype.Commit()
    comm.Send([source,1,newvectortype],dest=the_other)
    newvectortype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

For the full source of this example, see section 6.10.5

MPL note 41: Derived type handling. In MPL type creation routines are in the main namespace, templated over the datatypes.

```c
mpl::contiguous_layout<double> target_layout(count);
mpl::status_t recv_status =
    comm_world.recv(target.data(),target_layout, the_other);
recv_count = recv_status.get_count<double>();
```

For the full source of this example, see section 6.10.6

The commit call is part of the type creation, and freeing is done in the destructor.

6.3.1.1 Create calls

The `MPI_Datatype` variable gets its value by a call to one of the following routines:

• `MPI_Type_contiguous` for contiguous blocks of data; section 6.3.2;
6. MPI topic: Data types

### Figure 6.6 MPI_Type_commit

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_commit (</td>
<td>datatype</td>
<td>datatype that is committed</td>
<td>MPI_Datatype*</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
</tbody>
</table>

MPL:

Done as part of the type create call.

### Figure 6.7 MPI_Type_free

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_free (</td>
<td>datatype</td>
<td>datatype that is freed</td>
<td>MPI_Datatype*</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
</tbody>
</table>

MPL:

Done in the destructor.

- `MPI_Type_vector` for regularly strided data; section 6.3.3;
- `MPI_Type_create_subarray` for subsets out higher dimensional block; section 6.3.4;
- `MPI_Type_create_struct` for heterogeneous irregular data; section 6.3.6;
- `MPI_Type_indexed` and `MPI_Type_hindexed` for irregularly strided data; section 6.3.5.

These calls take an existing type, whether elementary or also derived, and produce a new type.

6.3.1.2 Commit and free

It is necessary to call `MPI_Type_commit` (figure 6.6) on a new data type, which makes MPI do the indexing calculations for the data type.

When you no longer need the data type, you call `MPI_Type_free` (figure 6.7). (This is typically not needed in OO APIs.) This has the following effects:

- The definition of the datatype identifier will be changed to `MPI_DATATYPE_NULL`.
- Any communication using this data type, that was already started, will be completed successfully.
- Datatypes that are defined in terms of this data type will still be usable.

6.3.2 Contiguous type

The simplest derived type is the 'contiguous' type, constructed with `MPI_Type_contiguous` (figure 6.8).

A contiguous type describes an array of items of an elementary or earlier defined type. There is no difference between sending one item of a contiguous type and multiple items of the constituent type. This is illustrated in figure 6.1.

```c
// contiguous.c
MPI_Datatype newvectortype;
if (procno==sender) {
```
6.3. Derived datatypes

Figure 6.8 MPI_Type_contiguous

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_contiguous</td>
<td></td>
<td></td>
<td></td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>replication count</td>
<td>int</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td>MPI_Type_contiguous_c</td>
<td></td>
<td></td>
<td></td>
<td>TYPE(MPI_Datatype)</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>replication count</td>
<td>int</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>oldtype</td>
<td>old datatype</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>newtype</td>
<td>new datatype</td>
<td>MPI_Datatype*</td>
<td>TYPE(MPI_Datatype)</td>
<td>OUT</td>
</tr>
</tbody>
</table>

Python:

Create_contiguous(self, int count)

Figure 6.1: A contiguous datatype is built up out of elements of a constituent type

MPI_Type_contiguous(count,MPI_DOUBLE,knewvectortype);
MPI_Type_commit(&newvectortype);
MPI_Send(source,1,newvectortype,receiver,0,comm);
MPI_Type_free(&newvectortype);
else if (procno==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE,receiver,0,comm,
              &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,krecv_count);
    ASSERT(count==recv_count);
}

For the full source of this example, see section 6.10.7

!! contiguous.F90
integer :: newvectortype
if (myid==sender) then
    call MPI_Type_contiguous(count,MPI_DOUBLE_PRECISION,newvectortype)
    call MPI_Type_commit(newvectortype)
    call MPI_Send(source,1,newvectortype,receiver,0,comm)
    call MPI_Type_free(newvectortype)
else if (myid==receiver) then
    call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,receiver,0,comm,
                  &recv_status)
    call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count)
    !ASSERT(count==recv_count);
end if

For the full source of this example, see section 6.10.8
6. MPI topic: Data types

Figure 6.9 MPI_Type_vector

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_vector</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Type_vector_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>number of blocks</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blocklength</td>
<td>number of elements in each</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stride</td>
<td>number of elements between</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>start of each block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>oldtype</td>
<td>old datatype</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td>TYPE(MPI_Datatype)IN</td>
<td></td>
</tr>
<tr>
<td>newtype</td>
<td>new datatype</td>
<td>MPI_Datatype*</td>
<td>TYPE(MPI_Datatype)OUT</td>
<td>TYPE(MPI_Datatype)OUT</td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Datatype.Create_vector(self, int count, int blocklength, int stride)
```

```python
# contiguous.py
source = np.empty(count, dtype=np.float64)
target = np.empty(count, dtype=np.float64)
if procid==sender:
    newcontiguoustype = MPI.DOUBLE.Create_contiguous(count)
    newcontiguoustype.Commit()
    comm.Send([source,1,newcontiguoustype],dest=the_other)
    newcontiguoustype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

For the full source of this example, see section 6.10.9

**MPL note 42: Contiguous type.** The MPL interface makes extensive use of contiguous_layout, as it is the main way to declare a nonscalar buffer; see note 11.

**MPL note 43: Contiguous composing.** Contiguous layouts can only use elementary types or other contiguous layouts as their ‘old’ type. To make a contiguous type for other layouts, use vector_layout:

```cpp
// contiguous.cpp
mp::contiguous_layout<int> type1(7);
mp::vector_layout<int> type2(8,type1);
```

(Contrast this with strided_vector_layout; note 44.)

6.3.3 Vector type

The simplest noncontiguous datatype is the ‘vector’ type, constructed with MPI_Type_vector (figure 6.9).

A vector type describes a series of blocks, all of equal size, spaced with a constant stride. This is illustrated in figure 6.2.
6.3. Derived datatypes

The vector datatype gives the first nontrivial illustration that datatypes can be different on the sender and receiver. If the sender sends b blocks of length l each, the receiver can receive them as bl contiguous elements, either as a contiguous datatype, or as a contiguous buffer of an elementary type; see figure 6.3. In this case, the receiver has no knowledge of the stride of the datatype on the sender.

In this example a vector type is created only on the sender, in order to send a strided subset of an array; the receiver receives the data as a contiguous block.

```c
// vector.c
source = (double*) malloc(stride*count*sizeof(double));
target = (double*) malloc(count*sizeof(double));
MPI_Datatype newvectortype;
if (procno==sender) {
    MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,other,0,comm);
    MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
```

Figure 6.2: A vector datatype is built up out of strided blocks of elements of a constituent type

Figure 6.3: Sending a vector datatype and receiving it as elementary or contiguous
6. MPI topic: Data types

MPI_Status recv_status;
int recv_count;
MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
          &recv_status);
MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
ASSERT(recv_count==count);
}

For the full source of this example, see section 6.10.10

We illustrate Fortran2008:

!! vector.F90
Type(MPI_Datatype) :: newvectortype
if (mytid==sender) then
  call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
                      newvectortype)
  call MPI_Type_commit(newvectortype)
  call MPI_Send(source,1,newvectortype,receiver,0,comm)
  call MPI_Type_free(newvectortype)
  if (.not. newvectortype==MPI_DATATYPE_NULL) then
    print *,"Trouble freeing datatype"
  else
    print *,"Datatype successfully freed"
  end if
else if (mytid==receiver) then
  call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,sender,0,comm,&
                recv_status)
  call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count)
end if

For the full source of this example, see section 6.10.11

In legacy mode Fortran90, code stays the same except that the type is declared as Integer:

!! vector.F90
integer :: newvectortype
integer :: recv_status(MPI_STATUS_SIZE),recv_count
call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
                      newvectortype,err)
call MPI_Type_commit(newvectortype, err)

For the full source of this example, see section 6.10.12

Python note 18: Vector type. The vector creation routine is a method of the datatype class. For the general discussion, see section 6.3.1.

# vector.py
source = np.empty(stride*count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
  newvectortype = MPI.DOUBLE.Create_vector(count,1,stride)
  newvectortype.Commit()
  comm.Send([source,1,newvectortype],dest=the_other)
  newvectortype.Free()
elif procid==receiver:
  comm.Recv([target,count,MPI.DOUBLE],source=the_other)
6.3. Derived datatypes

Figure 6.4: Memory layout of a row and column of a matrix in column-major storage

For the full source of this example, see section 6.10.5

MPL note 44: Vector type. MPL has the `strided_vector_layout` class as equivalent of the vector type:

```cxx
// vector.cxx
vector<double>
source(stride*count);
if (procno==sender) {
    mpl::strided_vector_layout<double>
    newvectortype(count,1,stride);
    comm_world.send
    (source.data(),newvectortype,the_other);
}
```

For the full source of this example, see section 6.10.6

(See note 43 for nonstrided vectors.)

6.3.3.1 Two-dimensional arrays

Figure 6.4 indicates one source of irregular data: with a matrix on column-major storage, a column is stored in contiguous memory. However, a row of such a matrix is not contiguous; its elements being separated by a stride equal to the column length.

**Exercise 6.1.** How would you describe the memory layout of a submatrix, if the whole matrix has size $M \times N$ and the submatrix $m \times n$?

As an example of this datatype, consider the example of transposing a matrix, for instance to convert between C and Fortran arrays (see section HPSC-38.4). Suppose that a processor has a matrix stored in C, row-major, layout, and it needs to send a column to another processor. If the matrix is declared as

```c
int M, N; double mat[M][N]
```
then a column has $M$ blocks of one element, spaced $N$ locations apart. In other words:

```c
MPI_Datatype MPI_column;
MPI_Type_vector(
    /* count= */ M, /* blocklength= */ 1, /* stride= */ N, 
    MPI_DOUBLE, &MPI_column);
```

Sending the first column is easy:

```c
MPI_Send( mat, 1, MPI_column, ... );
```

The second column is just a little trickier: you now need to pick out elements with the same stride, but starting at $A[0][1]$.

```c
MPI_Send( & (mat[0][1]), 1, MPI_column, ... );
```

You can make this marginally more efficient (and harder to read) by replacing the index expression by $mat+1$.

**Exercise 6.2.** Suppose you have a matrix of size $4N \times 4N$, and you want to send the elements $A[4*i][4*j]$ with $i,j = 0,\ldots,N-1$. How would you send these elements with a single transfer?

**Exercise 6.3.** Allocate a matrix on processor zero, using Fortran column-major storage. Using $P$ sendrecv calls, distribute the rows of this matrix among the processors.

**Python note 19: Sending from the middle of a matrix.** In C and Fortran it’s easy to apply a derived type to data in the middle of an array, for instance to extract an arbitrary column out of a C matrix, or row out of a Fortran matrix. While Python has no trouble describing sections from an array, usually it copies these instead of taking the address. Therefore, it is necessary to convert the matrix to a buffer and compute an explicit offset in bytes:

```python
## rowcol.py
rowsize = 4; colsiz = 5
coltype = MPI.INT.Create_vector(4, 1, 5)
coltype.Commit()
columntosend = 2
comm.Send( 
    [np.frombuffer(matrix.data, intc,
      offset=columntosend*np.dtype('intc').itemsize),
      1, coltype],
    receiver)
```

**Exercise 6.4.** Let processor 0 have an array $x$ of length $10P$, where $P$ is the number of processors. Elements $0, P, 2P, \ldots, 9P$ should go to processor zero, $1, P + 1, 2P + 1, \ldots$ to processor 1, et cetera. Code this as a sequence of send/recev calls, using a vector datatype for the send, and a contiguous buffer for the receive.

For simplicity, skip the send to/from zero. What is the most elegant solution if you want to include that case?

For testing, define the array as $x[i] = i$.

(There is a skeleton for this exercise under the name stridesend.)
6.3. Derived datatypes

Exercise 6.5. Write code to compare the time it takes to send a strided subset from an array: copy the elements by hand to a smaller buffer, or use a vector data type. What do you find? You may need to test on fairly large arrays.

6.3.4 Subarray type

The vector datatype can be used for blocks in an array of dimension more than 2 by using it recursively. However, this gets tedious. Instead, there is an explicit subarray type `MPI_Type_create_subarray` (figure 6.10). This describes the dimensionality and extent of the array, and the starting point (the 'upper left corner') and extent of the subarray.

*MPL note 45: Subarray layout.* The templated `subarray_layout` class is constructed from a vector of triplets of global size / subblock size / first coordinate.

```cpp
mpl::subarray_layout<int>(
    { {ny, ny_l, ny_0}, {nx, nx_l, nx_0} }
);
```

Exercise 6.6. Assume that your number of processors is \( P = Q^3 \), and that each process has an array of identical size. Use `MPI_Type_create_subarray` to gather all data onto a root process. Use a sequence of send and receive calls; `MPI_Gather` does not work here. (There is a skeleton for this exercise under the name `cubegather`.)

*Fortran note 9: Subarrays.* Subarrays are naturally supported in Fortran through array sections.

```fortran
!! section.F90
integer, parameter :: siz=20
real, dimension(siz,siz) :: matrix = [ ((j+(i-1)*siz,i=1,siz),j=1,siz) ]
real, dimension(2,2) :: submatrix
if (proccno==0) then
  call MPI_Send(matrix(1:2,1:2),4,MPI_REAL,1,0,comm)
else if (proccno==1) then
  call MPI_Recv(submatrix,4,MPI_REAL,0,0,comm,MPI_STATUS_IGNORE)
if (submatrix(2,2)==22) then
  print *, "Yay"
else
```

Figure 6.5: Send strided data from process zero to all others
6. MPI topic: Data types

Figure 6.10 MPI_Type_create_subarray

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_create_subarray</td>
<td>ndims</td>
<td>number of array dimensions</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td>MPI_Type_create_subarray_c</td>
<td>array_of_sizes</td>
<td>number of elements of type oldtype in each dimension of the full array</td>
<td>const int[]</td>
<td>const int[]</td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>array_of_sizes</td>
<td>number of elements of type oldtype in each dimension of the full array</td>
<td>const int[]</td>
<td>const int[]</td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>array_of_subsizes</td>
<td>number of elements of type oldtype in each dimension of the subarray</td>
<td>const int[]</td>
<td>const int[]</td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>array_of_starts</td>
<td>starting coordinates of the subarray in each dimension</td>
<td>const int[]</td>
<td>const int[]</td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>order</td>
<td>array storage order flag</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td>oldtype</td>
<td>newtype</td>
<td>old datatype</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td>newtype</td>
<td>newtype</td>
<td>new datatype</td>
<td>MPI_Datatype*</td>
<td>TYPE(MPI_Datatype)</td>
<td>OUT</td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Datatype.Create_subarray
(self, sizes, subsizes, starts, int order=ORDER_C)
```

```
print *,"nay...."
end if
end if
```

For the full source of this example, see section 6.10.13

However, there is a subtlety with non-blocking operations: for a non-contiguous buffer a temporary is created, which is released after the MPI call. This is correct for blocking sends, but for non-blocking the temporary has to stay around till the wait call.

```fortran
!! sectionisend.F90
integer :: siz
real, dimension(:,,:), allocatable :: matrix
real, dimension(2,2) :: submatrix
siz = 20
allocate( matrix(siz,siz) )
matrix = reshape( [ ((j+(i-1)*siz,i=1,siz),j=1,siz) ], (/siz,siz/) )
call MPI_Isend( matrix(1:2,1:2),4,MPI_REAL,1,0,comm,request)
call MPI_Wait(request,MPI_STATUS_IGNORE)
deallocate(matrix)
```

For the full source of this example, see section 6.10.14

In MPI-3 the variable MPI_SUBARRAYS_SUPPORTED indicates support for this mechanism:

```fortran
if (.not. MPI_SUBARRAYS_SUPPORTED ) then
  print *,"This code will not work"
call MPI_Abort(comm,0)
```

For the full source of this example, see section 6.10.14
6.3. Derived datatypes

For the full source of this example, see section 6.10.14

The possibilities for the order parameter are MPI_ORDER_C and MPI_ORDER_FORTRAN. However, this has nothing to do with the order of traversal of elements; it determines how the bounds of the subarray are interpreted. As an example, we fill a $4 \times 4$ array in C order with the numbers $0 \ldots 15$, and send the $[0,1] \times [0 \ldots 4]$ slice two ways, first C order, then Fortran order:

```c
// row2col.c
#define SIZE 4
int
sizes[2], subsizes[2], starts[2];
sizes[0] = SIZE; sizes[1] = SIZE;
subsizes[0] = SIZE/2; subsizes[1] = SIZE;
starts[0] = starts[1] = 0;
MPI_Type_create_subarray
(2, sizes, subsizes, starts,
 MPI_ORDER_C, MPI_DOUBLE,&rowtype);
MPI_Type_create_subarray
(2, sizes, subsizes, starts,
 MPI_ORDER_FORTRAN, MPI_DOUBLE,&coltype);
```

For the full source of this example, see section 6.10.15

The receiver receives the following, formatted to bring out where the numbers originate:

Received C order:
0.000 1.000 2.000 3.000
4.000 5.000 6.000 7.000
Received F order:
0.000 1.000
4.000 5.000
8.000 9.000
12.000 13.000

6.3.5 Indexed type

The indexed datatype, constructed with MPI_Type_indexed (figure 6.11) can send arbitrarily located elements from an array of a single datatype. You need to supply an array of index locations, plus an array of blocklengths with a separate blocklength for each index. The total number of elements sent is the sum of the blocklengths.

The following example picks items that are on prime number-indexed locations.

```c
// indexed.c
displacements = (int*) malloc(count*sizeof(int));
blocklengths = (int*) malloc(count*sizeof(int));
source = (int*) malloc(totalcount*sizeof(int));
target = (int*) malloc(targetbuffersize*sizeof(int));
MPI_Datatype newvectortype;
if (procno==sender) {
```
6. MPI topic: Data types

**Figure 6.11 MPI_Type_indexed**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_indexed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Type_indexed_c</td>
<td>count</td>
<td>number of blocks---also number of entries in array_of_displacements and array_of_blocklengths</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td></td>
<td></td>
<td>INTEGER</td>
<td></td>
<td>IN</td>
</tr>
<tr>
<td>array_of_blocklengths</td>
<td></td>
<td>number of elements per block</td>
<td>const int[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td></td>
<td></td>
<td>INTEGER(count)</td>
<td></td>
<td>IN</td>
</tr>
<tr>
<td>array_of_displacements</td>
<td></td>
<td>displacement for each block, in multiples of oldtype</td>
<td>const int[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td></td>
<td></td>
<td>INTEGER(count)</td>
<td></td>
<td>IN</td>
</tr>
<tr>
<td>oldtype</td>
<td></td>
<td>old datatype</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td>IN</td>
</tr>
<tr>
<td>newtype</td>
<td></td>
<td>new datatype</td>
<td>MPI_Datatype*</td>
<td>TYPE(MPI_Datatype)OUT</td>
<td>OUT</td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Datatype.Create_indexed(self, blocklengths, displacements)
```

**Figure 6.6: The elements of an MPI Indexed datatype**

For the full source of this example, see section 6.10.16

For Fortran we show the legacy syntax for once:

```fortran
MPI_Type_indexed(count, blocklengths, displacements, MPI_INT, &newvectortype);
MPI_Type_commit(&newvectortype);
MPI_Send(source, 1, newvectortype, the_other, 0, comm);
MPI_Type_free(&newvectortype);
```

```fortran
} else if (procno==receiver) {

MPI_Status recv_status;
int recv_count;
MPI_Recv(target, targetbuffersize, MPI_INT, the_other, 0, comm, &recv_status);
MPI_Get_count(&recv_status, MPI_INT, &recv_count);
ASSERT(recv_count==count);
}
```

For Fortran we show the legacy syntax for once:

```fortran
!! indexed.F90
```
6.3. Derived datatypes

```fortran
integer :: newvectortype;
ALLOCATE(indices(count))
ALLOCATE(blocklengths(count))
ALLOCATE(source(totalcount))
ALLOCATE(target(count))
if (mytid==sender) then
   call MPI_Type_indexed(count,blocklengths,indices,MPI_INT,&
                           newvectortype,err)
   call MPI_Type_commit(newvectortype,err)
   call MPI_Send(source,1,newvectortype,receiver,0,comm,err)
   call MPI_Type_free(newvectortype,err)
else if (mytid==receiver) then
   call MPI_Recv(target,count,MPI_INT,sender,0,comm,&
                  recv_status,err)
   call MPI_Get_count(recv_status,MPI_INT,recv_count,err)
   ! ASSERT(recv_count==count);
end if
```

For the full source of this example, see section 6.10.17

```python
# indexed.py
displacements = np.empty(count,dtype=int)
blocklengths = np.empty(count,dtype=int)
source = np.empty(totalcount,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
   newindextype = MPI.DOUBLE.Create_indexed(blocklengths,displacements)
   newindextype.Commit()
   comm.Send([source,1,newindextype],dest=the_other)
   newindextype.Free()
elif procid==receiver:
   comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

For the full source of this example, see section 6.10.18

MPL note 46: Indexed type. In MPL, the indexed_layout is based on a vector of 2-tuples denoting block length / block location.

```cpp
// indexed.cxx
const int count = 5;
mpl::contiguous_layout<int>
fiveints(count);
mpl::indexed_layout<int>
indexed_where( {{1,2}, {1,3}, {1,5}, {1,7}, {1,11}} );

if (procon==sender) {
   comm_world.send( source_buffer.data(),indexed_where, receiver );
} else if (procon==receiver) {
   auto recv_status =
   comm_world.recv( target_buffer.data(),fiveints, sender );
   int recv_count = recv_status.get_count<int>();
   assert(recv_count==count);
}
```

For the full source of this example, see section 6.10.19
6. MPI topic: Data types

**Figure 6.12 MPI_Type_create_hindexed_block**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_create_hindexed_block (</td>
<td></td>
<td>number of blocks---also number of entries in array_of_displacements</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>MPI_Type_create_hindexed_block_c (</td>
<td></td>
<td>count</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td></td>
<td>number of blocks---also number of entries in array_of_displacements</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>blocklength</td>
<td></td>
<td>number of elements in each block</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>array_of_displacements</td>
<td></td>
<td>byte displacement of each block</td>
<td>const MPI_Aint[]</td>
<td></td>
</tr>
<tr>
<td>oldtype</td>
<td></td>
<td>old datatype</td>
<td>MPI_Datatype</td>
<td></td>
</tr>
<tr>
<td>newtype</td>
<td></td>
<td>new datatype</td>
<td>MPI_Datatype*</td>
<td></td>
</tr>
</tbody>
</table>

MPL note 47: Layouts for gatherv. The size/displacement arrays for MPI_Gatherv / MPI_Alltoallv are handled through a layouts object, which is basically a vector of layout objects.

```cpp
mpi::layouts<int> receive_layout;
for ( int iproc=0,loc=0; iproc<nprocs; iproc++ ) {
    auto siz = size_buffer.at(iproc);
    receive_layout.push_back
        ( mpi::indexed_layout<int>({ { siz,loc } }) );
    loc += siz;
}
```

MPL note 48: Indexed block type. For the case where all block lengths are the same, use `indexed_block_layout`:

```cpp
// indexedblock.hxx
mpi::indexed_block_layout<int>
indexed_where( 1, {2,3,5,7,11} );
comm_world.send( source_buffer.data(),indexed_where, receiver );
```

For the full source of this example, see section 6.10.20

You can also `MPI_Type_create_hindexed` which describes blocks of a single old type, but with index locations in bytes, rather than in multiples of the old type.

```cpp
int MPI_Type_create_hindexed
    (int count, int blocklens[], MPI_Aint indices[],
    MPI_Datatype old_type,MPI_Datatype *newtype)
```

A slightly simpler version, `MPI_Type_create_hindexed_block` (figure 6.12) assumes constant block length.

There is an important difference between the hindexed and the above `MPI_Type_indexed`: that one described offsets from a base location; these routines describes absolute memory addresses. You can use this
6.3. Derived datatypes

Figure 6.13 MPI_Get_address

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Get_address</td>
<td>location</td>
<td>location in caller memory</td>
<td>const void* TYPE(*), DIMENSION(..)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>address</td>
<td>address of location</td>
<td>MPI_Aint* INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.14 MPI_Type_create_struct

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_create_struct</td>
<td>count</td>
<td>number of blocks---also number of entries in arrays array_of_types, array_of_displacements, and array_of_blocklengths</td>
<td>int</td>
<td>const int[]</td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>array_of_blocklengths</td>
<td>number of elements in each block array_of_types, array_of_displacements, and array_of_blocklengths</td>
<td></td>
<td>const int[]</td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>array_of_displacements</td>
<td>byte displacement of each block array_of_types, array_of_displacements, and array_of_blocklengths</td>
<td></td>
<td>const MPI_Aint[]</td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>array_of_types</td>
<td>type of elements in each block array_of_types, array_of_displacements, and array_of_blocklengths</td>
<td></td>
<td>const MPI_Datatype[]</td>
<td>TYPE(MPI_Datatype)</td>
</tr>
<tr>
<td></td>
<td>newtype</td>
<td>new datatype array_of_types, array_of_displacements, and array_of_blocklengths</td>
<td>MPI_Datatype*</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
</tbody>
</table>

...to send for instance the elements of a linked list. You would traverse the list, recording the addresses of the elements with MPI_Get_address (figure 6.13). (The routine MPI_Address is deprecated.)

In C++ you can use this to send an `std::vector`, that is, a vector object from the C++ standard library, if the component type is a pointer.

6.3.6 Struct type

The structure type, created with MPI_Type_create_struct (figure 6.14), can contain multiple data types. (The routine MPI_Type_struct is deprecated with MPI-3.) The specification contains a `count` parameter that specifies how many blocks there are in a single structure. For instance,

```c
struct {
  int i;
  float x,y;
} point;
```

has two blocks, one of a single integer, and one of two floats. This is illustrated in figure 6.7.
count  The number of blocks in this datatype. The blocklengths, displacements, types arguments have to be at least of this length.

blocklengths  array containing the lengths of the blocks of each datatype.

displacements  array describing the relative location of the blocks of each datatype.

types  array containing the datatypes; each block in the new type is of a single datatype; there can be multiple blocks consisting of the same type.

In this example, unlike the previous ones, both sender and receiver create the structure type. With structures it is no longer possible to send as a derived type and receive as a array of a simple type. (It would be possible to send as one structure type and receive as another, as long as they have the same datatype signature.)

```c
// struct.c
struct object {
  char c;
  double x[2];
  int i;
};
MPI_Datatype newstructuretype;
int structlen = 3;
int blocklengths[structlen]; MPI_Datatype types[structlen];
MPI_Aint displacements[structlen];

/*
 * where are the components relative to the structure?
 */
MPI_Aint current_displacement=0;

// one character
blocklengths[0] = 1; types[0] = MPI_CHAR;
displacements[0] = (size_t)&(myobject.c) - (size_t)&myobject;

// two doubles
blocklengths[1] = 2; types[1] = MPI_DOUBLE;
displacements[1] = (size_t)&(myobject.x) - (size_t)&myobject;

// one int
displacements[2] = (size_t)&(myobject.i) - (size_t)&myobject;
```
6.3. Derived datatypes

MPI_Type_create_struct(structlen,blocklengths,displacements,types,&newstructuretype);
MPI_Type_commit(&newstructuretype);
if (procno==sender) {
    MPI_Send(&myobject,1,newstructuretype,the_other,0,comm);
} else if (procno==receiver) {
    MPI_Recv(&myobject,1,newstructuretype,the_other,0,comm,MPI_STATUS_IGNORE);
}
MPI_Type_free(&newstructuretype);

For the full source of this example, see section 6.10.21

Note the displacement calculations in this example, which involve some not so elegant pointer arithmetic. The following Fortran code uses MPI_Get_address, which is more elegant, and in fact the only way address calculations can be done in Fortran.

!!! struct.F90

Type object
    character :: c
    real*8,dimension(2) :: x
    integer :: i
end type object
type(object) :: myobject
integer,parameter :: structlen = 3
type(MPI_Datatype) :: newstructuretype
integer,dimension(structlen) :: blocklengths
type(MPI_Datatype),dimension(structlen) :: types;
MPI_Aint,dimension(structlen) :: displacements
MPI_Aint :: base_displacement, next_displacement
if (procno==sender) then
    myobject%c = 'x'
    myobject%x(0) = 2.7; myobject%x(1) = 1.5
    myobject%i = 37

!! component 1: one character
blocklengths(1) = 1; types(1) = MPI_CHAR
call MPI_Get_address(myobject,base_displacement)
call MPI_Get_address(myobject%c,next_displacement)
displacements(1) = next_displacement-base_displacement

!! component 2: two doubles
blocklengths(2) = 2; types(2) = MPI_DOUBLE
call MPI_Get_address(myobject%x,next_displacement)
displacements(2) = next_displacement-base_displacement

!! component 3: one int
blocklengths(3) = 1; types(3) = MPI_INT
call MPI_Get_address(myobject%i,next_displacement)
displacements(3) = next_displacement-base_displacement

if (procno==sender) then
    call MPI_Send(myobject,1,newstructuretype,receiver,0,comm)
else if (procno==receiver) then
    call MPI_Recv(myobject,1,newstructuretype,sender,0,comm,MPI_STATUS_IGNORE)
end if
6. MPI topic: Data types

|| call MPI_Type_free(newstructuretype)

*For the full source of this example, see section 6.10.22*

It would have been incorrect to write

```
displacement[0] = 0;
displacement[1] = displacement[0] + sizeof(char);
```

since you do not know the way the compiler lays out the structure in memory.

If you want to send more than one structure, you have to worry more about padding in the structure. You can solve this by adding an extra type `MPI_UB` for the 'upper bound' on the structure:

```
MPI_Type_create_struct(struclen+1,.....);
```

*MPL note 49: Struct type scalar.* One could describe the MPI struct type as a collection of displacements, to be applied to any set of items that conforms to the specifications. An MPL heterogeneous_layout on the other hand, incorporates the actual data. Thus you could write

```
// structscalar.cxx
char c; double x; int i;
if (procno==sender) {
  c = 'x'; x = 2.4; i = 37; }
mlp::heterogeneous_layout object( c,x,i );
if (procno==sender)
  comm_world.send( mlp::absolute,object,receiver );
else if (procno==receiver)
  comm_world.recv( mlp::absolute,object,sender );
```

*For the full source of this example, see section 6.10.23*

Here, the `absolute` indicates the lack of an implicit buffer: the layout is absolute rather than a relative description.

*MPL note 50: Struct type general.* More complicated data than scalars takes more work:

```
// struct.cxx
char c; vector<double> x(2); int i;
if (procno==sender) {
  c = 'x'; x[0] = 2.7; x[1] = 1.5; i = 37; }
mlp::heterogeneous_layout object
  ( c,
    mlp::make_absolute(x.data(),mlp::vector_layout<double>(2)),
    i );
if (procno==sender) {
  comm_world.send( mlp::absolute,object,receiver );
} else if (procno==receiver) {
  comm_world.recv( mlp::absolute,object,sender );
}
```

*For the full source of this example, see section 6.10.23*

Note the `make_absolute` in addition to `absolute` mentioned above.

---

1. Homework question: what does the language standard say about this?
6.4 Big data types

The size parameter in MPI send and receive calls is of type integer, meaning that it's maximally (platform-dependent, but typically:) $2^{31} - 1$. These day computers are big enough that this is a limitation. As of the MPI-4 standard, this has been solved by allowing a larger count parameter. The implementation of this depends somewhat on the language.

The following material is for the recently released MPI-4 standard and may not be supported yet.

6.4.1 C

For every routine, such as `MPI_Send` with an integer count, there is a corresponding `MPI_Send_c` with a count of type `MPI_Count`.

```c
MPI_Count buffersize = 1000;
double *indata,*outdata;
indata = (double*) malloc( buffersize*sizeof(double) );
outdata = (double*) malloc( buffersize*sizeof(double) );
MPI_Allreduce_c(indata,outdata,buffersize,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
```

Code:

```c
// pingpongbig.c
assert( sizeof(MPI_Count)>4 );
for ( int power=3; power<10; power++ ) {
    MPI_Count length=pow(10,power);
    buffer = (double*)malloc( length*sizeof(double) );
    MPI_Send_c
        (buffer,length,MPI_DOUBLE, processB,0,comm);
    MPI_Recv_c
        (buffer,length,MPI_DOUBLE, processB,0,comm,MPI_STATUS_IGNORE);
```

Output:

```
make[3]: `pingpongbig' is up to date.
Ping-pong between ranks 0--1, repeated 10 times
MPI Count has 8 bytes
Size: $10^3$, (repeats=10000)
Time 1.399211e-05 for size $10^3$: 1.1435 Gb/sec
Size: $10^4$, (repeats=10000)
Time 4.077882e-05 for size $10^4$: 3.9236 Gb/sec
Size: $10^5$, (repeats=1000)
Time 1.532863e-04 for size $10^5$: 10.4380 Gb/sec
Size: $10^6$, (repeats=1000)
Time 1.418844e-03 for size $10^6$: 11.2768 Gb/sec
Size: $10^7$, (repeats=100)
Time 1.443470e-02 for size $10^7$: 11.0844 Gb/sec
Size: $10^8$, (repeats=100)
Time 1.540918e-01 for size $10^8$: 10.3834 Gb/sec
Size: $10^9$, (repeats=10)
Time 1.813220e+00 for size $10^9$: 8.8241 Gb/sec
Size: $10^{10}$, (repeats=10)
Time 1.846741e+01 for size $10^{10}$: 8.6639 Gb/sec
```

6.4.2 Fortran

The count parameter can be declared to be

```fortran
use mpi_f08
 Integer(kind=MPI_COUNT_KIND) :: count
```

Since Fortran has polymorphism, the same routine names can be used.
6. MPI topic: Data types

The legit way of coding:

```f90
!! typecheck.F90
integer(8) :: source
type(integer(kind=mpi_count_kind)) :: n=1
call MPI_Init()
call
    MPI_Send(source,n,MPI_INTEGER8,
    1,0,MPI_COMM_WORLD)
```

... but you can see what’s under the hood:

```f90
!! typecheck8.F90
integer(8) :: source, n=1
call MPI_Init()
call
    MPI_Send(source,n,MPI_INTEGER8,
    1,0,MPI_COMM_WORLD)
```

Routines using this type are not available unless using the mpi_f08 module.

End of MPI-4 material

6.4.3 Count datatype

The MPI_Count datatype is defined as being large enough to accommodate values of

- the ordinary 4-byte integer type;
- the MPI_Aint type, sections 6.2.4 and 6.2.4;
- the MPI_Offset type, section 10.2.2.

The size_t type in C/C++ is defined as big enough to contain the output of sizeof, that is, being big enough to measure any object.

6.4.4 MPI 3 temporary solution

Large messages were already possible by using derived types: to send a big data type of $10^{40}$ elements you would

- create a contiguous type with $10^{20}$ elements, and
- send $10^{20}$ elements of that type.

This often works, but it’s not perfect. For instance, the routine MPI_Get_elements returns the total number of basic elements sent (as opposed to MPI_Get_count which would return the number of elements of the derived type). Since its output argument is of integer type, it can’t store the right value.

The MPI-3 standard has addressed this through the introduction of an MPI_Count datatype, and new routines that return that type of count. (In view of the ‘embiggened’ routines, this solution is no longer needed, and will probably be deprecated in later standards.)

Let us consider an example.

Allocating a buffer of more than 4Gbyte is not hard:

```c
// vectorx.c
float *source=NULL,*target=NULL;
int mediumsize = 1<<30;
int nblocks = 8;
size_t datasize = (size_t)mediumsize * nblocks * sizeof(float);
if (procnosender) {
    source = (float*) malloc(datasize);
```
For the full source of this example, see section 6.10.24

We use the trick with sending elements of a derived type:

```c
MPI_Datatype blocktype;
MPI_Type_contiguous(mediumsize,MPI_FLOAT,&blocktype);
MPI_Type_commit(&blocktype);
if (procno==sender) {
  MPI_Send(source,nblocks,blocktype,receiver,0,comm);
}
```

For the full source of this example, see section 6.10.24

We use the same trick for the receive call, but now we catch the status parameter which will later tell us how many elements of the basic type were sent:

```c
} else if (procno==receiver) {
  MPI_Status recv_status;
  MPI_Recv(target,nblocks,blocktype,receiver,0,comm,
             &recv_status);
```

For the full source of this example, see section 6.10.24

When we query how many of the basic elements are in the buffer (remember that in the receive call the buffer length is an upper bound on the number of elements received) do we need a counter that is larger than an integer. MPI has introduced a type `MPI_Count` for this, and new routines such as `MPI_Get_elements_x` (figure 4.18) that return a count of this type:

```c
MPI_Count recv_count;
MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);
```

For the full source of this example, see section 6.10.24

**Remark 15** Computing a big number to allocate is not entirely simple.

```c
// getx.c
int gig = 1<<30;
int nblocks = 8;
sizet big1 = gig * nblocks * sizeof(double);
sizet big2 = (sizet)1 * gig * nblocks * sizeof(double);
sizet big3 = (sizet) gig * nblocks * sizeof(double);
sizet big4 = gig * nblocks * (sizet) (sizeof(double));
sizet big5 = sizeof(double) * gig * nblocks;
```

For the full source of this example, see section 6.10.25

gives as output:

```c
size of size_t = 8
0 68719476736 68719476736 0 68719476736
```

Clearly, not only do operations go left-to-right, but casting is done that way too: the computed subexpressions are only cast to `size_t` if one operand is.

Above, we did not actually create a datatype that was bigger than 2G, but if you do so, you can query its extent by `MPI_Type_get_extent_x` (figure 6.16) and `MPI_Type_get_true_extent_x` (figure 6.16).
6. MPI topic: Data types

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_get_extent</td>
<td></td>
<td>datatype to get information on</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
<tr>
<td>MPI_Type_get_extent_c</td>
<td>datatype</td>
<td>lower bound of datatype</td>
<td>MPI_Aint*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>lb</td>
<td>OUT</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td>MPI_Aint*</td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>extent</td>
<td>extent of datatype</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td>MPI_Aint*</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.15 MPI_Type_get_extent

Python note 20: Big data. Since python has unlimited size integers there is no explicit need for the 'x' variants of routines. Internally, MPI.Status.Get_elements is implemented in terms of MPI_Get_elements_x.

6.5 Type maps and type matching

With derived types, you saw that it was not necessary for the type of the sender and receiver to match. However, when the send buffer is constructed, and the receive buffer unpacked, it is necessary for the successive types in that buffer to match.

The types in the send and receive buffers also need to match the datatypes of the underlying architecture, with two exceptions. The MPI_PACKED and MPI_BYTE types can match any underlying type. However, this still does not mean that it is a good idea to use these types on only sender or receiver, and a specific type on the other.

6.6 Type extent

See section 6.2.5 about the related issue of type sizes.

6.6.1 Extent and true extent

The datatype extent, measured with MPI_Type_get_extent (figure 6.15), is strictly the distance from the first to the last data item of the type, that is, with counting the gaps in the type. It is measured in bytes so the output parameters are of type MPI_Aint.

In the following example (see also figure 6.8) we measure the extent of a vector type. Note that the extent is not the stride times the number of blocks, because that would count a 'trailing gap'.

```c
MPI_Aint lb, asize;
MPI_Type_vector(count, bs, stride, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
MPI_Type_get_extent(newtype, &lb, &asize);
ASSERT( lb==0 );
ASSERT( asize==((count-1)*stride+bs)*sizeof(double) );
MPI_Type_free(&newtype);
```
6.6. Type extent

Figure 6.8: Extent of a vector datatype

For the full source of this example, see section 6.10.4

Similarly, using MPI_Type_get_extent counts the gaps in a struct induced by alignment issues.

```c
size_t size_of_struct = sizeof(struct object);
MPI_Aint typesize, typelb;
MPI_Type_get_extent(newstructuretype,&typelb,&typesize);
assert( typesize==size_of_struct);
```

For the full source of this example, see section 6.10.21

See section 6.3.6 for the code defining the structure type.

**Remark 16** Routine MPI_Type_get_extent replaces deprecated functions MPI_Type_extent, MPI_Type_lb, MPI_Type_ub.

Figure 6.9: True lower bound and extent of a subarray data type

The subarray datatype need not start at the first element of the buffer, so the extent is an overstatement of how much data is involved. The routine MPI_Type_get_true_extent (figure 6.16) returns the lower bound, indicating where the data starts, and the extent from that point. This is illustrated in figure 6.9.

```c
// trueextent.c
int sender = 0, receiver = 1, the_other = 1-procno,
```
### 6. MPI topic: Data types

#### Figure 6.16 MPI_Type_get_true_extent

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_get_true_extent</td>
<td></td>
<td>datatype to get information on</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
<tr>
<td>MPI_Type_get_true_extent_c</td>
<td>datatype</td>
<td>true lower bound of datatype</td>
<td>MPI_Aint*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>true_lb</td>
<td>true lower bound of datatype</td>
<td>MPI_Aint*</td>
<td></td>
<td>OUT</td>
</tr>
<tr>
<td>MPI_Count</td>
<td>true_extent</td>
<td>true extent of datatype</td>
<td>MPI_Aint*</td>
<td></td>
<td>OUT</td>
</tr>
</tbody>
</table>

```c

```

```c

```c

```c

```

For the full source of this example, see section 6.10.26

There are also 'big data' routines `MPI_Type_get_extent_x` `MPI_Type_get_true_extent_x` that has an `MPI_Count` as output.

The following material is for the recently released MPI-4 standard and may not be supported yet.

End of MPI-4 material

#### 6.6.2  Extent resizing

A type is partly characterized by its lower bound and extent, or equivalently lower bound and upper bound. Somewhat miraculously, you can actually change these to achieve special effects. This is needed for:

- some cases of gather/scatter operations;

The problem here is that MPI uses the extent of the send type in a scatter, or the receive type in a gather: if that type is 20 bytes big from its first to its last element, then data will be read out
6.6. Type extent

20 bytes apart in a scatter, or written 20 bytes apart in a gather. This ignores the ‘gaps’ in the type! (See exercise 6.4.)

- When the count of derived items in a buffer is more than one.
  Consider the vector type from the previous section. It is clear that

\[
\text{MPI\_Type\_vector( 2*count, bs, stride, oldtype, &two\_n\_type );}
\]

will not give the same result as

\[
\text{MPI\_Type\_vector( count, bs, stride, oldtype, &one\_n\_type );}
\]

\[
\text{MPI\_Type\_contiguous( 2, &one\_n\_type, &two\_n\_type );}
\]

The technicality on which the solution hinges is that you can ‘resize’ a type with \text{MPI\_Type\_create\_resized} (figure 6.17) to give it a different extent, while not affecting how much data there actually is in it.

6.6.2.1 Example 1

![Diagram of contiguous type of two vectors, before and after resizing the extent.]

First consider sending more than one derived type, from a buffer containing consecutive integers:

\[
\text{for (int i=0; i<max\_elements; i++) sendbuffer[i] = i;}
\]
6. MPI topic: Data types

```c
MPI_Type_vector(count, blocklength, stride, MPI_INT, &stridetype);
MPI_Type_commit(&stridetype);
MPI_Send( sendbuffer, ntypes, stridetype, receiver, 0, comm );
```

For the full source of this example, see section 6.10.27

We receive into a contiguous buffer:

```c
MPI_Recv( recvbuffer, max_elements, MPI_INT, sender, 0, comm, &status );
int count; MPI_Get_count(&status, MPI_INT,&count);
printf("Receive %d elements:",count);
for (int i=0; i<count; i++) printf("%d",recvbuffer[i]);
printf("\n");
```

For the full source of this example, see section 6.10.27

giving an output of:
Receive 6 elements: 0 2 4 5 7 9

Next, we resize the type to ad the gap at the end. This is illustrated in figure 6.10.

Resizing the type looks like:

```c
MPI_Type_get_extent(stridetype,&l,&e);
printf("Stride type l=%ld e=%ld\n",l,e);
e += ( stride-blocklength) * sizeof(int);
MPI_Type_create_resized(stridetype,l,e,&paddedtype);
MPI_Type_get_extent(paddedtype,&l,&e);
printf("Padded type l=%ld e=%ld\n",l,e);
MPI_Type_commit(&paddedtype);
MPI_Send( sendbuffer, ntypes, paddedtype, receiver, 0, comm );
```

For the full source of this example, see section 6.10.27

and the corresponding output, including querying the extents, is:

Strided type l=0 e=20
Padded type l=0 e=24
Receive 6 elements: 0 2 4 6 8 10

6.6.2.2 Example 2

For another example, let's revisit exercise 6.4 (and figure 6.5) where each process makes a buffer of integers that will be interleaved in a gather call: Strided data was sent in individual transactions. Would it be possible to address all these interleaved packets in one gather or scatter call?

```c
int *mydata = (int*) malloc( localsize*sizeof(int) );
for (int i=0; i<localsize; i++)
  mydata[i] = i*nprocs+procn;
MPI_Gather( mydata,localsize, MPI_INT,
  /* rest to be determined */ );
```

An ordinary gather call will of course not interleave, but put the data end-to-end:
MPI_Gather( mydata, localsize, MPI_INT, gathered, localsize, MPI_INT, // abutting root, comm );

gather 4 elements from 3 procs:
0 3 6 9 1 4 7 10 2 5 8 11

Using a strided type still puts data end-to-end, but now there are unwritten gaps in the gather buffer:

MPI_Gather( mydata, localsize, MPI_INT, gathered, 1, stridetype, // abut with gaps root, comm );

0 1879048192 1100361260 3 3 0 6 0 0 9 1 198654

The trick is to use MPI_Type_create_resized to make the extent of the type only one int long:

// interleagather.c
MPI_Datatype interleavetype;
MPI_Type_create_resized( stridetype, 0, sizeof(int), &interleavetype );
MPI_Type_commit( &interleavetype );
MPI_Gather( mydata, localsize, MPI_INT, gathered, 1, interleavetype, // shrunk extent root, comm );

For the full source of this example, see section 6.10.28

Now data is written with the same stride, but at starting points equal to the shrunk extent:
0 1 2 3 4 5 6 7 8 9 10 11
This is illustrated in figure 6.11.

**Exercise 6.7.** Rewrite exercise 6.4 to use a gather, rather than individual messages.

*MPL note 51: Extent resizing.* Resizing a datatype does not give a new type, but does the resize ‘in place’:

```c
void layout::resize(ssize_t lb, ssize_t extent);
```

### 6.6.2.3 Example: dynamic vectors

Does it bother you (a little) that in the vector type you have to specify explicitly how many blocks there are? It would be nice if you could create a ‘block with padding’ and then send however many of those.

Well, you can introduce that padding by resizing a type, making it a little larger.

```c
// stridestretch.c
MPI_Datatype oneblock;
MPI_Type_vector(1,1,stride,MPI_DOUBLE,&oneblock);
MPI_Type_commit(&oneblock);
MPI_Aint block_lb,block_x;
MPI_Type_get_extent(oneblock,&block_lb,&block_x);
printf("One block has extent: \%ld\n",block_x);

MPI_Datatype paddedblock;
MPI_Type_create_resized(oneblock,0,stride*sizeof(double),&paddedblock);
MPI_Type_commit(&paddedblock);
MPI_Type_get_extent(paddedblock,&block_lb,&block_x);
printf("Padded block has extent: \%ld\n",block_x);

// now send a bunch of these padded blocks
MPI_Send(source,count,paddedblock,the_other,0,comm);
```

*For the full source of this example, see section 6.10.29*

There is a second solution to this problem, using a structure type. This does not use resizing, but rather indicates a displacement that reaches to the end of the structure. We do this by putting a type MPI_UB at this displacement:

```c
int blens[2]; MPI_Aint displs[2];
MPI_Datatype types[2], paddedblock;
blens[0] = 1; blens[1]= 1;
displs[0] = 0; displs[1] = 2 * sizeof(double);
types[0] = MPI_DOUBLE; types[1] = MPI_UB;
MPI_Type_struct(2, blens, displs, types, &paddedblock);
MPI_Type_commit(&paddedblock);
MPI_Status recv_status;
MPI_Recv(target,count,paddedblock,the_other,0,comm,&recv_status);
```

*For the full source of this example, see section 6.10.29*

### 6.6.2.4 Example: transpose

Transposing data is an important part of such operations as the FFT. We develop this in steps. Refer to figure 6.12.
6.7. Reconstructing types

The source data can be described as a vector type defined as:

- there are $b$ blocks,
- of blocksize $b$,
- spaced apart by the global $i$-size of the array.

```cpp
// transposeblock.cxx
MPI_Datatype sourceblock;
MPI_Type_vector(blocksize_j, blocksize_i, isize, MPI_INT, &sourceblock);
MPI_Type_commit(&sourceblock);
```

The target type is harder to describe. First we note that each contiguous block from the source type can be described as a vector type with:

- $b$ blocks,
- of size 1 each,
- stided by the global $j$-size of the matrix.

```cpp
MPI_Datatype targetcolumn;
MPI_Type_vector(blocksize_i, 1, jsize, MPI_INT, &targetcolumn);
MPI_Type_commit(&targetcolumn);
```

For the full type at the receiving process we now need to pack $b$ of these lines together.

**Exercise 6.8.** Finish the code.

- What is the extent of the `targetcolumn` type?
- What is the spacing of the first elements of the blocks? How do you therefore resize the `targetcolumn` type?

---

6.7 Reconstructing types

It is possible to find from a datatype how it was constructed. This uses the routines `MPI_Type_get_envelope` and `MPI_Type_get_contents`. The first routine returns the `combiner` (with values such as `MPI_COMBINER_VECTOR`) and the number of parameters; the second routine is then used to retrieve the actual parameters.
6. MPI topic: Data types

6.8 Packing

One of the reasons for derived datatypes is dealing with noncontiguous data. In older communication libraries this could only be done by packing data from its original containers into a buffer, and likewise unpacking it at the receiver into its destination data structures.

MPI offers this packing facility, partly for compatibility with such libraries, but also for reasons of flexibility. Unlike with derived datatypes, which transfers data atomically, packing routines add data sequentially to the buffer and unpacking takes them sequentially.

This means that one could pack an integer describing how many floating point numbers are in the rest of the packed message. Correspondingly, the unpack routine could then investigate the first integer and based on it unpack the right number of floating point numbers.

MPI offers the following:

- The `MPI_Pack` command adds data to a send buffer;
- the `MPI_Unpack` command retrieves data from a receive buffer;
- the buffer is sent with a datatype of `MPI_PACKED`.

With `MPI_Pack` data elements can be added to a buffer one at a time. The `position` parameter is updated each time by the packing routine.

```
int MPI_Pack(
    void *inbuf, int incount, MPI_Datatype datatype,
    void *outbuf, int outcount, int *position,
    MPI_Comm comm);
```

Conversely, `MPI_Unpack` retrieves one element from the buffer at a time. You need to specify the MPI datatype.

```
int MPI_Unpack(
    void *inbuf, int insize, int *position,
    void *outbuf, int outcount, MPI_Datatype datatype,
    MPI_Comm comm);
```

A packed buffer is sent or received with a datatype of `MPI_PACKED`. The sending routine uses the `position` parameter to specify how much data is sent, but the receiving routine does not know this value a priori, so has to specify an upper bound.
6.8. Packing

You can precompute the size of the required buffer with **MPI_Pack_size** (figure 6.18).

Victor Eijkhout 233
6. MPI topic: Data types

Code:

```c
// pack.c
for (int i=1; i<=4; i++) {
    MPI_Pack_size(i, MPI_CHAR, comm, &s);
    printf("%d chars: %d\n", i, s);
}
for (int i=1; i<=4; i++) {
    MPI_Pack_size(i, MPI_UNSIGNED_SHORT, comm, &s);
    printf("%d unsigned shorts: %d\n", i, s);
}
for (int i=1; i<=4; i++) {
    MPI_Pack_size(i, MPI_INT, comm, &s);
    printf("%d ints: %d\n", i, s);
}
```

Output:

```
1 chars: 1
2 chars: 2
3 chars: 3
4 chars: 4
1 unsigned shorts: 2
2 unsigned shorts: 4
3 unsigned shorts: 6
4 unsigned shorts: 8
1 ints: 4
2 ints: 8
3 ints: 12
4 ints: 16
```

Exercise 6.9. Suppose you have a ‘structure of arrays’

```c
struct aos {
    int length;
    double *reals;
    double *imags;
};
```

with dynamically created arrays. Write code to send and receive this structure.
6.9 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

1. Give two examples of MPI derived datatypes. What parameters are used to describe them?
2. Give a practical example where the sender uses a different type to send than the receiver uses in the corresponding receive call. Name the types involved.
3. Fortran only. True or false?
   (a) Array indices can be different between the send and receive buffer arrays.
   (b) It is allowed to send an array section.
   (c) You need to *Reshape* a multi-dimensional array to linear shape before you can send it.
   (d) An allocatable array, when dimensioned and allocated, is treated by MPI as if it were a normal static array, when used as send buffer.
   (e) An allocatable array is allocated if you use it as the receive buffer: it is filled with the incoming data.
4. Fortran only: how do you handle the case where you want to use an allocatable array as receive buffer, but it has not been allocated yet, and you do not know the size of the incoming data?
6. MPI topic: Data types

6.10 Sources used in this chapter

6.10.1 Listing of code header

6.10.2 Listing of code examples/mpi/p/inttype.py

```python
from mpi4py import MPI
import numpy as np
import sys

comm = MPI.COMM_WORLD
nprocs = comm.Get_size()
procno = comm.Get_rank()

if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

## set up sender and receiver
sender = 0; receiver = nprocs-1
## how many elements to each process?
count = 6

## Send a contiguous buffer as numpy ints
if procno==sender:
    data = np.empty(count,dtype=int32)
    for i in range(count):
        data[i] = i
    comm.Send( data, receiver )
elif procno==receiver:
    data = np.empty(count,dtype=int32)
    comm.Recv( data, sender )
    print(data)

## Send a strided buffer as numpy ints
## this is wrong because numpy ints are not C ints
if procno==sender:
    sizeofint = np.dtype('int32').itemsize
    print("Size of numpy int32: \{\}".format(sizeofint))
    data = np.empty(2*count,dtype=int32)
    for i in range(2*count):
        data[i] = i
    vectortype = MPI.INT.Create_vector(count,1,2)
    vectortype.Commit()
    comm.Send( [data,1,vectortype], receiver )
elif procno==receiver:
    data = np.empty(count,dtype=int32)
    comm.Recv( data, sender )
    print(data)

## Send strided buffer as C ints
```

236

Parallel Computing – r428
if procno==sender:
    sizeofint = np.dtype('intc').itemsize
    print("Size of C int: {}".format(sizeofint))
data = np.empty(2*count,dtype=intc)
for i in range(2*count):
    data[i] = i
vectortype = MPI.INT.Create_vector(count,1,2)
vectortype.Commit()
comm.Send( [data,1,vectortype], receiver )
elif procno==receiver:
    data = np.empty(count,dtype=intc)
    comm.Recv( data, sender )
    print(data)

6.10.3 Listing of code examples/mpi/c/typematch.c

   int main() {

        MPI_Init(0,0);

        float x5;
        double x10;
        int s5,s10;
        MPI_Datatype mpi_x5,mpi_x10;
        MPI_Type_match_size(MPI_TYPECLASS_REAL,sizeof(x5),&mpi_x5);
        MPI_Type_match_size(MPI_TYPECLASS_REAL,sizeof(x10),&mpi_x10);
        MPI_Type_size(mpi_x5,&s5);
        MPI_Type_size(mpi_x10,&s10);
        printf("%d, %d\n",s5,s10);
        MPI_Finalize();

        return 0;
   }

6.10.4 Listing of code examples/mpi/c/typesize.c

   #include <stdlib.h>
   #include <stdio.h>
   #include <string.h>
   #include "mpi.h"

   int main(int argc,char **argv) {

        #include "globalinit.c"

        int size, count, stride, bs;

        MPI_Datatype newtype;
6. MPI topic: Data types

```c
count = 3; bs = 2; stride = 5;
MPI_Type_vector(count, bs, stride, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
MPI_Type_size(&newtype, &size);
ASSERTE(size==(count*bs)*sizeof(double));
MPI_Type_free(&newtype);

printf("count=%d, stride=%d, bs=%d, size=%d\n", count, stride, bs, size);

MPI_Aint lb, asize;
MPI_Type_vector(count, bs, stride, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
MPI_Type_get_extent(&newtype, &lb, &asize);
ASSERTE(lb==0);
ASSERTE(asize==((count-1)*stride+bs)*sizeof(double));
MPI_Type_free(&newtype);

printf("count=%d, stride=%d, bs=%d: lb=%ld, extent=%ld\n", count, stride, bs, lb, asize);

if (procno==0)
    printf("Finished\n");
return 0;
```
6.10. Sources used in this chapter

```c
newvectortype.Free()
```

```c
eelif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

```c
if procid==sender:
    print("finished")
```

```c
if procid==receiver:
    for i in range(count):
        if target[i]!=source[stride*i]:
            print("error in location %d: %e s/b %e" % (i,target[i],source[stride*i]))
```

### 6.10.6 Listing of code examples/mpi/mpl/vector.cxx

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <vector>
using std::vector;
#include <cassert>
#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

    const mpl::communicator &comm_world = mpl::environment::comm_world();
    int nprocs,procno;
    // compute communicator rank and size
    nprocs = comm_world.size();
    procno = comm_world.rank();

    if (nprocs<2) {
        printf("This program needs at least two processes\n");
        return -1;
    }

    int sender = 0, receiver = 1, the_other = 1-procno,
    count = 5,stride=2;
    vector<double>
    source(stride*count);
    vector<double>
    target(count);

    for (int i=0; i<stride*count; i++)
        source[i] = i+.5;

    if (procno==sender) {
        mpl::strided_vector_layout<double>
        newvectortype(count,1,stride);
        comm_world.send
            (source.data(),newvectortype,the_other);
    }
    else if (procno==receiver) {
        int recv_count;
        mpl::contiguous_layout<double> target_layout(count);
```
6. MPI topic: Data types

```c
mpl::status_t recv_status =
    comm_world.recv(target.data(),target_layout, the_other);
recv_count = recv_status.get_count<double>();
assert(recv_count==count);
}

if (procno==receiver) {
    for (int i=0; i<count; i++)
        if (target[i]!=source[stride*i])
            printf("location \%d \%e s/b \%e\n",i,target[i],source[stride*i]);
}

if (procno==0)
    printf("Finished\n");

return 0;
}

6.10.7 Listing of code examples/mpi/c/contiguous.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
}

int sender = 0, receiver = 1, count = 5;
double *source,*target;
source = (double*) malloc(count*sizeof(double));
target = (double*) malloc(count*sizeof(double));

for (int i=0; i<count; i++)
    source[i] = i+.5;

MPI_Datatype newvectortype;
if (procno==sender) {
    MPI_Type_contiguous(count,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,receiver,0,comm);
    MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE,sender,0,comm,
        &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    MPI_Type_commit(&newvectortype);
```
6.10. Sources used in this chapter

```fortran
ASSERT(count==recv_count);
}
if (procno==receiver) {
    for (int i=0; i<count; i++)
        if (target[i]!=source[i])
            printf("location %d %e s/b %e\n",i,target[i],source[i]);
}
if (procno==0)
    printf("Finished\n");
MPI_Finalize();
return 0;
}

6.10.8 Listing of code examples/mpi/f08/contiguous.F90

Program Contiguous

use mpi_f08
implicit none

integer :: sender = 0, receiver = 1, count = 5
double precision, dimension(:), allocatable :: source, target

integer :: newvectortype
integer :: recv_status(MPI_STATUS_SIZE), recv_count

#include "globalinit.F90"

if (ntids<2) then
    print *, "This program needs at least two processes"
    stop
end if
ALLOCATE(source(count))
ALLOCATE(target(count))

do i=1,count
    source(i) = i+.5;
end do

if (mytid==sender) then
    call MPI_Type_contiguous(count,MPI_DOUBLE_PRECISION,newvectortype)
    call MPI_Type_commit(newvectortype)
    call MPI_Send(source,1,newvectortype,receiver,0,comm)
    call MPI_Type_free(newvectortype)
else if (mytid==receiver) then
    call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,sender,0,comm,&
                 recv_status)
    call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count)
    !ASSERT(count==recv_count);
```

Victor Eijkhout
end if

if (mytid==receiver) then
  ! for (i=0; i<count; i++)
  !  if (target[i]!=source[i])
  !    printf("location %d %e s/b %e\n",i,target[i],source[i]);
end if

call MPI_Finalize(err)

end Program Contiguous

6.10.9 Listing of code examples/mpi/p/contiguous.py

import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
sys.exit(1)

sender = 0; receiver = 1; the_other = 1-procid
count = 5

source = np.empty(count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)

for i in range(count):
    source[i] = i+.5

if procid==sender:
    newcontiguoustype = MPI.DOUBLE.Create_contiguous(count)
    newcontiguoustype.Commit()
    comm.Send([source,1,newcontiguoustype],dest=the_other)
    newcontiguoustype.Free()

elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)

if procid==sender:
    print("finished")
if procid==receiver:
    for i in range(count):
        if target[i]!=source[i]:
            print("error in location %d: %e s/b %e" % (i,target[i],source[i]))

6.10.10 Listing of code examples/mpi/c/vector.c

#include <stdlib.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

    if (nprocs<2) {
        printf("This program needs at least two processes\n");
        return -1;
    }
    int sender = 0, receiver = 1, the_other = 1-procno,
        count = 5,stride=2;
    double *source,*target;
    source = (double*) malloc(stride*count*sizeof(double));
    target = (double*) malloc(count*sizeof(double));

    for (int i=0; i<stride*count; i++)
        source[i] = i+.5;

    MPI_Datatype newvectortype;
    if (procno==sender) {
        MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
        MPI_Type_commit(&newvectortype);
        MPI_Send(source,1,newvectortype,the_other,0,comm);
        MPI_Type_free(&newvectortype);
    } else if (procno==receiver) {
        MPI_Status recv_status;
        int recv_count;
        MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
                        &recv_status);
        MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
        ASSERT(recv_count==count);
    }

    if (procno==receiver) {
        for (int i=0; i<count; i++)
            if (target[i]!=source[stride*i])
                printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
    }

    if (procno==0)
        printf("Finished\n");
    MPI_Finalize();
    return 0;
}

6.10.11 Listing of code examples/mpi/f08/vector.F90

Program Vector
use mpi_f08
implicit none

double precision, dimension(:,), allocatable :: source, target
integer :: sender = 0, receiver = 1, count = 5, stride = 2

Type(MPI_Datatype) :: newvectortype
integer :: recv_count
Type(MPI_Status) :: recv_status

Type(MPI_Comm) :: comm;
integer :: mytid, ntids, i, p, err;

call MPI_Init()
comm = MPI_COMM_WORLD
call MPI_Comm_rank(comm, mytid)
call MPI_Comm_size(comm, ntids)
call MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN)

if (ntids<2) then
   print *, "This program needs at least two processes"
stop
end if

ALLOCATE(source(stride*count))
ALLOCATE(target(stride*count))

do i=1,stride*count
   source(i) = i+.5;
end do

if (mytid==sender) then
   call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
         newvectortype)
call MPI_Type_commit(newvectortype)
call MPI_Send(source,1,newvectortype,receiver,0,comm)
call MPI_Type_free(newvectortype)
   if ( .not. newvectortype==MPI_DATATYPE_NULL) then
      print *, "Trouble freeing datatype"
   else
      print *, "Datatype successfully freed"
   end if
else if (mytid==receiver) then
   call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,sender,0,comm,&
        recv_status)
call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count)
end if

if (mytid==receiver) then
   ! for (i=0; i<count; i++)
   !   if (target[i]!=source[stride*i])
   !      printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
6.10. Sources used in this chapter

6.10.12 Listing of code examples/mpi/f/vector.F90

Program Vector

use mpi
implicit none

double precision, dimension(:,), allocatable :: source, target
integer :: sender = 0, receiver = 1, count = 5, stride = 2

integer :: newvector_type
integer :: recv_status(MPI_STATUS_SIZE), recv_count

#include "globalinit.F90"

if (ntids<2) then
   print *, "This program needs at least two processes"
   stop
end if

ALLOCATE(source(stride*count))
ALLOCATE(target(stride*count))

do i=1,stride*count
   source(i) = i+.5;
end do

if (mytid==sender) then
   call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
      newvector_type,err)
   call MPI_Type_commit(newvector_type,err)
   call MPI_Send(source,1,newvector_type,receiver,0,comm,err)
   call MPI_Type_free(newvector_type,err)
else if (mytid==receiver) then
   call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,sender,0,comm,&
      recv_status,err)
   call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count,err)
end if

if (mytid==receiver) then
   ! for (i=0; i<count; i++)
   !   if (target[i]!=source[stride*i])
   !      printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
end if

call MPI_Finalize(err)
6. MPI topic: Data types

end Program Vector

6.10.13 Listing of code examples/mpi/f08/section.F90

Program F90Section
use mpi_f08
implicit none

integer :: i,j, nprocs, procno
integer,parameter :: siz=20
real,dimension(siz,siz) :: matrix = ( ((j+(i-1)*siz,i=1,siz),j=1,siz) )
real,dimension(2,2) :: submatrix
Type(MPI_Comm) :: comm

call MPI_Init()
comm = MPI_COMM_WORLD

call MPI_Comm_size(comm,nprocs)
call MPI_Comm_rank(comm,procno)
if (nprocs<2) then
  print *,"This example really needs 2 processors"
call MPI_Abort(comm,0)
end if
if (procno==0) then
  call MPI_Send(matrix(1:2,1:2),4,MPI_REAL,1,0,comm)
else if (procno==1) then
  call MPI_Recv(submatrix,4,MPI_REAL,0,0,comm,MPI_STATUS_IGNORE)
  if (submatrix(2,2)==22) then
    print *,"Yay"
  else
    print *,"nay...."
  end if
else if (procno==1) then
  call MPI_Commsend(matrix(1:2,1:2),4,MPI_REAL,1,0,comm)
else if (procno==1) then
  call MPI_Recv(submatrix,4,MPI_REAL,0,0,comm,MPI_STATUS_IGNORE)
  if (submatrix(2,2)==22) then
    print *,"Yay"
  else
    print *,"nay...."
  end if
end if

call MPI_Finalize()
end Program F90Section

6.10.14 Listing of code examples/mpi/f08/sectionisend.F90

Program F90Section
use mpi_f08
implicit none

integer :: i,j, nprocs, procno
integer :: siz
real,dimension(:,,:),allocatable :: matrix
real,dimension(2,2) :: submatrix
Type(MPI_Comm) :: comm
Type(MPI_Request) :: request

call MPI_Init()
6.10. Sources used in this chapter

comm = MPI_COMM_WORLD

call MPI_Comm_size(comm,nprocs)
call MPI_Comm_rank(comm,procno)
if (nprocs<2) then
   print *,"This example really needs 2 processors"
call MPI_Abort(comm,0)
end if

if ( .not. MPI_SUBARRAYS_SUPPORTED ) then
   print *,"This code will not work"
call MPI_Abort(comm,0)
end if

if (procno==0) then
   siz = 20
   allocate( matrix(siz,siz) )
   matrix = reshape( [ ((j+(i-1)*siz,i=1,siz),j=1,siz) ], (/siz,siz/) )
call MPI_Isend(matrix(1:2,1:2),4,MPI_REAL,1,0,comm,request)
call MPI_Wait(request,MPI_STATUS_IGNORE)
deallocate(matrix)
else if (procno==1) then
   call MPI_IRecv(submatrix,4,MPI_REAL,0,0,comm,request)
call MPI_Wait(request,MPI_STATUS_IGNORE)
   if (submatrix(2,2)==22) then
      print *,"Yay"
   else
      print *,"nay...."
   end if
end if

call MPI_Finalize()

end Program F90Section

6.10.15 Listing of code code/mpi/c/row2col.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#define ERR(c,m) if (c) { printf("Error: %s\n",m); MPI_Abort(MPI_COMM_WORLD,c); }

int main(int argc,char **argv) {

#define MOD(i,n) (i+n)%n
#define ABS(x) ((x)<0 ? (-x) : (x))
#define MAX(x,y) ((x)>(y) ? (x) : (y))

#define MALLOC( t,v,n ) \
   t *v = (t*) malloc ( (n)*sizeof(t) ); \ 
   if (!v) { printf("Allocation failed in line %d\n",__LINE__);_LINE_; MPI_Abort(comm,0); }

Victor Eijkhout

MPI_Comm comm;
int procnos=-1,nprocs,ierr;
MPI_Init(&argc,&argv);
comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm,&procno);
MPI_Comm_size(comm,&nprocs);
MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN);

if (nprocs==1) {
    printf("This program needs at least 2 procs\n");
    MPI_Abort(comm,0);
} else {
    int sender = 0, receiver = nprocs-1;

    #define SIZE 4
    int
        sizes[2], subsizes[2], starts[2];
    sizes[0] = SIZE; sizes[1] = SIZE;
    subsizes[0] = SIZE/2; subsizes[1] = SIZE;
    starts[0] = starts[1] = 0;

    MPI_Request req;

    if (procno==sender) {
        /*
         * Write lexicographic test data
         */
        double data[SIZE][SIZE];
        for (int i=0; i<SIZE; i++)
            for (int j=0; j<SIZE; j++)
                data[i][j] = j+i*SIZE;
        /*
         * Make a datatype that enumerates the storage in C order
         */
        MPI_Datatype rowtype;
        ierr =
            MPI_Type_create_subarray
                (2,sizes,subsizes,starts,
                 MPI_ORDER_C,MPI_DOUBLE,&rowtype);
        ERR(ierr,"creating rowtype");
        MPI_Type_commit(&rowtype);
        MPI_Send(data,1,rowtype, receiver,0,comm);
        MPI_Type_free(&rowtype);
        
        /*
         * Make a datatype that enumerates the storage in F order
         */
        MPI_Datatype coltype;
        ierr =
            MPI_Type_create_subarray
                (2,sizes,subsizes,starts,
                 MPI_ORDER_FORTRAN,MPI_DOUBLE,&coltype);
        ERR(ierr,"creating rowtype");
    }

    MPI_Finalize();
}
MPI_Type_commit(&coltype);
MPI_Send(data,1,coltype, receiver,0,comm);
MPI_Type_free(&coltype);
}

} else if (procno==receiver) {
    int linearsize = SIZE * SIZE/2;
double lineardata[ linearsize ];

    /*
     * Receive msg in C order:
     */
    MPI_Recv(lineardata,linearsize,MPI_DOUBLE, sender,0,comm, MPI_STATUS_IGNORE);
    printf("Received C order:
");
    for (int i=0; i<SIZE/2; i++) {
        for (int j=0; j<SIZE; j++)
            printf(" %5.3f",lineardata[j+i*SIZE]);
        printf("\n");
    }

    /*
     * Receive msg in F order:
     */
    MPI_Recv(lineardata,linearsize,MPI_DOUBLE, sender,0,comm, MPI_STATUS_IGNORE);
    printf("Received F order:
");
    for (int j=0; j<SIZE; j++) {
        for (int i=0; i<SIZE/2; i++)
            printf(" %5.3f",lineardata[i+j*SIZE/2]);
        printf("\n");
    }
}

/* MPI_Finalize(); */
return 0;

6.10.16 Listing of code examples/mpi/c/indexed.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

    #include "globalinit.c"

    if (nprocs<2) {
        printf("This program needs at least two processes\n");
        return -1;
    }

    int sender = 0, receiver = 1, the_other = 1-procno,
    count = 5,totalcount = 15, targetbuffersize = 2*totalcount;

Victor Eijkhout
249
int *source,*target;
int *displacements,*blocklengths;

displacements = (int*) malloc(count*sizeof(int));
blocklengths = (int*) malloc(count*sizeof(int));
source = (int*) malloc(totalcount*sizeof(int));
target = (int*) malloc(targetbuffersize*sizeof(int));

displacements[0] = 2; displacements[1] = 3; displacements[2] = 5;
for (int i=0; i<count; ++i)
    blocklengths[i] = 1;
for (int i=0; i<totalcount; ++i)
    source[i] = i;

MPI_Datatype newvectortype;
if (procno==sender) {
    MPI_Type_indexed(count,blocklengths,displacements,MPI_INT,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,the_other,0,comm);
    MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,targetbuffersize,MPI_INT,the_other,0,comm,
             &recv_status);
    MPI_Get_count(&recv_status,MPI_INT,&recv_count);
    ASSERT(recv_count==count);
}
if (procno==receiver) {
    int i=3,val=7;
    if (target[i]!=val)
        printf("location %d %d s/b %d\n",i,target[i],val);
    i=4; val=11;
    if (target[i]!=val)
        printf("location %d %d s/b %d\n",i,target[i],val);
}
if (procno==0)
    printf("Finished\n");
MPI_Finalize();
return 0;

6.10.17 Listing of code examples/mpi/f/indexed.F90

Program Indexed

use mpi
implicit none

integer :: newvectortype;
integer, dimension(:,), allocatable :: indices, blocklengths, &
   source, targt
integer :: sender = 0, receiver = 1, count = 5, totalcount = 15
integer :: recv_status(MPI_STATUS_SIZE), recv_count

#include "globalinit.F90"

if (ntids<2) then
   print *, "This program needs at least two processes"
   stop
end if

ALLOCATE(indices(count))
ALLOCATE(blocklengths(count))
ALLOCATE(source(totalcount))
ALLOCATE(targt(count))

indices(0) = 2; indices(1) = 3; indices(2) = 5;
indices(3) = 7; indices(4) = 11;
do i=1,count
   blocklengths(i) = 1
end do

do i=1,totalcount
   source(i) = i
end do

if (mytid==sender) then
   call MPI_Type_indexed(count, blocklengths, indices, MPI_INT, &
      newvectortype, err)
   call MPI_Type_commit(newvectortype, err)
   call MPI_Send(source, 1, newvectortype, receiver, 0, comm, err)
   call MPI_Type_free(newvectortype, err)
else if (mytid==receiver) then
   call MPI_Recv(targt, count, MPI_INT, sender, 0, comm, &
      recv_status, err)
   call MPI_Get_count(recv_status, MPI_INT, recv_count, err)
   ! ASSERT(recv_count==count);
end if

! if (mytid==receiver) {
!   int i=3, val=7;
!   if (targt(i)!=val)
!      printf("location %d %d s/b %d\n", i, targt(i), val);
!   i=4; val=11;
!   if (targt(i)!=val)
!      printf("location %d %d s/b %d\n", i, targt(i), val);
! }

call MPI_Finalize(err)

end Program Indexed
6.10.18 Listing of code examples/mpi/p/indexed.py

```python
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

sender = 0; receiver = 1; the_other = 1-procid
count = 5; totalcount = 15

displacements = np.empty(count,dtype=int)
blocklengths = np.empty(count,dtype=int)
source = np.empty(totalcount,dtype=np.float64)
target = np.empty(count,dtype=np.float64)

idcs = [2,3,5,7,11]
for i in range(len(idcs)):
    displacements[i] = idcs[i]
    blocklengths[i] = 1
for i in range(totalcount):
    source[i] = i+.5

if procid==sender:
    newindextype = MPI.DOUBLE.Create_indexed(blocklengths,displacements)
    newindextype.Commit()
    comm.Send([source,1,newindextype],dest=the_other)
    newindextype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)

if procid==sender:
    print("finished")
if procid==receiver:
    target_loc = 0
    for block in range(count):
        for element in range(blocklengths[block]):
            source_loc = displacements[block]+element
            if target[target_loc]!=source[source_loc]:
                print("error in src/tar location %d/%d: %e s/b %e" %
                    (source_loc,target_loc,target[target_loc],source[source_loc]) )
            target_loc += 1
```

6.10.19 Listing of code examples/mpi/mpl/indexed.cxx

```c++
#include <iostream>
using std::cout;
using std::endl;
```

252 Parallel Computing – r428
6.10. Sources used in this chapter

```cpp
#include <vector>
using std::vector;

#include <cassert>

#include <mpl/mpl.hpp>

int main(int argc,char **argv) {
  // MPI Comm world
  const mpl::communicator &comm_world=mpl::environment::comm_world();
  int nprocs = comm_world.size(), procno = comm_world.rank();

  int sender = 0, receiver = 1, the_other = 1-procno,
      totalcount = 15, targetbuffersize = 2*totalcount;

  vector<int> source_buffer(totalcount),
              target_buffer(targetbuffersize);
  for (int i=0; i<totalcount; ++i)
    source_buffer[i] = i;

  const int count = 5;
  mpl::contiguous_layout<int> fiveints(count);
  mpl::indexed_layout<int> indexed_where{ { {1,2}, {1,3}, {1,5}, {1,7}, {1,11} } };

  if (procno==sender) {
    comm_world.send( source_buffer.data(),indexed_where, receiver );
  } else if (procno==receiver) {
    auto recv_status =
      comm_world.recv( target_buffer.data(),fiveints, sender );
    int recv_count = recv_status.get_count<int>();
    assert(recv_count==count);
  }

  if (procno==receiver) {
    int i=3,val=7;
    if (target_buffer[i]!=val)
      printf("Error: location %d %d s/b %d\n",i,target_buffer[i],val);
    i=4; val=11;
    if (target_buffer[i]!=val)
      printf("Error: location %d %d s/b %d\n",i,target_buffer[i],val);
    printf("Finished. Correctly sent indexed primes.\n");
  }

  return 0;
}
```
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

#include <cassert>

#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

    // MPI Comm world
    const mpl::communicator &comm_world=mpl::environment::comm_world();
    int nprocs = comm_world.size(), procno = comm_world.rank();

    int sender = 0, receiver = 1, the_other = 1-procno,
    totalcount = 15, targetbuffersize = 2*totalcount;

    vector<int>
    source_buffer(totalcount),
    target_buffer(targetbuffersize);
    for (int i=0; i<totalcount; ++i)
    source_buffer[i] = i;

    const int count = 5;
    if (procno==sender) {
        mpl::indexed_block_layout<int>
        indexed_where( 1, {2,3,5,7,11} );
        comm_world.send( source_buffer.data(),indexed_where, receiver );
    } else if (procno==receiver) {
        mpl::contiguous_layout<int>
        fiveints(count);
        auto recv_status =
        comm_world.recv( target_buffer.data(),fiveints, sender );
        int recv_count =
        recv_status.get_count<int>();
        assert(recv_count==count);
    }

    if (procno==receiver) {
        int i=3,val=7;
        if (target_buffer[i]!=val)
        printf("Error: location %d %d s/b %d\n",i,target_buffer[i],val);
        i=4; val=11;
        if (target_buffer[i]!=val)
        printf("Error: location %d %d s/b %d\n",i,target_buffer[i],val);
        printf("Finished. Correctly sent indexed primes.\n");
    }

    return 0;
}
Listing of code examples/mpi/c/struct.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "mpi.h"

int main(int argc,char **argv) {

    int sender = 0, receiver = 1, the_other = 1-procno;
    struct object {
        char c;
        double x[2];
        int i;
    };

    size_t size_of_struct = sizeof(struct object);
    struct object myobject;

    MPI_Datatype newstructuretype;
    int structlen = 3;
    int blocklengths[structlen]; MPI_Datatype types[structlen];
    MPI_Aint displacements[structlen];

    /*
     * where are the components relative to the structure?
     */
    MPI_Aint current_displacement=0;

    // one character
    blocklengths[0] = 1; types[0] = MPI_CHAR;
    displacements[0] = (size_t)&(myobject.c) - (size_t)&myobject;

    // two doubles
    blocklengths[1] = 2; types[1] = MPI_DOUBLE;

    if (procno==sender) {
        printf("Structure has size %ld, naive size %ld\n",
                size_of_struct,
                sizeof(char)+2*sizeof(double)+sizeof(int));
        myobject.c = 'x';
        myobject.x[0] = 2.7; myobject.x[1] = 1.5;
        myobject.i = 37;
    }
```

Victor Eijkhout
displacements[1] = (size_t)&(myobject.x) - (size_t)&myobject;

// one int
displacements[2] = (size_t)&(myobject.i) - (size_t)&myobject;

MPI_Type_create_struct(structlen,blocklengths,displacements,types,&newstructuretype);
MPI_Type_commit(&newstructuretype);

MPI_Aint typesize,typelb;
MPI_Type_get_extent(newstructuretype,&typelb,&typesize);
assert( typesize==size_of_struct );
if (procno==sender) {
    printf("Type extent: %ld bytes; displacements: %ld %ld %ld\n",
        typesize,displacements[0],displacements[1],displacements[2]);
}
if (procno==sender) {
    MPI_Send(&myobject,1,newstructuretype,the_other,0,comm);
} else if (procno==receiver) {
    MPI_Recv(&myobject,1,newstructuretype,the_other,0,comm,MPI_STATUS_IGNORE);
}
MPI_Type_free(&newstructuretype);

/* if (procno==sender) */
/* printf("char x=%ld, l=%ld; double x=%ld, l=%ld, int x=%ld, l=%ld\n", */
/*    char_extent,char_lb,double_extent,double_lb,int_extent,int_lb); */

if (procno==receiver) {
    printf("Char '%c' double0=%e double1=%e int=%d\n",
        myobject.c,myobject.x[0],myobject.x[1],myobject.i);
    ASSERT(myobject.x[1]==1.5);
    ASSERT(myobject.i==37);
}
if (procno==0)
    printf("Finished\n");
MPI_Finalize();
return 0;

#if 0
blocklengths[0] = 1; types[0] = MPI_CHAR;
displacements[0] = (size_t)&(myobject.c) - (size_t)&myobject;
blocklengths[1] = 2; types[1] = MPI_DOUBLE;
displacements[1] = (size_t)&(myobject.x[0]) - (size_t)&myobject;
displacements[2] = (size_t)&(myobject.i) - (size_t)&myobject;

MPI_Aint char_extent,char_lb;
MPI_Type_get_extent(MPI_CHAR,&char_lb,&char_extent);
/* if (procno==0) */
/* printf("CHAR lb=%ld xt=%ld disp=%ld\n",char_lb,char_extent,current_displacement); */
6.10. Sources used in this chapter

MPI_Aint double_extent,double_lb;
MPI_Type_get_extent(MPI_DOUBLE,&double_lb,&double_extent);
/* if (procno==0) */
/* printf("DOUBLE lb=%ld xt=%ld disp=%ld\n",double_lb,double_extent,current_displacement); */
MPI_Aint int_extent,int_lb;
MPI_Type_get_extent(MPI_INT,&int_lb,&int_extent);
/* if (procno==0) */
/* printf("INT lb=%ld xt=%ld disp=%ld\n",int_lb,int_extent,current_displacement); */
#endif

6.10.22 Listing of code examples/mpi/f/struct.F90

6.10.23 Listing of code examples/mpi/mpl/struct.hxx

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <vector>
using std::vector;
#include <cassert>
#include <mpl/mpl.hpp>

int main(int argc,char **argv) {
    const mpl::communicator &comm_world = mpl::environment::comm_world();
    int nprocs,procno;
    // compute communicator rank and size
    nprocs = comm_world.size();
    procno = comm_world.rank();

    if (nprocs<2) {
        printf("This program needs at least two processes\n");
        return -1;
    }
    int sender = 0, receiver = 1, the_other = 1-procno;
    char c; vector<double> x(2); int i;
    if (procno==sender) {
        c = 'x'; x[0] = 2.7; x[1] = 1.5; i = 37; }
    mpl::heterogeneous_layout object
        ( c,
          mpl::make_absolute(x.data(),mpl::vector_layout<double>(2)),
          i );
    if (procno==sender) {
        comm_world.send( mpl::absolute,object,receiver );
    } else if (procno==receiver) {
        comm_world.recv( mpl::absolute,object,sender );
    }
if (procno==receiver) {
    printf("Char '%c' double0=%e double1=%e int=%d\n", c,x[0],x[1],i);
    assert(x[1]==1.5);
    assert(i==37);
}

if (procno==0)
    printf("Finished\n");

return 0;

6.10.24 Listing of code examples/mpi/c/vectorx.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

    #include "globalinit.c"

    if (nprocs<2) {
        printf("This program needs at least two processes\n");
        return -1;
    }
    int sender = 0, receiver = 1, the_other = 1-procno,
        count = 5,stride=2;

    if (procno==sender) printf("size of size_t = %d\n",sizeof(size_t));

    float *source=NULL,*target=NULL;
    int mediumsize = 1<<30;
    int nbblocks = 8;
    size_t datasize = (size_t)mediumsize * nbblocks * sizeof(float);

    if (procno==sender)
        printf("datasize = %lld bytes =%7.3f giga-bytes = %7.3f gfloats\n", datasize,datasize*1.e-9,datasize*1.e-9/sizeof(float));

    if (procno==sender) {
        source = (float*) malloc(datasize);
        if (source) {
            printf("Source allocated\n");
        } else {
            printf("Could not allocate source data\n"); MPI_Abort(comm,1); }
        long int idx = 0;
        for (int iblock=0; iblock<nblocks; iblock++) {
            for (int element=0; element<mediumsize; element++) {
                source[idx] = idx+.5; idx++;
            }
        }
    }
if (idx<0) { printf("source index wrap\n"); MPI_Abort(comm,0); }
}
}

if (procno==receiver) {
target = (float*) malloc(datasize);
if (target) {
    printf("Target allocated\n");
} else {
    printf("Could not allocate target data\n"); MPI_Abort(comm,1); }
}

MPI_Datatype blocktype;
MPI_Type_contiguous(mediumsize,MPI_FLOAT,&blocktype);
MPI_Type_commit(&blocktype);
if (procno==sender) {
    MPI_Send(source,nblocks,blocktype,receiver,0,comm);
} else if (procno==receiver) {
    MPI_Status recv_status;
    MPI_Recv(target,nblocks,blocktype,sender,0,comm,
         &recv_status);
    MPI_Count recv_count;
    MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);
    printf("Received %7.3f medium size elements\n",recv_count * 1e-9);
}
MPI_Type_free(&blocktype);

if (0 && procno==receiver) {
    for (int i=0; i<count; i++)
        if (target[i]!=source[stride*i])
            printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
}

if (procno==0)
    printf("Finished\n");

if (procno==sender)
    free(source);
if (procno==receiver)
    free(target);

MPI_Finalize();
return 0;

6.10.25 Listing of code examples/mpi/c/getx.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char **argv) {
    #include "globalinit.c"

    if (procno==0) {
        printf("size of size_t = %d\n",sizeof(size_t));
        int gig = 1<<30;
        int nblocks = 8;
        size_t big1 = gig * nblocks * sizeof(double);
        size_t big2 = (size_t)1 * gig * nblocks * sizeof(double);
        size_t big3 = (size_t) gig * nblocks * sizeof(double);
        size_t big4 = gig * nblocks * (size_t) ( sizeof(double) );
        size_t big5 = sizeof(double) * gig * nblocks;
        printf("%lld %lld %lld %lld %lld\n",big1,big2,big3,big4,big5);
    }
    MPI_Finalize();
    return 0;
}

6.10.26 Listing of code examples/mpi/c/trueextent.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char **argv) {
    #include "globalinit.c"

    if (nprocs<2) {
        printf("This program needs at least two processes\n");
        return -1;
    }
    int sender = 0, receiver = 1, the_other = 1-procno,
    count = 4;
    double *source,*target;
    source = (double*) malloc(sizes[0]*sizes[1]*sizeof(double));
    target = (double*) malloc(sizes[0]*subsizes[1]*sizeof(double));

    for (int i=0; i<sizes[0]*sizes[1]; i++)
        source[i] = i+.5;

    if (procno==sender) {
        MPI_Datatype subarraytype;
        MPI_Type_create_subarray
            (2,sizes,subsizes,starts,MPI_ORDER_C,MPI_DOUBLE,&subarraytype);
        MPI_Type_commit(&subarraytype);
    }
MPI_Aint true_lb, true_extent, extent;
MPI_Type_get_true_extent
  (subarraytype, &true_lb, &true_extent);
MPI_Aint
  comp_lb = sizeof(double) *
    ( starts[0] * sizes[1] + starts[1] ),
  comp_extent = sizeof(double) *
      + ( subsizes[0] > 1 ? subsizes[0] - 1 : 0 ) * sizes[1] );
printf("Found lb=%ld, extent=%ld\n", true_lb, true_extent);
printf("Computing lb=%ld extent=%ld\n", comp_lb, comp_extent);
ASSERT(true_lb == comp_lb);
ASSERT(true_extent == comp_extent);
MPI_Send(source, 1, subarraytype, the_other, 0, comm);
{
  MPI_Aint lb, extent,
  comp_extent = sizeof(double) *
    ( ( starts[0] + subsizes[0] - 1 ) * sizes[1]
      + starts[1] + subsizes[1] );
MPI_Type_get_extent(subarraytype, &lb, &extent);
printf("Non-true lb=%ld, extent=%ld, computed=%ld\n",
  lb, extent, comp_extent);
ASSERT(extent == comp_extent);
}
MPI_Type_free(&subarraytype);
} else if (procno==receiver) {
  MPI_Status recv_status;
  int recv_count;
  MPI_Recv(target, count, MPI_DOUBLE, the_other, 0, comm,
    &recv_status);
  MPI_Get_count(&recv_status, MPI_DOUBLE, &recv_count);
  ASSERT(recv_count==count);
}
if (procno==receiver) {
  printf("received: ");
  for (int i=0; i<count; i++)
    printf("%6.3f", target[i]);
  printf("\n");
  int icnt = 0;
  for (int i=starts[0]; i<starts[0]+subsizes[0]; i++)
    for (int j=starts[1]; j<starts[1]+subsizes[1]; j++)
      printf("%d,%d\n", i, j);
  ASSERT(icnt<=count-1); %icnt too high%
  int isrc = i*sizes[1]+j;
  if (source[isrc]!=target[icnt])
    printf("target location (%d,%d)->%d %e s/b %e\n", i, j, icnt, target[icnt], source[isrc]);
  icnt ++;
}
6. MPI topic: Data types

```c
if (procno==0)
    printf("Finished\n");

free(source); free(target);
MPI_Finalize();
return 0;
```

6.10.27 Listing of code examples/mpi/c/vectorpadsend.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

    MPI_Comm comm; int nprocs,procno;
    MPI_Init(&argc,&argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_rank(comm,&procno);
    MPI_Comm_size(comm,&nprocs);

    if (nprocs<2) {
        printf("Needs at least 2 processes\n");
        MPI_Abort(comm,0);
    }
    int sender=0, receiver=1;

    /*
    * Datatype for strided destinations
    */
    MPI_Datatype stridetype;
    int count = 3, stride = 2, blocklength = 1;
    int ntypes = 2, max_elements = ntypes*stride*count;
    if (procno==sender) {
        int *sendbuffer = (int*)malloc( max_elements*sizeof(int) );
        for (int i=0; i<count; i++) sendbuffer[i] = i;
        MPI_Type_vector(count,blocklength,stride,MPI_INT,&stridetype);
        MPI_Type_commit(&stridetype);
        MPI_Send( sendbuffer,ntypes,stridetype, receiver,0, comm );
        free(sendbuffer);
    } else if (procno==receiver) {
        int *recvbuffer = (int*)malloc( max_elements*sizeof(int) );
        MPI_Status status;
        MPI_Recv( recvbuffer,max_elements,MPI_INT, sender,0, comm,&status );
        int count; MPI_Get_count(&status,MPI_INT,&count);
        printf("Receive %d elements: ",count);
        for (int i=0; i<count; i++) printf(" %d",recvbuffer[i]);
        printf("\n");
        free(recvbuffer);
    }
```
// ntypes*stride

MPI_Datatype paddedtype;
if (procno==sender) {
    MPI_Aint l,e;
    int *sendbuffer = (int*)malloc( max_elements*sizeof(int) );
    for (int i=0; i<max_elements; i++) sendbuffer[i] = i;
    MPI_Type_get_extent(stridetype,&l,&e);
    printf("Stride type l=%ld e=%ld\n",l,e);
    e += ( stride-blocklength) * sizeof(int);
    MPI_Type_create_resized(stridetype,l,e,&paddedtype);
    MPI_Type_get_extent(paddedtype,&l,&e);
    printf("Padded type l=%ld e=%ld\n",l,e);
    MPI_Type_commit(&paddedtype);
    MPI_Send( sendbuffer,ntypes,paddedtype, receiver,0, comm );
    free(sendbuffer);
} else if (procno==receiver) {
    int *recvbuffer = (int*)malloc( max_elements*sizeof(int) );
    MPI_Status status;
    MPI_Recv( recvbuffer,max_elements,MPI_INT, sender,0, comm,&status );
    int count; MPI_Get_count(&status,MPI_INT,&count);
    printf("Receive %d elements:",count);
    for (int i=0; i<count; i++) printf(" %d",recvbuffer[i]);
    printf("\n");
    free(recvbuffer);
}

if (procno==sender) {
    MPI_Type_free(&paddedtype);
    MPI_Type_free(&stridetype);
}

if (procno==0)
    printf("Finished\n");

MPI_Finalize();
return 0;

6.10.28 Listing of code examples/mpi/c/interleavegather

6.10.29 Listing of code examples/mpi/c/stridestretch.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int argc,char **argv) {

#include "globalinit.c"

if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
}

int sender = 0, receiver = 1, the_other = 1-procno,
    count = 5, stride=2;
double *source,*target;
source = (double*) malloc(stride*count*sizeof(double));
target = (double*) malloc(stride*count*sizeof(double));

for (int i=0; i<stride*count; i++)
    source[i] = i+.5;

if (procno==sender) {
    MPI_Datatype oneblock;
    MPI_Type_vector(1,1,stride,MPI_DOUBLE,&oneblock);
    MPI_Type_commit(&oneblock);
    MPI_Aint block_lb,block_x;
    MPI_Type_get_extent(oneblock,&block_lb,&block_x);
    printf("One block has extent: %ld\n",block_x);

    MPI_Datatype paddedblock;
    MPI_Type_create_resized(oneblock,0,stride*sizeof(double),&paddedblock);
    MPI_Type_commit(&paddedblock);
    MPI_Type_get_extent(paddedblock,&block_lb,&block_x);
    printf("Padded block has extent: %ld\n",block_x);

    // now send a bunch of these padded blocks
    MPI_Send(source,count,paddedblock,the_other,0,comm);
    MPI_Type_free(&oneblock);
    MPI_Type_free(&paddedblock);
} else if (procno==receiver) {
    int blens[2]; MPI_Aint displs[2];
    MPI_Datatype types[2], paddedblock;
    blens[0] = 1; blens[1] = 1;
    displs[0] = 0; displs[1] = 2 * sizeof(double);
    types[0] = MPI_DOUBLE; types[1] = MPI_UB;
    MPI_Type_struct(2, blens, displs, types, &paddedblock);
    MPI_Type_commit(&paddedblock);
    MPI_Status recv_status;
    MPI_Recv(target,count,paddedblock,the_other,0,comm,&recv_status);
    /* MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm, */
    /* &recv_status); */
    int recv_count;
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(recv_count==count);
}

if (procno==receiver) {
    for (int i=0; i<count; i++)
        if (target[i*stride]!=source[i*stride])
            break;
printf("location %d %e s/b %e\n", i, target[i*stride], source[stride*i]);
}

if (procno==0)
    printf("Finished\n");

MPI_Finalize();
return 0;
}
Chapter 7

MPI topic: Communicators

A communicator is an object describing a group of processes. In many applications all processes work together closely coupled, and the only communicator you need is MPI_COMM_WORLD, the group describing all processes that your job starts with.

In this chapter you will see ways to make new groups of MPI processes: subgroups of the original world communicator. Chapter 8 discusses dynamic process management, which, while not extending MPI_COMM_WORLD does extend the set of available processes. That chapter also discusses the ‘sessions model’, which is another way to constructing communicators.

7.1 Basic communicators

There are three predefined communicators:

- MPI_COMM_WORLD comprises all processes that were started together by mpiexec (or some related program).
- MPI_COMM_SELF is the communicator that contains only the current process.
- MPI_COMM_NULL is the invalid communicator. This values results
  - when a communicator is freed; see section 7.3;
  - as error return value from routines that construct communicators;
  - for processes outside a created Cartesian communicator (section 11.1.1);
  - on non-spawned processes when querying their parent (section 7.6.3).

These values are constants, though not necessarily compile-time constants. Thus, they can not be used in switch statements, array declarations, or constexpr evaluations.

If you don’t want to write MPI_COMM_WORLD repeatedly, you can assign that value to a variable of type MPI_Comm.

Examples:

```c
// C:
#include <mpi.h>
MPI_Comm comm = MPI_COMM_WORLD;
```
7.2. Duplicating communicators

With MPI_Comm_dup (figure 7.1) you can make an exact duplicate of a communicator (see section 7.2.2 for an application). There is a nonblocking variant MPI_Comm_idup (figure 7.2).

These calls do not propagate info hints (sections 15.1.1 and 15.1.1.2); to achieve this, use MPI_Comm_dup_with_info and MPI_Comm_idup_with_info; section 15.1.1.2.

Python note 22: Communicator duplication. Duplicate communicators are created as output of the duplication routine:
7. MPI topic: Communicators

Figure 7.1 MPI_Comm_dup

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_dup (</td>
<td>comm communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>newcomm copy of comm</td>
<td>MPI_Comm*</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MPI:

Done as part of the copy assignment operator.

Figure 7.2 MPI_Comm_idup

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_idup (</td>
<td>comm communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>newcomm copy of comm</td>
<td>MPI_Comm*</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>request communication request</td>
<td>MPI_Request*</td>
<td>TYPE(MPI_Request)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\
newcomm = comm.Dup()

**MPI note 54: Communicator duplication.** Communicators can be duplicated but only during initialization. Copy assignment has been deleted. Thus:

// LEGAL:
mpi::communicator init = comm;
// WRONG:
mpi::communicator init;
init = comm;

7.2.1 Communicator comparing

You may wonder what ‘an exact copy’ means precisely. For this, think of a communicator as a context label that you can attach to, among others, operations such as sends and receives. And it’s that label that counts, not what processes are in the communicator. A send and a receive ‘belong together’ if they have the same communicator context. Conversely, a send in one communicator can not be matched to a receive in a duplicate communicator, made by MPI_Comm_dup.

Testing whether two communicators are really the same is then more than testing if they comprise the same processes. The call MPI_Comm_compare returns MPI_IDENT if two communicator values are the same, and not if one is derived from the other by duplication:
Communicators that are not actually the same can be

- consisting of the same processes, in the same order, giving MPI_CONGRUENT;
- merely consisting of the same processes, but not in the same order, giving MPI_SIMILAR;
- different, giving MPI_UNEQUAL.

Comparing against MPI_COMM_NULL is not allowed.

### 7.2.2 Communicator duplication for library use

Duplicating a communicator may seem pointless, but it is actually very useful for the design of software libraries. Imagine that you have a code

```c
MPI_Isend(...); MPI_Irecv(...);
// library call
MPI_Waitall(...);
```

and suppose that the library has receive calls. Now it is possible that the receive in the library inadvertently catches the message that was sent in the outer environment.

Let us consider an example. First of all, here is code where the library stores the communicator of the calling program:

```c
// commdupwrong.cxx
class library {
private:
    MPI_Comm comm;
    int procnos, nprocs, other;
    MPI_Request request[2];
public:
    library(MPI_Comm incomm) {
```
7. MPI topic: Communicators

```cpp
comm = incomm;
MPI_Comm_rank(comm,&procno);
other = 1-procno;
int communication_start();
int communication_end();
```

*For the full source of this example, see section 7.8.2*

This models a main program that does a simple message exchange, and it makes two calls to library routines. Unbeknown to the user, the library also issues send and receive calls, and they turn out to interfere.

Here

- The main program does a send,
- the library call `function_start` does a send and a receive; because the receive can match either send, it is paired with the first one;
- the main program does a receive, which will be paired with the send of the library call;
- both the main program and the library do a wait call, and in both cases all requests are successfully fulfilled, just not the way you intended.

To prevent this confusion, the library should duplicate the outer communicator with `MPI_Comm_dup` and send all messages with respect to its duplicate. Now messages from the user code can never reach the library software, since they are on different communicators.

```cpp
// commdupright.cxx
class Library {
private:
    MPI_Comm comm;
    intprocno,nprocs,other;
    MPI_Request request[2];
public:
    Library(MPI_Comm incomm) {
        MPI_Comm_dup(incomm,&comm);
        MPI_Comm_rank(comm,&procno);
    }
    -Library() {
        MPI_Comm_free(&comm);
    }
    int communication_start();
    int communication_end();
};
```

*For the full source of this example, see section 7.8.3*

Note how the preceding example performs the `MPI_Comm_free` call in a C++ destructor.

```python
## commdup.py
class Library():
    def __init__(self,comm):
        # wrong: self.comm = comm
```
7.3 Sub-communicators

In many scenarios you divide a large job over all the available processors. However, your job may have two or more parts that can be considered as jobs by themselves. In that case it makes sense to divide your processors into subgroups accordingly.

Suppose for instance that you are running a simulation where inputs are generated, a computation is performed on them, and the results of this computation are analyzed or rendered graphically. You could then consider dividing your processors in three groups corresponding to generation, computation, rendering. As long as you only do sends and receives, this division works fine. However, if one group of processes needs to perform a collective operation, you don’t want the other groups involved in this. Thus, you really want the three groups to be distinct from each other.

In order to make such subsets of processes, MPI has the mechanism of taking a subset of `MPI_COMM_WORLD` (or other communicator) and turning that subset into a new communicator.

Now you understand why the MPI collective calls had an argument for the communicator: a collective involves all processes of that communicator. By making a communicator that contains a subset of all available processes, you can do a collective on that subset.

The usage is as follows:

- You create a new communicator with routines such as `MPI_Comm_dup` (section 7.2), `MPI_Comm_split` (section 7.4), `MPI_Comm_create` (section 7.5), `MPI_Intercomm_create` (section 7.6), `MPI_Comm_spawn` (section 8.1);
- you use that communicator for a while;

For the full source of this example, see section 7.8.4

```python
def __del__(self):
    if self.comm.Get_rank()==0:
        print(".. freeing communicator")
    self.comm.Free()
def communication_start(self):
    sendbuf = np.empty(1,dtype=int); sendbuf[0] = 37
    recvbuf = np.empty(1,dtype=int)
    self.requests[0] = self.comm.Isend( sendbuf, dest=other,tag=2 )
    self.requests[1] = self.comm.Irecv( recvbuf, source=other )

def communication_end(self):
    MPI.Request.Waitall(self.requests)
```

```python
mylibrary = Library(comm)
my_requests[0] = comm.Isend( sendbuffer, dest=other,tag=1 )
mylibrary.communication_start()
my_requests[1] = comm.Irecv( recvbuffer,source=other )
MPI.Request.Waitall(my_requests,my_status)
mylibrary.communication_end()
```
• and you call `MPI_Comm_free` when you are done with it; this also sets the communicator variable to `MPI_COMM_NULL`. A similar routine, `MPI_Comm_disconnect` waits for all pending communication to finish. Both are collective.

### 7.3.1 Scenario: distributed linear algebra

For *scalability* reasons, matrices should often be distributed in a 2D manner, that is, each process receives a subblock that is not a block of full columns or rows. This means that the processors themselves are, at least logically, organized in a 2D grid. Operations then involve reductions or broadcasts inside rows or columns. For this, a row or column of processors needs to be in a subcommunicator.

### 7.3.2 Scenario: climate model

A climate simulation code has several components, for instance corresponding to land, air, ocean, and ice. You can imagine that each needs a different set of equations and algorithms to simulate. You can then divide your processes, where each subset simulates one component of the climate, occasionally communicating with the other components.

### 7.3.3 Scenario: quicksort

The popular quicksort algorithm works by splitting the data into two subsets that each can be sorted individually. If you want to sort in parallel, you could implement this by making two subcommunicators, and sorting the data on these, creating recursively more subcommunicators.

### 7.3.4 Shared memory

There is an important application of communicator splitting in the context of one-sided communication, grouping processes by whether they access the same shared memory area; see section 12.1.

### 7.3.5 Process spawning

Finally, newly created communicators do not always need to be subset of the initial `MPI_COMM_WORLD`. MPI can dynamically spawn new processes (see chapter 8) which start in a `MPI_COMM_WORLD` of their own. However, another communicator will be created that spawns the old and new worlds so that you can communicate with the new processes.

### 7.4 Splitting a communicator

Above we saw several scenarios where it makes sense to divide `MPI_COMM_WORLD` into disjoint subcommunicators. The command `MPI_Comm_split` (figure 7.3) uses a ‘color’ to define these subcommunicators: all processes in the old communicator with the same color wind up in a new communicator together. The old communicator still exists, so processes now have two different contexts in which to communicate.
The ranking of processes in the new communicator is determined by a 'key' value: in a subcommunicator the process with lowest key is given the lowest rank, et cetera. Most of the time, there is no reason to use a relative ranking that is different from the global ranking, so the `MPI_Comm_rank` value of the global communicator is a good choice. Any ties between identical key values are broken by using the rank from the original communicator. Thus, specifying zero are the key will also retain the original process ordering.

Here is one example of communicator splitting. Suppose your processors are in a two-dimensional grid:

```c
MPI_Comm_rank( MPI_COMM_WORLD, &mytid );
proc_i = mytid % proc_column_length;
proc_j = mytid / proc_column_length;
```

You can now create a communicator per column:

```c
MPI_Comm column_comm;
MPI_Comm_split( MPI_COMM_WORLD, proc_j, mytid, &column_comm );
```

and do a broadcast in that column:

```c
MPI_Bcast( data, /* stuff */, column_comm );
```

Because of the SPMD nature of the program, you are now doing in parallel a broadcast in every processor column. Such operations often appear in dense linear algebra.

**Exercise 7.1.** Organize your processes in a grid, and make subcommunicators for the rows and columns. For this compute the row and column number of each process.
7. MPI topic: Communicators

In the row and column communicator, compute the rank. For instance, on a $2 \times 3$ processor grid you should find:

Global ranks: Ranks in row: Ranks in column:

```
0 1 2 0 1 2 0 0 0
3 4 5 0 1 2 1 1 1
```

Check that the rank in the row communicator is the column number, and the other way around.

Run your code on different number of processes, for instance a number of rows and columns that is a power of 2, or that is a prime number.

(There is a skeleton for this exercise under the name procgrid.)

**Python note 23: Comm split key is optional.** In Python, the `key` argument is optional:

```python
code:
# commsplit.py
mydata = procid

# communicator modulo 2
color = procid2
mod2comm = comm.Split(color)
procid2 = mod2comm.Get_rank()

# communicator modulo 4 recursively
color = procid2 % 2
mod4comm = mod2comm.Split(color)
procid4 = mod4comm.Get_rank()
```

**Output:**

```
Proc 0 -> 0 -> 0
Proc 2 -> 1 -> 0
Proc 6 -> 3 -> 1
Proc 4 -> 2 -> 1
Proc 3 -> 1 -> 0
Proc 7 -> 3 -> 1
Proc 1 -> 0 -> 0
Proc 5 -> 2 -> 1
```

**MPL note 55: Communicator splitting.** In MPL, splitting a communicator is done as one of the overloads of the communicator constructor;

```cpp
// commsplit.cpp
// create sub communicator modulo 2
int color2 = procn % 2;
mpl::communicator comm2( mpl::communicator::split, comm_world, color2 );
auto procn2 = comm2.rank();

// create sub communicator modulo 4 recursively
int color4 = procn2 % 2;
mpl::communicator comm4( mpl::communicator::split, comm2, color4 );
auto procn4 = comm4.rank();
```

For the full source of this example, see section 7.8.5

**MPL implementation note:** The `communicator::split` identifier is an object of class `communicator::split_tag`, itself is an otherwise empty subclass of `communicator`:

```
class split_tag {};
static constexpr split_tag split{};
```

There is also a routine `MPI_Comm_split_type` which uses a type rather than a key to split the communicator. We will see this in action in section 12.1.

As another example of communicator splitting, consider the recursive algorithm for *matrix transposition*. Processors are organized in a square grid. The matrix is divided on $2 \times 2$ block form.
7.5. Communicators and groups

You saw in section 7.4 that it is possible derive communicators that have a subset of the processes of another communicator. There is a more general mechanism, using MPI_Group objects.

Using groups, it takes three steps to create a new communicator:

1. Access the MPI_Group of a communicator object using MPI_Comm_group (figure 7.4).
2. Use various routines, discussed next, to form a new group.
3. Make a new communicator object from the group with MPI_Group, using MPI_Comm_create (figure 7.5)

Exercise 7.2. Implement a recursive algorithm for matrix transposition:

- Swap blocks (1, 2) and (2, 1); then
- Divide the processors into four subcommunicators, and apply this algorithm recursively on each;
- If the communicator has only one process, transpose the matrix in place.
7. MPI topic: Communicators

### Figure 7.6 MPI_Group_incl

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>group</td>
<td>number of elements in array</td>
<td>MPI_Group</td>
<td>TYPE(MPI_Group)</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>n</td>
<td>number of elements in array</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>ranks</td>
<td>ranks</td>
<td>ranks of processes in group to appear in newgroup</td>
<td>const int[]</td>
<td>INTEGER(n)</td>
<td></td>
</tr>
<tr>
<td>newgroup</td>
<td>newgroup</td>
<td>new group derived from above, in the order defined by ranks</td>
<td>MPI_Group*</td>
<td>TYPE(MPI_Group)</td>
<td></td>
</tr>
</tbody>
</table>

### Figure 7.7 MPI_Group_excl

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>group</td>
<td>number of elements in array</td>
<td>MPI_Group</td>
<td>TYPE(MPI_Group)</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>n</td>
<td>number of elements in array</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>ranks</td>
<td>ranks</td>
<td>array of integer ranks of processes in group not to appear in newgroup</td>
<td>const int[]</td>
<td>INTEGER(n)</td>
<td></td>
</tr>
<tr>
<td>newgroup</td>
<td>newgroup</td>
<td>new group derived from above, preserving the order defined by group</td>
<td>MPI_Group*</td>
<td>TYPE(MPI_Group)</td>
<td></td>
</tr>
</tbody>
</table>

Creating a new communicator from a group is collective on the old communicator. There is also a routine `MPI_Comm_create_group` that only needs to be called on the group that constitutes the new communicator.

#### 7.5.1 Process groups

Groups are manipulated with `MPI_Group_incl` (figure 7.6), `MPI_Group_excl` (figure 7.7), `MPI_Group_difference` and a few more.

```c
MPI_Comm_group (comm, group, ierr)
MPI_Comm_create (MPI_Comm comm, MPI_Group group, MPI_Comm newcomm, ierr)

MPI_Group_union (group1, group2, newgroup, ierr)
MPI_Group_intersection (group1, group2, newgroup, ierr)
MPI_Group_difference (group1, group2, newgroup, ierr)

MPI_Group_size (group, size, ierr)
MPI_Group_rank (group, rank, ierr)
```

Certain MPI types, `MPI_Win` and `MPI_File`, are created on a communicator. While you can not directly extract that communicator from the object, you can get the group with `MPI_Win_get_group` and `MPI_File_get_group`. 

276
7.6. Intercommunicators

7.5.2 Example

Suppose you want to split the world communicator into one manager process, with the remaining processes workers.

```c
// portapp.c
MPI_Comm comm_work;
{
    MPI_Group group_world, group_work;
    MPI_Comm_group( comm_world, &group_world );
    int manager[] = {0};
    MPI_Group_excl( group_world, 1, manager, &group_work );
    MPI_Comm_create( comm_world, group_work, &comm_work );
    MPI_Group_free( &group_world ); MPI_Group_free( &group_work );
}
```

7.6 Intercommunicators

In several scenarios it may be desirable to have a way to communicate between communicators. For instance, an application can have clearly functionally separated modules (preprocessor, simulation, post-processor) that need to stream data pairwise. In another example, dynamically spawned processes (section 8.1) get their own value of `MPI_COMM_WORLD`, but still need to communicate with the process(es) that spawned them. In this section we will discuss the inter-communicator mechanism that serves such use cases.

Communicating between disjoint communicators can of course be done by having a communicator that overlaps them, but this would be complicated: since the ‘inter’ communication happens in the overlap communicator, you have to translate its ordering into those of the two worker communicators. It would be easier to express messages directly in terms of those communicators, and this is what happens in an inter-communicator.

![Figure 7.2: Illustration of ranks in an intercommunicator setup](image)

A call to `MPI_Intercomm_create` (figure 7.8) involves the following communicators:
7. MPI topic: Communicators

**Figure 7.8 MPI_Intercomm_create**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MPI_Intercomm_create</td>
<td>local_comm</td>
<td>local intra-communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>local_leader</td>
<td>rank of local group leader in local_comm</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>peer_comm</td>
<td>`peer' communicator; significant only at the local_leader</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>remote_leader</td>
<td>rank of remote group leader in peer_comm; significant only at the local_leader</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tag</td>
<td>tag int</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>newintercomm</td>
<td>new inter-communicator</td>
<td>MPI_Comm*</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
</tbody>
</table>

- Two local communicators, which in this context are known as *intra-communicators*: one process in each will act as the local leader, connected to the remote leader;
- The *peer communicator*, often MPI_COMM_WORLD, that contains the local communicators;
- An *inter-communicator* that allows the leaders of the subcommunicators to communicate with the other subcommunicator.

Even though the intercommunicator connects only two processes, it is collective on the peer communicator.

### 7.6.1 Intercommunicator point-to-point

The local leaders can now communicate with each other.

- The sender specifies as target the local number of the other leader in the other sub-communicator;
- Likewise, the receiver specifies as source the local number of the sender in its sub-communicator.

In one way, this design makes sense: processors are referred to in their natural, local, numbering. On the other hand, it means that each group needs to know how the local ordering of the other group is arranged. Using a complicated key value makes this difficult.

```c
if (i_am_local_leader) {
    if (color==0) {
        int inter_target = local_number_of_other_leader;
        printf("[%d] sending interdata %e to %d\n", procno, interdata, inter_target);
        MPI_Send(&interdata,1,MPI_DOUBLE,inter_target,0,intercomm);
    } else {
        MPI_Status status;
        MPI_Recv(&interdata,1,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,intercomm,&status);
        int inter_source = status.MPI_SOURCE;
        printf("[%d] received interdata %e from %d\n", procno, interdata, inter_source);
        if (inter_source!=local_number_of_other_leader)
            fprintf(stderr,
```
7.6. Intercommunicators

Figure 7.9 MPI_Comm_get_parent

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_get_parent()</td>
<td>parent</td>
<td>the parent communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
</tbody>
</table>

"Got inter communication from unexpected %d; s/b %d\n", inter_source, local_number_of_other_leader);

For the full source of this example, see section 7.8.6

7.6.2 Intercommunicator collectives

The intercommunicator can be used in collectives such as a broadcast.

- In the sending group, the root process passes MPI_ROOT as ‘root’ value; all others use MPI_PROC_NULL.
- In the receiving group, all processes use a ‘root’ value that is the rank of the root process in the root group. Note: this is not the global rank!

Gather and scatter behave similarly; the allgather is different: all send buffers of group A are concatenated in rank order, and places on all processes of group B.

Intercommunicators can be used if two groups of process work asynchronously with respect to each other; another application is fault tolerance (section 15.5).

if (color==0) { // sending group: the local leader sends
  if (i_am_local_leader)
    root = MPI_ROOT;
  else
    root = MPI_PROC_NULL;
} else { // receiving group: everyone indicates leader of other group
  root = local_number_of_other_leader;
}
if (DEBUG) fprintf(stderr, "[\%d] using root value \%d\n", procno, root);
MPI_Bcast(&bcast_data,1,MPI_INT,root,intercomm);

For the full source of this example, see section 7.8.6

7.6.3 Intercommunicator querying

Some of the operations you have seen before for intra-communicators behave differently with intercommunicator:

- MPI_Comm_size returns the size of the local group, not the size of the intercommunicator.
- MPI_Comm_rank returns the rank in the local group.
- MPI_Comm_group returns the local group.
7. MPI topic: Communicators

**Figure 7.10 MPI_Comm_test_inter**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_test_inter (</td>
<td></td>
<td></td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>communicator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>flag</td>
<td>true if comm is an inter-communicator</td>
<td>int*</td>
<td>LOGICAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7.11 MPI_Comm_remote_size**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_remote_size (</td>
<td></td>
<td></td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>inter-communicator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>number of processes in the remote group of comm</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Python:

Intercomm.Get_remote_size(self)

Spawned processes can find their parent communicator with MPI_Comm_get_parent (figure 7.9) (see examples in section 8.1). On other processes this returns MPI_COMM_NULL.

Test whether a communicator is intra or inter: MPI_Comm_test_inter (figure 7.10).

**MPI_Comm_compare** works for intercommunicators.

Processes connected through an intercommunicator can query the size of the ‘other’ communicator with MPI_Comm_remote_size (figure 7.11). The actual group can be obtained with MPI_Comm_remote_group (figure 7.12).

Virtual topologies (chapter 11) cannot be created with an intercommunicator. To set up virtual topologies, first transform the intercommunicator to an intracommunicator with the function MPI_Intercomm_merge (figure 7.13).
7.7. Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

1. True or false: in each communicator, processes are numbered consecutively from zero.
2. If a process is in two communicators, it has the same rank in both.
3. Any communicator that is not MPI_COMM_WORLD is a strict subset of it.
4. The subcommunicators derived by MPI_Comm_split are disjoint.
5. If two processes have ranks \( p < q \) in some communicator, and they are in the same subcommunicator, then their ranks \( p', q' \) in the subcommunicator also obey \( p' < q' \).
7. MPI topic: Communicators

7.8 Sources used in this chapter

7.8.1 Listing of code header

7.8.2 Listing of code examples/mpi/c/commdupwrong.cxx

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

class library {
private:
    MPI_Comm comm;
    int procno,nprocs,other;
    MPI_Request request[2];
public:
    library(MPI_Comm incomm) {
        comm = incomm;
        MPI_Comm_rank(comm,&procno);
        other = 1-procno;
    }
    int communication_start();
    int communication_end();
};

int main(int argc,char **argv) {

    #include "globalinit.c"
    int other = 1-procno, sdata=5,rdata;
    MPI_Request request[2];
    MPI_Status status[2];

    if (procno>1) { MPI_Finalize(); return 0; }

    library my_library(comm);
    MPI_Isend(&sdata,1,MPI_INT,other,1,comm,&(request[0]));
    my_library.communication_start();
    MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,comm,&(request[1]));
    MPI_Waitall(2,request,status);
    my_library.communication_end();

    if ((status[1].MPI_TAG==2)
        printf("wrong!\n");  
        MPI_Finalize();
        return 0;
    }

    int library::communication_start() {
```
7.8. Sources used in this chapter

int sdata=6,rdata;
MPI_Isend(&sdata,1,MPI_INT,other,2,comm,&(request[0]));
MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,
   comm,&(request[1]));
return 0;
}

int library::communication_end() {
   MPI_Status status[2];
   MPI_Waitall(2,request,status);
   return 0;
}

7.8.3 Listing of code examples/mpi/c/commdupright.cxx

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

class library {  
private:
   MPI_Comm comm;
   int procno,nprocs,other;
   MPI_Request request[2];
public:
   library(MPI_Comm incomm) {
      MPI_Comm_dup(incomm,&comm);
      MPI_Comm_rank(comm,&procno);
      other = 1-procno;
   };
   -library() {  
      MPI_Comm_free(&comm);
   }
   int communication_start();
   int communication_end();
};

int main(int argc,char **argv) {

   #include "globalinit.c"

   int other = 1-procno, sdata=5,rdata;
   MPI_Request request[2];
   MPI_Status status[2];

   if (procno>1) { MPI_Finalize(); return 0; }

   library my_library(comm);
   MPI_Isend(&sdata,1,MPI_INT,other,1,comm,&(request[0]));
   my_library.communication_start();
7. MPI topic: Communicators

MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG, comm,&(request[1]));
MPI_Waitall(2,request,status);
my_library.communication_end();

if (status[1].MPI_TAG==2)
        printf("wrong!\n");

MPI_Finalize();
return 0;
}

int library::communication_start() {
    int sdata=6,rdata, ierr;
    MPI_Isend(&sdata,1,MPI_INT,other,2,comm,&(request[0]));
    MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,comm,&(request[1]));
    return 0;
}

int library::communication_end() { 
    MPI_Status status[2];
    int ierr;
    ierr = MPI_Waitall(2,request,status); CHK(ierr);
    return 0;
}

7.8.4 Listing of code examples/mpi/p/commdup.py

import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

other = nprocs-procid-1
my_requests = [ None ] * 2
my_status = [ MPI.Status() ] * 2
sendbuffer = np.empty(1,dtype=int)
recvbuffer = np.empty(1,dtype=int)

class Library():
    def __init__(self,comm):
        # wrong: self.comm = comm
        self.comm = comm.Dup()
        self.other = self.comm.Get_size()-self.comm.Get_rank()-1
        self.requests = [ None ] * 2
def __del__(self):
    if self.comm.Get_rank()==0: print(".. freeing communicator")
    self.comm.Free()

def communication_start(self):
    sendbuf = np.empty(1,dtype=int); sendbuf[0] = 37
    recvbuf = np.empty(1,dtype=int)
    self.requests[0] = self.comm.Isend( sendbuf, dest=other,tag=2 )
    self.requests[1] = self.comm.Irecv( recvbuf, source=other )

def communication_end(self):
    MPI.Request.Waitall(self.requests)

mylibrary = Library(comm)
my_requests[0] = comm.Isend( sendbuffer,dest=other,tag=1 )
mylibrary.communication_start()
my_requests[1] = comm.Irecv( recvbuffer,source=other )
MPI.Request.Waitall(my_requests,my_status)
mylibrary.communication_end()

if my_status[1].Get_tag()==2:
    print("Caught wrong message!")

7.8.5 Listing of code examples/mpi/ml/commsplit.cxx

#include <iostream>
using std::cout;
using std::endl;
#include <mpl/ml.hpp>

int main(int argc,char **argv) {

  const mpl::communicator &comm_world=mpl::environment::comm_world();
  auto procno = comm_world.rank();
  auto nprocs = comm_world.size();

  // create sub communicator modulo 2
  int color2 = procno % 2;
  mpl::communicator comm2( mpl::communicator::split, comm_world, color2 );
  auto procno2 = comm2.rank();

  // create sub communicator modulo 4 recursively
  int color4 = procno2 % 2;
  mpl::communicator comm4( mpl::communicator::split, comm2, color4 );
  auto procno4 = comm4.rank();

  int mod4ranks[nprocs];
  comm_world.gather( 0, procno4,mod4ranks );
  if (procno==0) {
    cout << "Ranks mod 4: ";
    for (int ip=0; ip<nprocs; ip++)
      cout << " " << mod4ranks[ip];
    cout << endl;
  }

Victor Eijkhout 285
if (procno/4!=procno4)
    printf("Error %d %d\n",procno,procno4);

if (procno==0)
    printf("Finished\n");

return 0;
}

7.8.6 Listing of code examples/mpi/c/intercomm.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#ifndef DEBUG
#define DEBUG 0
#endif

int main(int argc,char **argv) {
    #include "globalinit.c"

    if (nprocs<4) {
        fprintf(stderr,"This program needs at least four processes\n");
        return -1;
    }

    if (nprocs%2>0) {
        fprintf(stderr,"This program needs an even number of processes\n");
        return -1;
    }

    int color/colors=2;
    MPI_Comm_split_half_comm;

    int mydata = procno;
    // create sub communicator first & second half
    color = procno<nprocs/2 ? 0 : 1;
    int key=procno;
    if (color==0)
        // first half rotated
        key = ( procno+1 ) % (nprocs/2);
    else
        // second half numbered backwards
        key = nprocs-procno;
    MPI_Comm_split(MPI_COMM_WORLD,color,key,&split_half_comm);
    int sub_procno,sub_nprocs;
    MPI_Comm_rank(split_half_comm,&sub_procno);
    MPI_Comm_size(split_half_comm,&sub_nprocs);
    if (DEBUG) fprintf(stderr,"[%d] key=%d local rank: %d\n",procno,key,sub_procno);
7.8. Sources used in this chapter

```c
int
local_leader_in_inter_comm
= color==0 ? 2 : (sub_nprocs-2);
,
local_number_of_other_leader
= color==1 ? 2 : (sub_nprocs-2);

if (local_leader_in_inter_comm<0 || local_leader_in_inter_comm>=sub_nprocs) {
    fprintf(stderr,
        "[%d] invalid local member: %d
", procno,local_leader_in_inter_comm);
    MPI_Abort(2,comm);
}

int
global_rank_of_other_leader =
1 + ( procno<nprocs/2 ? nprocs/2 : 0 );

int
i_am_local_leader = sub_procno==local_leader_in_inter_comm,
inter_tag = 314;
if (i_am_local_leader)
    fprintf(stderr, 
        "[%d] creating intercomm with %d
", procno,global_rank_of_other_leader);
MPI_Comm intercomm;
MPI_Intercomm_create
/* local_comm: */ split_half_comm,
/* local_leader: */ local_leader_in_inter_comm,
/* peer_comm: */ MPI_COMM_WORLD,
/* remote_peer_rank: */ global_rank_of_other_leader,
/* tag: */ inter_tag,
/* newintercomm: */ &intercomm );
if (DEBUG) fprintf(stderr, 
    "[%d] intercomm created.
",procno);

if (i_am_local_leader) {
    int inter_rank,inter_size;
    MPI_Comm_size(intercomm,&inter_size);
    MPI_Comm_rank(intercomm,&inter_rank);
    if (DEBUG) fprintf(stderr, 
        "[%d] inter rank/size: %d/%d
",procno,inter_rank,inter_size);
}

double interdata=0.;
if (i_am_local_leader) {
    if (color==0) {
        interdata = 1.2;
        int inter_target = local_number_of_other_leader;
        printf("[%d] sending interdata 1.2 to %d
", procno,interdata,inter_target);
        MPI_Send(&interdata,1,MPI_DOUBLE,inter_target,0,intercomm);
    } else {
        MPI_Status status;
        MPI_Recv(&interdata,1,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,intercomm,&status);
    }
    }```

Victor Eijkhout
int inter_source = status.MPI_SOURCE;
    printf("[%d] received interdata %e from %d\n", procno, interdata, inter_source);
    if (inter_source!=local_number_of_other_leader) fprintf(stderr, "Got inter communication from unexpected %d; s/b %d\n", inter_source, local_number_of_other_leader);
}

int root; int bcast_data = procno;
if (color==0) { // sending group: the local leader sends
    if (i_am_local_leader)
        root = MPI_ROOT;
    else
        root = MPI_PROC_NULL;
} else { // receiving group: everyone indicates leader of other group
    root = local_number_of_other_leader;
}
if (DEBUG) fprintf(stderr,"[%d] using root value %d\n",procno,root);
MPI_Bcast(&bcast_data,1,MPI_INT,root,intercomm);
fprintf(stderr,"[%d] bcast data: %d\n",procno,bcast_data);

if (procno==0)
    fprintf(stderr,"Finished\n");
MPI_Finalize();
return 0;
Chapter 8

MPI topic: Process management

In this course we have up to now only considered the SPMD model of running MPI programs. In some rare cases you may want to run in an MPMD mode, rather than SPMD. This can be achieved either on the OS level, using options of the mpiexec mechanism, or you can use MPI’s built-in process management. Read on if you’re interested in the latter.

8.1 Process spawning

The first version of MPI did not contain any process management routines, even though the earlier PVM project did have that functionality. Process management was later added with MPI-2.

Unlike what you might think, newly added processes do not become part of MPI_COMM_WORLD; rather, they get their own communicator, and an inter-communicator (section 7.6) is established between this new group and the existing one. The first routine is MPI_Comm_spawn (figure 8.1), which tries to fire up multiple copies of a single named executable. Errors in starting up these codes are returned in an array of integers, or if you’re feeling sure of yourself, specify MPI_ERRCODES_IGNORE.

It is not immediately clear whether there is opportunity for spawning new executables; after all, MPI_COMM_WORLD contains all your available processors. You can probably tell your job starter to reserve space for a few extra processes, but that is installation-dependent (see below). However, there is a standard mechanism for querying whether such space has been reserved. The attribute MPI_UNIVERSE_SIZE, retrieved with MPI_Comm_get_attr (section 15.1.2), will tell you to the total number of hosts available.

If this option is not supported, you can determine yourself how many processes you want to spawn. If you exceed the hardware resources, your multi-tasking operating system (which is some variant of Unix for almost everyone) will use time-slicing to start the spawned processes, but you will not gain any performance.

Here is an example of a work manager. First we query how much space we have for new processes:

```c
int universe_size, *universe_size_attr, uflag;
MPI_Comm_get_attr( comm_world, MPI_UNIVERSE_SIZE,
                    &universe_size_attr, &uflag);
universe_size = *universe_size_attr;
```
8. MPI topic: Process management

Figure 8.1 MPI_Comm_spawn

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_spawn</td>
<td>command</td>
<td>name of program to be spawned</td>
<td>const char*</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>argv</td>
<td>arguments to command</td>
<td>char*[]</td>
<td>CHARACTER(*)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>maxprocs</td>
<td>maximum number of processes to start</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>info</td>
<td>a set of key-value pairs telling the runtime system where and how to start the processes</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>root</td>
<td>rank of process in which previous arguments are examined</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>intra-communicator containing group of spawning processes</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>intercomm</td>
<td>inter-communicator between original group and the newly spawned group</td>
<td>MPI_Comm*</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>array_of_errcodes</td>
<td>one code per process</td>
<td>int[]</td>
<td>INTEGER(*)</td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Intracomm.Spawn(self,
    command, args=None, int maxprocs=1, Info info=INFO_NULL,
    int root=0, errcodes=None)
returns an intracommunicator
```

```c
if (!uflag) universe_size = world_n;
int work_n = universe_size - world_n;
if (world_p==0) {
    printf("A universe of size %d leaves room for %d workers\n",
        universe_size, work_n);
    printf(".. spawning from %s\n", proname);
}
```

**For the full source of this example, see section 8.5.2**

(See section 15.1.2 for that dereference behavior.)

Use the flag to see if this option is supported:

```c
// spawnmanager.c
if (!flag) {
    if (manager_rank==0) {
        printf("This MPI does not support UNIVERSE_SIZE.\nHow many processes total?\n");
        scanf("%d", &universe_size);
    }
    MPI_Bcast(&universe_size,1,MPI_INTEGER,0,MPI_COMM_WORLD);
```

**For the full source of this example, see section 8.5.2**
8.1. Process spawning

Then we actually spawn the processes:

```c
const char *workerprogram = "./spawnapp";
MPI_Comm_spawn(workerprogram, MPI_ARGV_NULL, 
    work_n, MPI_INFO_NULL, 
    0, comm_world, &comm_inter, NULL);
```

For the full source of this example, see section 8.5.2

```python
try:
    universe_size = comm.Get_attr(MPI.UNIVERSE_SIZE)
    if universe_size is None:
        print("Universe query returned None")
        universe_size = nprocs + 4
    else:
        print("World has {} ranks in a universe of {}")
        .format(nprocs,universe_size))
except:
    print("Exception querying universe size")
universe_size = nprocs + 4
nworkers = universe_size - nprocs
itercomm = comm.Spawn("./spawn_worker.py", maxprocs=nworkers)
```

For the full source of this example, see section 8.5.3

A process can detect whether it was a spawning or a spawned process by using `MPI_Comm_get_parent`: the resulting intercommunicator is `MPI_COMM_NULL` on the parent processes.

```c
// spawnapp.c
MPI_Comm comm_parent;
MPI_Comm_get_parent(&comm_parent);
int is_child = (comm_parent!=MPI_COMM_NULL);
if (is_child) {
    int nworkers, workerno;
    MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
    MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
    printf("I detect I am worker %d/%d running on %s\n", 
        workerno, nworkers, procname);
}
```

For the full source of this example, see section 8.5.4

The spawned program looks very much like a regular MPI program, with its own initialization and finalize calls.

```c
// spawnworker.c
MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
MPI_Comm_get_parent(&parent);
```

For the full source of this example, see section 8.5.5

```python
## spawnworker.py
parentcomm = comm.Get_parent()
parents = parentcomm.Get_remote_size()
```
8. MPI topic: Process management

For the full source of this example, see section 8.5.6

Spawned processes wind up with a value of MPI_COMM_WORLD of their own, but managers and workers can find each other regardless. The spawn routine returns the intercommunicator to the parent; the children can find it through MPI_Comm_get_parent (section 7.6.3). The number of spawning processes can be found through MPI_Comm_remote_size on the parent communicator.

Running spawnapp with usize=12, wsize=4

```
%%
%% manager output
%%
A universe of size 12 leaves room for 8 workers
.. spawning from c209-026.frontera.tacc.utexas.edu
%%
%% worker output
%%
Worker deduces 8 workers and 4 parents
I detect I am worker 0/8 running on c209-027.frontera.tacc.utexas.edu
I detect I am worker 1/8 running on c209-027.frontera.tacc.utexas.edu
I detect I am worker 2/8 running on c209-027.frontera.tacc.utexas.edu
I detect I am worker 3/8 running on c209-027.frontera.tacc.utexas.edu
I detect I am worker 4/8 running on c209-028.frontera.tacc.utexas.edu
I detect I am worker 5/8 running on c209-028.frontera.tacc.utexas.edu
I detect I am worker 6/8 running on c209-028.frontera.tacc.utexas.edu
I detect I am worker 7/8 running on c209-028.frontera.tacc.utexas.edu
```

8.1.1 MPI startup with universe

You could start up a single copy of this program with

```
mpiexec -n 1 spawnmanager
```

but with a hostfile that has more than one host.

TACC note. Intel MPI requires you to pass an option -usize to mpiexec indicating the size of the comm universe. With the TACC jobs starter ibrun do the following:

```
export FI_MLX_ENABLE_SPAWN=yes
# specific
MY_MPIRUN_OPTIONS="-usize 8" ibrun -np 4 spawnmanager
# more generic
MY_MPIRUN_OPTIONS="-usize ${SLURM_NPROCS}" ibrun -np 4 spawnmanager
# using mpiexec:
mpiexec -np 2 -usize ${SLURM_NPROCS} spawnmanager
```

8.1.2 MPMD

Instead of spawning a single executable, you can spawn multiple with MPI_Comm_spawn_multiple. In that case a process can retrieve with the attribute MPI_APPNUM which of the executables it is; section 15.1.2.
8.2 Socket-style communications

It is possible to establish connections with running MPI programs that have their own world communicator.

- The server process establishes a port with `MPI_Open_port`, and calls `MPI_Comm_accept` to accept connections to its port.
- The client process specifies that port in an `MPI_Comm_connect` call. This establishes the connection.

### 8.2.1 Server calls

The server calls `MPI_Open_port` (figure 8.2), yielding a port name. Port names are generated by the system and copied into a character buffer of length at most `MPI_MAX_PORT_NAME`.

The server then needs to call `MPI_Comm_accept` (figure 8.3) prior to the client doing a connect call. This is collective over the calling communicator. It returns an intercommunicator (section 7.6) that allows communication with the client.

```c
MPI_Comm intercomm;
char myport[MPI_MAX_PORT_NAME];
MPI_Open_port( MPI_INFO_NULL,myport );
int portlen = strlen(myport);
MPI_Send( myport,portlen+1,MPI_CHAR,1,0,comm_world );
printf("Host sent port <<%s>>\n",myport);
MPI_Comm_accept( myport,MPI_INFO_NULL,0,comm_self,&intercomm );
printf("host accepted connection\n");
```
8. MPI topic: Process management

### Figure 8.4 MPI_Comm_connect

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_connect (</td>
<td>port_name</td>
<td>network address</td>
<td>const char*</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>info</td>
<td>implementation-dependent information</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>root</td>
<td>rank in comm of root node</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>intra-communicator over which call is collective</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>newcomm</td>
<td>inter-communicator with server as remote group</td>
<td>MPI_Comm*</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
</tbody>
</table>

The port can be closed with MPI_Close_port.

#### 8.2.2 Client calls

After the server has generated a port name, the client needs to connect to it with MPI_Comm_connect (figure 8.4), again specifying the port through a character buffer. The connect call is collective over its communicator.

```c
    char myport[MPI_MAX_PORT_NAME];
    if (work_p==0) {
        MPI_Recv( myport,MPI_MAX_PORT_NAME,MPI_CHAR,
                   MPI_ANY_SOURCE,0, comm_world,MPI_STATUS_IGNORE );
        printf("Worker received port <<%s>>\n",myport);
    }
    MPI_Bcast( myport,MPI_MAX_PORT_NAME,MPI_CHAR,0,comm_work );

    /*
     * The workers collective connect over the inter communicator
     */
    MPI_Comm intercomm;
    MPI_Comm_connect( myport,MPI_INFO_NULL,0,comm_work,&intercomm );
    if (work_p==0) {
        int manage_n;
        MPI_Comm_remote_size(intercomm,&manage_n);
        printf("%d workers connected to %d managers\n",work_n,manage_n);
    }
```

For the full source of this example, see section 8.5.7

If the named port does not exist (or has been closed), MPI_Comm_connect raises an error of class MPI_ERR_PORT.

The client can sever the connection with MPI_Comm_disconnect.

Running the above code on 5 processes gives:

```
# exchange port name:
Host sent port <<tag#0$0FA#000010e1:0001cde9:0001cdeed$rdma_port#1024$rdma_host#10:16:225:0:1:205:199:254:
```
8.2. Socket-style communications

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Publish_name (</td>
<td>service_name</td>
<td>a service name to associate with the port</td>
<td>const char*</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>info</td>
<td>implementation-specific information</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>port_name</td>
<td>a port name</td>
<td>const char*</td>
<td>CHARACTER</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Worker received port <<tag#0$OFA#000010e1:0001cde9:0001cdee$rdma_port#1024$rdma_host#10:16:225:0:1:205:199:254:128:0:0:0:0:0:0$>>

# Comm accept/connect
host accepted connection 4 workers connected to 1 managers

# Send/recv over the intercommunicator
Manager sent 4 items over intercomm
Worker zero received data

8.2.3 Published service names

More elegantly than the port mechanism above, it is possible to publish a named service, with MPI_Publish_name (figure 8.5), which can then be discovered by other processes.

```c
// publishapp.c
MPI_Comm intercomm;
char myport[MPI_MAX_PORT_NAME];
MPI_Open_port( MPI_INFO_NULL, myport );
MPI_Publish_name( service_name, MPI_INFO_NULL, myport );
MPI_Comm_accept( myport, MPI_INFO_NULL, 0, comm_self, &intercomm );
```

For the full source of this example, see section 8.5.8

Worker processes connect to the intercommunicator by

```c
char myport[MPI_MAX_PORT_NAME];
MPI.Lookup_name( service_name, MPI_INFO_NULL, myport );
MPI_Comm intercomm;
MPI_Comm_connect( myport, MPI_INFO_NULL, 0, comm_work, &intercomm );
```

For the full source of this example, see section 8.5.8

For this it is necessary to have a name server running.

Intel note. Start the hydra name server and use the corresponding mpi starter:

```
hydra_nameserver &
MPIEXEC=mpiexec.hydra
```

There is an environment variable, but that doesn’t seem to be needed.
8. MPI topic: Process management

**Figure 8.6 MPI_Unpublish_name**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Unpublish_name</td>
<td>service_name</td>
<td>a service name</td>
<td>const char*</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>info</td>
<td>implementation-specific information</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>port_name</td>
<td>a port name</td>
<td>const char*</td>
<td>CHARACTER</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8.7 MPI_Comm_join**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_join</td>
<td>fd</td>
<td>socket file descriptor</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>intercomm</td>
<td>new inter-communicator</td>
<td>MPI_Comm*</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
</tbody>
</table>

export I_MPI_HYDRA_NAMESERVER=`hostname`:8008

It is also possible to specify the name server as an argument to the job starter.

At the end of a run, the service should be unpublished with **MPI_Unpublish_name** (figure 8.6). Unpublishing a nonexisting or already unpublished service gives an error code of **MPI_ERR_SERVICE**.

MPI provides no guarantee of fairness in servicing connection attempts. That is, connection attempts are not necessarily satisfied in the order in which they were initiated, and competition from other connection attempts may prevent a particular connection attempt from being satisfied.

### 8.2.4 Unix sockets

It is also possible to create an intercommunicator from a Unix socket with **MPI_Comm_join** (figure 8.7).

### 8.3 Sessions

The most common way of initializing MPI, with **MPI_Init** (or **MPI_Init_thread**) and **MPI_Finalize**, is known as the world model. This model suffers from some disadvantages:

1. There is no error handling during **MPI_Init**.
2. If multiple libraries are active, they can not initialize or finalize MPI, but have to base themselves on subcommunicators; section 7.2.2.
3. A library can’t even

   ```c
   MPI_Initialized(&flag);
   if (!flag) MPI_Init(0,0);
   ```

   if it is running in a multi-threaded environment.
The following material is for the recently released MPI-4 standard and may not be supported yet.

In addition to the world, where all MPI is bracketed by `MPI_Init` (or `MPI_Init_thread`) and `MPI_Finalize`, there is the session model, where entities such as libraries can start/end their MPI session independently.

The two models can be used in the same program, but there are limitations on how they can mix.

### 8.3.1 World model versus sessions model

The world model of using MPI can be described as:

1. There is a single call to `MPI_Init` or `MPI_Init_thread`;
2. There is a single call to `MPI_Finalize`;
3. With very few exceptions, all MPI calls appear in between the initialize and finalize calls.

This has some problems:

- MPI can not be finalized and restarted;
- there is no threadsafe way of initializing MPI.

In the session model, the world model has become a single session, and it is possible to start multiple sessions, each on their own set of processes, possibly identical or overlapping.

You can not mix in a single call objects from different sessions, from a session and from the world model, or from a session and from `MPI_Comm_get_parent` or `MPI_Comm_join`.

### 8.3.2 Session creation

An MPI session is initialized and finalized with `MPI_Session_init` and `MPI_Session_finalize`, somewhat similar to `MPI_Init` and `MPI_Finalize`.

```c
MPI_Session the_session;
MPI_Session_init ( session_request_info, MPI_ERRORS_ARE_FATAL, &the_session );
MPI_Session_finalize ( &the_session );
```

This call is thread-safe, in view of the above reasoning.

### 8.3.2.1 Session info

The `MPI_Info` object can be null, or it can be used to request a threading level:

```c
// session.c
MPI_Info session_request_info = MPI_INFO_NULL;
MPI_Info_create(&session_request_info);
char thread_key[] = "mpi_thread_support_level";
MPI_Info_set(session_request_info, thread_key,"MPI_THREAD_MULTIPLE");
```

Other info keys can be implementation-dependent, but the key `thread_support` is pre-defined.

Info keys can be retrieved again with `MPI_Session_get_info`.

```c
MPI_Info session_request_info = NULL;
MPI_Info_create(&session_request_info);
char thread_key[] = "mpi_thread_support_level";
MPI_Info_set(session_request_info, thread_key,"MPI_THREAD_MULTIPLE");
```
8. MPI topic: Process management

8.3.2.2 Session error handler

The error handler argument accepts a pre-defined error handler (section 15.2.2) or one create by `MPI_Session_create_errhandler`.

8.3.3 Process sets and communicators

A session has a number of process sets. Process sets are indicated with a Uniform Resource Identifier (URI), where the URIs `mpi://WORLD` and `mpi://SELF` are always defined.

You query the ‘psets’ with `MPI_Session_get_num_psets` and `MPI_Session_get_nth_pset`:

```c
int npsets;
MPI_Session_get_num_psets( the_session, MPI_INFO_NULL, &npsets );
if (mainproc) printf("Number of process sets: %d\n", npsets);
for (int ipset=0; ipset<npsets; ipset++) {
    int len_pset; char name_pset[MPI_MAX_PSET_NAME_LEN];
    MPI_Session_get_nth_pset( the_session, MPI_INFO_NULL, ipset, &len_pset, name_pset );
    if (mainproc)
        printf("Process set %2d: <<%s>>\n", ipset, name_pset);
```

The following partial code creates a communicator equivalent to `MPI_COMM_WORLD` in the session model:

```c
MPI_Group world_group = MPI_GROUP_NULL;
MPI_Comm world_comm = MPI_COMM_NULL;
MPI_Group_from_session_pset( the_session, world_name, &world_group );
MPI_Comm_create_from_group( world_group, "victor-code-session.c",
                          MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL,
                          &world_comm );
MPI_Group_free( &world_group );
int procid = -1, nprocs = 0;
MPI_Comm_size(world_comm, &nprocs);
MPI_Comm_rank(world_comm, &procid);
```

However, comparing communicators (with `MPI_Comm_compare`) from the session and world model, or from different sessions, is undefined behavior.

Get the info object (section 15.1.1) from a process set: `MPI_Session_get_pset_info`. This info object always has the key `mpi_size`. 
8.4. Functionality available outside init/finalize

8.3.4 Example

As an example of the use of sessions, we declare a library class, where each library object starts and ends its own session:

```cpp
// sessionlib.cxx
class Library {
private:
    MPI_Comm world_comm; MPI_Session session;
public:
    Library() {
        MPI_Info info = MPI_INFO_NULL;
        MPI_Session_init
            ( MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL, &session );
        char world_name[] = "mpi://WORLD";
        MPI_Group world_group;
        MPI_Group_from_session_pset
            ( session, world_name, &world_group );
        MPI_Comm_create_from_group
            ( world_group, "world-session", MPI_INFO_NULL, MPI_ERRORS_ARE_FATAL,
              &world_comm );
        MPI_Group_free( &world_group );
    }
    ~Library() { MPI_Session_finalized(&session); }
    int compute(int x) {
```

Now we create a main program, using the world model, which activates two libraries, passing data to them by parameter:

```cpp
int main(int argc, char **argv) {
    Library lib1, lib2;
    MPI_Init(0,0);
    MPI_Comm world = MPI_COMM_WORLD;
    int procnos, nprocs;
    MPI_Comm_rank(world,&procnos);
    MPI_Comm_size(world,&nprocs);
    auto sum1 = lib1.compute(procnos);
    auto sum2 = lib2.compute(procnos+1);
}
```

Note that no mpi calls will go between main program and either of the libraries, or between the two libraries, but this seems to make sense in this scenario.

*End of MPI-4 material*

8.4 Functionality available outside init/finalize

MPI_Initiated MPI_Finalized MPI_Get_version MPI_Get_library_version MPI_Info_create
MPI_Info_create_env MPI_Info_set MPI_Info_delete MPI_Info_get MPI_Info_get_valuen
MPI_Info_get_nkeys MPI_Info_get_nthkey MPI_Info_dup MPI_Info_free MPI_Info_f2c MPI_Info_c2f
8. MPI topic: Process management

MPI_Session_create_errhandler MPI_Session_call_errhandler MPI_Errhandler_free MPI_Errhandler_f2c
MPI_Errhandler_c2f MPI_Error_string MPI_Error_class

Also all routines starting with MPI_Txxx.
8.5 Sources used in this chapter

8.5.1 Listing of code header

8.5.2 Listing of code examples/mpi/c/spawnmanager.c

```c
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    #define ASSERT(p) if (!(p)) {printf("Assertion failed for proc %d at line %d\n",procno, __LINE__); return -1;}
    #define ASSERTm(p,m) if (!(p)) {printf("Message<<%s>> for proc %d at line %d\n",m,procno, __LINE__); return -1;}

    MPI_Comm comm;
    int procno=-1, nprocs, err;
    MPI_Init(&argc, &argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_rank(comm, &procno);
    MPI_Comm_size(comm, &nprocs);
    //MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN);
    
    /*
     * To investigate process placement, get host name
     */
    {
        int namelen = MPI_MAX_PROCESSOR_NAME;
        char procname[namelen];
        MPI_Get_processor_name(procname, &namelen);
        printf("[%d] manager process runs on <<%s>>\n", procno, procname);
    }

    int world_size, manager_rank, universe_size, *universe_sizep, flag;

    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &manager_rank);
    MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, (void**)&universe_sizep, &flag);
    
    if (!flag) {
        if (manager_rank==0) {
            printf("This MPI does not support UNIVERSE_SIZE.\nHow many processes total?\n");
            scanf("%d", &universe_size);
        }
        MPI_Bcast(&universe_size, 1, MPI_INTEGER, 0, MPI_COMM_WORLD);
    } else {
        universe_size = *universe_sizep;
        if (manager_rank==0)
            printf("Universe size deduced as %d\n", universe_size);
        ASSERTm(universe_size>world_size,"No room to start workers");
    }

    int main(int argc, char *argv[])
    {
        #define ASSERT(p) if (!(p)) {printf("Assertion failed for proc %d at line %d\n",procno, __LINE__); return -1;}
        #define ASSERTm(p,m) if (!(p)) {printf("Message<<%s>> for proc %d at line %d\n",m,procno, __LINE__); return -1;}

        MPI_Comm comm;
        int procno=-1, nprocs, err;
        MPI_Init(&argc, &argv);
        comm = MPI_COMM_WORLD;
        MPI_Comm_rank(comm, &procno);
        MPI_Comm_size(comm, &nprocs);
        //MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN);
    
    /*
     * To investigate process placement, get host name
     */
    {
        int namelen = MPI_MAX_PROCESSOR_NAME;
        char procname[namelen];
        MPI_Get_processor_name(procname, &namelen);
        printf("[%d] manager process runs on <<%s>>\n", procno, procname);
    }

    int world_size, manager_rank, universe_size, *universe_sizep, flag;

    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &manager_rank);
    MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, 
        (void**)&universe_sizep, &flag);
    
    if (!flag) {
        if (manager_rank==0) {
            printf("This MPI does not support UNIVERSE_SIZE.\nHow many processes total?\n");
            scanf("%d", &universe_size);
        }
        MPI_Bcast(&universe_size, 1, MPI_INTEGER, 0, MPI_COMM_WORLD);
    } else {
        universe_size = *universe_sizep;
        if (manager_rank==0)
            printf("Universe size deduced as %d\n", universe_size);
        ASSERTm(universe_size>world_size,"No room to start workers");
    }
```

Victor Eijkhout
int nworkers = universe_size-world_size;

/*
 * Now spawn the workers. Note that there is a run-time determination
 * of what type of worker to spawn, and presumably this calculation must
 * be done at run time and cannot be calculated before starting
 * the program. If everything is known when the application is
 * first started, it is generally better to start them all at once
 * in a single MPI_COMM_WORLD.
 */

if (manager_rank==0)
   printf("Now spawning %d workers\n",nworkers);
const char *worker_program = "spawnworker";
int errorcodes[nworkers];
MPI_Comm inter_to_workers; /* intercommunicator */
MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, nworkers,
                  MPI_INFO_NULL, 0, MPI_COMM_WORLD, &inter_to_workers,
                  errorcodes);
for (int ie=0; ie<nworkers; ie++)
    if (errorcodes[ie]!=0)
        printf("Error %d in spawning worker %d\n",errorcodes[ie],ie);

/*
 * Parallel code here. The communicator "inter_to_workers" can be used
 * to communicate with the spawned processes, which have ranks 0,..
 * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
 * "inter_to_workers".
 */

MPI_Finalize();
return 0;
}

8.5.3 Listing of code examples/mpi/p/spawnmanager.py

import numpy as np
import random # random.randint(1,N), random.random()
import sys

from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()

try :
    universe_size = comm.Get_attr(MPI.UNIVERSE_SIZE)
    if universe_size is None:
        print("Universe query returned None")
        universe_size = nprocs + 4
    else:
        # code...

else:
8.5. Sources used in this chapter

```python
print("World has {} ranks in a universe of {}")
.print(nprocs,universe_size)
except :
    print("Exception querying universe size")
    universe_size = nprocs + 4
nworkers = universe_size - nprocs
itercomm = comm.Spawn("./spawn_worker.py", maxprocs=nworkers)

8.5.4 Listing of code examples/mpi/c/spawnapp.c

#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <mpi.h>

int main(int argc,char **argv) {

    MPI_Init(&argc,&argv);
    MPI_Comm
        comm_world = MPI_COMM_WORLD,
        comm_self = MPI_COMM_SELF,
        comm_inter;
    int world_p,world_n;
    MPI_Comm_size(comm_world,&world_n);
    MPI_Comm_rank(comm_world,&world_p);
    int len = MPI_MAX_PROCESSOR_NAME;
    char procname[MPI_MAX_PROCESSOR_NAME];
    MPI_Get_processor_name(procname,&len);

    MPI_Comm comm_parent;
    MPI_Comm_get_parent(&comm_parent);
    int is_child = (comm_parent!=MPI_COMM_NULL);
    if (is_child) {
        int nworkers,workerno;
        MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
        MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
        printf("I detect I am worker %d/%d running on %s\n",
                workerno,nworkers,procname);

        int remotesize;
        MPI_Comm_remote_size(comm_parent, &remotesize);
        if (workerno==0) {
            printf("Worker deduces %d workers and %d parents\n",nworkers,remotesize);
        }
    } else { /*
    * Detect how many workers we can spawn
    */
        int universe_size, *universe_size_attr,uflag;
        MPI_Comm_get_attr
```
8. MPI topic: Process management

```
(comm_world,MPI_UNIVERSE_SIZE,
 &universe_size_attr,&uflag);
universe_size = *universe_size_attr;
if (!uflag) universe_size = world_n;
int work_n = universe_size - world_n;
if (world_p==0) {
    printf("A universe of size %d leaves room for %d workers\n",
        universe_size,work_n);
    printf("... spawning from %s\n",procname);
}
if (work_n<=0)
    MPI_Abort(comm_world,1);
const char *workerprogram = "./spawnapp";
MPI_Comm_spawn(workerprogram,MPI_ARGV_NULL,
    work_n,MPI_INFO_NULL,
    0,comm_world,&comm_inter,NULL);
}
MPI_Finalize();
return 0;
```

8.5.5 Listing of code examples/mpi/c/spawnworker.c

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    #define ASSERT(p) if (!(p)) {printf("Assertion failed for proc %d at line %d\n",procno,__LINE__); return -1;}
    #define ASSERTm(p,m) if (!(p)) {printf("Message<<%s>> for proc %d at line %d\n",m,procno,__LINE__); return -1;}

    MPI_Comm comm;
    int procno=-1,nprocs,err;
    MPI_Init(&argc,&argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_rank(comm,&procno);
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN);

    int remotesize,nworkers,workerno;
    MPI_Comm parent;

    MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
    MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
    MPI_Comm_get_parent(&parent);
    ASSERTm(parent!=MPI_COMM_NULL,"No parent!");
```
8.5. Sources used in this chapter

8.5.6 Listing of code examples/mpi/p/spawnworker.py

```python
import numpy as np
import random # random.randint(1,N), random.random()

from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()

if procid==0:
    print("#workers:",nprocs)

parentcomm = comm.Get_parent()
nparents = parentcomm.Get_remote_size()

print("#parents=",nparents)
```

8.5.7 Listing of code examples/mpi/c/portapp.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <mpi.h>

Victor Eijkhout
```

Victor Eijkhout
int main(int argc,char **argv) {

MPI_Init(&argc,&argv);
MPI_Comm
    comm_world = MPI_COMM_WORLD,
    comm_self = MPI_COMM_SELF;

int world_p,world_n;
MPI_Comm_size(comm_world,&world_n);
MPI_Comm_rank(comm_world,&world_p);

/*
 * Set up a communicator for all the worker ranks
 */
MPI_Comm comm_work;
{
    MPI_Group group_world,group_work;
    MPI_Comm_group( comm_world,&group_world );
    int manager[] = {0};
    MPI_Group_excl( group_world,1,manager,&group_work );
    MPI_Comm_create( comm_world,group_work,&comm_work );
    MPI_Group_free( &group_world ); MPI_Group_free( &group_work );
}

if (world_p==0) {
    /*
     * On world process zero open a port, and
     * send its name to world process 1,
     * which is zero in the worker comm.
     */
    MPI_Comm intercomm;
    char myport[MPI_MAX_PORT_NAME];
    MPI_Open_port( MPI_INFO_NULL,myport );
    int portlen = strlen(myport);
    MPI_Send( myport,portlen+1,MPI_CHAR,1,0,comm_world );
    printf("Host sent port <<%s>>\n",myport);
    MPI_Comm_accept( myport,MPI_INFO_NULL,0,comm_self,&intercomm );
    printf("host accepted connection\n");

    /*
     * After the workers have accept the connection,
     * we can talk over the inter communicator
     */
    int work_n;
    MPI_Comm_remote_size(intercomm,&work_n);
    double work_data[work_n];
    MPI_Send( work_data,work_n,MPI_DOUBLE,
             /* to rank zero of worker comm */ 0,0,intercomm );
    printf("Manager sent %d items over intercomm\n",work_n);
    /*
     * After we're done, close the port
     */
    MPI_Close_port(myport);
}

8.5. Sources used in this chapter

} else {

    int work_p, work_n;
    MPI_Comm_size( comm_work, &work_n );
    MPI_Comm_rank( comm_work, &work_p );
    /*
     * In the workers communicator, rank 0
     * (which is 1 in the global)
     * receives the port name and passes it on.
     */
    char myport[MPI_MAX_PORT_NAME];
    if (work_p==0) {
        MPI_Recv( myport, MPI_MAX_PORT_NAME, MPI_CHAR,
                   MPI_ANY_SOURCE, 0, comm_world, MPI_STATUS_IGNORE );
        printf("Worker received port "
                ""<%s>"",myport);
    }
    MPI_Bcast( myport, MPI_MAX_PORT_NAME, MPI_CHAR, 0, comm_work );

    /*
     * The workers collective connect over the inter communicator
     */
    MPI_Comm intercomm;
    MPI_Comm_connect( myport, MPI_INFO_NULL, 0, comm_work, &intercomm );
    if (work_p==0) {
        int manage_n;
        MPI_Comm_remote_size(intercomm, &manage_n);
        printf("%d workers connected to %d managers\n", work_n, manage_n);
    }

    /*
     * The local leader receives work from the manager
     */
    if (work_p==0) {
        double work_data[work_n];
        MPI_Status work_status;
        MPI_Recv( work_data, work_n, MPI_DOUBLE, /* from rank zero of manager comm */ 0, 0, intercomm, &work_status );
        int work_count;
        MPI_Get_count(&work_status, MPI_DOUBLE, &work_count);
        printf("Worker zero received %d data items from manager\n", work_count);
    }

    /*
     * After we're done, close the connection
     */
    MPI_Close_port(myport);
}

MPI_Finalize();

return 0;
}
8. MPI topic: Process management

8.5.8 Listing of code examples/mpi/c/publishapp.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <mpi.h>

int main(int argc,char **argv) {

    MPI_Init(&argc,&argv);
    MPI_Comm
        comm_world = MPI_COMM_WORLD,
        comm_self = MPI_COMM_SELF;

    int world_p,world_n;
    MPI_Comm_size(comm_world,&world_n);
    MPI_Comm_rank(comm_world,&world_p);

    /*
     * Set up a communicator for all the worker ranks
     */
    MPI_Comm comm_work;
    {
        MPI_Group group_world,group_work;
        MPI_Comm_group( comm_world,&group_world );
        int manager[] = {0};
        MPI_Group_excl( group_world,1,manager,&group_work );
        MPI_Comm_create( comm_world,group_work,&comm_work );
        MPI_Group_free( &group_world ); MPI_Group_free( &group_work );
    }

    char
        service_name[] = "exampleservice";
    if (world_p==0) {
        /*
         * On world process zero open a port, and
         * send its name to world process 1,
         * which is zero in the worker comm.
         */
        MPI_Comm intercomm;
        char myport[MPI_MAX_PORT_NAME];
        MPI_Open_port( MPI_INFO_NULL,myport );
        MPI_Publish_name( service_name, MPI_INFO_NULL, myport );
        MPI_Comm_accept( myport,MPI_INFO_NULL,0,comm_self,&intercomm );
        printf("Manager accepted connection on port <<%s>>\n",myport);

        /*
         * After the workers have accept the connection,
         * we can talk over the inter communicator
         */
        int work_n;
        MPI_Comm_remote_size(intercomm,&work_n);
        double work_data[work_n];
        MPI_Send( work_data,work_n,MPI_DOUBLE,
```
8.5. Sources used in this chapter

/* to rank zero of worker comm */ 0,0,intercomm);
printf("Manager sent %d items over intercomm\n",work_n);
/
 * After we're done, close the port, and unpublish the service
 */
MPI_Close_port(mypnrt);
MPI_Unpublish_name(service_name, MPI_INFO_NULL, myport);
} else {
/
 * See if we can find the service
 */
char myport[MPI_MAX_PORT_NAME];
MPI_Lookup_name(service_name,MPI_INFO_NULL,myport);
/
 * The workers collective connect over the inter communicator
 */
MPI_Comm intercomm;
MPI_Comm_connect(mypnrt,MPI_INFO_NULL,0,comm_work,&intercomm);

int work_p,work_n;
MPI_Comm_size(comm_work,&work_n);
MPI_Comm_rank(comm_work,&work_p);

if (work_p==0) {
    int manage_n;
    MPI_Comm_remote_size(intercomm,&manage_n);
    printf("%d workers connected to %d managers\n",work_n,manage_n);
}
/
 * The local leader receives work from the manager
 */
if (work_p==0) {
    double work_data[work_n];
    MPI_Status work_status;
    MPI_Recv(work_data,work_n,MPI_DOUBLE,
    / from rank zero of manager comm */ 0,0,intercomm,&work_status);
    int work_count;
    MPI_Get_count(&work_status,MPI_DOUBLE,&work_count);
    printf("Worker zero received %d data items from manager\n",work_count);
}
/
 * After we're done, close the connection
 */
MPI_Close_port(mypnrt);
}

MPI_Finalize();

return 0;

Victor Eijkhout 309
8. MPI topic: Process management

>
Chapter 9

MPI topic: One-sided communication

Above, you saw point-to-point operations of the two-sided type: they require the co-operation of a sender and receiver. This co-operation could be loose: you can post a receive with MPI_ANY_SOURCE as sender, but there had to be both a send and receive call. This two-sidedness can be limiting. Consider code where the receiving process is a dynamic function of the data:

\[
\begin{align*}
  x &= f(); \\
  p &= \text{hash}(x); \\
  \text{MPI\_Send}(x, /* to: */ p); 
\end{align*}
\]

The problem is now: how does \( p \) know to post a receive, and how does everyone else know not to?

In this section, you will see one-sided communication routines where a process can do a ‘put’ or ‘get’ operation, writing data to or reading it from another processor, without that other processor’s involvement.

In one-sided MPI operations, known as Remote Memory Access (RMA) operations in the standard, or as Remote Direct Memory Access (RDMA) in other literature, there are still two processes involved: the origin, which is the process that originates the transfer, whether this is a ‘put’ or a ‘get’, and the target whose memory is being accessed. Unlike with two-sided operations, the target does not perform an action that is the counterpart of the action on the origin.

That does not mean that the origin can access arbitrary data on the target at arbitrary times. First of all, one-sided communication in MPI is limited to accessing only a specifically declared memory area on the target: the target declares an area of user-space memory that is accessible to other processes. This is known as a window. Windows limit how origin processes can access the target’s memory: you can only ‘get’ data from a window or ‘put’ it into a window; all the other memory is not reachable from other processes. On the origin there is no such limitation; any data can function as the source of a ‘put’ or the recipient of a ‘get’ operation.

The alternative to having windows is to use distributed shared memory or virtual shared memory: memory is distributed but acts as if it shared. The so-called Partitioned Global Address Space (PGAS) languages such as Unified Parallel C (UPC) use this model. The MPI RMA model makes it possible to lock a window which makes programming slightly more cumbersome, but the implementation more efficient.

Within one-sided communication, MPI has two modes: active RMA and passive RMA. In active RMA, or active target synchronization, the target sets boundaries on the time period (the ‘epoch’) during which
9. MPI topic: One-sided communication

its window can be accessed. The main advantage of this mode is that the origin program can perform
many small transfers, which are aggregated behind the scenes. Active RMA acts much like asynchronous
transfer with a concluding MPI_Waitall.

In passive RMA, or passive target synchronization, the target process puts no limitation on when its window
can be accessed. (PGAS languages such as UPC are based on this model: data is simply read or written at
will.) While intuitively it is attractive to be able to write to and read from a target at arbitrary time, there
are problems. For instance, it requires a remote agent on the target, which may interfere with execution
of the main thread, or conversely it may not be activated at the optimal time. Passive RMA is also very
hard to debug and can lead to strange deadlocks.

9.1 Windows

In one-sided communication, each processor can make an area of memory, called a window, available to
one-sided transfers. This is stored in a variable of type MPI_Win. A process can put an arbitrary item from
its own memory (not limited to any window) to the window of another process, or get something from
the other process' window in its own memory.

A window can be characterized as follows:

- The window is defined on a communicator, so the create call is collective; see figure 9.1.
- The window size can be set individually on each process. A zero size is allowed, but since win-
dow creation is collective, it is not possible to skip the create call.
- You can set a 'displacement unit' for the window: this is a number of bytes that will be used as
  the indexing unit. For example if you use sizeof(double) as the displacement unit, an MPI_Put
to location 8 will go to the 8th double. That's easier than having to specify the 64th byte.
- The window is the target of data in a put operation, or the source of data in a get operation; see
  figure 9.2.
- There can be memory associated with a window, so it needs to be freed explicitly with MPI_Win_free.

The typical calls involved are:

```c
MPI_Info info;
MPI_Win window;
```
9.1. Windows

Figure 9.1 MPI_Win_create

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_create</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Win_create_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>base</td>
<td>initial address of window</td>
<td>void*</td>
<td>TYPE(*)</td>
<td>TYPE(MPI_Win)</td>
<td>IN</td>
</tr>
<tr>
<td>size</td>
<td>size of window in bytes</td>
<td>MPI_Aint</td>
<td>INTEGER</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td>IN</td>
</tr>
<tr>
<td>disp_unit</td>
<td>local unit size for displacements, in bytes</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>info</td>
<td>info argument</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td>intra-communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>win</td>
<td>window object</td>
<td>MPI_Win*</td>
<td>TYPE(MPI_Win)</td>
<td>OUT</td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Win.Create
(memory, int disp_unit=1,
Info info=INFO_NULL, Intracomm comm=COMM_SELF)
```

```c
MPI_Win_allocate( /* size info */, info, comm, &memory, &window );
// do put and get calls
MPI_Win_free( &window );
```

Figure 9.2: Put and get between process memory and windows

### 9.1.1 Window creation and allocation

The memory for a window is at first sight ordinary data in user space. There are multiple ways you can associate data with a window:

1. You can pass a user buffer to `MPI_Win_create` (figure 9.1). This buffer can be an ordinary array, or it can be created with `MPI_Alloc_mem`. (In the former case, it may not be possible to lock the window; section 9.4.)
2. You can let MPI do the allocation, so that MPI can perform various optimizations regarding placement of the memory. The user code then receives the pointer to the data from MPI. This can again be done in two ways:

Victor Eijkhout 313
9. MPI topic: One-sided communication

Figure 9.2 MPI_Win_allocate

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_allocate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Win_allocate_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>size</td>
<td>size of window in bytes</td>
<td>MPI_Aint</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
</tr>
<tr>
<td>disp_unit</td>
<td>disp_unit</td>
<td>local unit size for</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>displacements, in bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Aint</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>info</td>
<td>info</td>
<td>info argument</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td>IN</td>
</tr>
<tr>
<td>comm</td>
<td>intra-communicator</td>
<td></td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
<tr>
<td>baseptr</td>
<td>initial address of window</td>
<td>void*</td>
<td>TYPE(C_PTR)</td>
<td>OUT</td>
<td></td>
</tr>
<tr>
<td>win</td>
<td>window object returned by call</td>
<td>MPI_Win*</td>
<td>TYPE(MPI_Win)</td>
<td>OUT</td>
<td></td>
</tr>
</tbody>
</table>

• Use **MPI_Win_allocate** (figure 9.2) to create the data and the window in one call.
• If a communicator is on a shared memory (see section 12.1) you can create a window in that shared memory with **MPI_Win_allocate_shared**. This will be useful for **MPI shared memory**; see chapter 12.

3. Finally, you can create a window with **MPI_Win_create_dynamic** which postpones the allocation; see section 9.5.2.

First of all, **MPI_Win_create** creates a window from a pointer to memory. The data array must not be **PARAMETER** or **static const**.

The size parameter is measured in bytes. In C this can be done with the `sizeof` operator;

```c
// putfencealloc.c
MPI_Win the_window;
int *window_data;
MPI_Win_allocate(2*sizeof(int),sizeof(int),
                     MPI_INFO_NULL,comm,
                     &window_data,&the_window);
```

*For the full source of this example, see section 9.9.2*

for doing this calculation in Fortran, see section 15.3.1.

**Python note 24: Displacement byte computations.** For computing the displacement in bytes, here is a good way for finding the size of **numpy** datatypes:

```python
# putfence.py
intsize = np.dtype('int').itemsize
window_data = np.zeros(2,dtype=int)
win = MPI.Win.Create(window_data,intsize,comm=comm)
```

*For the full source of this example, see section 9.9.3*

Next, one can obtain the memory from MPI by using **MPI_Win_allocate**, which has the data pointer as output. Note the `void*` in the C signature; it is still necessary to pass a pointer to a pointer:

```c
double *window_data;
MPI_Win_allocate( ... &window_data ... );
```
### Figure 9.3 MPI_Alloc_mem

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Alloc_mem (</td>
<td>size</td>
<td>size of memory segment in bytes</td>
<td>MPI_Aint</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>info</td>
<td>info argument</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>baseptr</td>
<td>pointer to beginning of memory segment allocated</td>
<td>void*</td>
<td>TYPE(C_PTR)</td>
<td></td>
</tr>
</tbody>
</table>

The routine **MPI_Alloc_mem** (figure 9.3) performs only the allocation part of **MPI_Win_allocate**, after which you need to **MPI_Win_create**.

- An error of **MPI_ERR_NO_MEM** indicates that no memory could be allocated.

The following material is for the recently released MPI-4 standard and may not be supported yet.

- Allocated memory can be aligned by specifying an **MPI_Info** key of **mpi_minimum_memory_alignment**.

*End of MPI-4 material*

This memory is freed with

```
MPI_Free_mem()
```

These calls reduce to `malloc` and `free` if there is no special memory area; SGI is an example where such memory does exist.

There will be more discussion of window memory in section 9.5.1.

**Python note 25: Window buffers.** Unlike in C, the python window allocate call does not return a pointer to the buffer memory, but an **MPI.memory** object. Should you need the bare memory, there are the following options:

- Window objects expose the Python buffer interface. So you can do Pythonic things like
  ```python
  mview = memoryview(win)
  array = numpy.frombuffer(win, dtype='i4')
  ```

- If you really want the raw base pointer (as an integer), you can do any of these:
  ```python
  base, size, disp_unit = win.atts
  base = win.Get_attr(MPI.WIN_BASE)
  ```

- You can use mpi4py’s builtin memoryview/buffer-like type, but I do not recommend it, much better to use NumPy as above:
  ```python
  mem = win.tomemory()  # type(mem) is MPI.memory, similar to memoryview, but
  # *quite limited in functionality
  base = mem.address
  size = mem.nbytes
  ```

#### 9.1.2 Address arithmetic

Working with windows involves a certain amount of arithmetic on addresses, meaning **MPI_Aint**. See **MPI_Aint_add** and **MPI_Aint_diff** in section 6.2.4.
9. MPI topic: One-sided communication

Figure 9.4 MPI_Win_fence

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_fence</td>
<td>assert</td>
<td>program assertion</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>window object</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
win.Fence(self, int assertion=0)
```

9.2 Active target synchronization: epochs

One-sided communication has an obvious complication over two-sided: if you do a put call instead of a send, how does the recipient know that the data is there? This process of letting the target know the state of affairs is called 'synchronization', and there are various mechanisms for it. First of all we will consider active target synchronization. Here the target knows when the transfer may happen (the communication epoch), but does not do any data-related calls.

In this section we look at the first mechanism, which is to use a fence operation: MPI_Win_fence (figure 9.4). This operation is collective on the communicator of the window. (Another, more sophisticated mechanism for active target synchronization is discussed in section 9.2.2.)

The interval between two fences is known as an epoch. Roughly speaking, in an epoch you can make one-sided communication calls, and after the concluding fence all these communications are concluded.

```c
MPI_Win_fence(0, win);  // start an epoch
MPI_Get( /* operands */ , win);    
MPI_Win_fence(0, win);       // the 'got' data is available
```

In between the two fences the window is exposed, and while it is you should not access it locally. If you absolutely need to access it locally, you can use an RMA operation for that. Also, there can be only one remote process that does a put; multiple accumulate accesses are allowed.

Fences are, together with other window calls, collective operations. That means they imply some amount of synchronization between processes. Consider:

```c
MPI_Win_fence( ... win ... ); // start an epoch
if (mytid==0) // do lots of work
   MPI_Win_fence( ... win ... ); // end the epoch
```

and assume that all processes execute the first fence more or less at the same time. The zero process does work before it can do the second fence call, but all other processes can call it immediately. However, they can not finish that second fence call until all one-sided communication is finished, which means they wait for the zero process.

As a further restriction, you can not mix MPI_Get with MPI_Put or MPI_Accumulate calls in a single epoch. Hence, we can characterize an epoch as an access epoch on the origin, and as an exposure epoch on the target.
9.2. Active target synchronization: epochs

Figure 9.3: A trace of a one-sided communication epoch where process zero only originates a one-sided transfer

9.2.1 Fence assertions

You can give various hints to the system about this epoch versus the ones before and after through the assert parameter.

- **MPI_MODE_NOSTORE** This value can be specified or not per process.
- **MPI_MODE_NOPUT** This value can be specified or not per process.
- **MPI_MODE_NOPRECEDE** This value has to be specified or not the same on all processes.
- **MPI_MODE_NOSUCCEED** This value has to be specified or not the same on all processes.

Example:

```c
MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
MPI_Get(/* operands */ , win);
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

Assertions are an integer parameter: you can combine assertions by adding them or using logical-or. The value zero is always correct. For further information, see section 9.6.

9.2.2 Non-global target synchronization

The 'fence' mechanism (section 9.2) uses a global synchronization on the communicator of the window, giving a program a Bulk Synchronous Parallel (BSP) like character. As such it is good for applications where the processes are largely synchronized, but it may lead to performance inefficiencies if processors are not in step which each other. Also, global synchronization may have hardware support, making this less restrictive than it may at first seem.

There is a mechanism that is more fine-grained, by using synchronization only on a processor group. This takes four different calls, two for starting and two for ending the epoch, separately for target and origin.
You start and complete an exposure epoch with `MPI_Win_post` / `MPI_Win_wait`:

```c
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_wait(MPI_Win win)
```

In other words, this turns your window into the target for a remote access. (There is a non-blocking version `MPI_Win_test` of `MPI_Win_wait`. See also `MPI_Win_complete`.)

You start and complete an access epoch with `MPI_Win_start` / `MPI_Win_complete`:

```c
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_complete(MPI_Win win)
```

In other words, these calls border the access to a remote window, with the current processor being the origin of the remote access.

In the following snippet a single processor puts data on one other. Note that they both have their own definition of the group, and that the receiving process only does the post and wait calls.

```c
// postwaitwin.c
if (procno==origin) {
    MPI_Group_incl(all_group,1,target,&two_group);
    // access
    MPI_Win_start(two_group,0, the_window);
    MPI_Put( /* data on origin: */ &my_number, 1, MPI_INT,
             /* data on target: */ target,0, 1, MPI_INT,
             the_window);
    MPI_Win_complete(the_window);
}

if (procno==target) {
    MPI_Group_incl(all_group,1,&origin,&two_group);
    // exposure
    MPI_Win_post(two_group,0, the_window);
```
9.3 Put, get, accumulate

We will now look at the first three routines for doing one-sided operations: the Put, Get, and Accumulate call. (We will look at so-called ‘atomic’ operations in section 9.3.7.) These calls are somewhat similar to a Send, Receive and Reduce, except that of course only one process makes a call. Since one process does all the work, its calling sequence contains both a description of the data on the origin (the calling process) and the target (the affected other process).

As in the two-sided case, MPI_PROC_NULL can be used as a target rank.

The Accumulate routine has an MPI_Op argument that can be any of the usual operators, but no user-defined ones (see section 3.10.1).

9.3.1 Put

The MPI_Put (figure 9.5) call can be considered as a one-sided send. As such, it needs to specify

- the target rank
- the data to be sent from the origin, and
- the location where it is to be written on the target.

The description of the data on the origin is the usual trio of buffer/count/datatype. However, the description of the data on the target is more complicated. It has a count and a datatype, but instead of an address it has a displacement with respect to the start of the window on the target. This displacement can be given in bytes, so its type is MPI_Aint, but strictly speaking it is a multiple of the displacement unit that was specified in the window definition.

Specifically, data is written starting at

\[ \text{window_base} + \text{target\_disp} \times \text{disp\_unit}. \]

Here is a single put operation. Note that the window create and window fence calls are collective, so they have to be performed on all processors of the communicator that was used in the create call.
9. MPI topic: One-sided communication

---

**Figure 9.5 MPI_Put**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Put (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Put_c (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>origin_addr</td>
<td>initial address of origin buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>origin_count</td>
<td>number of entries in origin buffer</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td>datatable of each entry in origin buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype) IN</td>
<td>IN</td>
</tr>
<tr>
<td>origin_datatype</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>target_rank</td>
<td>rank of target</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>target_disp</td>
<td>displacement from start of window to target buffer</td>
<td>MPI_Aint</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>target_count</td>
<td>number of entries in target buffer</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td>datatable of each entry in target buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype) IN</td>
<td>IN</td>
</tr>
<tr>
<td>target_datatype</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
win.Put(self, origin, int target_rank, target=None)
```

---

---

```
// putfence.c
MPI_Win the_window;
MPI_Win_create
    (kwindow_data,2*sizeof(int),sizeof(int),
     MPI_INFO_NULL,comm,&the_window);
MPI_Win_fence(0, the_window);
if (procno==0) {
    MPI_Put
        ( /* data on origin: */ &my_number, 1,MPI_INT,
          /* data on target: */ other,1, 1,MPI_INT,
           the_window);
    }
MPI_Win_fence(0, the_window);
MPI_Win_free(&the_window);
```

---

For the full source of this example, see section 9.9.5

Fortran note 10: Displacement unit. The disp_unit variable is declared as an integer of 'kind' MPI_ADDRESS_KIND:

```fortran
!! putfence.F90
integer(kind=MPI_ADDRESS_KIND) :: target_displacement
target_displacement = 1
call MPI_Put( my_number, 1,MPI_INTEGER, &
               other,target_displacement, &
               1,MPI_INTEGER, &
```

---
9.3. Put, get, accumulate

For the full source of this example, see section 9.9.6

Prior to Fortran2008, specifying a literal constant, such as 0, could lead to bizarre runtime errors; the solution was to specify a zero-valued variable of the right type. With the mpi_f08 module this is no longer allowed. Instead you get an error such as error #6285: There is no matching specific subroutine for this generic subroutine call. [MPI]

Python note 26: MPI one-sided transfer routines. MPI_Put (and Get and Accumulate) accept at minimum the origin buffer and the target rank. The displacement is by default zero.

Exercise 9.1. Revisit exercise 4.3 and solve it using MPI_Put.
(There is a skeleton for this exercise under the name rightput.)

Exercise 9.2. Write code where:
• process 0 computes a random number \( r \)
• if \( r < .5 \), zero writes in the window on 1;
• if \( r \geq .5 \), zero writes in the window on 2.
(There is a skeleton for this exercise under the name randomput.)

9.3.2 Get

The MPI_Get (figure 9.6) call is very similar.

Example:

```c
// getfence.c
MPI_Win_create(&other_number,2*sizeof(int),sizeof(int),
    MPI_INFO_NULL,comm,&the_window);
test_window(the_window,comm);

MPI_Win_fence(0,the_window);
if (procno==0) {
    MPI_Get( /* data on origin: */ &my_number, 1,MPI_INT,
            /* data on target: */ other,1, 1,MPI_INT,
            the_window);
}
MPI_Win_fence(0,the_window);
```

For the full source of this example, see section 9.9.7

We make a null window on processes that do not participate.

```python
# getfence.py
if procid==0 or procid==nprocs-1:
    win_mem = np.empty( 1,dtype=np.float64 )
    win = MPI.Win.Create( win_mem,comm=comm )
else:
    win = MPI.Win.Create( None,comm=comm )

# put data on another process
win.Fence()
if procid==0 or procid==nprocs-1:
```
### 9. MPI topic: One-sided communication

#### Figure 9.6 MPI_Get

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Get</td>
<td>origin_addr</td>
<td>initial address of origin buffer</td>
<td>void*</td>
<td>TYPE(*), OUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>origin_count</td>
<td>number of entries in origin buffer</td>
<td>int</td>
<td>DIMENSION(..)</td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>origin_datatype</td>
<td>datatype of each entry in origin buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>target_rank</td>
<td>rank of target</td>
<td>int</td>
<td>INTEGER IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target_disp</td>
<td>displacement from window start to the beginning of the target buffer</td>
<td>MPI_Aint</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target_count</td>
<td>number of entries in target buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>target_datatype</td>
<td>datatype of each entry in target buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)IN</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>window object used for communication</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win) IN</td>
<td></td>
</tr>
</tbody>
</table>

**Python:**

```python
win.Get(self, origin, int target_rank, target=None)
```

```python
putdata = np.empty(1, dtype=np.float64)
putdata[0] = mydata
print("%d putting %e" % (procid, mydata))
win.Put( putdata, other )
win.Fence()
```

For the full source of this example, see section 9.9.8

#### 9.3.3 Put and get example: halo update

As an example, let’s look at **halo update**. The array A is updated using the local values and the halo that comes from bordering processors, either through Put or Get operations.

In a first version we separate computation and communication. Each iteration has two fences. Between the two fences in the loop body we do the **MPI_Put** operation; between the second and and first one of the next iteration there is only computation, so we add the **NOPRECEDE** and **NOSUCCEED** assertions. The **NOSTORE** assertion states that the local window was
not updated: the Put operation only works on remote windows.

```c
for ( ..... ) {
    update(A);
    MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
    for(i=0; i < toneighbors; i++)
        MPI_Put( ... );
    MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
}
```

For much more about assertions, see section 9.6 below.

Next, we split the update in the core part, which can be done purely from local values, and the boundary, which needs local and halo values. Update of the core can overlap the communication of the halo.

```c
for ( ..... ) {
    update_boundary(A);
    MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
    for(i=0; i < fromneighbors; i++)
        MPI_Get( ... );
    update_core(A);
    MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
}
```

The NOPRECEDE and NOSUCCEED assertions still hold, but the Get operation implies that instead of NOSTORE in the second fence, we use NOPUT in the first.

### 9.3.4 Accumulate

A third one-sided routine is `MPI_Accumulate` (figure 9.7) which does a reduction operation on the results that are being put.

Accumulate is an atomic reduction with remote result. This means that multiple accumulates to a single target gives the correct result. As with `MPI_Reduce`, the order in which the operands are accumulated is undefined.

The same predefined operators are available, but no user-defined ones. There is one extra operator: `MPI_REPLACE`, this has the effect that only the last result to arrive is retained.

**Exercise 9.3.** Implement an 'all-gather' operation using one-sided communication: each processor stores a single number, and you want each processor to build up an array that contains the values from all processors. Note that you do not need a special case for a processor collecting its own value: doing ‘communication’ between a processor and itself is perfectly legal.

**Exercise 9.4.**

   Implement a shared counter:
     - One process maintains a counter;
     - Iterate: all others at random moments update this counter.
     - When the counter is no longer positive, everyone stops iterating.
9. MPI topic: One-sided communication

Figure 9.7 MPI_Accumulate

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Accumulate()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Win.Accumulate(self, origin, int target_rank, target=None, Op op=SUM)
```

The problem here is data synchronization: does everyone see the counter the same way?

9.3.5 Ordering and coherence of RMA operations

There are few guarantees about what happens inside one epoch.

- No ordering of Get and Put/Accumulate operations: if you do both, there is no guarantee whether the Get will find the value before or after the update.
- No ordering of multiple Puts. It is safer to do an Accumulate.

The following operations are well-defined inside one epoch:

- Instead of multiple Put operations, use Accumulate with `MPI_REPLACE`.
- `MPI_Get_accumulate` with `MPI_NO_OP` is safe.
- Multiple Accumulate operations from one origin are done in program order by default. To allow reordering, for instance to have all reads happen after all writes, use the info parameter when the window is created; section 9.5.3.

9.3.6 Request-based operations

Analogous to `MPI_Isend` there are request based one-sided operations: `MPI_Rput` (figure 9.8) and similarly `MPI_Rget` and `MPI_Raccumulate` and `MPI_Rget_accumulate`.
9.3. Put, get, accumulate

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Rput</td>
<td>origin_addr</td>
<td>initial address of origin buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>origin_count</td>
<td>number of entries in origin buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Cnt</td>
<td>origin_datatype</td>
<td>datatype of each entry in origin buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>target_rank</td>
<td>rank of target</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>target_disp</td>
<td>displacement from start of window to target buffer</td>
<td>MPI_Aint</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target_count</td>
<td>number of entries in target buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Cnt</td>
<td>target_datatype</td>
<td>datatype of each entry in target buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>window object used for communication</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>request</td>
<td>RMA request</td>
<td>MPI_Request*</td>
<td>TYPE(MPI_Request)</td>
<td>OUT</td>
</tr>
</tbody>
</table>

These only apply to passive target synchronization. Any MPI_Win_flush... call also terminates these transfers.

9.3.7 Atomic operations

One-sided calls are said to emulate shared memory in MPI, but the put and get calls are not enough for certain scenarios with shared data. Consider the scenario where:
- One process stores a table of work descriptors, and a pointer to the first unprocessed descriptor;
- Each process reads the pointer, reads the corresponding descriptor, and increments the pointer; and
- A process that has read a descriptor then executes the corresponding task.

The problem is that reading and updating the pointer is not an atomic operation, so it is possible that multiple processes get hold of the same value; conversely, multiple updates of the pointer may lead to work descriptors being skipped. These different overall behaviors, depending on precise timing of lower level events, are called a race condition.

In MPI-3 some atomic routines have been added. Both MPI_Fetch_and_op (figure 9.9) and MPI_Get_accumulate (figure 9.10) atomically retrieve data from the window indicated, and apply an operator, combining the data on the target with the data on the origin. Unlike Put and Get, it is safe to have multiple atomic operations in the same epoch.

Both routines perform the same operations: return data before the operation, then atomically update data on the target, but MPI_Get_accumulate is more flexible in data type handling. The more simple routine,
9. MPI topic: One-sided communication

**Figure 9.9 MPIFetch_and_op**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIFetch_and_op (</td>
<td>origin_addr</td>
<td>initial address of buffer</td>
<td>const void*,</td>
<td>TYPE(*),</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DIMENSION(..)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>result_addr</td>
<td>initial address of result buffer</td>
<td>void*</td>
<td>TYPE(*),</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DIMENSION(..)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>datatype</td>
<td>datatype of the entry in origin, result, and target buffers</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target_rank</td>
<td>rank of target</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target Disp</td>
<td>displacement from start of window to beginning of target buffer</td>
<td>MPI_Aint</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>op</td>
<td>reduce operation</td>
<td>MPI_Op</td>
<td>TYPE(MPI_Op)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>window object</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 9.10 MPIGet_accumulate**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIGet_accumulate (</td>
<td>origin_addr</td>
<td>initial address of buffer</td>
<td>const void*,</td>
<td>TYPE(*),</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DIMENSION(..)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>origin_count</td>
<td>number of entries in origin buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>origin_datatype</td>
<td>datatype of each entry in origin buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>result_addr</td>
<td>initial address of result buffer</td>
<td>void*</td>
<td>TYPE(*),</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DIMENSION(..)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>result_count</td>
<td>number of entries in result buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>result_datatype</td>
<td>datatype of each entry in result buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>target_rank</td>
<td>rank of target</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target Disp</td>
<td>displacement from start of window to beginning of target buffer</td>
<td>MPI_Aint</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target_count</td>
<td>number of entries in target buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>target_datatype</td>
<td>datatype of each entry in target buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>op</td>
<td>reduce operation</td>
<td>MPI_Op</td>
<td>TYPE(MPI_Op)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>window object</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td></td>
</tr>
</tbody>
</table>
9.3. Put, get, accumulate

**MPI Fetch and Op**, which operates on only a single element, allows for faster implementations, in particular through hardware support.

Use of **MPI NO OP** as the **MPI Op** turns these routines into an atomic Get. Similarly, using **MPI REPLACE** turns them into an atomic Put.

**Exercise 9.5.** Redo exercise 9.4 using **MPI Fetch and Op**. The problem is again to make sure all processes have the same view of the shared counter.

Does it work to make the fetch-and-op conditional? Is there a way to do it unconditionally? What should the ‘break’ test be, seeing that multiple processes can update the counter at the same time?

**Example.** A root process has a table of data; the other processes do atomic gets and update of that data using passive target synchronization through **MPI Win lock**.

```c++
// passive.cxx
if (procno==repository) {
  // Repository processor creates a table of inputs
  // and associates that with the window
}
if (procno!=repository) {
  float contribution=(float)procno,table_element;
  int loc=0;
  MPI_Win_lock(MPI_LOCK_EXCLUSIVE,repository,0,the_window);
  // read the table element by getting the result from adding zero
  MPI_Fetch_and_op
    (&contribution,&table_element,MPI_FLOAT,repository,loc,MPI_SUM,\the_window);
}
MPI_Win_unlock(repository,the_window);
```

For the full source of this example, see section 9.9.9

```python
# passive.py
if procid==repository:
  # repository process creates a table of inputs
  # and associates it with the window
  win_mem = np.empty( ninputs,dtype=np.float32 )
  win = MPI.Win.Create( win_mem,comm=comm )
else:
  # everyone else has an empty window
  win = MPI.Win.Create( None,comm=comm )
if procid!=repository:
  contribution = np.empty( 1,dtype=np.float32 )
  contribution[0] = 1.*procid
  table_element = np.empty( 1,dtype=np.float32 )
  win.Lock( repository,lock_type=MPI.LOCK_EXCLUSIVE )
  win.Fetch_and_op( contribution,table_element,repository,0,MPI.SUM)
  win.Unlock( repository )
```

For the full source of this example, see section 9.9.10

Finally, **MPI Compare and swap** (figure 9.11) swaps the origin and target data if the target data equals some comparison value.
### 9. MPI topic: One-sided communication

#### Figure 9.11 MPI_Compare_and_swap

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>origin_addr</td>
<td>initial address of buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>compare_addr</td>
<td>initial address of compare buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>result_addr</td>
<td>initial address of result buffer</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>datatype</td>
<td>datatype of the element in all buffers</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target_rank</td>
<td>rank of target</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target_disp</td>
<td>displacement from start of window to beginning of target buffer</td>
<td>MPI_Aint</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>window object</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td></td>
</tr>
</tbody>
</table>

### 9.3.7.1 A case study in atomic operations

Let us consider an example where a process, identified by `counter_process`, has a table of work descriptors, and all processes, including the counter process, take items from it to work on. To avoid duplicate work, the counter process has as counter that indicates the highest numbered available item. The part of this application that we simulate is this:

1. a process reads the counter, to find an available work item; and
2. subsequently decrements the counter by one.

We initialize the window content, under the separate memory model:

```c
// countdownop.c
MPI_Win_fence(0, the_window);
if (procno==counter_process)
    MPI_Put(&counter_init,1,MPI_INT, counter_process,0,1,MPI_INT, the_window);
MPI_Win_fence(0, the_window);
```

*For the full source of this example, see section 9.9.11*

We start by considering the naive approach, where we execute the above scheme literally with `MPI_Get` and `MPI_Put`:

```c
// countdownput.c
MPI_Win_fence(0, the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT, counter_process,0,1,MPI_INT, the_window);
MPI_Win_fence(0, the_window);
if (i_am_available) {
    my_counter_values[ n_my_counter_values++ ] = counter_value;
```
9.3. Put, get, accumulate

```c
total_decrement++;
int decrement = -1;
counter_value += decrement;
MPI_Put
    ( &counter_value, 1, MPI_INT,
        counter_process, 0, 1, MPI_INT,
        the_window);
}
MPI_Win_fence(0, the_window);
```

For the full source of this example, see section 9.9.12

This scheme is correct if only process has a true value for `i_am_available`: that processes ‘owns’ the current counter values, and it correctly updates the counter through the `MPI_Put` operation. However, if more than one process is available, they get duplicate counter values, and the update is also incorrect. If we run this program, we see that the counter did not get decremented by the total number of ‘put’ calls.

**Exercise 9.6.** Supposing only one process is available, what is the function of the middle of the three fences? Can it be omitted?

We can fix the decrement of the counter by using `MPI_Accumulate` for the counter update, since it is atomic: multiple updates in the same epoch all get processed.

```c
// countdownacc.c
MPI_Win_fence(0, the_window);
int counter_value;
MPI_Get( &counter_value, 1, MPI_INT,
    counter_process, 0, 1, MPI_INT,
    the_window);
MPI_Win_fence(0, the_window);
if (i_am_available) {
    my_counter_values[n_my_counter_values++] = counter_value;
    total_decrement++;
    int decrement = -1;
    MPI_Accumulate
    ( &decrement, 1, MPI_INT,
        counter_process, 0, 1, MPI_INT,
        MPI_SUM,
        the_window);
}
MPI_Win_fence(0, the_window);
```

For the full source of this example, see section 9.9.13

This scheme still suffers from the problem that processes will obtain duplicate counter values. The true solution is to combine the ‘get’ and ‘put’ operations into one atomic action; in this case `MPI_Fetch_and_op`:

```c
MPI_Win_fence(0, the_window);
int counter_value;
if (i_am_available) {
    int decrement = -1;
    total_decrement++;
    MPI_Fetch_and_op
```
9. MPI topic: One-sided communication

**Figure 9.12 MPI_Win_lock**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_lock</td>
<td>lock_type</td>
<td>either MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rank</td>
<td>rank of locked window</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>assert</td>
<td>program assertion</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>window object</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Win.Lock(self, int rank, int lock_type=LOCK_EXCLUSIVE, int assertion=0)
```

```c
( /* operate with data from origin: */ &decrement,
  /* retrieve data from target: */ &counter_value,
  MPI_INT, counter_process, 0, MPI_SUM,
  the_window);
```

```c
MPI.Win_fence(0, the_window);
if (i_am_available) {
  my_counter_values[n_my_counter_values++] = counter_value;
}
```

*For the full source of this example, see section 9.9.11*

Now, if there are multiple accesses, each retrieves the counter value and updates it in one atomic, that is, indivisible, action.

### 9.4 Passive target synchronization

In **passive target synchronization** only the origin is actively involved: the target makes no calls whatsoever. This means that the origin process remotely locks the window on the target, performs a one-sided transfer, and releases the window by unlocking it again.

During an access epoch, also called an **passive target epoch** in this case (the concept of ‘exposure epoch’ makes no sense with passive target synchronization), a process can initiate and finish a one-sided transfer. Typically it will lock the window with **MPI_Win_lock** (figure 9.12):

```c
if (rank == 0) {
  MPI_Win_lock (MPI_LOCK_EXCLUSIVE, 1, 0, win);
  MPI_Put (outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win);
  MPI_Win_unlock (1, win);
}
```

**Remark 17** The possibility to lock a window is not guaranteed for windows that are not created (possibly internally) by **MPI_Alloc_mem**, that is, all but **MPI_Win_create**.
9.4. Passive target synchronization

Figure 9.13 MPI_Win_unlock

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_unlock()</td>
<td></td>
<td></td>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>rank</td>
<td>rank of window</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>win</td>
<td>window object</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9.4.1 Lock types

A lock is needed to start an access epoch, that is, for an origin to acquire the capability to access a target. You can either acquire a lock on a specific process with MPI_Win_lock, or on all processes (in a communicator) with MPI_Win_lock_all. Unlike MPI_Win_fence, this is not a collective call. Also, it is possible to have multiple access epochs through MPI_Win_lock active simultaneously.

The two lock types are:

- **MPI_LOCK_SHARED**: multiple processes can access the window on the same rank. If multiple processes perform a MPI_Get call there is no problem; with MPI_Put and similar calls there is a consistency problem; see below.
- **MPI_LOCK_EXCLUSIVE**: an origin gets exclusive access to the window on a certain target. Unlike the shared lock, this has no consistency problems.

You can only specify a lock type in MPI_Win_lock; MPI_Win_lock_all is always shared.

To unlock a window, use MPI_Win_unlock (figure 9.13), respectively MPI_Win_unlock_all.

**Exercise 9.7.** Investigate atomic updates using passive target synchronization. Use

MPI_Win_lock with an exclusive lock, which means that each process only acquires the lock when it absolutely has to.

- All processs but one update a window:

  ```
  int one=1;
  MPI_Fetch_and_op(&one, &readout,
      MPI_INT, repo, zero_disp, MPI_SUM,
      the_win);
  ```

  - while the remaining process spins until the others have performed their update.

Use an atomic operation for the latter process to read out the shared value.

Can you replace the exclusive lock with a shared one?

(There is a skeleton for this exercise under the name lockfetch.)

**Exercise 9.8.** As exercise 9.7, but now use a shared lock: all processes acquire the lock simultaneously and keep it as long as is needed.

The problem here is that coherence between window buffers and local variables is now not forced by a fence or releasing a lock. Use MPI_Win_flush_local to force coherence of a window (on another process) and the local variable from

MPI_Fetch_and_op.

(There is a skeleton for this exercise under the name lockfetchshared.)
9. MPI topic: One-sided communication

Figure 9.14 MPI_Win_lock_all

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_lock_all</td>
<td>assert</td>
<td>program assertion</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>window object</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td></td>
</tr>
</tbody>
</table>

9.4.2 Lock all

To lock the windows of all processes in the group of the windows, use MPI_Win_lock_all (figure 9.14). This is not a collective call: the ‘all’ part refers to the fact that one process is locking the window on all processes.

- The assertion value can be zero, or MPI_MODE_NOCHECK, which asserts that no other process will acquire a competing lock.
- There is no ‘locktype’ parameter: this is a shared lock.

The corresponding unlock is MPI_Win_unlock_all.

The expected use of a ‘lock/unlock all’ is that they surround an extended epoch with get/put and flush calls.

9.4.3 Completion and consistency in passive target synchronization

In one-sided transfer one should keep straight the multiple instances of the data, and the various completions that effect their consistency.

- The user data. This is the buffer that is passed to a Put or Get call. For instance, after a Put call, but still in an access epoch, the user buffer is not safe to reuse. Making sure the buffer has been transferred is called local completion.
- The window data. While this may be publicly accessible, it is not necessarily always consistent with internal copies.
- The remote data. Even a successful Put does not guarantee that the other process has received the data. A successful transfer is a remote completion.

As observed, RMA operations are nonblocking, so we need mechanisms to ensure that an operation is completed, and to ensure consistency of the user and window data.

Completion of the RMA operations in a passive target epoch is ensured with MPI_Win_unlock or MPI_Win_unlock_all, similar to the use of MPI_Win_fence in active target synchronization.

If the passive target epoch is of greater duration, and no unlock operation is used to ensure completion, the following calls are available.

Remark 18 Using flush routines with active target synchronization (or generally outside a passive target epoch) you are likely to get a message

Wrong synchronization of RMA calls
9.5. More about window memory

### Local completion

The call `MPI_Win_flush_local` (figure 9.15) ensure that all operations with a given target is completed at the origin. For instance, for calls to `MPI_Get` or `MPI_Fetch_and_op` the local result is available after the `MPI_Win_flush_local`.

With `MPI_Win_flush_local_all` local operations are concluded for all targets. This will typically be used with `MPI_Win_lock_all` (section 9.4.2).

### Remote completion

The calls `MPI_Win_flush` (figure 9.16) and `MPI_Win_flush_all` effect completion of all outstanding RMA operations on the target, so that other processes can access its data. This is useful for `MPI_Put` operations, but can also be used for atomic operations such as `MPI_Fetch_and_op`.

### Window synchronization

Under the separate memory model, the user code can hold a buffer that is not coherent with the internal window data. The call `MPI_Win_sync` synchronizes private and public copies of the window.

9.5  More about window memory

9.5.1  Memory models

You may think that the window memory is the same as the buffer you pass to `MPI_Win_create` or that you get from `MPI_Win_allocate` (section 9.1.1). This is not necessarily true, and the actual state of affairs is called the memory model. There are two memory models:

---

**Figure 9.15 MPI_Win_flush_local**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_flush_local (</td>
<td>rank</td>
<td>rank of target window</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>window object</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 9.16 MPI_Win_flush**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_flush (</td>
<td>rank</td>
<td>rank of target window</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>window object</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
9. MPI topic: One-sided communication

Figure 9.17 MPI_Win_create_dynamic

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_create_dynamic</td>
<td>info</td>
<td>info argument</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>intra-communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>window object returned by the call</td>
<td>MPI_Win*</td>
<td>TYPE(MPI_Win)</td>
<td></td>
</tr>
</tbody>
</table>

- Under the *unified* memory model, the buffer in process space is indeed the window memory, or at least they are kept *coherent*. This means that after *completion* of an epoch you can read the window contents from the buffer. To get this, the window needs to be created with MPI_Win_allocate_shared. This memory model is required for MPI shared memory; chapter 12.
- Under the *separate* memory model, the buffer in process space is the *private window* and the target of put/get operations is the *public window* and the two are not the same and are not kept coherent. Under this model, you need to do an explicit get to read the window contents.

You can query the model of a window using the MPI_Win_get_attr call with the MPI_WIN_MODEL keyword:

```c
// window.c
int *modelstar, flag;
MPI_Win_get_attr(the_window, MPI_WIN_MODEL, &modelstar, &flag);
int model = *modelstar;
if (procno==0)
    printf("Window model is unified: %d\n", model==MPI_WIN_UNIFIED);
```

with possible values:
- MPI_WIN_SEPARATE,
- MPI_WIN_UNIFIED,

For more on attributes, see section 9.5.4.

9.5.2 Dynamically attached memory

In section 9.1.1 we looked at simple ways to create a window and its memory.

It is also possible to have windows where the size is dynamically set. Create a dynamic window with MPI_Win_create_dynamic (figure 9.17) and attach memory to the window with MPI_Win_attach (figure 9.18).

At first sight, the code looks like splitting up a MPI_Win_create call into separate creation of the window and declaration of the buffer:

```c
// windynamic.c
MPI_Win_create_dynamic(MPI_INFO_NULL, comm, &the_window);
if (procno==data_proc)
    window_buffer = (int*) malloc( 2*sizeof(int) );
MPI_Win_attach(the_window, window_buffer, 2*sizeof(int));
```

For the full source of this example, see section 9.9.14
9.5. More about window memory

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_attach (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>win</td>
<td>window object</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>base</td>
<td>initial address of memory to be attached</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>size of memory to be attached in bytes</td>
<td>MPI_Aint</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9.19 MPI_Win_detach

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_detach (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>win</td>
<td>window object</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>base</td>
<td>initial address of memory to be detached</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(where the window_buffer represents memory that has been allocated.)

However, there is an important difference in how the window is addressed in RMA operations. With all other window models, the displacement parameter is measured relative in units from the start of the buffer, here the displacement is an absolute address. This means that we need to get the address of the window buffer with MPI_Get_address and communicate it to the other processes:

```c
MPI_Aint data_address;
if (procno==data_proc) {
    MPI_Get_address(window_buffer,&data_address);
}
MPI_Bcast(&data_address,1,MPI_LONG,data_proc,comm);
```

*For the full source of this example, see section 9.9.14*

Location of the data, that is, the displacement parameter, is then given as an absolute location of the start of the buffer plus a count in bytes; in other words, the displacement unit is 1. In this example we use MPI_Get to find the second integer in a window buffer:

```c
MPI_Aint disp = data_address+1*sizeof(int);
MPI_Get( /* data on origin: */ retrieve, 1,MPI_INT,
    /* data on target: */ data_proc,disp, 1,MPI_INT,
    the_window);
```

*For the full source of this example, see section 9.9.14*

Notes.

- The attached memory can be released with MPI_Win_detach (figure 9.19).
- The above fragments show that an origin process has the actual address of the window buffer. It is an error to use this if the buffer is not attached to a window.
9. MPI topic: One-sided communication

- In particular, one has to make sure that the attach call is concluded before performing RMA operations on the window.

9.5.3 Window usage hints

The following keys can be passed as info argument:

- no_locks: if set to true, passive target synchronization (section 9.4) will not be used on this window.
- accumulate_ordering: a comma-separated list of the keywords rar, raw, war, waw can be specified. This indicates that reads or writes from MPI_Accumulate or MPI_Get_accumulate can be reordered, subject to certain constraints.
- accumulate_ops: the value same_op indicates that concurrent Accumulate calls use the same operator; same_op_no_op indicates the same operator or MPI_NO_OP.

9.5.4 Window information

The MPI_Info parameter (see section 15.1.1 for info objects) can be used to pass implementation-dependent information.

A number of attributes are stored with a window when it is created.

- **MPI_WIN_BASE** for obtaining a pointer to the start of the window area:

  ```c
  void *base;
  MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag)
  ```

- **MPI_WIN_SIZE** and **MPI_WIN_DISP_UNIT** for obtaining the size and window displacement unit:

  ```c
  MPI_Aint *size;
  MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag),
  int *disp_unit;
  MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag),
  ```

- **MPI_WIN_CREATE_FLAVOR** for determining the type of create call used:

  ```c
  int *create_kind;
  MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, &create_kind, &flag)
  ```

  with possible values:
  - **MPI_WIN_FLAVOR_CREATE** if the window was create with MPI_Win_create;
  - **MPI_WIN_FLAVOR_ALLOCATE** if the window was create with MPI_Win_allocate;
  - **MPI_WIN_FLAVOR_DYNAMIC** if the window was create with MPI_Win_create_dynamic. In this case the base is MPI_BOTTOM and the size is zero;
  - **MPI_WIN_FLAVOR_SHARED** if the window was create with MPI_Win_allocate_shared;
  - **MPI_WIN_MODEL** for querying the window memory model; see section 9.5.1.

Get the group of processes (see section 7.5) associated with a window:

```c
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
```
9.6 Assertions

The routines

- (Active target synchronization) `MPI_Win_fence`, `MPI_Win_post`, `MPI_Win_start`;
- (Passive target synchronization) `MPI_Win_lock`, `MPI_Win_lockall`,

take an argument through which assertions can be passed about the activity before, after, and during the epoch. The value zero is always allowed, by you can make your program more efficient by specifying one or more of the following, combined by bitwise OR in C/C++ or IOR in Fortran.

- **MPI_Win_start** Supports the option:
  - `MPI_MODE_NOCHECK` the matching calls to `MPI_Win_post` have already completed on all target processes when the call to `MPI_Win_start` is made. The nocheck option can be specified in a start call if and only if it is specified in each matching post call. This is similar to the optimization of “ready-send” that may save a handshake when the handshake is implicit in the code. (However, ready-send is matched by a regular receive, whereas both start and post must specify the nocheck option.)

- **MPI_Win_post** supports the following options:
  - `MPI_MODE_NOCHECK` the matching calls to `MPI_Win_start` have not yet occurred on any origin processes when the call to `MPI_Win_post` is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.
  - `MPI_MODE_NOSTORE` the local window was not updated by local stores (or local get or receive calls) since last synchronization. This may avoid the need for cache synchronization at the post call.
  - `MPI_MODE_NOPUT` the local window will not be updated by put or accumulate calls after the post call, until the ensuing (wait) synchronization. This may avoid the need for cache synchronization at the wait call.

- **MPI_Win_fence** supports the following options:
  - `MPI_MODE_NOSTORE` the local window was not updated by local stores (or local get or receive calls) since last synchronization.
  - `MPI_MODE_NOPUT` the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.
  - `MPI_MODE_NOPRECEDE` the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.
  - `MPI_MODE_NOSUCCEED` the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.

- **MPI_Win_lock** and **MPI_Win_lock_all** support the following option:
9. MPI topic: One-sided communication

- MPI MODE NOCHECK: no other process holds, or will attempt to acquire a conflicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.

9.7 Implementation

You may wonder how one-sided communication is realized\(^1\). Can a processor somehow get at another processor’s data? Unfortunately, no.

Active target synchronization is implemented in terms of two-sided communication. Imagine that the first fence operation does nothing, unless it concludes prior one-sided operations. The Put and Get calls do nothing involving communication, except for marking with what processors they exchange data. The concluding fence is where everything happens: first a global operation determines which targets need to issue send or receive calls, then the actual sends and receive are executed.

**Exercise 9.9.** Assume that only Get operations are performed during an epoch. Sketch how these are translated to send/receive pairs. The problem here is how the senders find out that they need to send. Show that you can solve this with an MPI Reduce_scatter call.

The previous paragraph noted that a collective operation was necessary to determine the two-sided traffic. Since collective operations induce some amount of synchronization, you may want to limit this.

**Exercise 9.10.** Argue that the mechanism with window post/wait/start/complete operations still needs a collective, but that this is less burdensome.

Passive target synchronization needs another mechanism entirely. Here the target process needs to have a background task (process, thread, daemon,...) running that listens for requests to lock the window. This can potentially be expensive.

\(^1\) For more on this subject, see [27].
Find all the errors in this code.

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define MASTER 0

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    MPI_Comm comm = MPI_COMM_WORLD;
    int r, p;
    MPI_Comm_rank(comm, &r);
    MPI_Comm_size(comm, &p);
    printf("Hello from %d\n", r);
    int result[1] = {0};
    //int assert = MPI_MODE_NOCHECK;
    int assert = 0;
    int one = 1;
    MPI_Win win_res;
    MPI_Win_allocate(1 * sizeof(MPI_INT), sizeof(MPI_INT), MPI_INFO_NULL, comm,
        &result[0], &win_res);
    MPI_Win_lock_all(assert, win_res);
    if (r == MASTER) {
        result[0] = 0;
        do{
            MPI_Fetch_and_op(&result, &result, MPI_INT, r, 0, MPI_NO_OP, win_res);
            printf("result: %d\n", result[0]);
        } while(result[0] != 4);
        printf("Master is done!\n");
    } else {
        MPI_Fetch_and_op(&one, &result, MPI_INT, 0, 0, MPI_SUM, win_res);
    }
    MPI_Win_unlock_all(win_res);
    MPI_Win_free(&win_res);
    MPI_Finalize();
    return 0;
}
```
9. MPI topic: One-sided communication

9.9 Sources used in this chapter

9.9.1 Listing of code header

9.9.2 Listing of code examples/mpi/c/examples/putfencealloc.c

import numpy as np
import random # random.randint(1,N), random.random()
import sys

from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

intsize = np.dtype('int').itemsize
window_data = np.zeros(2,dtype=int)
win = MPI.Win.Create(window_data,intsize,comm=comm)

my_number = np.empty(1,dtype=int)
src = 0; tgt = nprocs-1
if procid==src:
    my_number[0] = 37
else:
    my_number[0] = 1

win.Fence()
if procid==src:
    # put data in the second element of the window
    win.Put(my_number,tgt,target=1)

win.Fence()

if procid==tgt:
    print("Window after put:",window_data)

9.9.3 Listing of code examples/mpi/p/putfence.py

import numpy as np
import random # random.randint(1,N), random.random()
import sys

from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

intsize = np.dtype('int').itemsize
window_data = np.zeros(2,dtype=int)
win = MPI.Win.Create(window_data,intsize,comm=comm)

my_number = np.empty(1,dtype=int)
src = 0; tgt = nprocs-1
if procid==src:
    my_number[0] = 37
else:
    my_number[0] = 1
win.Fence()
if procid==src:
    # put data in the second element of the window
    win.Put(my_number,tgt,target=1)

win.Fence()

if procid==tgt:
    print("Window after put:",window_data)

9.9.4 Listing of code examples/mpi/c/postwaitwin.c

#include <stdlib.h>
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

340
int main(int argc,char **argv) {

#include "globalinit.c"

{

MPI_Win the_window;
MPI_Group all_group,two_group;
int my_number = 37, other_number,
	twotids[2],origin,target;

MPI_Win_create(&other_number,1,sizeof(int),

MII_INFO_NULL,comm,&the_window);
if (procno>0 && procno<nprocs-1) goto skip;

origin = 0; target = nprocs-1;
MPI_Comm_group(comm,&all_group);

if (procno==origin) {

MPI_Group_incl(all_group,1,&target,&two_group);
// access
MPI_Win_start(two_group,0,the_window);
MPI_Put( /* data on origin: */ &my_number, 1,MPI_INT,

/* data on target: */ target,0, 1,MPI_INT,

the_window);
MPI_Win_complete(the_window);
}

if (procno==target) {

MPI_Group_incl(all_group,1,&origin,&two_group);
// exposure
MPI_Win_post(two_group,0,the_window);
MPI_Win_wait(the_window);
}

if (procno==target)

printf("Got the following: %d\n",other_number);

MPI_Group_free(&all_group);
MPI_Group_free(&two_group);
skip:
MPI_Win_free(&the_window);
}

MPI_Finalize();
return 0;
}

9.9.5 Listing of code examples/mpi/c/putfence.c

#include <stdlib.h>
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

Victor Eijkhout
int main(int argc,char **argv) {
    
    #include "globalinit.c"
    
    MPI_Win the_window;
    int my_number=0, window_data[2], other = nprocs-1;
    if (procno==0)
        my_number = 37;
    
    MPI_Win_create
    ( &window_data,2*sizeof(int),sizeof(int),
    MPI_INFO_NULL,comm,&the_window);
    MPI_Win_fence(0,the_window);
    if (procno==0) {
        MPI_Put
        ( /* data on origin: */ &my_number, 1,MPI_INT,
        /* data on target: */ other,1, 1,MPI_INT,
        the_window);
    }
    MPI_Win_fence(0,the_window);
    if (procno==other)
        printf("I got the following: %d
",window_data[1]);
    MPI_Win_free(&the_window);

    MPI_Finalize();
    return 0;
}

9.9.6 Listing of code examples/mpi/f08/putfence.F90

Program PutFence

use mpi_f08
implicit none

Type(MPI_Win) :: the_window
integer :: window_elt_size
integer(kind=MPI_ADDRESS_KIND) :: window_size
integer(kind=MPI_ADDRESS_KIND) :: target_displacement
integer :: my_number=0, window_data(2), other

Type(MPI_Comm) :: comm;
integer :: mytid,ntids,i,p,err;

call MPI_Init() 
comm = MPI_COMM_WORLD 
call MPI_Comm_rank(comm,mytid) 
call MPI_Comm_size(comm,ntids) 
call MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN)

other = ntids-1
if (mytid.eq.0) my_number = 37
9.9. Sources used in this chapter


call MPI_Sizeof(window_data,window_elt_size)
window_size = 2*window_elt_size
call MPI_Win_create(window_data,&
    window_size,window_elt_size, &
    MPI_INFO_NULL,comm,the_window)
call MPI_Win_fence(0,the_window)
if (mytid.eq.0) then
#ifndef F90STYLE
    target_displacement = 1
    call MPI_Put( my_number, 1,MPI_INTEGER, &
        other,target_displacement, &
        1,MPI_INTEGER, &
        the_window)
#else
    call MPI_Put( my_number, 1,MPI_INTEGER, &
        other,1, &
        1,MPI_INTEGER, &
        the_window)
#endif
endif
endif

call MPI_Win_fence(0,the_window)
if (mytid.eq.other) then
    print *,"I got:",window_data(1+target_displacement)
end if

call MPI_Win_free(the_window)
call MPI_Finalize(err);
end Program PutFence

9.9.7 Listing of code examples/mpi/c/getfence.c

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>
#include "window.c"

int main(int argc,char **argv) {

    #include "globalinit.c"

    MPI_Win the_window;
    int my_number, other_number[2], other = nprocs-1;
    if (procno==other)
        other_number[1] = 27;
    MPI_Win_create(&other_number,2*sizeof(int),sizeof(int),
        MPI_INFO_NULL,comm,&the_window);
9. MPI topic: One-sided communication

test_window(the_window,comm);

MPI_Win_fence(0,the_window);
if (procno==0) {
    MPI_Get( /* data on origin: */ &my_number, 1,MPI_INT,
            /* data on target: */ other,1, 1,MPI_INT,
            the_window);
}

MPI_Win_fence(0,the_window);
if (procno==0)
    printf("I got the following: %d
",my_number);

MPI_Win_free(&the_window);

MPI_Finalize();
return 0;
}

9.9.8 Listing of code examples/mpi/p/getfence.py

import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)
other = nprocs-1-procid
mydata = random.random()

if procid==0 or procid==nprocs-1:
    win_mem = np.empty( 1,dtype=np.float64 )
    win = MPI.Win.Create( win_mem,comm=comm )
else:
    win = MPI.Win.Create( None,comm=comm )

# put data on another process
win.Fence()
if procid==0 or procid==nprocs-1:
    putdata = np.empty( 1,dtype=np.float64 )
    putdata[0] = mydata
    print("[%d] putting %e % (procid,mydata))
    win.Put( putdata,other )
win.Fence()

# see what you got
if procid==0 or procid==nprocs-1:
    print("[%d] getting %e % (procid,win_mem[0])")
9.9. Sources used in this chapter

9.9.9 Listing of code examples/mpi/c/fetchop.c

```c
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)
repository = 0 # process that keeps the repository
ninputs = nprocs-1
checksum = nprocs*(nprocs-1)/2.

if procid==repository:
    # repository process creates a table of inputs
    # and associates it with the window
    win_mem = np.empty( ninputs,dtype=np.float32 )
    win = MPI.Win.Create( win_mem,comm=comm )
else:
    # everyone else has an empty window
    win = MPI.Win.Create( None,comm=comm )

# bookkeeping: which jobs did I process
myjobs = np.zeros( ninputs,dtype=int )

if procid!=repository:
    contribution = np.empty( 1,dtype=np.float32 )
    contribution[0] = 1.*procid
    table_element = np.empty( 1,dtype=np.float32 )
    win.Lock( repository,lock_type=MPI.LOCK_EXCLUSIVE )
    win.Fetch_and_op( contribution,table_element,repository,0,MPI.SUM)
    win.Unlock( repository )
    print(procid,"added its contribution to partial sum",table_element[0])

win.Free() 

if procid==repository:
    if abs(win_mem[0]-checksum)>1.e-12:
        print("Incorrect result %e s/b %e" % (win_mem[0],checksum))
    print("finished")
```

9.9.10 Listing of code examples/mpi/p/passive.py

```python
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

repository = 0 # process that keeps the repository
ninputs = nprocs-1
checksum = nprocs*(nprocs-1)/2.

if procid==repository:
    # repository process creates a table of inputs
    # and associates it with the window
    win_mem = np.empty( ninputs,dtype=np.float32 )
    win = MPI.Win.Create( win_mem,comm=comm )
else:
    # everyone else has an empty window
    win = MPI.Win.Create( None,comm=comm )

# bookkeeping: which jobs did I process
myjobs = np.zeros( ninputs,dtype=int )

if procid!=repository:
    contribution = np.empty( 1,dtype=np.float32 )
    contribution[0] = 1.*procid
    table_element = np.empty( 1,dtype=np.float32 )
    win.Lock( repository,lock_type=MPI.LOCK_EXCLUSIVE )
    win.Fetch_and_op( contribution,table_element,repository,0,MPI.SUM)
    win.Unlock( repository )
    print(procid,"added its contribution to partial sum",table_element[0])

win.Free() 

if procid==repository:
    if abs(win_mem[0]-checksum)>1.e-12:
        print("Incorrect result %e s/b %e" % (win_mem[0],checksum))
    print("finished")
```
9. MPI topic: One-sided communication

9.9.11 Listing of code examples/mpi/c/countdownop.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>
#include "gather_sort_print.h"

int main(int argc,char **argv) {
    int nprocs,procno;
    MPI_Init(0,0);
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procno);
    if (nprocs<2) {
        printf("Need at least 2 procs\n");
        MPI_Abort(comm,0);
    }
    // first set a unique random seed
    srand(procno*time(0));

    {
        /*
        * Create a window.
        * We only need a nonzero size on the last process,
        * which we label the 'counter_process';
        * everyone else makes a window of size zero.
        */
        MPI_Win the_window;
        int counter_process = nprocs-1;
        int window_data,check_data;
        if (procno==counter_process) {
            window_data = 2*nprocs-1;
            check_data = window_data;
            MPI_Win_create(&window_data,sizeof(int),sizeof(int),
                            MPI_INFO_NULL,comm,&the_window);
        } else {
            MPI_Win_create(&window_data,0,sizeof(int),
                            MPI_INFO_NULL,comm,&the_window);
        }
        /*
        * Initialize the window
        * - PROC_WRITES is approx the number of writes we want each process to do
        * - COLLISION is approx how many processes will collide on a write
        */
        #ifndef COLLISION
        #define COLLISION 2
        #endif
        #ifndef PROC_WRITES
        #define PROC_WRITES 40
        ```
int counter_init = nprocs * PROCWRITES;
MPI_Win_fence(0, the_window);
if (procno==counter_process)
    MPI_Put(&counter_init,1,MPI_INT,
            counter_process,0,1,MPI_INT,
            the_window);
MPI_Win_fence(0, the_window);

/*
 * Allocate an array (grossly over-dimensioned)
 * for the counter values that belong to me
 */
int *my_counter_values = (int*) malloc(counter_init * sizeof(int));
if (!my_counter_values) {
    printf("[%d] could not allocate counter values\n", procno);
    MPI_Abort(comm,0);
}
int n_my_counter_values = 0;

/*
 * Loop:
 * - at random times update the counter on the counter process
 * - and read out the counter to see if we stop
 */
int total_decrement = 0;
nsteps = PROCWRITES / COLLISION;
if (procno==0)
    printf("Doing %d steps, counter starting: %d\n probably %d-way collision on each step\n", nsteps, counter_init, COLLISION);
for (int step=0; step<nsteps ; step++) {
    /*
     * Basic probability of a write is 1/P,
     * so each step only one proc will write.
     * Increase chance of collision by upping
     * the value of COLLISION.
     */
    float randomfraction = (rand() / (double)RAND_MAX);
    int i_am_available = randomfraction < ( COLLISION * 1./nprocs );
    /*
     * Exercise:
     * - atomically read and decrement the counter
     */
    MPI_Win_fence(0, the_window);
    int counter_value;
    if (i_am_available) {
        int decrement = -1;
        total_decrement++;
        total_decrement++;
9. MPI topic: One-sided communication

( /* operate with data from origin: */ &decrement,
/* retrieve data from target: */ &counter_value,
MPI_INT, counter_process, 0, MPI_SUM,
the_window);

#ifdef DEBUG
printf("[%d updating in step %d; retrieved %d\n",procno,step,counter_value);
#endif
MPI_Win_fence(0,the_window);
if (i_am_available) {
    my_counter_values[n_my_counter_values++] = counter_value;
}

/*
 * What counter values were actually obtained?
 */
gather_sort_print( my_counter_values,n_my_counter_values, comm );

/*
 * We do a correctness test by computing what the
 * window_data is supposed to be
 */
{
    MPI_Win_fence(0,the_window);
    int counter_value;
    MPI_Get( /* origin data to set: */ &counter_value,1,MPI_INT,
        /* window data to get: */ counter_process,0,1,MPI_INT,
        the_window);
    MPI_Win_fence(0,the_window);
    MPI_Allreduce(MPI_IN_PLACE,&total_decrement,1,MPI_INT,MPI_SUM,comm);
    if (procno==counter_process) {
        if (counter_init-total_decrement==counter_value)
            printf("[%d] initial counter %d decreased by %d correctly giving %d
", procno,counter_init,total_decrement,counter_value);
        else
            printf("[%d] initial counter %d decreased by %d, giving %d s/b %d
", procno,counter_init,total_decrement,counter_value,counter_init-total_decrement);
    }
}
MPI_Win_free(&the_window);

MPI_Finalize();
return 0;
}

9.9.12 Listing of code examples/mpi/c/countdownput.c

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "time.h"
#include <mpi.h>
#include "gather_sort_print.h"

int main(int argc,char **argv) {

    int nprocs,procno;
    MPI_Init(0,0);
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procno);

    if (nprocs<2) {
        printf("Need at least 2 procs\n");
        MPI_Abort(comm,0);
    }

    // first set a unique random seed
    srand(procno*time(0));

    {
        /*
         * Create a window.
         * We only need a nonzero size on the last process,
         * which we label the `counter_process';
         * everyone else makes a window of size zero.
         */
        MPI_Win the_window;
        int counter_process = nprocs-1;
        int window_data,check_data;
        if (procno==counter_process) {
            window_data = 2*nprocs-1;
            check_data = window_data;
            MPI_Win_create(&window_data,sizeof(int),sizeof(int),
                            MPI_INFO_NULL,comm,&the_window);
        } else {
            MPI_Win_create(&window_data,0,sizeof(int),
                            MPI_INFO_NULL,comm,&the_window);
        }

        /*
         * Initialize the window
         * - PROCWRITES is approx the number of writes we want each process to do
         * - COLLISION is approx how many processes will collide on a write
         */
        #ifndef COLLISION
        #define COLLISION 1
        #endif
        #ifndef PROCWRITES
        #define PROCWRITES 10
        #endif

        int counter_init = nprocs * PROCWRITES;
        MPI_Win_fence(0, the_window);
        if (procno==counter_process)
MPI_Put(&counter_init,1,MPI_INT, 
    counter_process,0,1,MPI_INT, 
    the_window);
MPI_Win_fence(0,the_window);

/*
 * Allocate an array (grossly over-dimensioned)
 * for the counter values that belong to me
 */
int *my_counter_values = (int*) malloc( counter_init * sizeof(int) );
if (!my_counter_values) {
    printf("[%d] could not allocate counter values\n",procno);
    MPI_Abort(comm,0);
}
int n_my_counter_values = 0;

/*
 * Loop forever:
 * - at random times update the counter on the counter process
 * - and read out the counter to see if we stop
 */
int total_decrement = 0;
int nsteps = PROC_WRITES / COLLISION;
if (procno==0)
    printf("Doing %d steps, %d writes per proc,\n.. probably %d-way collision on each step\n", 
        nsteps,PROC_WRITES,COLLISION);
for (int step=0; step<nsteps ; step++) {

    /*
    * Basic probability of a write is 1/P,
    * so each step only one proc will write.
    * Increase chance of collision by upping
    * the value of COLLISION.
    */
    float random_fraction = (rand() / (double)RAND_MAX);
    int i_am_available = random_fraction < ( COLLISION * .8/nprocs );

    /*
    * Exercise:
    * - decrement the counter by Get, compute new value, Put
    */
    MPI_Win_fence(0,the_window);
    int counter_value;
    MPI_Get( &counter_value,1,MPI_INT, 
        counter_process,0,1,MPI_INT, 
        the_window);
    MPI_Win_fence(0,the_window);
    if (i_am_available) {
        #ifdef DEBUG
        printf("[%d] obtaining value %d in step %d\n", 
            procno,counter_value,step);
        #endif
        my_counter_values[ n_my_counter_values++ ] = counter_value;
    }
9.9.13 Listing of code examples/mpi/c/countdownacc.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>
#include "gather_sort_print.h"

Victor Eijkhout
351
```
int main(int argc,char **argv) {

#include "globalinit.c"

if (nprocs<2) {
    printf("Need at least 2 procs\n");
    MPI_Abort(comm,0);
}

// first set a unique random seed
srand(procno*time(0));

{
    /*
     * We only need a nonzero size on the last process,
     * which we label the 'counter_process';
     * everyone else makes a window of size zero.
     */
    MPI_Win the_window;
    int counter_process = nprocs-1;
    int window_data,check_data;
    if (procno==counter_process) {
        window_data = 2*nprocs-1;
        check_data = window_data;
        MPI_Win_create(&window_data,sizeof(int),sizeof(int),
                        MPI_INFO_NULL,comm,&the_window);
    } else {
        MPI_Win_create(&window_data,0,sizeof(int),
                       MPI_INFO_NULL,comm,&the_window);
    }
    /*
     * Initialize the window
     * - PROCWRITES is approx the number of writes we want each process to do
     * - COLLISION is approx how many processes will collide on a write
     */
    #ifndef COLLISION
    #define COLLISION 2
    #endif
    #ifndef PROCWRITES
    #define PROCWRITES 40
    #endif
    int counter_init = nprocs * PROCWRITES;
    MPI_Win_fence(0,the_window);
    if (procno==counter_process)
        MPI_Put(&counter_init,1,MPI_INT,
                 counter_process,0,1,MPI_INT,
                 the_window);
    MPI_Win_fence(0,the_window);

    /*
     * Allocate an array (grossly over-dimensioned)
     * for the counter values that belong to me
     */
}
int *my_counter_values = (int*) malloc(counter_init * sizeof(int));
if (!my_counter_values) {
    printf("[%d] could not allocate counter values\n",procno);
    MPI_Abort(comm,0);
}
int n_my_counter_values = 0;

/*
 * Loop:
 * - at random times update the counter on the counter process
 * - and read out the counter to see if we stop
 */
int total_decrement = 0;
int nsteps = PROCWRITES / COLLISION;
if (procno==0)
    printf("Doing %d steps, counter starting: %d\n probably %d-way collision on each step\n",nsteps,counter_init,COLLISION);
for (int step=0; step<nsteps ; step++) {

/*
 * Basic probability of a write is 1/P,
 * so each step only one proc will write.
 * Increase chance of collision by upping
 * the value of COLLISION.
 */
float randomfraction = (rand() / (double)RAND_MAX);
int i_am_available = randomfraction < ( COLLISION * 1./nprocs );

/*
 * Exercise:
 * - decrement the counter by Get, compute new value, Accumulate
 */
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
counter_process,0,1,MPI_INT,
the_window);
MPI_Win_fence(0,the_window);
if (i_am_available) {
    #ifdef DEBUG
        printf("[%d] updating in step %d\n",procno,step);
    #endif
    my_counter_values[n_my_counter_values++] = counter_value;
    total_decrement--;
    int decrement = -1;
    MPI_Accumulate
        ( &decrement, 1,MPI_INT,
counter_process,0,1,MPI_INT,
        MPI_SUM,
        the_window);
    #endif
    MPI_Win_fence(0,the_window);
}
9.9.14 Listing of code examples/mpi/c/windynamic.c

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>
#include "window.c"

#include "globalinit.c"

int main(int argc,char **argv) {

    {
        MPI_Win the_window;
        int origin=0, data_proc = nprocs-1;
        int *retrieve=NULL,*window_buffer=NULL;

        MPI_Win_create_dynamic(MPI_INFO_NULL,comm,&the_window);
        
        MPI_Win_fence(0,the_window);
        int counter_value;
        MPI_Get( /* origin data to set: */ &counter_value,1,MPI_INT,
                /* window data to get: */ counter_process,0,1,MPI_INT,
                the_window);
        MPI_Win_fence(0,the_window);
        MPI_Allreduce(MPI_IN_PLACE,&total_decrement,1,MPI_INT,MPI_SUM,comm);
        if (procno==counter_process) {
            if (counter_init-total_decrement==counter_value)
                printf("[%d] initial counter %d decreased by %d correctly giving %d\n", procno,counter_init,total_decrement,counter_value);
            else
                printf("[%d] initial counter %d decreased by %d, giving %d s/b %d\n", procno,counter_init,total_decrement,counter_value,counter_init-total_decrement);
        }
    }
    MPI_Win_free(&the_window);
    
    MPI_Finalize();
    return 0;
}
if (procno==data_proc)
    window_buffer = (int*) malloc( 2*sizeof(int) );
MPI_Win_attach(the_window,window_buffer,2*sizeof(int));

    test_window( the_window,comm );

if (procno==data_proc) {
    window_buffer[0] = 1;
    window_buffer[1] = 27;
}
if (procno==origin) {
    retrieve = (int*) malloc( sizeof(int) );
}

MPI_Aint data_address;
if (procno==data_proc) {
    MPI_Get_address(window_buffer,&data_address);
}
MPI_Bcast(&data_address,1,MPI_LONG,data_proc,comm);

MPI_Win_fence(0,the_window);
if (procno==origin) {
    MPI_Aint disp = data_address+1*sizeof(int);
    MPI_Get( /* data on origin: */ retrieve, 1,MPI_INT, /* data on target: */ data_proc,disp, 1,MPI_INT, the_window);
}
MPI_Win_fence(0,the_window);

if (procno==origin)
    printf("I got the following: %d\n",retrieve[0]);
MPI_Win_free(&the_window);
}

MPI_Finalize();
return 0;
}
Chapter 10

MPI topic: File I/O

This chapter discusses the I/O support of MPI, which is intended to alleviate the problems inherent in parallel file access. Let us first explore the issues. This story partly depends on what sort of parallel computer you are running on.

- On networks of workstations each node will have a separate drive with its own file system.
- On many clusters there will be a shared file system that acts as if every process can access every file.
- Cluster nodes may or may not have a private file system.

Based on this, the following strategies are possible, even before we start talking about MPI I/O.

- One process can collect all data with MPI_Gather and write it out. There are at least three things wrong with this: it uses network bandwidth for the gather, it may require a large amount of memory on the root process, and centralized writing is a bottleneck.
- Absent a shared file system, writing can be parallelized by letting every process create a unique file and merge these after the run. This makes the I/O symmetric, but collecting all the files is a bottleneck.
- Even with a with a shared file system this approach is possible, but it can put a lot of strain on the file system, and the post-processing can be a significant task.
- Using a shared file system, there is nothing against every process opening the same existing file for reading, and using an individual file pointer to get its unique data.
- ... but having every process open the same file for output is probably not a good idea. For instance, if two processes try to write at the end of the file, you may need to synchronize them, and synchronize the file system flushes.

For these reasons, MPI has a number of routines that make it possible to read and write a single file from a large number of processes, giving each well-defined locations where to access the data. In fact, MPI-IO uses MPI derived datatypes for both the source data (that is, in memory) and target data (that is, on disk). Thus, in one call that is collective on a communicator each process can address data that is not contiguous in memory, and place it in locations that are not contiguous on disc.

There are dedicated libraries for file I/O, such as hdf5, netcdf, or silo. However, these often add header information to a file that may not be understandable to post-processing applications. With MPI I/O you are in complete control of what goes to the file. (A useful tool for viewing your file is the unix utility od.)
10.1 File handling

**Figure 10.1 MPI_File_open**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_File_open (</td>
<td>communicator</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td>filename</td>
<td>name of file to open</td>
<td>const char*</td>
<td>CHARACTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>amode</td>
<td>file access mode</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>info</td>
<td>info object</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fh</td>
<td>new file handle</td>
<td>MPI_File*</td>
<td>TYPE(MPI_File)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
Open(type cls, Intracomm comm, filename,
    int amode=MODE_RDONLY, Info info=INFO_NULL)
```

*TACC note.* Each node has a private `/tmp` file system (typically flash storage), to which you can write files. Considerations:

- Since these drives are separate from the shared file system, you don’t have to worry about stress on the file servers.
- These temporary file systems are wiped after your job finishes, so you have to do the post-processing in your job script.
- The capacity of these local drives are fairly limited; see the userguide for exact numbers.

### 10.1 File handling

MPI has its own file handle: `MPI_File`.

You open a file with `MPI_File_open` (figure 10.1). This routine is collective, even if only certain processes will access the file with a read or write call. Similarly, `MPI_File_close` is collective.

*Python note 27: File open is class method.* Note the slightly unusual syntax for opening a file:

```python
mpifile = MPI.File.Open(comm, filename, mode)
```

Even though the file is opened on a communicator, it is a class method for the `MPI.File` class, rather than for the communicator object. The latter is passed in as an argument.

File access modes:

- `MPI_MODE_RDONLY`: read only,
- `MPI_MODE_RDWR`: reading and writing,
- `MPI_MODE_WRONLY`: write only,
- `MPI_MODE_CREATE`: create the file if it does not exist,
- `MPI_MODE_EXCL`: error if creating file that already exists,
- `MPI_MODE_DELETE_ON_CLOSE`: delete file on close,
- `MPI_MODE_UNIQUE_OPEN`: file will not be concurrently opened elsewhere,
- `MPI_MODE_SEQUENTIAL`: file will only be accessed sequentially,
10. MPI topic: File I/O

Figure 10.2 MPI_File_seek

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_File_seek (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fh</td>
<td>file handle</td>
<td>MPI_File</td>
<td>TYPE(MPI_File)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>offset</td>
<td>file offset</td>
<td>MPI_Offset</td>
<td>INTEGER(KIND=MPI_OFFSET_KIND)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>whence</td>
<td>update mode</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10.3 MPI_File_write

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_File_write (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fh</td>
<td>file handle</td>
<td>MPI_File</td>
<td>TYPE(MPI_File)</td>
<td>INOUT</td>
</tr>
<tr>
<td></td>
<td>buf</td>
<td>initial address of buffer</td>
<td>const void*</td>
<td>TYPE(*)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>number of elements in buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>datatype</td>
<td>datatyp of each buffer</td>
<td></td>
<td>TYPE(MPI_Datatype)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>status</td>
<td>status object</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• MPI_MODE_APPEND: set initial position of all file pointers to end of file.

These modes can be added or bitwise-or’ed.

You can delete a file with MPI_File_delete.

Buffers can be flushed with MPI_File_sync, which is a collective call.

10.2 File reading and writing

The basic file operations, in between the open and close calls, are the POSIX-like, noncollective, calls

• MPI_File_seek (figure 10.2). The whence parameter can be:
  – MPI_SEEK_SET The pointer is set to offset.
  – MPI_SEEK_CUR The pointer is set to the current pointer position plus offset.
  – MPI_SEEK_END The pointer is set to the end of the file plus offset.

• MPI_File_write (figure 10.3). This routine writes the specified data in the locations specified with the current file view. The number of items written is returned in the MPI_Status argument; all other fields of this argument are undefined. It can not be used if the file was opened with MPI_MODE_SEQUENTIAL.

• If all processes execute a write at the same logical time, it is better to use the collective call MPI_File_write_all.
10.2. File reading and writing

Figure 10.4 MPI_File_read

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_File_read</td>
<td>fh</td>
<td>file handle</td>
<td>MPI_File</td>
<td>TYPE(MPI_File)</td>
<td>INOUT</td>
</tr>
<tr>
<td></td>
<td>buf</td>
<td>initial address of buffer</td>
<td>void*</td>
<td>TYPE(*),</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DIMENSION(..)</td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>count</td>
<td>number of elements in buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>datatype</td>
<td></td>
<td>datatype of each buffer</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td>status</td>
<td></td>
<td>status object</td>
<td>MPI_Status*</td>
<td>TYPE(MPI_Status)</td>
<td>OUT</td>
</tr>
</tbody>
</table>

• MPI_File_read (figure 10.4) This routine attempts to read the specified data from the locations specified in the current file view. The number of items read is returned in the MPI_Status argument; all other fields of this argument are undefined. It can not be used if the file was opened with MPI_MODE_SEQUENTIAL.

• If all processes execute a read at the same logical time, it is better to use the collective call MPI_File_read_all (figure 10.5).

For thread safety it is good to combine seek and read/write operations:

• MPI_File_read_at: combine read and seek. The collective variant is MPI_File_read_at_all.
• MPI_File_write_at: combine write and seek! The collective variant is MPI_File_write_at_all; section 10.2.2.

Writing to and reading from a parallel file is rather similar to sending a receiving:

• The process uses an elementary data type or a derived datatype to describe what elements in an array go to file, or are read from file.
• In the simplest case, your read or write that data to the file using an offset, or first having done a seek operation.
• But you can also set a ‘file view’ to describe explicitly what elements in the file will be involved.
10. MPI topic: File I/O

**Figure 10.6 MPI_File_iwrite**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_File_iwrite</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_File_iwrite_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fh</td>
<td>file handle</td>
<td>initial address of buffer</td>
<td>MPI_File</td>
<td>TYPE(MPI_File)</td>
<td>INOUT</td>
</tr>
<tr>
<td>buf</td>
<td></td>
<td></td>
<td>const void*</td>
<td>TYPE(*)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>INTEGER</td>
<td>number of elements in buffer</td>
<td>MPI_Count</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td>datatype</td>
<td>datatype of each buffer element</td>
<td></td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td>request</td>
<td>request object</td>
<td></td>
<td>MPI_Request*</td>
<td>TYPE(MPI_Request)</td>
<td>OUT</td>
</tr>
</tbody>
</table>

10.2.1 Nonblocking read/write

Just like there are blocking and nonblocking sends, there are also nonblocking writes and reads: MPI_File_iwrite (figure 10.6), MPI_File_iread operations, and their collective versions MPI_File_iwrite_all, MPI_File_iread_all.

Also MPI_File_iwrite_at, MPI_File_iwrite_at_all, MPI_File_iread_at, MPI_File_iread_at_all.

These routines output an MPI_Request object, which can then be tested with MPI_Wait or MPI_Test.

Nonblocking collective I/O functions much like other nonblocking collectives (section 3.11): the request is satisfied if all processes finish the collective.

There are also split collectives that function like nonblocking collective I/O, but with the request/wait mechanism: MPI_File_write_all_begin / MPI_File_write_all_end (and similarly MPI_File_read_all_begin / MPI_File_read_all_end) where the second routine blocks until the collective write/read has been concluded.

Also MPI_File_iread_shared, MPI_File_iwrite_shared.

10.2.2 Individual file pointers, contiguous writes

After the collective open call, each process holds an individual file pointer each process can individually position the pointer somewhere in the shared file. Let’s explore this modality.

The simplest way of writing a data to file is much like a send call: a buffer is specified with the usual count/datatype specification, and a target location in the file is given. The routine MPI_File_write_at (figure 10.7) gives this location in absolute terms with a parameter of type MPI_Offset, which counts bytes.

**Exercise 10.1.** Create a buffer of length nwords=3 on each process, and write these buffers as a sequence to one file with MPI_File_write_at.

(There is a skeleton for this exercise under the name blockwrite.)

Instead of giving the position in the file explicitly, you can also use a MPI_File_seek call to position the file pointer, and write with MPI_File_write at the pointer location. The write call itself also advances the file pointer so separate calls for writing contiguous elements need no seek calls with MPI_SEEK_CUR.
10.2. File reading and writing

Figure 10.7 MPI_File_write_at

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_File_write_at (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fh</td>
<td>file handle</td>
<td>MPI_File</td>
<td>TYPE(MPI_File)</td>
<td>INOUT</td>
</tr>
<tr>
<td></td>
<td>offset</td>
<td>file offset</td>
<td>MPI_Offset</td>
<td>INTEGER(KIND=MPI_OFFSET_KIND)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>buf</td>
<td>initial address of buffer</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>number of elements in buffer</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>datatype</td>
<td>datatype of each buffer element</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>status</td>
<td>status object</td>
<td>MPI_Status*</td>
<td>TYPE(MPI_Status)</td>
<td>OUT</td>
</tr>
</tbody>
</table>

Python:

```python
MPI.File.Write_at(self, Offset offset, buf, Status status=None)
```

Figure 10.1: Writing at an offset

Exercise 10.2. Rewrite the code of exercise 10.1 to use a loop where each iteration writes only one item to file. Note that no explicit advance of the file pointer is needed.

Exercise 10.3. Construct a file with the consecutive integers 0,...,WP where W some integer, and P the number of processes. Each process p writes the numbers p, p + W, p + 2W,.... Use a loop where each iteration
1. writes a single number with MPI_File_write, and
2. advanced the file pointer with MPI_File_seek with a whence parameter of MPI_SEEK_CUR.

10.2.3 File views

The previous mode of writing is enough for writing simple contiguous blocks in the file. However, you can also access noncontiguous areas in the file. For this you use MPI_File_set_view (figure 10.8). This call is collective, even if not all processes access the file.

- The disp displacement parameters is measured in bytes. It can differ between processes. On sequential files such as tapes or network streams it does not make sense to set a displacement; for those the MPI_DISPLACEMENT_CURRENT value can be used.
10. MPI topic: File I/O

Figure 10.8 MPI_File_set_view

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_File_set_view()</td>
<td></td>
<td>Mpi file handle</td>
<td>Mpi_File</td>
<td>type(MPI_File)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>disp</td>
<td>displacement</td>
<td>Mpi_Offset</td>
<td>INTEGER(KIND=MPI_OFFSET_KIND)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>etype</td>
<td>elementary datatype</td>
<td>Mpi_Datatype</td>
<td>type(MPI_Datatype)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>filetype</td>
<td>filetype</td>
<td>Mpi_Datatype</td>
<td>type(MPI_Datatype)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>datarep</td>
<td>data representation</td>
<td>const char*</td>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>info</td>
<td>info object</td>
<td>Mpi_Info</td>
<td>type(MPI_Info)</td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
mpifile = MPI.File.Open( .... )
mpifile.Set_view
( self,
  Offset disp=0, Datatype etype=None, Datatype filetype=None,
  datarep=None, Info info=INFO_NULL)
```

- The etype describes the data type of the file, it needs to be the same on all processes.
- The filetype describes how this process sees the file, so it can differ between processes.
- The datarep string can have the following values:
  - native: data on disk is represented in exactly the same format as in memory;
  - internal: data on disk is represented in whatever internal format is used by the MPI implementation;
  - external: data on disk is represented using XDR portable data formats.
- The info parameter is an MPI_Info object, or MPI_INFO_NULL. See section 15.1.1.3 for more. (See T3PIO [19] for a tool that assists in setting this object.)

Figure 10.2: Writing at a view
Exercise 10.4.

(There is a skeleton for this exercise under the name `viewwrite`.) Write a file in the same way as in exercise 10.1, but now use `MPI_File_write` and use `MPI_File_set_view` to set a view that determines where the data is written.

You can get very creative effects by setting the view to a derived datatype.

Figure 10.3: Writing at a derived type

Fortran note 11: Offset literals. In Fortran you have to assure that the displacement parameter is of ‘kind’ `MPI_OFFSET_KIND`. In particular, you can not specify a literal zero ‘0’ as the displacement; use `0_MPI_OFFSET_KIND` instead.

More: `MPI_File_set_size` `MPI_File_get_size` `MPI_File_preallocate` `MPI_File_get_view`

10.2.4 Shared file pointers

It is possible to have a file pointer that is shared (and therefore identical) between all processes of the communicator that was used to open the file. This file pointer is set with `MPI_File_seek_shared`. For reading and writing there are then two sets of routines:

- Individual accesses are done with `MPI_File_read_shared` and `MPI_File_write_shared`. Nonblocking variants are `MPI_File_iread_shared` and `MPI_File_iwrite_shared`.
- Collective accesses are done with `MPI_File_read_ordered` and `MPI_File_write_ordered`, which execute the operations in order ascending by rank.

Shared file pointers require that the same view is used on all processes. Also, these operations are less efficient because of the need to maintain the shared pointer.
10.3 Consistency

It is possible for one process to read data previously written by another process. For this it is of course necessary to impose a temporal order, for instance by using `MPI_Barrier`, or using a zero-byte send from the writing to the reading process.

However, the file also needs to be declared atomic: `MPI_File_set_atomicity`.

10.4 Constants

`MPI_SEEK_SET` used to be called `SEEK_SET` which gave conflicts with the C++ library. This had to be circumvented with

```make
make CPPFLAGS="-DMPICH_IGNORE_CXX_SEEK -DMPICH_SKIP_MPICXX"
```

and such.

10.5 Error handling

By default, MPI uses `MPI_ERRORS_ARE_FATAL` since parallel errors are almost impossible to recover from. File handling errors, on the other hand, are less serious: if a file is not found, the operation can be abandoned. For this reason, the default error handler for file operations is `MPI_ERRORS_RETURN`.

The default I/O error handler can be queried and set with `MPI_File_get_errhandler` and `MPI_File_set_errhandler` respectively, passing `MPI_FILE_NULL` as argument.
10.6 Review questions

Exercise 10.5. T/F? After your SLURM job ends, you can copy from the login node the files you’ve written to /tmp.

Exercise 10.6. T/F? File views (MPI_File_set_view) are intended to
- write MPI derived types to file; without them you can only write contiguous buffers;
- prevent collisions in collective writes; they are not needed for individual writes.

Exercise 10.7. The sequence MPI_File_seek_shared, MPI_File_read_shared can be replaced by MPI_File_seek, MPI_File_read if you make what changes?
10.7 Sources used in this chapter

10.7.1 Listing of code header
Chapter 11

MPI topic: Topologies

A communicator describes a group of processes, but the structure of your computation may not be such that every process will communicate with every other process. For instance, in a computation that is mathematically defined on a Cartesian 2D grid, the processes themselves act as if they are two-dimensionally ordered and communicate with N/S/E/W neighbors. If MPI had this knowledge about your application, it could conceivably optimize for it, for instance by renumbering the ranks so that communicating processes are closer together physically in your cluster.

The mechanism to declare this structure of a computation to MPI is known as a virtual topology. The following types of topology are defined:

- **MPI_UNDEFINED**: this values holds for communicators where no topology has explicitly been specified.
- **MPI_CART**: this value holds for Cartesian topologies, where processes act as if they are ordered in a multi-dimensional ‘brick’; see section 11.1.
- **MPI_GRAPH**: this value describes the graph topology that was defined in MPI-1; section 11.2.4. It is unnecessarily burdensome, since each process needs to know the total graph, and should therefore be considered obsolete; the type MPI_DIST_GRAPH should be used instead.
- **MPI_DIST_GRAPH**: this value describes the distributed graph topology where each process only describes the edges in the process graph that touch itself; see section 11.2.

These values can be discovered with the routine MPI_Topo_test.

11.1 Cartesian grid topology

A Cartesian grid is a structure, typically in 2 or 3 dimensions, of points that have two neighbors in each of the dimensions. Thus, if a Cartesian grid has sizes $K \times M \times N$, its points have coordinates $(k, m, n)$ with $0 \leq k < K$ et cetera. Most points have six neighbors $(k \pm 1, m, n), (k, m \pm 1, n), (k, m, n \pm 1)$; the exception are the edge points. A grid where edge processors are connected through wraparound connections is called a periodic grid.

The auxiliary routine MPI_Dims_create assists in finding a grid of a given dimension, attempting to minimize the diameter.
11. MPI topic: Topologies

Figure 11.1 MPI_Cart_create

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Cart_create (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>comm_old</td>
<td>input communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ndims</td>
<td>number of dimensions of Cartesian grid</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dims</td>
<td>integer array of size ndims specifying the number of processes in each dimension</td>
<td>const int[]</td>
<td>INTEGER(ndims)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>periods</td>
<td>logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension</td>
<td>const int[]</td>
<td>LOGICAL(ndims)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reorder</td>
<td>ranking may be reordered (true) or not (false)</td>
<td>int</td>
<td>LOGICAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>comm_cart</td>
<td>communicator with new Cartesian topology</td>
<td>MPI_Comm*</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Code:

```c
// cartdims.c
int *dimensions = (int*) malloc(dim*sizeof(int));
for (int idim=0; idim<dim; idim++)
  dimensions[idim] = 0;
MPI_Dims_create(nprocs, dim, dimensions);
```

Output:

```
mpicc -o cartdims cartdims.o
Cartesian grid size: 3 dim: 1 3
Cartesian grid size: 3 dim: 2 3 x 1
Cartesian grid size: 4 dim: 1 4
Cartesian grid size: 4 dim: 2 2 x 2
Cartesian grid size: 4 dim: 3 2 x 2 x 1
Cartesian grid size: 12 dim: 1 12
Cartesian grid size: 12 dim: 2 4 x 3
Cartesian grid size: 12 dim: 3 3 x 2 x 2
Cartesian grid size: 12 dim: 4 3 x 2 x 2 x 1
```

If the dimensions array is nonzero in a component, that one is not touched. Of course, the product of the specified dimensions has to divide in the input number of nodes.

### 11.1.1 Cartesian topology communicator

The cartesian topology is specified by giving `MPI_Cart_create` (figure 11.1) the sizes of the processor grid along each axis, and whether the grid is periodic along that axis.

```c
MPI_Comm cart_comm;
```
### 11.1. Cartesian grid topology

#### Figure 11.2 MPI_Topo_test

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Topo_test (</td>
<td></td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>status</td>
<td>topology type of communicator</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>comm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
int *periods = (int*) malloc(dim*sizeof(int));
for ( int id=0; id<dim; id++ ) periods[id] = 0;
MPI_Cart_create
  (comm,dim,dimensions,periods,
   0,&cart_comm );
```

(The Cartesian grid can have fewer processes than the input communicator: any processes not included
get MPI_COMM_NULL as output.)

For a given communicator, you can test what type it is with MPI_Topo_test (figure 11.2):

```c
int world_type,cart_type;
MPI_Topo_test( comm,&world_type);
MPI_Topo_test( cart_comm,&cart_type );
if (procno==0) {
  printf("World comm type=%d, Cart comm type=%d\n",
         world_type,cart_type);
  printf("no topo =%d, cart top =%d\n",
         MPI_UNDEFINED,MPI_CART);
}
```

For a Cartesian communicator, you can retrieve its information with MPI_Cartdim_get and MPI_Cart_get:

```c
int dim;
MPI_Cartdim_get( cart_comm,&dim );
int *dimensions = (int*) malloc(dim*sizeof(int));
int *periods   = (int*) malloc(dim*sizeof(int));
int *coords    = (int*) malloc(dim*sizeof(int));
MPI_Cart_get( cart_comm,dim,dimensions,periods,coords );
```

### 11.1.2 Cartesian vs world rank

Each point in this new communicator has a coordinate and a rank. The translation from rank to Cartesian coordinate is done by MPI_Cart_coords (figure 11.3), and translation from coordinates to a rank is done by MPI_Cart_rank (figure 11.4). In both cases, this translation can be done on any process; for the latter routine note that coordinates outside the Cartesian grid are erroneous, if the grid is not periodic in the offending coordinate.

```c
// cart.c
MPI_Com comm2d;
int periodic[ndim]; periodic[0] = periodic[1] = 0;
```
11. MPI topic: Topologies

Figure 11.3 MPI_Cart_coords

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Cart_coords</td>
<td>comm</td>
<td>communicator with Cartesian structure</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rank</td>
<td>rank of a process within group of comm</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>maxdims</td>
<td>length of vector coords in the calling program</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>coords</td>
<td>integer array (of size maxdims) containing the Cartesian coordinates of specified process</td>
<td>int[]</td>
<td>INTEGER(maxdims)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 11.4 MPI_Cart_rank

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Cart_rank</td>
<td>comm</td>
<td>communicator with Cartesian structure</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>coords</td>
<td>integer array (of size ndims) specifying the Cartesian coordinates of a process</td>
<td>const int[]</td>
<td>INTEGER(*)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rank</td>
<td>rank of specified process</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
</tbody>
</table>

MPI_Cart_create(comm,ndim,dimensions,periodic,1,&comm2d);
MPI_Cart_coords(comm2d,procno,ndim,coord_2d);
MPI_Cart_rank(comm2d,coord_2d,&rank_2d);
printf("I am %d: (%d,%d); originally %d\n", rank_2d,coord_2d[0],coord_2d[1],procno);

For the full source of this example, see section 11.3.2

The reorder parameter to MPI_Cart_create indicates whether processes can have a rank in the new communicator that is different from in the old one.

11.1.3 Cartesian communication

A common communication pattern in Cartesian grids is to do an MPI_Sendrecv with processes that are adjacent along one coordinate axis.

By way of example, consider a 3D grid that is periodic in the first dimension:

```c
// cartcoord.c
for ( int id=0; id<dim; id++)
    periods[id] = id==0 ? 1 : 0;
MPI_Cart_create`
```
11.1. Cartesian grid topology

Figure 11.5 MPI_Cart_sub

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Cart_sub</td>
<td>comm</td>
<td>communicator with Cartesian structure</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>remain_dims</td>
<td>the i-th entry of remain_dims specifies whether the i-th dimension is kept in the subgrid (true) or is dropped (false)</td>
<td>const int[]</td>
<td>LOGICAL(*)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>newcomm</td>
<td>communicator containing the subgrid that includes the calling process</td>
<td>MPI_Comm*</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
</tbody>
</table>

```c
MPI_Cart_sub (comm, dim, dimensions, periods, 0, &period_comm);
```

For the full source of this example, see section 11.3.3

We shift process 0 in dimensions 0 and 1. In dimension 0 we get a wrapped-around source, and a target that is the next process in row-major ordering; in dimension 1 we get MPI_PROC_NULL as source, and a legitimate target.

Code:

```c
int pred, succ;
MPI_Cart_shift
    (period_comm,/* dim: */ 0,/* up: */ 1,
     &pred,&succ);
printf("periodic dimension 0:\n src=%d, tgt=%d\n",
      pred,succ);
MPI_Cart_shift
    (period_comm,/* dim: */ 1,/* up: */ 1,
     &pred,&succ);
printf("non-periodic dimension 1:\n src=%d, tgt=%d\n",
      pred,succ);
```

Output:

Grid of size 6 in 3 dimensions:
3 x 2 x 1
Shifting process 0.
periodic dimension 0:
src=4, tgt=2
non-periodic dimension 1:
src=-1, tgt=1

11.1.4 Communicators in subgrids

The routine MPI_Cart_sub (figure 11.5) is similar to MPI_Comm_split, in that it splits a communicator into disjoint subcommunicators. In this case, it splits a Cartesian communicator into disjoint Cartesian communicators, each corresponding to a subset of the dimensions. This subset inherits both sizes and periodicity from the original communicator.
11. MPI topic: Topologies

Code:

```c
MPI_Cart_sub( period_comm, remain, &hyperplane);
if (procno==0) {
    MPI_Topo_test( hyperplane, &topo_type );
    MPI_Cartdim_get( hyperplane, &hyperdim );
    printf("hyperplane has dimension %d, type %d\n", hyperdim, topo_type);
    MPI_Cart_get( hyperplane, dim, dims, period, coords );
    printf(" periodic: ");
    for (int id=0; id<2; id++)
        printf("%d," ,period[id]);
    printf("\n");
}
```

Output:

```
hyperplane has dimension 2, type 2
periodic: 1,0,
```

11.1.5 Reordering

The routine `MPI_Cart_map` gives a re-ordered rank for the calling process.

11.2 Distributed graph topology

![Figure 11.1: Illustration of a distributed graph topology where each node has four neighbors](image)

In many calculations on a grid (using the term in its mathematical, Finite Element Method (FEM), sense), a grid point will collect information from grid points around it. Under a sensible distribution of the grid over processes, this means that each process will collect information from a number of neighbor processes. The number of neighbors is dependent on that process. For instance, in a 2D grid (and assuming a five-point stencil for the computation) most processes communicate with four neighbors; processes on the edge with three, and processes in the corners with two.

Such a topology is illustrated in figure 11.1.
MPI’s notion of graph topology, and the neighborhood collectives, offer an elegant way of expressing such communication structures. There are various reasons for using graph topologies over the older, simpler methods.

- MPI is allowed to reorder the processes, so that network proximity in the cluster corresponds to proximity in the structure of the code.
- Ordinary collectives could not directly be used for graph problems, unless one would adopt a subcommunicator for each graph neighborhood. However, scheduling would then lead to deadlock or serialization.
- The normal way of dealing with graph problems is through nonblocking communications. However, since the user indicates an explicit order in which they are posted, congestion at certain processes may occur.
- Collectives can pipeline data, while send/receive operations need to transfer their data in its entirety.
- Collectives can use spanning trees, while send/receive uses a direct connection.

Thus the minimal description of a process graph contains for each process:

- Degree: the number of neighbor processes; and
- the ranks of the processes to communicate with.

However, this ignores that communication is not always symmetric: maybe the processes you receive from are not the ones you send to. Worse, maybe only one side of this duality is easily described. Therefore, there are two routines:

- MPI_Dist_graph_create_adjacent assumes that a process knows both who it is sending it, and who will send to it. This is the most work for the programmer to specify, but it is ultimately the most efficient.
- MPI_Dist_graph_create specifies on each process only what it is the source for; that is, who this process will be sending to. Consequently, some amount of processing – including communication – is needed to build the converse information, the ranks that will be sending to a process.

### 11.2.1 Graph creation

There are two creation routines for process graphs. These routines are fairly general in that they allow any process to specify any part of the topology. In practice, of course, you will mostly let each process describe its own neighbor structure.

The routine MPI_Dist_graph_create_adjacent assumes that a process knows both who it is sending it, and who will send to it. This means that every edge in the communication graph is represented twice, so the memory footprint is double of what is strictly necessary. However, no communication is needed to build the graph.

The second creation routine, MPI_Dist_graph_create (figure 11.6), is probably easier to use, especially in cases where the communication structure of your program is symmetric, meaning that a process sends to the same neighbors that it receives from. Now you specify on each process only what it is the source for; that is, who this process will be sending to. Consequently, some amount of processing – including communication – is needed to build the converse information, the ranks that will be sending to a process.

---

1. I disagree with this design decision. Specifying your sources is usually easier than specifying your destinations.
11. MPI topic: Topologies

Figure 11.6 MPI_Dist_graph_create

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Dist_graph_create</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>comm_old</td>
<td></td>
<td>input communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td>number of source nodes for which this process specifies edges</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>sources</td>
<td></td>
<td>array containing the n source nodes for which this process specifies edges</td>
<td>const int[]</td>
<td>INTEGER(n)</td>
<td></td>
</tr>
<tr>
<td>degrees</td>
<td></td>
<td>array specifying the number of destinations for each source node in the source node array</td>
<td>const int[]</td>
<td>INTEGER(n)</td>
<td></td>
</tr>
<tr>
<td>destinations</td>
<td></td>
<td>destination nodes for the source nodes in the source node array</td>
<td>const int[]</td>
<td>INTEGER(*)</td>
<td></td>
</tr>
<tr>
<td>weights</td>
<td></td>
<td>weights for source to destination edges</td>
<td>const int[]</td>
<td>INTEGER(*)</td>
<td></td>
</tr>
<tr>
<td>info</td>
<td></td>
<td>hints on optimization and interpretation of weights</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
</tr>
<tr>
<td>reorder</td>
<td></td>
<td>the ranks may be reordered (true) or not (false)</td>
<td>int</td>
<td>LOGICAL</td>
<td></td>
</tr>
<tr>
<td>comm_dist_graph</td>
<td></td>
<td>communicator with distributed graph topology added</td>
<td>MPI_Comm*</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
</tbody>
</table>

MPL:

dist_graph_communicator
    (const communicator &old_comm,
     const source_set &ss, const dest_set &ds, bool reorder=true)

where:

class dist_graph_communicator::source_set : private set< pair<int,int> >

class dist_graph_communicator::dest_set : private set< pair<int,int> >

Python:

MPI.Comm.Create_dist_graph
    (self, sources, degrees, destinations, weights=None, Info info=INFO_NULL, bool reorder=False)

returns graph communicator

MPL note 56: Distributed graph creation. The class mpl::dist_graph_communicator only has a constructor corresponding to MPI_Dist_graph_create.

Figure 11.1 describes the common five-point stencil structure. If we let each process only describe itself, we get the following:

- nsources = 1 because the calling process describes on node in the graph: itself.
- sources is an array of length 1, containing the rank of the calling process.
- degrees is an array of length 1, containing the degree (probably: 4) of this process.
- destinations is an array of length the degree of this process, probably again 4. The elements of this array are the ranks of the neighbor nodes; strictly speaking the ones that this process will send to.
11.2. Distributed graph topology

- weights is an array declaring the relative importance of the destinations. For an unweighted graph use MPI_UNWEIGHTED. In the case the graph is weighted, but the degree of a source is zero, you can pass an empty array as MPI_WEIGHTS_EMPTY.
- reorder (int in C, LOGICAL in Fortran) indicates whether MPI is allowed to shuffle processes to achieve greater locality.

The resulting communicator has all the processes of the original communicator, with the same ranks. In other words MPI_Comm_size and MPI_Comm_rank gives the same values on the graph communicator, as on the intra-communicator that it is constructed from. To get information about the grouping, use MPI_Dist_graph_neighbors and MPI_Dist_graph_neighbors_count; section 11.2.3.

By way of example we build an unsymmetric graph, that is, an edge $v_1 \to v_2$ between vertices $v_1, v_2$ does not imply an edge $v_2 \to v_1$.

```c
// graph.c
for ( int i=0; i<1; i++ ) {
  int neighb_i = proci+i;
  if (neighb_i<0 || neighb_i>=idim) continue;
  for ( int j=0; j<1; j++ ) {
    int neighb_j = procj+j;
    if (neighb_j<0 || neighb_j>=jdim) continue;
    destinations[ degree++ ] = PROC(neighb_i,neighb_j,idim,jdim);
  }
}
MPI_Dist_graph_create
(comm,
  /* I specify just one proc: me */ 1,
  &procno,&degree,destinations,weights,
  MPI_INFO_NULL,0,
  &comm2d
);
```

![Diagram of an unsymmetric graph](image_url)
11. MPI topic: Topologies

**Code:**

```c
int indegree, outdegree, weighted;
MPI_Dist_graph_neighbors_count(comm2d, &indegree, &outdegree, &weighted);
int my_ij[2] = {proci, procj}, other_ij[4][2];
MPI_Neighbor_allgather (my_ij, 2, MPI_INT, other_ij, 2, MPI_INT, comm2d);
```

**Output:**

- `[ 0 = (0,0) ]` has 4 outbound neighbours: 0, 1, 2, 3, 1 inbound neighbors: ( 0, 0)= 0
- `[ 1 = (0,1) ]` has 2 outbound neighbours: 1, 3, 2 inbound neighbors: ( 0, 1)= 1 ( 0, 0)= 0
- `[ 2 = (1,0) ]` has 4 outbound neighbours: 2, 3, 4, 5, 2 inbound neighbors: ( 1, 0)= 2 ( 0, 0)= 0
- `[ 3 = (1,1) ]` has 2 outbound neighbours: 3, 5, 4 inbound neighbors: ( 1, 1)= 3 ( 0, 1)= 1 ( 1, 0)= 2
- `[ 4 = (2,0) ]` has 2 outbound neighbours: 4, 5, 2 inbound neighbors: ( 2, 0)= 4 ( 1, 0)= 2
- `[ 5 = (2,1) ]` has 1 outbound neighbours: 5, 4 inbound neighbors: ( 2, 1)= 5 ( 1, 1)= 3 ( 2, 0)= 4

**Python note 28: Graph communicators.** Graph communicator creation is a method of the `Comm` class, and
the graph communicator is a function return result:

```python
    || graph_comm = oldcomm.Create_dist_graph(sources, degrees, destinations)
```

The weights, info, and reorder arguments have default values.

**MPL note 57: Graph communicators.** The constructor `dist_graph_communicator`

```python
    (const communicator old_comm, const source_set ss, const dest_set ds, bool reorder = true);
```

is a wrapper around `MPI_Dist_graph_create_adjacent`.

**MPL note 58: Graph communicator querying.** Methods `indegree, outdegree` are wrappers around `MPI_Dist_graph_neighbors_count`.
Sources and targets can be queried with `inneighbors` and `outneighbors`, which are wrappers around `MPI_Dist_graph_neighbors`.

### 11.2.2 Neighbor collectives

We can now use the graph topology to perform a gather or allgather `MPI_Neighbor_allgather` (figure 11.7) that combines only the processes directly connected to the calling process.

The neighbor collectives have the same argument list as the regular collectives, but they apply to a graph communicator.

![Figure 11.2: Solving the right-send exercise with neighborhood collectives](image)
11.2. Distributed graph topology

Figure 11.7 MPI Neighbor_allgather

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>List of Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Neighbor_allgather (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Neighbor_allgather_c (</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendbuf</td>
<td>starting address of send buffer</td>
<td></td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(...)</td>
<td>IN</td>
</tr>
<tr>
<td>sendcount</td>
<td>number of elements sent to each neighbor</td>
<td></td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendtype</td>
<td>datatype of send buffer elements</td>
<td></td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td>recvbuf</td>
<td>starting address of receive buffer</td>
<td></td>
<td>void*</td>
<td>TYPE(*), DIMENSION(...)</td>
<td>OUT</td>
</tr>
<tr>
<td>recvcount</td>
<td>number of elements received from each neighbor</td>
<td></td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Count</td>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>recvtype</td>
<td>datatype of receive buffer elements</td>
<td></td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
</tr>
<tr>
<td>comm</td>
<td>communicator with topology structure</td>
<td></td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
</tbody>
</table>

Exercise 11.1. Revisit exercise 4.3 and solve it using MPI_Dist_graph_create. Use figure 11.2 for inspiration.
Use a degree value of 1.
(There is a skeleton for this exercise under the name rightgraph.)

The previous exercise can be done with a degree value of:

- 1, reflecting that each process communicates with just 1 other; or
- 2, reflecting that you really gather from two processes.

In the latter case, results do not wind up in the receive buffer in order of increasing process number as with a traditional gather. Rather, you need to use MPI_Dist_graph_neighbors to find their sequencing; see section 11.2.3.

Another neighbor collective is MPI_Neighbor_alltoall.

The vector variants are MPI_Neighbor_allgatherv and MPI_Neighbor_alltoallv.

There is a heterogenous (multiple datatypes) variant: MPI_Neighbor_alltoallw.

The list is: MPI_Neighbor_allgather, MPI_Neighbor_allgatherv, MPI_Neighbor_alltoall, MPI_Neighbor_alltoallv, MPI_Neighbor_alltoallw.

Nonblocking: MPI_Ineighbor_allgather, MPI_Ineighbor_allgatherv, MPI_Ineighbor_alltoall, MPI_Ineighbor_alltoallv, MPI_Ineighbor_alltoallw.

For unclear reasons there is no MPI_Neighbor_allreduce.

Victor Eijkhout 377
11. MPI topic: Topologies

### Figure 11.8 MPI_Dist_graph_neighbors_count

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Dist_graph_neighbors_count (</td>
<td>comm</td>
<td>communicator with distributed graph topology</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>indegree</td>
<td>number of edges into this process</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>outdegree</td>
<td>number of edges out of this process</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>weighted</td>
<td>false if MPI_UNWEIGHTED was supplied during creation, true otherwise</td>
<td>int*</td>
<td>LOGICAL</td>
<td></td>
</tr>
</tbody>
</table>

### Figure 11.9 MPI_Dist_graph_neighbors

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Dist_graph_neighbors (</td>
<td>comm</td>
<td>communicator with distributed graph topology</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>maxindegree</td>
<td>size of sources and sourceweights arrays</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sources</td>
<td>processes for which the calling process is a destination</td>
<td>int[]</td>
<td>INTEGER(maxindegree)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sourceweights</td>
<td>weights of the edges into the calling process</td>
<td>int[]</td>
<td>INTEGER(*)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>maxoutdegree</td>
<td>size of destinations and destweights arrays</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>destinations</td>
<td>processes for which the calling process is a source</td>
<td>int[]</td>
<td>INTEGER(maxoutdegree)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>destweights</td>
<td>weights of the edges out of the calling process</td>
<td>int[]</td>
<td>INTEGER(*)</td>
<td></td>
</tr>
</tbody>
</table>

### 11.2.3 Query

There are two routines for querying the neighbors of a process: MPI_Dist_graph_neighbors_count (figure 11.8) and MPI_Dist_graph_neighbors (figure 11.9).

While this information seems derivable from the graph construction, that is not entirely true for two reasons.

1. With the nonadjoinig version MPI_Dist_graph_create, only outdegrees and destinations are specified; this call then supplies the indegrees and sources;
2. As observed above, the order in which data is placed in the receive buffer of a gather call is not determined by the create call, but can only be queried this way.
11.2.4 Graph topology (deprecated)

The original MPI-1 had a graph topology interface `MPI_Graph_create` which required each process to specify the full process graph. Since this is not scalable, it should be considered deprecated. Use the distributed graph topology (section 11.2) instead.

Other legacy routines: `MPI_Graph_neighbors`, `MPI_Graph_neighbors_count`, `MPI_Graph_get`, `MPI_Graphdims_get`.

11.2.5 Re-ordering

The routine `MPI_Graph_map` gives a re-ordered rank for the calling process.
11.3 Sources used in this chapter

11.3.1 Listing of code header

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"

int main(int argc,char **argv) {
    #include "globalinit.c"
    int ndim = 2; int dimensions[2];
    dimensions[0] = 0; dimensions[1] = 0;
    MPI_Dims_create( nprocs,ndim,dimensions );

    MPI_Comm comm2d;
    int periodic[ndim]; periodic[0] = periodic[1] = 0;
    MPI_Cart_create(comm,ndim,dimensions,periodic,1,&comm2d);
    MPI_Cart_coords(comm2d,procno,ndim,coord_2d);
    MPI_Cart_rank(comm2d,coord_2d,&rank_2d);
    printf("I am %d: (%d,%d); originally %d\n",
        rank_2d,coord_2d[0],coord_2d[1],procno);

    int rank_left,rank_right,rank_up,rank_down;
    char indata[4]; int idata=0,sdata=0;
    for (int i=0; i<4; i++)
        indata[i] = 32;
    char mychar = 65+procno;
    MPI_Cart_shift(comm2d,0,+1,&rank_2d,&rank_right);
    MPI_Cart_shift(comm2d,0,-1,&rank_2d,&rank_left);
    MPI_Cart_shift(comm2d,1,+1,&rank_2d,&rank_up);
    MPI_Cart_shift(comm2d,1,-1,&rank_2d,&rank_down);
    int irequest = 0; MPI_Request *requests = malloc(8*sizeof(MPI_Request));
    MPI_Isend(&mychar,1,MPI_CHAR,rank_right, 0,comm, requests+irequest++);
    MPI_Isend(&mychar,1,MPI_CHAR,rank_left, 0,comm, requests+irequest++);
    MPI_Isend(&mychar,1,MPI_CHAR,rank_up, 0,comm, requests+irequest++);
    MPI_Isend(&mychar,1,MPI_CHAR,rank_down, 0,comm, requests+irequest++);
    MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_right, 0,comm, requests+irequest++);
    MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_left, 0,comm, requests+irequest++);
    MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_up, 0,comm, requests+irequest++);
    MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_down, 0,comm, requests+irequest++);
    MPI_Waitall(irequest,requests,MPI_STATUSES_IGNORE);
    printf("[%d] %s\n",procno,indata);
    /* for (int i=0; i<4; i++) */
    /* sdata += indata[i]; */
    /* printf("[%d] %d,%d,%d,%d sum=%d\n",procno,indata[0],indata[1],indata[2],indata[3],sdata); */
    if (procno==0)
```

380
11.3. Sources used in this chapter

```c
printf("Finished\n");
MPI_Finalize();
return 0;
}

11.3.3 Listing of code examples/mpi/c/cartcoord.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"

void print_grid( int nprocs,int dim,int dimensions[] ) {
    printf("Grid of size %d in %d dimensions:\n ",nprocs,dim);
    for (int idim=0; idim<dim; idim++) {
        printf("%2d",dimensions[idim]);
        if (idim<dim-1) printf(" x ");
    }
    printf("\n");
}

int main(int argc,char **argv) {

    #include "globalinit.c"
    const int dim = 3;
    if (procno==0)
        printf("dim=%d\n",dim);

    /*
     * Create 3D brick
     */
    int *dimensions = (int*) malloc(dim*sizeof(int));
    for (int idim=0; idim<dim; idim++)
        dimensions[idim] = 0;
    MPI_Dims_create(nprocs,dim,dimensions);
    int *periods = (int*) malloc(dim*sizeof(int));
    for ( int id=0; id<dim; id++)
        periods[id] = 0;
    if (procno==0) {
        print_grid( nprocs,dim,dimensions );
    }
    MPI_Comm cart_comm;
    MPI_Cart_create
        ( comm,dim,dimensions,periods,
          0,&cart_comm );
    MPI_Comm period_comm;
    for ( int id=0; id<dim; id++)
        periods[id] = id==0 ? 1 : 0;
    MPI_Cart_create
        ( comm,dim,dimensions,periods,
          0,&period_comm );

Victor Eijkhout 381
/*
 * Translation rank -> coord
 */
if (procno==0) {
  int *coord = (int*) malloc( dim*sizeof(int) );
  for ( int ip=0; ip<nprocs; ip++ ) {
    /*
      * Translate process rank to cartesian coordinate
      */
    MPI_Cart_coords( cart_comm,ip,dim,coord );
    printf(" [%2d] coord: [",ip);
    for ( int id=0; id<dim; id++ )
      printf("%d,",coord[id]);
    printf(" ]\n");
    /*
      * Shift the coordinate and translate back to rank
      * This is erroneous for a non-periodic Cartesian grid
      */
    int rank_check;
    coord[0]++;
    MPI_Cart_rank( cart_comm,coord,&rank_check );
    printf(" shifted neighbor : %2d\n",rank_check);
    MPI_Cart_rank( period_comm,coord,&rank_check );
    printf(" periodic neighbor: %2d\n",rank_check);
  }
  free(coord);
}

/*
 * Shifted coordinates
 */
if (procno==0) {
  if (dimensions[1]==1) {
    printf("Too few processes: need non-trivial dimensions[1]\n");
  } else {
    printf("\nCartShift\n");
    print_grid(nprocs,dim,dimensions);
    printf("Shifting process 0.\n");
    int pred,succ;
    MPI_Cart_shift
      (period_comm,/* dim: */ 0,/* up: */ 1,
       &pred,&succ);
    printf("periodic dimension 0: src=%d, tgt=%d\n", pred,succ);
    MPI_Cart_shift
      (period_comm,/* dim: */ 1,/* up: */ 1,
       &pred,&succ);
    printf("non-periodic dimension 1: src=%d, tgt=%d\n", pred,succ);
    printf("cartshift\n");
  }
}
11.3. Sources used in this chapter

```c
/* Subdimensions */
{
  int remain[] = {1,0,1};
  int topo_type, hyperdim, dims[3], period[3], coords[3];
  MPI_Comm hyperplane;

  if (procno==0) printf("Hyperplane13\n");
  MPI_Cart_sub( cart_comm,remain,&hyperplane );
  if (procno==0) {
    MPI_Topo_test( hyperplane,&topo_type );
    MPI_Cartdim_get( hyperplane,&hyperdim );
    printf("hyperplane has dimension %d, type %d\n", hyperdim,topo_type);
    MPI_Cart_get( hyperplane,dim,dims,period,coords );
    printf(" periodic: ");
    for (int id=0; id<2; id++) printf("%d,",period[id]);
    printf("\n");
  }
  MPI_Comm_free( &hyperplane );
  if (procno==0) printf("hyperplane13\n");

  if (procno==0) printf("Hyperplane13p\n");
  MPI_Cart_sub( period_comm,remain,&hyperplane );
  if (procno==0) {
    MPI_Topo_test( hyperplane,&topo_type );
    MPI_Cartdim_get( hyperplane,&hyperdim );
    printf("hyperplane has dimension %d, type %d\n", hyperdim,topo_type);
    MPI_Cart_get( hyperplane,dim,dims,period,coords );
    printf(" periodic: ");
    for (int id=0; id<2; id++) printf("%d,",period[id]);
    printf("\n");
  }
  MPI_Comm_free( &hyperplane );
  if (procno==0) printf("hyperplane13p\n");

  free(dimensions); free(periods);
  MPI_Finalize();
  return 0;
}
```
Chapter 12

MPI topic: Shared memory

Some programmers are under the impression that MPI would not be efficient on shared memory, since all operations are done through what looks like network calls. This is not correct: many MPI implementations have optimizations that detect shared memory and can exploit it, so that data is copied, rather than going through a communication layer. (Conversely, programming systems for shared memory such as OpenMP can actually have inefficiencies associated with thread handling.) The main inefficiency associated with using MPI on shared memory is then that processes can not actually share data.

The one-sided MPI calls (chapter 9) can also be used to emulate shared memory, in the sense that an origin process can access data from a target process without the target’s active involvement. However, these calls do not distinguish between actually shared memory and one-sided access across the network.

In this chapter we will look at the ways MPI can interact with the presence of actual shared memory. (This functionality was added in the MPI-3 standard.) This relies on the MPI_Win windows concept, but otherwise uses direct access of other processes’ memory.

12.1 Recognizing shared memory

MPI’s one-sided routines take a very symmetric view of processes: each process can access the window of every other process (within a communicator). Of course, in practice there will be a difference in performance depending on whether the origin and target are actually on the same shared memory, or whether they can only communicate through the network. For this reason MPI makes it easy to group processes by shared memory domains using MPI_Comm_split_type (figure 12.1).

Here the split_type parameter has to be from the following (short) list:

- **MPI_COMM_TYPE_SHARED**: split the communicator into subcommunicators of processes sharing a memory area.
- **MPI_COMM_TYPE_HW_GUIDED** (MPI-4): split using an info value from MPI_Get_hw_resource_types.
- **MPI_COMM_TYPE_HW_UNGUIDED** (MPI-4): similar to MPI_COMM_TYPE_HW_GUIDED, but the resulting communicators should be a strict subset of the original communicator. On processes where this condition can not be fullfilled, MPI_COMM_NULL will be returned.

**End of MPI-4 material**
12.1. Recognizing shared memory

**Figure 12.1 MPI_Comm_split_type**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_split_type()</td>
<td>comm</td>
<td>communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>split_type</td>
<td>type of processes to be grouped together</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>key</td>
<td>control of rank assignment</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>info</td>
<td>info argument</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>newcomm</td>
<td>new communicator</td>
<td>MPI_Comm*</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Comm.Split_type(
    self, int split_type, int key=0, Info info=INFO_NULL)
```

Splitting by shared memory:

**Code:**

```c
// commsplittype.c
MPI_Info info;
MPI_Comm_split_type
    (MPI_COMM_WORLD,
     MPI_COMM_TYPE_SHARED,
     procno,info,&sharedcomm);
MPI_Comm_size
    (sharedcomm,&new_nprocs);
MPI_Comm_rank
    (sharedcomm,&new_procno);
```

**Output:**

```
make[2]: `commsplittype' is up to date.
TACC: Starting up job 3954439
TACC: Starting parallel tasks...
There are 8 ranks total
[0] is processor 0 in a shared group of 4, running
[4] is processor 0 in a shared group of 4, running
TACC: Shutdown complete. Exiting.
```

**Exercise 12.1.** Write a program that uses `MPI_Comm_split_type` to analyze for a run

1. How many nodes there are;
2. How many processes there are on each node.

If you run this program on an unequal distribution, say 10 processes on 3 nodes, what distribution do you find?

```
Nodes: 3; processes: 10
TACC: Starting up job 4210429
TACC: Starting parallel tasks...
There are 3 nodes
Node sizes: 4 3 3
TACC: Shutdown complete. Exiting.
```

**Remark 19** The OpenMPI implementation of MPI has a number of non-standard split types, such as `OMPI_COMM_TYPE_SOCKET`; see https://www.open-mpi.org/doc/v4.1/man3/MPI_Comm_split_type.3.php

**MPL note 59:** Split by shared memory. Similar to ordinary communicator splitting 55: `communicator::split_shared`. 

Victor Eijkhout 385
12. MPI topic: Shared memory

Figure 12.2 MPI_Win_allocate_shared

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_allocate_shared</td>
<td>size</td>
<td>size of local window in bytes</td>
<td>MPI_Aint</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
</tr>
<tr>
<td>MPI_Win_allocate_shared_c</td>
<td>disp_unit</td>
<td>local unit size for displacements, in bytes</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>info</td>
<td>info argument</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>comm</td>
<td>intra-communicator</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>baseptr</td>
<td>address of local allocated window segment</td>
<td>void*</td>
<td>TYPE(C_PTR)</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>window object returned by the call</td>
<td>MPI_Win*</td>
<td>TYPE(MPI_Win)</td>
<td>OUT</td>
</tr>
</tbody>
</table>

12.2 Shared memory for windows

Processes that exist on the same physical shared memory should be able to move data by copying, rather than through MPI send/receive calls – which of course will do a copy operation under the hood. In order to do such user-level copying:

1. We need to create a shared memory area with MPI_Win_allocate_shared. This creates a window with the unified memory model; and
2. We need to get pointers to where a process’ area is in this shared space; this is done with MPI_Win_shared_query.

12.2.1 Pointers to a shared window

The first step is to create a window (in the sense of one-sided MPI; section 9.1) on the processes on one node. Using the MPI_Win_allocate_shared (figure 12.2) call presumably will put the memory close to the socket on which the process runs.

```c
// sharedbulk.c
MPI_Win node_window;
MPI_Aint window_size; double *window_data;
if (onnode_procid==0)
    window_size = sizeof(double);
else window_size = 0;
MPI_Win_allocate_shared
    (window_size,sizeof(double),MPI_INFO_NULL,
         nodecomm,
         &window_data,&node_window);
```

For the full source of this example, see section 12.3.2

The memory allocated by MPI_Win_allocate_shared is contiguous between the processes. This makes it possible to do address calculation. However, if a cluster node has a Non-Uniform Memory Access (NUMA) structure, for instance if two sockets have memory directly attached to each, this would increase latency.
for some processes. To prevent this, the key `alloc_shared_noncontig` can be set to `true` in the `MPI_Info` object.

The following material is for the recently released MPI-4 standard and may not be supported yet.

In the contiguous case, the `mpi_minimum_memory_alignment info` argument (section 9.1.1) applies only to the memory on the first process; in the noncontiguous case it applies to all.

End of MPI-4 material

```c
// numa.c
MPI_Info window_info;
MPI_Info_create(&window_info);
MPI_Info_set(window_info,"alloc_shared_noncontig","true");
MPI_Win_allocate_shared( window_size, sizeof(double),window_info,
nodecomm,
&window_data,&node_window);
MPI_Info_free(&window_info);
```

For the full source of this example, see section 12.3.3

Let’s explore this. We create a shared window where each process stores exactly one double, that is, 8 bytes. The following code fragment queries the window locations, and prints the distance in bytes to the window on process 0.

```c
for (int p=1; p<onnnode_procs; p++) {
    MPI_Aint window_sizep; int windowp_unit; double *winp_addr;
    MPI_Win_shared_query( node_window,p,
        &window_sizep,&windowp_unit, &winp_addr);
    distp = (size_t)winp_addr-(size_t)win0_addr;
    if (procno==0)
        printf("Distance %d to zero: %ld\n",p,(long)distp);
```

For the full source of this example, see section 12.3.3

With the default strategy, these windows are contiguous, and so the distances are multiples of 8 bytes. Not so for the the non-contiguous allocation:

<table>
<thead>
<tr>
<th>Strategy: default behavior of shared window allocation</th>
<th>Strategy: allow non-contiguous shared window allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance 1 to zero: 8</td>
<td>Distance 1 to zero: 4096</td>
</tr>
<tr>
<td>Distance 2 to zero: 16</td>
<td>Distance 2 to zero: 8192</td>
</tr>
<tr>
<td>Distance 3 to zero: 24</td>
<td>Distance 3 to zero: 12288</td>
</tr>
<tr>
<td>Distance 4 to zero: 32</td>
<td>Distance 4 to zero: 16384</td>
</tr>
<tr>
<td>Distance 5 to zero: 40</td>
<td>Distance 5 to zero: 20480</td>
</tr>
<tr>
<td>Distance 6 to zero: 48</td>
<td>Distance 6 to zero: 24576</td>
</tr>
<tr>
<td>Distance 7 to zero: 56</td>
<td>Distance 7 to zero: 28672</td>
</tr>
<tr>
<td>Distance 8 to zero: 64</td>
<td>Distance 8 to zero: 32768</td>
</tr>
<tr>
<td>Distance 9 to zero: 72</td>
<td>Distance 9 to zero: 36864</td>
</tr>
</tbody>
</table>

The explanation here is that each window is placed on its own `small page`, which on this particular system has a size of 4K.

Victor Eijkhout
12. MPI topic: Shared memory

Figure 12.3 MPI_Win_shared_query

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_shared_query</td>
<td>win</td>
<td>shared memory window object</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>rank</td>
<td>rank in the group of window</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>win</td>
<td>win or MPI_PROC_NULL</td>
<td>win or MPI_PROC_NULL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>size of the window segment</td>
<td>MPI_Aint*</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>disp_unit</td>
<td>local unit size for displacements, in bytes</td>
<td>int*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Aint</td>
<td>integer</td>
<td>MPI_Aint</td>
<td></td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td>baseptr</td>
<td>address for load/store access</td>
<td>void*</td>
<td>TYPE(C_PTR)</td>
<td>OUT</td>
</tr>
</tbody>
</table>

Remark 20 The ampersand operator in C is not a physical address, but a virtual address. The translation of where pages are placed in physical memory is determined by the page table.

12.2.2 Querying the shared structure

Even though the window created above is shared, that doesn’t mean it’s contiguous. Hence it is necessary to retrieve the pointer to the area of each process that you want to communicate with: MPI_Win_shared_query (figure 12.3).

```c
MPI_Aint window_size0; int window_unit; double *win0_addr;
MPI_Win_shared_query
( node_window, 0,
  &window_size0, &window_unit, &win0_addr );
```

For the full source of this example, see section 12.3.4

12.2.3 Heat equation example

As an example, which consider the 1D heat equation. On each process we create a local area of three point:

```c
// sharedshared.c
MPI_Win_allocate_shared(3, sizeof(int), info, sharedcomm, &shared_baseptr, &shared_window);
```

For the full source of this example, see section 12.3.5

12.2.4 Shared bulk data

In applications such as ray tracing, there is a read-only large data object (the objects in the scene to be rendered) that is needed by all processes. In traditional MPI, this would need to be stored redundantly on each process, which leads to large memory demands. With MPI shared memory we can store the data object once per node. Using as above MPI_Comm_split_type to find a communicator per NUMA domain, we store the object on process zero of this node communicator.
Exercise 12.2. Let the 'shared' data originate on process zero in MPI_COMM_WORLD. Then:

- create a communicator per shared memory domain;
- create a communicator for all the processes with number zero on their node;
- broadcast the shared data to the processes zero on each node.

(There is a skeleton for this exercise under the name shareddata.)
12. MPI topic: Shared memory

12.3 Sources used in this chapter

12.3.1 Listing of code header

12.3.2 Listing of code examples/mpi/c/sharedbulk.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {
    MPI_Comm comm;
    int nprocs,procid;

    MPI_Init(&argc,&argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procid);

    /*
     * Find the subcommunicator on the node,
     * and get the procid on the node.
     */
    MPI_Comm nodecomm; int onnode_procid;
    MPI_Comm_split_type
        (comm,MPI_COMM_TYPE_SHARED,procid,MPI_INFO_NULL,
         &nodecomm);
    MPI_Comm_rank(nodecomm,&onnode_procid);

    /*
     * Find the subcommunicators of
     * identical `onnode_procid' processes;
     * the procid on that communicator is the node ID
     */
    MPI_Comm crosscomm; int nodeid;
    MPI_Comm_split
        (comm,onnode_procid,procid,&crosscomm);
    MPI_Comm_rank(crosscomm,&nodeid);
    printf("[%d] %dx%d
",procid,nodeid,onnode_procid);

    /*
     * Create data on global process zero,
     * and broadcast it to the zero processes on other nodes
     */
    double shared_data = 0;
    if (procid==0) shared_data = 3.14;
    if (onnode_procid==0)
        MPI_Bcast(&shared_data,1,MPI_DOUBLE,0,crosscomm);
    printf("[%d] Head nodes should have shared data: %e
",procid,shared_data);
```
/* create window on the node communicator; * it only has nonzero size on the first process */ MPI_Win node_window; MPI_Aint window_size; double *window_data; if (onnode_procid==0) window_size = sizeof(double); else window_size = 0; MPI_Win_allocate_shared ( window_size,sizeof(double),MPI_INFO_NULL, nodecomm, &window_data,&node_window); /* * put data on process zero of the node window * we use a put call rather than a straight copy: * the fence calls enforce coherence */ MPI_Win_fence(0,node_window); if (onnode_procid==0) { MPI_Aint disp = 0; MPI_Put( &shared_data,1,MPI_DOUBLE,0,disp,1,MPI_DOUBLE,node_window); } MPI_Win_fence(0,node_window); /* * now get on each process the address of the window of process zero. */ MPI_Aint window_size0; int window_unit; double *win0_addr; MPI_Win_shared_query ( node_window,0, &window_size0,&window_unit, &win0_addr ); /* * check that we can indeed get at the data in the shared memory */ printf("[%d,%d] data at shared window: %e\n",nodeid,onnode_procid,*win0_addr); /* * cleanup */ MPI_Win_free(&node_window); MPI_Finalize(); return 0; }
#include <mpi.h>
#include "window.c"

int main(int argc, char **argv) {
#include "globalinit.c"

    /*
    * Find the subcommunicator on the node,
    * and get the procid on the node.
    */
    MPI_Comm nodecomm;
    int onnode_procno, onnode_nprocs;
    MPI_Comm_split_type
        (comm, MPI_COMM_TYPE_SHARED, procno, MPI_INFO_NULL,
         &nodecomm);
    MPI_Comm_size(nodecomm, &onnode_nprocs);
    /* if (onnode_nprocs<2) {
    *     printf("This example needs at least two ranks per node\n"); */
    /*     // MPI_Abort(comm,0); */
    /* } */
    MPI_Comm_rank(nodecomm, &onnode_procno);

    for (int strategy=0; strategy<2; strategy++) {
        /*
        * Create window on the node communicator;
        * one item on each process
        */
        MPI_Aint window_size; double *window_data; MPI_Win node_window;
        window_size = sizeof(double);
        MPI_Info window_info;
        MPI_Info_create(&window_info);
        if (strategy==0) {
            if (procno==0)
                printf("Strategy 0 : default behavior of shared window allocation\n");
            MPI_Info_set(window_info,"alloc_shared_noncontig","false");
        } else {
            if (procno==0)
                printf("Strategy 1 : allow non-contiguous shared window allocation\n");
            MPI_Info_set(window_info,"alloc_shared_noncontig","true");
        }
        MPI_Win_allocate_shared( window_size,sizeof(double),window_info,
                                      nodecomm,
                                      &window_data,&node_window);
        MPI_Info_free(&window_info);
        test_window(node_window,nodecomm);

        /*
        * Now process zero checks on window placement
        */
        if (onnode_procno==0) {
            MPI_Aint window_size0; int window0_unit; double *win0_addr;
            /*
            */
        }
    }
}
12.3. Sources used in this chapter

```c
MPI_Win_shared_query( node_window,0,
    &window_size0,&window0_unit, &win0_addr );
size_t dist1,distp;
for (int p=1; p<onnode_nprocs; p++) {
    MPI_Aint window_sizep; int windowp_unit; double *winp_addr;
    MPI_Win_shared_query( node_window,p,
        &window_sizep,&windowp_unit, &winp_addr );
    distp = (size_t)winp_addr-(size_t)win0_addr;
    if (procno==0)
        printf("Distance %d to zero: %ld\n",p,(long)distp);
    if (p==1)
        dist1 = distp;
    else {
        if (distp%dist1!=0)
            printf("!!!! not a multiple of distance 0--1 !!!!\n");
    }
}
MPI_Win_free(&node_window);

/*@ cleanup */
MPI_Finalize();
return 0;
```

12.3.4 Listing of code code/mpi/shared.c

12.3.5 Listing of code examples/mpi/c/sharedshared.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#ifndef CORES_PER_NODE
#define CORES_PER_NODE 16
#endif

int main(int argc,char **argv) {
    #include "globalinit.c"
    if (nprocs<3) {
        printf("This program needs at least three processes\n");
        return -1;
    }
    return 0;
}
```
if (procno==0)
    printf("There are %d ranks total\n",nprocs);

int new_procn, new_nprocs;
MPI_Comm sharedcomm;

MPI_Info info;
MPI_Comm_split_type(MPI_COMM_WORLD,MPI_COMM_TYPE_SHARED,procno,info,&sharedcomm);
MPI_Comm_size(sharedcomm,&new_nprocs);
MPI_Comm_rank(sharedcomm,&new_procn);

ASSERT(new_procn<CORES_PER_NODE);

if (new_nprocs!=nprocs) {
    printf("This example can only run on shared memory\n");
    MPI_Abort(comm,0);
}

MPI_Win shared_window; int *shared_baseptr;
MPI_Win_allocate_shared(3,sizeof(int),info,sharedcomm,&shared_baseptr,&shared_window);
{
    MPI_Aint check_size; int check_unit; int *check_baseptr;
    MPI_Win_shared_query
        (shared_window,new_procn,
         &check_size,&check_unit,&check_baseptr);
    printf("[%d;%d] size=%ld\n",procno,new_procn,check_size);
}

int *left_ptr,*right_ptr;
int left_proc = new_procn>0 ? new_procn-1 : MPI_PROC_NULL,
    right_proc = new_procn<new_nprocs-1 ? new_procn+1 : MPI_PROC_NULL;
MPI_Win_shared_query(shared_window,left_proc,NULL,NULL,&left_ptr);
MPI_Win_shared_query(shared_window,right_proc,NULL,NULL,&right_ptr);

if (procno==0)
    printf("Finished\n");

MPI_Finalize();
return 0;
Chapter 13

MPI topic: Hybrid computing

While the MPI standard itself makes no mention of threads – process being the primary unit of computation – the use of threads is allowed. Below we will discuss what provisions exist for doing so.

Using threads and other shared memory models in combination with MPI leads of course to the question how race conditions are handled. Example of a code with a data race that pertains to MPI:

```c
#pragma omp sections
#pragma omp section
  MPI_Send( x, /* to process 2 */ )
#pragma omp section
  MPI_Recv( x, /* from process 3 */ )
```

The MPI standard here puts the burden on the user: this code is not legal, and behavior is not defined.

13.1 MPI support for threading

In hybrid execution, the main question is whether all threads are allowed to make MPI calls. To determine this, replace the `MPI_Init` call by `MPI_Init_thread` (figure 13.1) Here the required and provided parameters can take the following (monotonically increasing) values:

- **MPI_THREAD_SINGLE**: Only a single thread will execute.
- **MPI_THREAD_FUNNELED**: The program may use multiple threads, but only the main thread will make MPI calls.
  
  The main thread is usually the one selected by the `master` directive, but technically it is the only that executes `MPI_Init_thread`. If you call this routine in a parallel region, the main thread may be different from the master.
- **MPI_THREAD_SERIALIZED**: The program may use multiple threads, all of which may make MPI calls, but there will never be simultaneous MPI calls in more than one thread.
- **MPI_THREAD_MULTIPLE**: Multiple threads may issue MPI calls, without restrictions.

After the initialization call, you can query the support level with `MPI_Query_thread` (figure 13.2).

In case more than one thread performs communication, `MPI_Is_thread_main` (figure 13.3) can determine whether a thread is the main thread.

**Python note 29: Thread level.** The thread level can be set through the `mpi4py.rc` object (section 2.2.2):
13. MPI topic: Hybrid computing

Figure 13.1 MPI_Init_thread

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init_thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 13.2 MPI_Query_thread

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Query_thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>provided</td>
<td>provided level of thread support</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

mpi4py.rc.threads # default True
mpi4py.rc.thread_level # default "multiple"

Available levels are multiple, serialized, funneled, single.

**MPL note 60: Threading support.** MPL always calls `MPI_Init_thread` requesting the highest level `MPI_THREAD_MULTIPLE`.

```python
enum mpl::threading_modes {
    mpl::threading_modes::single = MPI_THREAD_SINGLE,
    mpl::threading_modes::funneled = MPI_THREAD_FUNNELED,
    mpl::threading_modes::serialized = MPI_THREAD_SERIALIZED,
    mpl::threading_modes::multiple = MPI_THREAD_MULTIPLE
};
threading_modes mpl::environment::threading_mode ();
bool mpl::environment::is_thread_main ();
```

The *mvapich* implementation of MPI does have the required threading support, but you need to set this environment variable:

```
export MV2_ENABLE_AFFINITY=0
```

Another solution is to run your code like this:

```
ibrun tacc_affinity <my_multithreaded_mpi_executable
```

Intel MPI uses an environment variable to turn on thread support:

```
I_MPI_LIBRARY_KIND=<value>
```

where

```
release : multi-threaded with global lock
release_mt : multi-threaded with per-object lock for thread-split
```

The *mpiexec* program usually propagates *environment variables*, so the value of `OMP_NUM_THREADS` when you call `mpiexec` will be seen by each MPI process.

- It is possible to use blocking sends in threads, and let the threads block. This does away with the need for polling.
- You can not send to a thread number: use the MPI *message tag* to send to a specific thread.
13.1. MPI support for threading

Exercise 13.1. Consider the 2D heat equation and explore the mix of MPI/OpenMP parallelism:

- Give each node one MPI process that is fully multi-threaded.
- Give each core an MPI process and don’t use multi-threading.

Discuss theoretically why the former can give higher performance. Implement both schemes as special cases of the general hybrid case, and run tests to find the optimal mix.

```c
// thread.c
MPI_Init_thread(&argc,&argv,MPI_THREAD_MULTIPLE,&threading);
comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm,&procno);
MPI_Comm_size(comm,&nprocs);

if (procno==0) {
    switch (threading) {
    case MPI_THREAD_MULTIPLE : printf("Glorious multithreaded MPI\n"); break;
    case MPI_THREAD_SERIALIZED : printf("No simultaneous MPI from threads\n"); break;
    case MPI_THREAD_FUNNELED : printf("MPI from main thread\n"); break;
    case MPI_THREAD_SINGLE : printf("no threading supported\n"); break;
    }
}
MPI_Finalize();
```

*For the full source of this example, see section 13.2.2*
13. MPI topic: Hybrid computing

13.2 Sources used in this chapter

13.2.1 Listing of code header

13.2.2 Listing of code examples/mpi/c/thread.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

  MPI_Comm comm;
  int procnos=-1,nprocs,threading,err;

  MPI_Init_thread(&argc,&argv,MPI_THREAD_MULTIPLE,&threading);
  comm = MPI_COMM_WORLD;
  MPI_Comm_rank(comm,&procno);
  MPI_Comm_size(comm,&nprocs);

  if (procno==0) {
    switch (threading) {
      case MPI_THREAD_MULTIPLE : printf("Glorious multithreaded MPI\n"); break;
      case MPI_THREAD_SERIALIZED : printf("No simultaneous MPI from threads\n"); break;
      case MPI_THREAD_FUNNELED : printf("MPI from main thread\n"); break;
      case MPI_THREAD_SINGLE : printf("no threading supported\n"); break;
    }
  }
  MPI_Finalize();
  return 0;
}
```

---

Parallel Computing – r428
Chapter 14

MPI topic: Tools interface

Recent versions of MPI have a standardized way of reading out performance variables: the tools interface which improves on the old interface described in section 15.6.2.

14.1 Initializing the tools interface

The tools interface requires a different initialization routine `MPI_T_init_thread`

```c
int MPI_T_init_thread( int required, int *provided );
```

Likewise, there is `MPI_T_finalize`

```c
int MPI_T_finalize();
```

These matching calls can be made multiple times, after MPI has already been initialized with `MPI_Init` or `MPI_Init_thread`.

Verbosity level is an integer parameter.

`MPI_T_VERBOSITY_{USER,TUNER,MPIDEV}_{BASIC,DETAIL,ALL}`

14.2 Control variables

`Control variables` are implementation-dependent variables that can be used to inspect and/or control the internal workings of MPI. Accessing control variables requires initializing the tools interface; section 14.1.

We query how many `control variables` are available with `MPI_T_cvar_get_num` (figure 14.1). A description of the control variable can be obtained from `MPI_T_cvar_get_info` (figure 14.2).

- An invalid index leads to a function result of `MPI_T_ERR_INVALID_INDEX`.
- Any output parameter can be specified as `NULL` and MPI will not set this.
- The `bind` variable is an object type or `MPI_T_BIND_NO_OBJECT`.
- The `enumtype` variable is `MPI_T_ENUM_NULL` if the variable is not an enum type.
14. MPI topic: Tools interface

Figure 14.1 MPI_T_cvar_get_num

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_T_cvar_get_num</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 14.2 MPI_T_cvar_get_info

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_T_cvar_get_info</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
// cvar.c
MPI_T_cvar_get_num(&ncvar);
printf("%d\n",ncvar);
for (int ivar=0; ivar<ncvar; ivar++) {
  char name[100]; int namelen = 100;
  char desc[256]; int desclen = 256;
  int verbosity,bind,scope;
  MPI_Datatype datatype;
  MPI_T_enum enumtype;
  MPI_T_cvar_get_info
    (ivar,
     name,&namelen,
     &verbosity,&datatype,&enumtype,desc,&desclen,&bind,&scope
    );
  printf("\%3d: %s\n %s\n",ivar,name,desc);
```

Remark 21 There is no constant indicating a maximum buffer length for these variables. However, you can do the following:

1. Call the info routine with NULL values for the buffers, reading out the buffer lengths;
2. allocate the buffers with sufficient length, that is, including an extra position for the null terminator;
3. calling the info routine a second time, filling in the string buffers.

Conversely, given a variable name, its index can be retrieved with MPI_T_cvar_get_index:

```c
int MPI_T_cvar_get_index(const char *name, int *cvar_index)
```

If the name can not be matched, the index is MPI_T_ERR_INVALID_NAME.

Accessing a control variable is done through a control variable handle.

```c
int MPI_T_cvar_handle_alloc
  (int cvar_index, void *obj_handle,
   MPI_T_cvar_handle *handle, int *count)
```

The handle is freed with MPI_T_cvar_handle_free.
14.3 Performance variables

The realization of the tools interface is installation-dependent, you first need to query how much of the tools interface is provided.

```c
// mpitpvar.c
MPI_Init_thread(&argc,&argv,MPI_THREAD_SINGLE,&tlevel);
MPI_T_init_thread(MPI_THREAD_SINGLE,&tlevel);
int npvar;
MPI_T_pvar_get_num(&npvar);
```

For the full source of this example, see section 14.6.2

```
int name_len=256,desc_len=256,
    verbosity, var_class, binding, isreadonly, iscontiguous, isatomic;
char var_name[256], description[256];
MPI_Datatype datatype; MPI_T_enum enumtype;
for (int pvar=0; pvar<npvar; pvar++) {
    name_len = 256; desc_len=256;
    MPI_T_pvar_get_info(pvar, var_name,&name_len,
        &verbosity,&var_class, &datatype,&enumtype, 
        description,&desc_len,
        &binding,&isreadonly,&iscontiguous,&isatomic);
    if (procid==0)
        printf("pvar %d: %d/%s = %s\n",pvar, var_class, var_name, description);
}
```

For the full source of this example, see section 14.6.2

Performance variables come in classes:

- MPI_T_PVAR_CLASS_STATE
- MPI_T_PVAR_CLASS_LEVEL
- MPI_T_PVAR_CLASS_SIZE
- MPI_T_PVAR_CLASS_PERCENTAGE
- MPI_T_PVAR_CLASS_HIGHWATERMARK
- MPI_T_PVAR_CLASS_LOWWATERMARK
- MPI_T_PVAR_CLASS_COUNTER
- MPI_T_PVAR_CLASS_AGGREGATE
- MPI_T_PVAR_CLASS_TIMER
- MPI_T_PVAR_CLASS_GENERIC

Query the number of performance variables with `MPI_T_pvar_get_num`:

```c
int MPI_T_pvar_get_num(int *num_pvar);
```

Get information about each variable, by index, with `MPI_T_pvar_get_info`:

```c
int MPI_T_pvar_get_info
    (int pvar_index, char *name, int *name_len,
     int *verbosity, int *var_class, MPI_Datatype *datatype,
     MPI_T_enum *enumtype, char *desc, int *desc_len, int *bind, 
     int *readonly, int *continuous, int *atomic)
```
See general remarks about these in section 14.2.

- The *readonly* variable indicates that the variable can not be written.
- The *continuous* variable requires use of `MPI_T_pvar_start` and `MPI_T_pvar_stop`.

Given a name, the index can be retrieved with `MPI_T_pvar_get_index`:

```c
int MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)
```

Again, see section 14.2.

### 14.3.1 Performance experiment sessions

To prevent measurements from getting mixed up, they need to be done in *performance experiment sessions*, to be called 'sessions' in this chapter. However see section 8.3.

Create a session with `MPI_T_pvar_session_create`:

```c
int MPI_T_pvar_session_create(MPI_T_pvar_session *session)
```

and release it with `MPI_T_pvar_session_free`:

```c
int MPI_T_pvar_session_free(MPI_T_pvar_session *session)
```

which sets the session variable to `MPI_T_PVAR_SESSION_NULL`.

We access a variable through a handle, associated with a certain session. The handle is created with `MPI_T_pvar_handle_alloc`:

```c
int MPI_T_pvar_handle_alloc(MPI_T_pvar_session session, int pvar_index, void *obj_handle, MPI_T_pvar_handle *handle, int *count)
```

(If a routine takes both a session and handle argument, and the two are not associated, an error of `MPI_T_ERR_INVALID_HANDLE` is returned.)

Free the handle with `MPI_T_pvar_handle_free`:

```c
int MPI_T_pvar_handle_free(MPI_T_pvar_session session, MPI_T_pvar_handle *handle)
```

which sets the variable to `MPI_T_PVAR_HANDLE_NULL`.

Continuous variables (see `MPI_T_pvar_get_info` above, which outputs this) can be started and stopped with `MPI_T_pvar_start` and `MPI_T_pvar_stop`:

```c
int MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle);
int MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
```

Passing `MPI_T_PVAR_ALL_HANDLES` to the stop call attempts to stop all variables within the session. Failure to stop a variable returns `MPI_T_ERR_PVAR_NO_STARTSTOP`.

Variables can be read and written with `MPI_T_pvar_read` and `MPI_T_pvar_write`:
14.4 Categories of variables

Variables, both the control and performance kind, can be grouped into categories by the MPI implementation.

The number of categories is queried with `MPI_T_category_get_num`:

\[
\text{int } \text{MPI_T_category_get_num} \left( \text{int } \ast \text{num}_\text{cat} \right)
\]

and for each category the information is retrieved with `MPI_T_category_get_info`:

\[
\text{int } \text{MPI_T_category_get_info} \left( \text{int } \ast \text{cat}_\text{index},
\begin{align*}
\text{char } \ast \text{name}, \text{int } \ast \text{name}_\text{len}, \text{char } \ast \text{desc}, \text{int } \ast \text{desc}_\text{len},
\text{int } \ast \text{num}_\text{cvars}, \text{int } \ast \text{num}_\text{pvars}, \text{int } \ast \text{num}_\text{categories}
\end{align*}
\right)
\]

For a given category name the index can be found with `MPI_T_category_get_index`:

\[
\text{int } \text{MPI_T_category_get_index} \left( \text{const char } \ast \text{name}, \text{int } \ast \text{cat}_\text{index} \right)
\]

The contents of a category are retrieved with `MPI_T_category_get_cvars`, `MPI_T_category_get_pvars`, and `MPI_T_category_get_categories`:

\[
\begin{align*}
\text{int } & \text{MPI_T_category_get_cvars} \left( \text{int } \ast \text{cat}_\text{index}, \text{int } \ast \text{len}, \text{int } \ast \text{indices}[] \right) \\
\text{int } & \text{MPI_T_category_get_pvars} \left( \text{int } \ast \text{cat}_\text{index}, \text{int } \ast \text{len}, \text{int } \ast \text{indices}[] \right) \\
\text{int } & \text{MPI_T_category_get_categories} \left( \text{int } \ast \text{cat}_\text{index}, \text{int } \ast \text{len}, \text{int } \ast \text{indices}[] \right)
\end{align*}
\]

These indices can subsequently be used in the calls `MPI_T_cvar_get_info`, `MPI_T_pvar_get_info`, `MPI_T_category_get_info`.

If categories change dynamically, this can be detected with `MPI_T_category_changed`:

\[
\text{int } \text{MPI_T_category_changed} \left( \text{int } \ast \text{stamp} \right)
\]
14. **MPI topic: Tools interface**

### 14.5 Events

```c
// mpitevent.c
int nsource;
MPI_T_source_get_num(&nsource);

int name_len=256, desc_len=256;
char var_name[256], description[256];
MPI_T_source_order ordering;
MPI_Count ticks_per_second, max_ticks;
MPI_Info info;
MPI_Datatype datatype; MPI_T_enum enumtype;
for (int source=0; source<nsource; source++) {
    name_len = 256; desc_len=256;
    MPI_T_source_get_info(source, var_name, &name_len,
                           description, &desc_len,
                           &ordering, &ticks_per_second, &max_ticks, &info);
```
14.6 Sources used in this chapter

14.6.1 Listing of code header

14.6.2 Listing of code code/mpi/c/mpitpvar.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc,char **argv) {
    int procid,nprocs;

    int tlevel;
    MPI_Init_thread(&argc,&argv,MPI_THREAD_SINGLE,&tlevel);
    MPI_T_init_thread(MPI_THREAD_SINGLE,&tlevel);
    int npvar;
    MPI_T_pvar_get_num(&npvar);

    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&procid);

    if (procid==0)
        printf("#pvars: %d\n",npvar);
    int name_len=256,desc_len=256,
        verbosity,var_class,binding,isreadonly,iscontiguous,isatomic;
    char var_name[256],description[256];
    MPI_Datatype datatype; MPI_T_enum enumtype;
    for (int pvar=0; pvar<npvar; pvar++) {
        name_len = 256; desc_len=256;
        MPI_T_pvar_get_info(pvar,var_name,&name_len,
                          &verbosity,&var_class,  
                          &datatype,&enumtype,  
                          description,&desc_len,
                          &binding,&isreadonly,&iscontiguous,&isatomic);
        if (procid==0)
            printf("pvar %d: %d/%s = %s\n",pvar,var_class,var_name,description);
    }

    MPI_T_finalize();
    MPI_Finalize();

    return 0;
}
```
Chapter 15

MPI leftover topics

15.1 Contextual information, attributes, etc.

15.1.1 Info objects

Certain MPI routines can accept MPI_Info objects. (For files, see section 15.1.1.3, for windows, see section 9.5.4.) These contain key-value pairs that can offer system or implementation dependent information.

Create an info object with MPI_Info_create (figure 15.1) and delete it with MPI_Info_free (figure 15.2).

Keys are then set with MPI_Info_set (figure 15.3), and they can be queried with MPI_Info_get (figure 15.4). Note that the output of the 'get' routine is not allocated: it is a buffer that is passed. The maximum length of a key is given by the parameter MPI_MAX_INFO_KEY. You can delete a key from an info object with MPI_Info_delete (figure 15.5).

There is a straightforward duplication of info objects: MPI_Info_dup (figure 15.6).

You can also query the number of keys in an info object with MPI_Info_get_nkeys (figure 15.7), after which the keys can be queried in succession with MPI_Info_get_nthkey.

Info objects that are marked as 'In' or 'Inout' arguments are parsed before that routine returns. This means that in nonblocking routines they can be freed immediately, unlike, for instance, send buffers.

*The following material is for the recently released MPI-4 standard and may not be supported yet.*

The routines MPI_Info_get and MPI_Info_get_valuelen are not robust with respect to the C language null terminator. Therefore, they are deprecated, and should be replaced with MPI_Info_get_string, which always returns a null-terminated string.

```
int MPI_Info_get_string
(MPI_Info info, const char *key,
 int *buflen, char *value, int *flag)
```

End of MPI-4 material

15.1.1.1 Environment information

The object MPI_INFO_ENV is predefined, containing:

- command Name of program executed.
15.1. Contextual information, attributes, etc.

### Figure 15.1 MPI_Info_create

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Info_create (</td>
<td>info info object created</td>
<td>MPI_Info*</td>
<td>TYPE(MPI_Info)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 15.2 MPI_Info_free

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Info_free (</td>
<td>info info object</td>
<td>MPI_Info*</td>
<td>TYPE(MPI_Info)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 15.3 MPI_Info_set

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Info_set (</td>
<td>info info object</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>key key key</td>
<td>const char*</td>
<td>CHARACTER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value value</td>
<td>const char*</td>
<td>CHARACTER</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 15.4 MPI_Info_get

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Info_get (</td>
<td>info info object</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>key key key</td>
<td>const char*</td>
<td>CHARACTER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>values len length of value associated with key</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value value</td>
<td>char*</td>
<td>CHARACTER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flag true if key defined, false if not</td>
<td>int*</td>
<td>LOGICAL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 15.5 MPI_Info_delete

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Info_delete (</td>
<td>info info object</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>key key key</td>
<td>const char*</td>
<td>CHARACTER</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
15. MPI leftover topics

Figure 15.6 MPI_Info_dup

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Info_dup (</td>
<td>info</td>
<td>info object</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>newinfo</td>
<td>info object created</td>
<td>MPI_Info*</td>
<td>TYPE(MPI_Info)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 15.7 MPI_Info_get_nkeys

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Info_get_nkeys (</td>
<td>info</td>
<td>info object</td>
<td>MPI_Info</td>
<td>TYPE(MPI_Info)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nkeys</td>
<td>number of defined keys</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
</tbody>
</table>

- argv Space separated arguments to command.
- maxprocs Maximum number of MPI processes to start.
- soft Allowed values for number of processors.
- host Hostname.
- arch Architecture name.
- vdir Working directory of the MPI process.
- file Value is the name of a file in which additional information is specified.
- thread_level Requested level of thread support, if requested before the program started execution.

Note that these are the requested values; the running program can for instance have lower thread support.

15.1.1.2 Communicator and window information

MPI has a built-in possibility of attaching information to communicators and windows using the calls 

MPI_Comm_get_info, MPI_Comm_set_info, MPI_Win_get_info, MPI_Win_set_info.

Copying a communicator with MPI_Comm_dup does not cause the info to be copied; to propagate information to the copy there is MPI_Comm_dup_with_info (section 7.2).

15.1.1.3 File information

An MPI_Info object can be passed to the following file routines:

- MPI_File_open
- MPI_File_set_view
- MPI_File_set_info; collective. The converse routine is MPI_File_get_info.

The following keys are defined in the MPI-2 standard:

- access_style: A comma separated list of one or more of: read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential, random
15.1. Contextual information, attributes, etc.

**Figure 15.8 MPI_Comm_set_attr**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_set_attr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td></td>
<td>communicator to which attribute will be attached</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td>comm_keyval</td>
<td></td>
<td>key value</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>attribute_val</td>
<td></td>
<td>attribute value</td>
<td>void*</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
</tr>
</tbody>
</table>

- `collective_buffering`: true or false; enables or disables buffering on collective I/O operations
- `cb_block_size`: integer block size for collective buffering, in bytes
- `cb_buffer_size`: integer buffer size for collective buffering, in bytes
- `cb_nodes`: integer number of MPI processes used in collective buffering
- `chunked`: a comma separated list of integers describing the dimensions of a multidimensional array to be accessed using subarrays, starting with the most significant dimension (1st in C, last in Fortran)
- `chunked_item`: a comma separated list specifying the size of each array entry, in bytes
- `chunked_size`: a comma separated list specifying the size of the subarrays used in chunking
- `file_perm`: UNIX file permissions at time of creation, in octal
- `io_node_list`: a comma separated list of I/O nodes to use

The following material is for the recently released MPI-4 standard and may not be supported yet.

- `mpi_minimum_memory_alignment`: alignment of allocated memory.

**End of MPI-4 material**

- `nb_proc`: integer number of processes expected to access a file simultaneously
- `num_io_nodes`: integer number of I/O nodes to use
- `striping_factor`: integer number of I/O nodes/devices a file should be striped across
- `striping_unit`: integer stripe size, in bytes

Additionally, file system-specific keys can exist.

### 15.1.2 Attributes

Some runtime (or installation dependent) values are available as attributes through `MPI_Comm_set_attr` (figure 15.8) and `MPI_Comm_get_attr` (figure 15.9) for communicators, or `MPI_Win_get_attr`, `MPI_Type_get_attr`. (The MPI-2 routine `MPI_Attr_get` is deprecated). The flag parameter has two functions:

- it returns whether the attributed was found;
- if on entry it was set to false, the value parameter is ignored and the routines only tests whether the key is present.

The return value parameter is subtle: while it is declared `void*`, it is actually the address of a `void*` pointer.

```c
// tags.c
int tag_upperbound;
void *v; int flag=1;
ierr = MPI_Comm_set_attr(comm, MPI_TAG_UB, &v, &flag);
tag_upperbound = *(int*)v;
```
Figure 15.9 MPI_Comm_get_attr

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_get_attr (</td>
<td></td>
<td>communicator to which the attribute is attached</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td>comm_keyval</td>
<td>key value</td>
<td>int</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>attribute_val</td>
<td>flag</td>
<td>attribute value, unless flag = false</td>
<td>void*</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>false if no attribute is associated with the key</td>
<td>int*</td>
<td>LOGICAL</td>
<td></td>
</tr>
</tbody>
</table>

Python:

```python
MPI.Comm.Get_attr(self, int keyval)
```

For the full source of this example, see section 15.12.2

```python
## tags.py
tag_upperbound = comm.Get_attr(MPI.TAG_UB)
if procid==0:
    print("Determined tag upperbound: {}\).format(tag_upperbound))
```

For the full source of this example, see section 15.12.3

Attributes are:

- **MPI_TAG_UB**: Upper bound for *tag* value. (The lower bound is zero.) Note that **MPI_TAG_UB** is the key, not the actual upper bound! This value has to be at least 32767.
- **MPI_HOST**: Host process rank, if such exists, **MPI_PROC_NULL**, otherwise.
- **MPI_IO**: Rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.
- **MPI_WTIME_IS_GLOBAL**: Boolean variable that indicates whether clocks are synchronized.

Also:

- **MPI_UNIVERSE_SIZE**: the total number of processes that can be created. This can be more than the size of **MPI_COMM_WORLD** if the host list is larger than the number of initially started processes. See section 8.1.

Python: `mpi4py.MPI.UNIVERSE_SIZE`.
- **MPI_APPNUM**: if MPI is used in MPMD mode (section 15.9.4), or if **MPI_Comm_spawn_multiple** is used (section 8.1), this attribute reports the how-manieth program we are in.

15.1.3 Create new keyval attributes

Create a key value with **MPI_Comm_create_keyval** (figure 15.10), **MPI_Type_create_keyval**, **MPI_Win_create_keyval**. Use this key to set new attributes with **MPI_Comm_set_attr**, **MPI_Type_set_attr**, **MPI_Win_set_attr**. Free the attributed with **MPI_Comm_delete_attr**, **MPI_Type_delete_attr**, **MPI_Win_delete_attr**.

This uses a function type **MPI_Comm_attr_function**. This function is copied when a communicator is duplicated; section 7.2. Free with **MPI_Comm_free_keyval**.
### 15.1. Contextual information, attributes, etc.

**Figure 15.10 MPI_Comm_create_keyval**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_create_keyval</td>
<td>comm_copy_attr_fn</td>
<td>copy callback function for comm_keyval</td>
<td><strong>MPI_Comm_copy_attr_function</strong></td>
<td><strong>PROCEDURE(MPI_Comm_copy_attr_function)</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm_delete_attr_fn</td>
<td>delete callback function for comm_keyval</td>
<td><strong>MPI_Comm_delete_attr_function</strong></td>
<td><strong>PROCEDURE(MPI_Comm_delete_attr_function)</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>comm_keyval</td>
<td>key value for future access</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extra_state</td>
<td>extra state for callback function</td>
<td>void*</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 15.11 MPI_Get_version**

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Get_version</td>
<td>version</td>
<td>version number</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>subversion</td>
<td>subversion number</td>
<td>int*</td>
<td>INTEGER</td>
<td></td>
</tr>
</tbody>
</table>

### 15.1.4 Processor name

You can query the hostname of a processor with `MPI_Get_processor_name`. This name need not be unique between different processor ranks.

You have to pass in the character storage: the character array must be at least `MPI_MAX_PROCESSOR_NAME` characters long. The actual length of the name is returned in the `resultlen` parameter.

### 15.1.5 Version information

For runtime determination, the MPI version is available through two parameters `MPI_VERSION` and `MPI_SUBVERSION` or the function `MPI_Get_version` (figure 15.11).

### 15.1.6 Python utility functions

*Python note 30: Utility functions.*

```python
## util.py
print(f"Configuration:
{mpi4py.get_config()}")
print(f"Include dir:
{mpi4py.get_include()}")

Mac OS X with Python installed through macports:

Configuration:
{'mpicc': '/opt/local/bin/mpicc-mpich-mp'}
Include dir:
/opt/local/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages/mpi4py
```

Victor Eijkhout 411
15. MPI leftover topics

Intel compiler and locally installed Python:

Configuration:

```python
class Configuration:
    mpicc: '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpicc',
    mpicxx: '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpicxx',
    mpifort: '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpif90',
    mpif90: '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpif90',
    mpif77: '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpif77'
```

Include dir:

```
/opt/apps/intel19/impi19_0/python3/3.9.2/lib/python3.9/site-packages/mpi4py/include
```

15.2 Error handling

Errors in normal programs can be tricky to deal with; errors in parallel programs can be even harder. This is because in addition to everything that can go wrong with a single executable (floating point errors, memory violation) you now get errors that come from faulty interaction between multiple executables.

A few examples of what can go wrong:

- **MPI errors**: An MPI routine can exit prematurely for various reasons, such as receiving much more data than its buffer can accommodate. Such errors, as well as the more common type mentioned above, typically cause your whole execution to terminate. That is, if one incarnation of your executable exits, the MPI runtime will kill all others.
- **Deadlocks and other hanging executions**: There are various scenarios where your processes individually do not exit, but are all waiting for each other. This can happen if two processes are both waiting for a message from each other, and this can be helped by using nonblocking calls. In another scenario, through an error in program logic, one process will be waiting for more messages (including nonblocking ones) than are sent to it.

While it is desirable for an MPI implementation to return an error, this is not always possible. Therefore, some scenarios, whether supplying certain procedure arguments, or doing a certain sequence of procedure calls, are simply marked as 'erroneous', and the state of MPI after an erroneous call is undefined.

15.2.1 Error codes

There are a bunch of error codes. These are all positive int values, while MPI_SUCCESS is zero. The maximum value of any built-in error code is MPI_ERR_LASTCODE. User-defined error codes are all larger than this.

- **MPI_ERR_ARG**: An argument was invalid that is not covered by another error code.
- **MPI_ERR_BUFFER**: The buffer pointer is invalid; this typically means that you have supplied a null pointer.
- **MPI_ERR_COMM**: Invalid communicator. A common error is to use a null communicator in a call.
- **MPI_ERR_COUNT**: Invalid count argument, usually this is caused by a negative count value; zero is often a valid count.
- **MPI_ERR_INTERNAL**: An internal error in MPI has been detected.
15.2. Error handling

- **MPI_ERR_IN_STATUS** A functioning returning an array of statuses has at least one status where the MPI_ERROR field is set to other than MPI_SUCCESS. See section 4.3.2.3.
- **MPI_ERR_INFO**: invalid info object.
- **MPI_ERR_NO_MEM** is returned by MPI_Alloc_mem if memory is exhausted.
- **MPI_ERR_OTHER**: an error occurred; use MPI_Error_string to retrieve further information about this error; see section 15.2.2.3.
- **MPI_ERR_PORT**: invalid port; this applies to MPI_Comm_connect and such.

*The following material is for the recently released MPI-4 standard and may not be supported yet.*

- **MPI_ERR_PROC_ABORTED** is returned if a process tries to communicate with a process that has aborted.

*End of MPI-4 material*

- **MPI_ERR_RANK**: an invalid source or destination rank is specified. Valid ranks are $0 \ldots s - 1$ where $s$ is the size of the communicator, or MPI_PROC_NULL, or MPI_ANY_SOURCE for receive operations.
- **MPI_ERR_SERVICE**: invalid service in MPI_Unpublish_name; section 8.2.3.

15.2.2 Error handling

The MPI library has a general mechanism for dealing with errors that it detects: one can specify an error handler, specific to MPI objects.

- Most commonly, an error handler is associated with a communicator: MPI_Comm_set_errhandler (and likewise it can be retrieved with MPI_Comm_get_errhandler);
- other possibilities are MPI_File_set_errhandler, MPI_File_call_errhandler, MPI_Session_set_errhandler, MPI_Session_call_errhandler, MPI_Win_set_errhandler, MPI_Win_call_errhandler.

*The following material is for the recently released MPI-4 standard and may not be supported yet.*

*End of MPI-4 material*

**Remark 22** The routine **MPI_Errhandler_set** is deprecated, replaced by its MPI-2 variant **MPI_Comm_set_errhandler**.

Some handlers of type MPI_Errhandler are predefined (**MPI_ERRORS_ARE_FATAL, MPI_ERRORS_ABORT, MPI_ERRORS_RETURN**; see below), but you can define your own with **MPI_Errhandler_create**, to be freed later with **MPI_Errhandler_free**.

By default, MPI uses **MPI_ERRORS_ARE_FATAL**, except for file operations; see section 10.5.

**Python note 31: Error policy.** The policy for dealing with errors can be set through the mpi4py.rc object (section 2.2.2):

```python
# mpi4py.rc.errors default: "exception"
```

Available levels are exception, default, fatal.

15.2.2.1 Abort

The default behavior, where the full run is aborted, is equivalent to your code having the following call to

```python
MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_ARE_FATAL);
```
The handler `MPI_ERRORS_ARE_FATAL`, even though it is associated with a communicator, causes the whole application to abort.

The following material is for the recently released MPI-4 standard and may not be supported yet.

The handler `MPI_ERRORS_ABORT` (MPI-4) aborts on the processes in the communicator for which it is specified.

*End of MPI-4 material*

### 15.2.2.2 Return

Another simple possibility is to specify `MPI_ERRORS_RETURN`:

```c
MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
```

which causes the error code to be returned to the user. This gives you the opportunity to write code that handles the error return value.

### 15.2.2.3 Error printing

If the `MPI_Errhandler` value `MPI_ERRORS_RETURN` is used, you can compare the return code to `MPI_SUCCESS` and print out debugging information:

```c
int ierr;
ierr = MPI_Something();
if (ierr!=MPI_SUCCESS) {
    // print out information about what your programming is doing
    MPI_Abort();
}
```

For instance,

**Fatal error in MPI_Waitall:**

See the `MPI_ERROR` field in `MPI_Status` for the error code

You could then retrieve the `MPI_ERROR` field of the status, and print out an error string with `MPI_Error_string` or maximal size `MPI_MAX_ERROR_STRING`:

```c
MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
ierr = MPI_Waitall(2*ntids-2,requests,status);
if (ierr!=0) {
    char errtxt[MPI_MAX_ERROR_STRING];
    for (int i=0; i<2*ntids-2; i++) {
        int err = status[i].MPI_ERROR;
        int len=MPI_MAX_ERROR_STRING;
        MPI_Error_string(err, errtxt, &len);
        printf("Waitall error: %d %s\n", err, errtxt);
    }
    MPI_Abort(MPI_COMM_WORLD,0);
}
```

One cases where errors can be handled is that of `MPI file I/O`: if an output file has the wrong permissions, code can possibly progress without writing data, or writing to a temporary file.
15.2. Error handling

MPI operators (MPI_Op) do not return an error code. In case of an error they call MPI_Abort; if MPI_ERRORS_RETURN is the error handler, error codes may be silently ignored.

You can create your own error handler with MPI_Comm_create_errhandler (figure 15.12), which is then installed with MPI_Comm_set_errhandler. You can retrieve the error handler with MPI_Comm_get_errhandler.

*MPL note 61: Communicator errhandler.* MPL does not allow for access to the wrapped communicators. However, for MPI_COMM_WORLD, the routine MPI_Comm_set_errhandler can be called directly.

15.2.3 Defining your own MPI errors

You can define your own errors that behave like MPI errors. As an example, let’s write a send routine that refuses to send zero-sized data.

The first step to defining a new error is to define an error class with MPI_Add_error_class:

```c
int nonzero_class;
MPI_Add_error_class(&nonzero_class);
```

This error number is larger than MPI_ERR_LASTCODE, the upper bound on built-in error codes. The attribute MPI_LASTUSED_CODE records the last issued value.

Your new error code is then defined in this class with MPI_Add_error_code, and an error string can be added with MPI_Add_error_string:

```c
int nonzero_code;
MPI_Add_error_code(nonzero_class,&nonzero_code);
MPI_Add_error_string(nonzero_code,"Attempting to send zero buffer");
```

You can then call an error handler with this code. For instance to have a wrapped send routine that will not send zero-sized messages:

```c
// errorclass.c
int MyPI_Send( void *buffer, int n, MPI_Datatype type, int target, int tag, MPI_Comm comm) {
if (n==0)
   MPI_Comm_call_errhandler( comm, nonzero_code );
MPI_Ssend(buffer,n,type,target,tag,comm);
return MPI_SUCCESS;
}
```

Here we used the default error handler associated with the communicator, but one can set a different one with MPI_Comm_create_errhandler.
We test our example:

```c
for (int msgsize=1; msgsize>=0; msgsize--) {
    double buffer;
    if (procno==0) {
        printf("Trying to send buffer of length %d\n",msgsize);
        MPI_Send(&buffer,msgsize,MPI_DOUBLE, 1,0,comm);
        printf(".. success\n");
    } else if (procno==1) {
        MPI_Recv (&buffer,msgsize,MPI_DOUBLE, 0,0,comm,MPI_STATUS_IGNORE);
    }
}
```

which gives:

Trying to send buffer of length 1
.. success
Trying to send buffer of length 0
Abort(1073742081) on node 0 (rank 0 in comm 0):
Fatal error in MPI_Comm_call_errhandler: Attempting to send zero buffer

## 15.3 Fortran issues

MPI is typically written in C, what if you program *Fortran*?

See section 6.2.2.3 for MPI types corresponding to *Fortran90* types.

### 15.3.1 Assumed-shape arrays

Use of other than contiguous data, for instance `A(1:N:2)`, was a problem in MPI calls, especially non-blocking ones. In that case it was best to copy the data to a contiguous array. This has been fixed in MPI-3.

- Fortran routines have the same signature as C routines except for the addition of an integer error parameter.
- The call for *MPI_Init* in Fortran does not have the commandline arguments; they need to be handled separately.
- The routine *MPI_Sizeof* is only available in Fortran, it provides the functionality of the C/C++ operator `sizeof`.

### 15.3.2 Prevent compiler optimizations

The Fortran compiler can aggressively optimize by rearranging instructions. This may lead to incorrect behavior in MPI code. In the sequence:

```fortran
call MPI_Isend( buf, ..., request )
call MPI_Wait(request)
print *,buf(1)
```
the wait call does not involve the buffer, so the compiler can translate this into

```fortran
  call MPI_Isend(buf, ..., request)
  register = buf(1)
  call MPI_Wait(request)
  print *, register
```

Preventing this is possible with a Fortran2018 mechanism. First of all the buffer should be declared asynchronous

```fortran
  type Asynchronous :: buf
```

and introducing

```fortran
  IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING) &
      CALL MPI_F_SYNC_REG(buf)
```

The call to `MPI_F_sync_reg` will be removed at compile time if `MPI_ASYNC_PROTECTS_NONBLOCKING` is true.

### 15.4 Progress

The concept *asynchronous progress* describes that MPI messages continue on their way through the network, while the application is otherwise busy.

The problem here is that, unlike straight `MPI_Send` and `MPI_Recv` calls, communication of this sort can typically not be off-loaded to the network card, so different mechanisms are needed.

This can happen in a number of ways:

- **Compute nodes may have a dedicated communications processor.** The *Intel Paragon* was of this design; modern multicore processors are a more efficient realization of this idea.
- **The MPI library may reserve a core or thread for communications processing.** This is implementation dependent; see *Intel MPI* information below.
- **Reserving a core, or a thread in a continuous busy-wait spin loop,** takes away possible performance from the code. For this reason, Ruhela *et al.* [24] propose using a *pthreads* signal to wake up the progress thread.
- **Absent such dedicated resources, the application can force MPI to make progress by occasional calls to a polling routine such as **`MPI_Iprobe`**. **

**Remark 23** The `MPI_Probe` call is somewhat similar, in spirit if not quite in functionality, as `MPI_Test`. However, they behave differently with respect to progress. Quoting the standard:

> The **MPI implementation of MPI_Probe and MPI_Iprobe needs to guarantee progress:** if a call to **MPI_Probe** has been issued by a process, and a **send** that matches the probe has been initiated by some process, then the call to **MPI_Probe** will return.

In other words: probing causes MPI to make progress. On the other hand,

> A call to **MPI_Test** returns **flag = true** if the **operation identified by request is complete**.
In other words, if progress has been made, then testing will report completion, but by itself it does not cause completion.

A similar problem arises with passive target synchronization: it is possible that the origin process may hang until the target process makes an MPI call.

The following commands force progress: `MPI_Win_test`, `MPI_Request_get_status`.

**Intel note.** Only available with the `release_mt` and `debug_mt` versions of the Intel MPI library. Set `I_MPI_ASYNC_PROGRESS` to 1 to enable asynchronous progress threads, and `I_MPI_ASYNC_PROGRESS_THREADS` to set the number of progress threads.


Progress issues play with: `MPI_Test`, `MPI_Request_get_status`, `MPI_Win_test`.

### 15.5 Fault tolerance

Processors are not completely reliable, so it may happen that one ‘breaks’: for software or hardware reasons it becomes unresponsive. For an MPI program this means that it becomes impossible to send data to it, and any collective operation involving it will hang. Can we deal with this case? Yes, but it involves some programming.

First of all, one of the possible MPI error return codes (section 15.2) is `MPI_ERR_COMM`, which can be returned if a processor in the communicator is unavailable. You may want to catch this error, and add a ‘replacement processor’ to the program. For this, the `MPI_Comm_spawn` can be used (see 8.1 for details). But this requires a change of program design: the communicator containing the new process(es) is not part of the old `MPI_COMM_WORLD`, so it is better to set up your code as a collection of inter-communicators to begin with.

### 15.6 Performance, tools, and profiling

In most of this book we talk about functionality of the MPI library. There are cases where a problem can be solved in more than one way, and then we wonder which one is the most efficient. In this section we will explicitly address performance. We start with two sections on the mere act of measuring performance.

#### 15.6.1 Timing

MPI has a wall clock timer: `MPI_Wtime` (figure 15.13) which gives the number of seconds from a certain point in the past. (Note the absence of the error parameter in the fortran call.)
15.6. Performance, tools, and profiling

Figure 15.13 MPI_Wtime

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Wtime</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Python:

```c
MPI.Wtime()
```

Figure 15.14 MPI_Wtick

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>Explanation</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Wtick</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Python:

```c
MPI.Wtick()
```

```c
double t;
t = MPI_Wtime();
for (int n=0; n<NEXPERIMENTS; n++) {
    // do something;
}
t = MPI_Wtime()-t; t /= NEXPERIMENTS;
```

The timer has a resolution of MPI_Wtick (figure 15.14).

*MPL note 62: Timing.* The timing routines `wtime` and `wtick` and `wtime_is_global` are environment methods:

```c
double mpl::environment::wtime();
double mpl::environment::wtick();
bool mpl::environment::wtime_is_global();
```

Timing in parallel is a tricky issue. For instance, most clusters do not have a central clock, so you cannot relate start and stop times on one process to those on another. You can test for a global clock as follows:

```c
int *v, flag;
MPI_Attr_get( comm, MPI_WTIME_IS_GLOBAL, &v, &flag );
if (mytid==0) printf("Time synchronized? %d->%d\n", flag,*v);
```

Normally you don't worry about the starting point for this timer: you call it before and after an event and subtract the values.

```c
t = MPI_Wtime();
// something happens here
t = MPI_Wtime()-t;
```

If you execute this on a single processor you get fairly reliable timings, except that you would need to subtract the overhead for the timer. This is the usual way to measure timer overhead:
MPI leftover topics

\[ t = \text{MPI\_Wtime}(); \]
// absolutely nothing here
\[ t = \text{MPI\_Wtime}() - t; \]

15.6.1 Global timing

However, if you try to time a parallel application you will most likely get different times for each process, so you would have to take the average or maximum. Another solution is to synchronize the processors by using a barrier through \text{MPI\_Barrier}:

\[
\text{MPI\_Barrier}(\text{comm})
\]
\[ t = \text{MPI\_Wtime}(); \]
// something happens here
\[
\text{MPI\_Barrier}(\text{comm})
\]
\[ t = \text{MPI\_Wtime}() - t; \]

Exercise 15.1. This scheme also has some overhead associated with it. How would you measure that?

15.6.1.2 Local timing

Now suppose you want to measure the time for a single send. It is not possible to start a clock on the sender and do the second measurement on the receiver, because the two clocks need not be synchronized. Usually a ping-pong is done:

\[
\text{if ( proc\_source )} \{
\]
\[
\text{MPI\_Send( /* to target */);}
\]
\[
\text{MPI\_Recv( /* from target */);} \}
\]
\[
\text{else if ( proc\_target )} \{
\]
\[
\text{MPI\_Recv( /* from source */);} \}
\]
\[
\text{MPI\_Send( /* to source */);} \}
\]

No matter what sort of timing you are doing, it is good to know the accuracy of your timer. The routine \text{MPI\_Wtick} gives the smallest possible timer increment. If you find that your timing result is too close to this 'tick', you need to find a better timer (for CPU measurements there are cycle-accurate timers), or you need to increase your running time, for instance by increasing the amount of data.

15.6.2 Simple profiling

MPI allows you to write your own profiling interface. To make this possible, every routine \text{MPI\_Something} calls a routine \text{PMPI\_Something} that does the actual work. You can now write your \text{MPI\_...} routine which calls \text{PMPI\_...}, and inserting your own profiling calls. See figure 15.1.

By default, the MPI routines are defined as \text{weak linker symbols} as a synonym of the PMPI ones. In the gcc case:
\#pragma weak \text{MPI\_Send} = \text{PMPI\_Send}

As you can see in figure 15.2, normally only the PMPI routines show up in the stack trace.
15.6. Performance, tools, and profiling

15.6.3 Programming for performance

We outline some issues pertaining to performance.

Eager limit Short blocking messages are handled by a simpler mechanism than longer. The limit on what is considered ‘short’ is known as the eager limit (section 4.1.4.2), and you could tune your code by increasing its value. However, note that a process may likely have a buffer accommodating eager sends for every single other process. This may eat into your available memory.

Blocking versus nonblocking The issue of blocking versus nonblocking communication is something of a red herring. While nonblocking communication allows latency hiding, we can not consider it an alternative to blocking sends, since replacing nonblocking by blocking calls will usually give deadlock.

Still, even if you use nonblocking communication for the mere avoidance of deadlock or serialization (section 4.1.4.3), bear in mind the possibility of overlap of communication and computation. This also brings us to our next point.

Looking at it the other way around, in a code with blocking sends you may get better performance from nonblocking, even if that is not structurally necessary.

Progress MPI is not magically active in the background, especially if the user code is doing scalar work that does not involve MPI. As sketched in section 15.4, there are various ways of ensuring that latency hiding actually happens.
15. MPI leftover topics

Figure 15.2: A stack trace, showing the PMPI calls.

**Persistent sends** If a communication between the same pair of processes, involving the same buffer, happens regularly, it is possible to set up a persistent communication. See section 5.1.

**Buffering** MPI uses internal buffers, and the copying from user data to these buffers may affect performance. For instance, derived types (section 6.3) can typically not be streamed straight through the network (this requires special hardware support [17]) so they are first copied. Somewhat surprisingly, we find that buffered communication (section 5.5) does not help. Perhaps MPI implementors have not optimized this mode since it is so rarely used.

This is issue is extensively investigated in [9].

**Graph topology and neighborhood collectives** Load balancing and communication minimization are important in irregular applications. There are dedicated programs for this (ParMetis, Zoltan), and libraries such as PETSc may offer convenient access to such capabilities.

In the declaration of a graph topology (section 11.2) MPI is allowed to reorder processes, which could be used to support such activities. It can also serve for better message sequencing when neighborhood collectives are used.

**Network issues** In the discussion so far we have assumed that the network is a perfect conduit for data. However, there are issues of port design, in particular caused by oversubscription that adversely affect performance. While in an ideal world it may be possible to set up routine to avoid this, in the actual practice of a supercomputer cluster, network contention or message collision from different user jobs is hard to avoid.

**Offloading and onloading** There are different philosophies of network card design: Mellanox, being a network card manufacturer, believes in off-loading network activity to the Network Interface Card (NIC),
while Intel, being a processor manufacturer, believes in ‘on-loading’ activity to the process. There are argument either way.

Either way, investigate the capabilities of your network.

15.6.4 MPIR

MPIR is the informally specified debugging interface for processes acquisition and message queue extraction.

15.7 Determinism

MPI processes are only synchronized to a certain extent, so you may wonder what guarantees there are that running a code twice will give the same result. You need to consider two cases: first of all, if the two runs are on different numbers of processors there are already numerical problems; see HPSC-3.6.5.

Let us then limit ourselves to two runs on the same set of processors. In that case, MPI is deterministic as long as you do not use wildcards such as MPI_ANY_SOURCE. Formally, MPI messages are ‘nonovertaking’: two messages between the same sender-receiver pair will arrive in sequence. Actually, they may not arrive in sequence: they are matched in sequence in the user program. If the second message is much smaller than the first, it may actually arrive earlier in the lower transport layer.

15.8 Subtleties with processor synchronization

Blocking communication involves a complicated dialog between the two processors involved. Processor one says ‘I have this much data to send; do you have space for that?’, to which processor two replies ‘yes, I do; go ahead and send’, upon which processor one does the actual send. This back-and-forth (technically known as a handshake) takes a certain amount of communication overhead. For this reason, network hardware will sometimes forgo the handshake for small messages, and just send them regardless, knowing that the other process has a small buffer for such occasions.

One strange side-effect of this strategy is that a code that should deadlock according to the MPI specification does not do so. In effect, you may be shielded from your own programming mistake! Of course, if you then run a larger problem, and the small message becomes larger than the threshold, the deadlock will suddenly occur. So you find yourself in the situation that a bug only manifests itself on large problems, which are usually harder to debug. In this case, replacing every MPI_Send with a MPI_Ssend will force the handshake, even for small messages.

Conversely, you may sometimes wish to avoid the handshake on large messages. MPI as a solution for this: the MPI_Rsend (‘ready send’) routine sends its data immediately, but it needs the receiver to be ready for this. How can you guarantee that the receiving process is ready? You could, for instance, do the following (this uses nonblocking routines, which are explained below in section 4.2.1):
15. MPI leftover topics

```c
if ( receiving ) {
    MPI_Irecv() // post nonblocking receive
    MPI_Barrier() // synchronize
} else if ( sending ) {
    MPI_Barrier() // synchronize
    MPI_Rsend() // send data fast
```

When the barrier is reached, the receive has been posted, so it is safe to do a ready send. However, global barriers are not a good idea. Instead you would just synchronize the two processes involved.

**Exercise 15.2.** Give pseudo-code for a scheme where you synchronize the two processes through the exchange of a blocking zero-size message.

15.9 Shell interaction

MPI programs are not run directly from the shell, but are started through an *ssh tunnel*. We briefly discuss ramifications of this.

15.9.1 Standard input

Letting MPI processes interact with the environment is not entirely straightforward. For instance, *shell input redirection* as in

```
mpiexec -n 2 mpiprogram < someinput
```

may not work.

Instead, use a script `programs script` that has one parameter:

```bash
#!/bin/bash
mpirunprogram < $1
```

and run this in parallel:

```
mpiexec -n 2 programscript someinput
```

15.9.2 Standard out and error

The *stdout* and *stderr* streams of an MPI process are returned through the ssh tunnel. Thus they can be caught as the *stdout/err* of `mpiexec`.

```c
#include <stdio.h>

int main (int argc, char **argv) {
    // outerr.c
    fprintf(stdout,"This goes to std out\n");
    fprintf(stderr,"This goes to std err\n");
    return 0;
}
```

*For the full source of this example, see section 15.12.4*
15.9.3 Process status

The return code of `MPI_Abort` is returned as the processes status of `mpiexec`. Running

```c
// abort.c
if (procno==nprocs-1)
    MPI_Abort(comm,37);
```

For the full source of this example, see section 15.12.5

as

```bash
mpiexec -n 4 ./abort ; \
echo "Return code from $MPIRUN is <<$$?>>"
```

gives

TACC: Starting up job 3760534
TACC: Starting parallel tasks...
application called MPI_Abort(MPI_COMM_WORLD, 37) - process 3
TACC: MPI job exited with code: 37
TACC: Shutdown complete. Exiting.
Return code from ibrun is <<37>>

15.9.4 Multiple program start

If the MPI application consists of sub-applications, that is, if we have a true `MPMD` runs, there are usually two ways of starting this up. (Once started, each process can retrieve with `MPI_APPNUM` to which application it belongs.)

The first possibility is that the job starter, `mpiexec` or `mpirun` or a local variant, accepts multiple executables:

`mpiexec spec0 [ : spec1 [ : spec2 : ... ] ]`

Absent this mechanism, the sort of script of section 15.9.1 can also be used to implement `MPMD` runs. We let the script start one of a number of programs, and we use the fact that the MPI rank is known in the environment. The name of the variable is implementation dependent, for `mpich` and its derivates such as `Intel MPI` it is `PMI_RANK`. (There is a similar `PMI_SIZE`.)

Use a script `mpmdscript`:

```bash
#!/bin/bash
rank=$PMI_RANK
half=$(( ${PMI_SIZE} / 2 ))
if [ $rank -lt $half ] ; then
    ./prog1
else
    ./prog2
```

Victor Eijkhout
15. MPI leftover topics

TACC: Starting up job 4032931
TACC: Starting parallel tasks...
Program 1 has process 1 out of 4
Program 2 has process 2 out of 4
Program 2 has process 3 out of 4
Program 1 has process 0 out of 4
TACC: Shutdown complete. Exiting.

This script is run in parallel:
mpiexec -n 25 mpmdscript

15.10 Leftover topics
15.10.1 MPI constants

MPI has a number of built-in constants. These do not all behave the same.

- Some are compile-time constants. Examples are MPI_VERSION and MPI_MAX_PROCESSOR_NAME. Thus, they can be used in array size declarations, even before MPI_Init.
- Some link-time constants get their value by MPI initialization, such as MPI_COMM_WORLD. Such symbols, which include all predefined handles, can be used in initialization expressions.
- Some link-time symbols can not be used in initialization expressions, such as MPI_BOTTOM and MPI_STATUS_IGNORE.

For symbols, the binary realization is not defined. For instance, MPI_COMM_WORLD is of type MPI_Comm, but the implementation of that type is not specified.

See Annex A of the MPI-3.1 standard for full lists.

The following are the compile-time constants:

- MPI_MAX_PROCESSOR_NAME
- MPI_MAX_LIBRARY_VERSION_STRING
- MPI_MAX_ERROR_STRING
- MPI_MAX_DATAREP_STRING
- MPI_MAX_INFO_KEY
- MPI_MAX_INFO_VAL
- MPI_MAX_OBJECT_NAME
- MPI_MAX_PORT_NAME
- MPI_VERSION
- MPI_SUBVERSION
- MPI_STATUS_SIZE (Fortran only)
- MPI_ADDRESS_KIND (Fortran only)
- MPI_COUNT_KIND (Fortran only)
- MPI_INTEGER_KIND (Fortran only)
- MPI_OFFSET_KIND (Fortran only)
- MPI_SUBARRAYS_SUPPORTED (Fortran only)
- MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)

The following are the link-time constants:
• MPI_BOTTOM
• MPI_STATUS_IGNORE
• MPI_STATUSES_IGNORE
• MPI_ERRCODES_IGNORE
• MPI_IN_PLACE
• MPI_ARGV_NULL
• MPI_ARGVS_NULL
• MPI_UNWEIGHTED
• MPI_WEIGHTS_EMPTY

Assorted constants:
• MPI_PROC_NULL and other ..._NULL constants.
• MPI_ANY_SOURCE
• MPI_ANY_TAG
• MPI_UNDEFINED
• MPI_BSEND_OVERHEAD
• MPI_KEYVAL_INVALID
• MPI_LOCK_EXCLUSIVE
• MPI_LOCK_SHARED
• MPI_ROOT

(This section was inspired by http://blogs.cisco.com/performance/mpi-outside-of-c-and-fortran.)

15.10.2 Cancelling messages

In section 4.3.2.1 we showed a master-worker example where the master accepts in arbitrary order the messages from the workers. Here we will show a slightly more complicated example, where only the result of the first task to complete is needed. Thus, we issue an MPI_Recv with MPI_ANY_SOURCE as source. When a result comes, we broadcast its source to all processes. All the other workers then use this information to cancel their message with an MPI_Cancel operation.

```c
// cancel.c
fprintf(stderr,"get set, go!\n");
if (procno==nprocs-1) {
  MPI_Status status;
  MPI_Recv(dummy,0,MPI_INT, MPI_ANY_SOURCE,0,comm, &status);
  first_tid = status.MPI_SOURCE;
  MPI_Bcast(&first_tid,1,MPI_INT, nprocs-1,comm);
  fprintf(stderr,"[%d] first msg came from %d\n",procno,first_tid);
} else {
  float randomfraction = (rand() / (double)RAND_MAX);
  int randomwait = (int) ( nprocs * randomfraction );
  MPI_Request request;
  fprintf(stderr,"[%d] waits for %e/%d=%%%d\n",
          procno,randomfraction,nprocs,randomwait);
  sleep(randomwait);
  MPI_Isend(dummy,0,MPI_INT, nprocs-1,0,comm, &request);
```
MPI_Bcast(&first_tid,1,_MPI_INT, nprocs-1,comm);
if (procno!=first_tid) {
  MPI_Cancel(&request);
  fprintf(stderr, "[%d] canceled\n",procno);
}

For the full source of this example, see section 15.12.6

After the cancelling operation it is still necessary to call MPI_Request_free, MPI_Wait, or MPI_Test in order to free the request object.

The MPI_Cancel operation is local, so it can not be used for nonblocking collectives or one-sided transfers.

Remark 24 As of MPI-3.2, cancelling a send is deprecated.

15.10.3 The origin of one-sided communication in ShMem

The Cray T3E had a library called shmem which offered a type of shared memory. Rather than having a true global address space it worked by supporting variables that were guaranteed to be identical between processors, and indeed, were guaranteed to occupy the same location in memory. Variables could be declared to be shared a 'symmetric' pragma or directive; their values could be retrieved or set by shmem_get and shmem_put calls.

15.11 Literature

Online resources:

- Official MPI documents: http://www.mpi-forum.org/docs/

Tutorial books on MPI:

- Using MPI [11] by some of the original authors.
15.12 Sources used in this chapter

15.12.1 Listing of code header

15.12.2 Listing of code examples/mpi/c/tags.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc,char **argv) {
    #ifndef FREQUENCY
    #define FREQUENCY -1
    #endif

    /*
     * Standard initialization
     */
    MPI_Comm comm = MPI_COMM_WORLD;
    int nprocs, procid;
    MPI_Init(&argc,&argv);
    MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN);
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procid);
    int ierr;

    if (nprocs<2) {
        printf("This test needs at least 2 processes, not %d\n",nprocs);
        MPI_Abort(comm,0);
    }
    int sender = 0, receiver = nprocs-1;
    if (procid==0) {
        printf("Running on comm world of %d procs; communicating between %d--%d\n",
                nprocs,sender,receiver);
    }

    int tag_upperbound;
    void *v; int flag=1;
    ierr = MPI_Comm_get_attr(comm,MPI_TAG_UB,&v,&flag);
    tag_upperbound = *(int*)v;
    if (ierr!=MPI_SUCCESS) {
        printf("Error getting attribute: return code=%d\n",ierr);
        if (ierr==MPI_ERR_COMM)
            printf("invalid communicator\n");
        if (ierr==MPI_ERR_KEYVAL)
            printf("erroneous keyval\n");
        MPI_Abort(comm,0);
    }
    if (!flag) {
        printf("Could not get keyval\n");
        MPI_Abort(comm,0);
    }
```

Victor Eijkhout 429
15. MPI leftover topics

```python
if (procid==sender)
    printf("Determined tag upperbound: %d\n",tag_upperbound);

MPI_Finalize();
return 0;

15.12.3 Listing of code examples/mpi/p/tags.py

import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<4:
    print( "Need 4 procs at least")
    sys.exit(1)

tag_upperbound = comm.Get_attr(MPI.TAG_UB)
if procid==0:
    print("Determined tag upperbound: {}".format(tag_upperbound))

15.12.4 Listing of code examples/mpi/c/outerr.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
    fprintf(stdout,"This goes to std out\n");
    fprintf(stderr,"This goes to std err\n");
    return 0;
}

15.12.5 Listing of code examples/mpi/c/abort.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
    #include "globalinit.c"
    return 0;
}
```
if (procno==nprocs-1)
    MPI_Abort(comm,37);

MPI_Finalize();
return 0;
}

15.12.6 Listing of code examples/mpi/c/cancel.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

int main(int argc,char **argv) {
    int first_tid,dummy[11];
    
    #include "globalinit.c"
    // Initialize the random number generator
    srand((int)(procno*(double)RAND_MAX/nprocs));
    
    fprintf(stderr,"get set, go!\n");
    if (procno==nprocs-1) { 
        MPI_Status status;
        MPI_Recv(dummy,0,MPI_INT, MPI_ANY_SOURCE,0,comm, &status);
        first_tid = status.MPI_SOURCE;
        MPI_Bcast(&first_tid,1,MPI_INT, nprocs-1,comm);
        fprintf(stderr,\[%d\] first msg came from %d\n",procno,first_tid);
    } else {
        float randomfraction = (rand() / (double)RAND_MAX);
        int randomwait = (int) ( nprocs * randomfraction );
        MPI_Request request;
        fprintf(stderr,\[%d\] waits for %e/%d=%d\n", 
            procno,randomfraction,nprocs,randomwait);
        sleep(randomwait);
        MPI_Isend(dummy,0,MPI_INT, nprocs-1,0,comm, 
            &request);
        MPI_Bcast(&first_tid,1,MPI_INT, nprocs-1,comm 
            );
        if (procno!=first_tid) {
            MPI_Cancel(&request);
            fprintf(stderr,\[%d\] canceled\n",procno);
        }
    }

    MPI_Finalize();
    return 0;
}
Chapter 16

MPI Examples

16.1 Bandwidth and halfbandwidth

Bandwidth is the quantity that measures the number of bytes per second that can go through a connection. This definition seems straightforward, but comes with many footnotes.

- The size of the message used matters, since there is a latency cost to merely starting the message. Often, the bandwidth number quoted is an asymptotic figure, hard to achieve in practice.
- If a certain bandwidth figure is attained between a pair of processes, will two pairs, sending simultaneously, reach the same number?
- Does the bandwidth depend on the choice of processes to measure?
- And combinations of these considerations.

A useful measure comes from asking what bandwidth is achievable if all processes are either sending or receiving. As a further refinement, we ask what the least favorable choice is for the communicating pairs:

Halfbandwidth is defined as the minimum total bandwidth, over all possible choices of splitting the processes into a sending and receiving half.

Figure 16.1: Intra and inter schemes for bandwidth

Figure 16.1 illustrates the ‘intra’ (left) and ‘inter’ (right) scheme for letting all processes communicate in pairs. With intra-communication, the messages do not rely on the network so we expect to measure high bandwidth. With inter-communication, all messages go through the network and we expect to measure a lower number.

However, there are more issues to explore, which we will now do.
First of all we need to find pairs of processes. Consecutive pairs:

```cpp
int sender = procid - procid % 2, receiver = sender + 1;
```

Pairs that are $P/2$ apart:

```cpp
int sender = procid < halfprocs ? procid : procid - halfprocs, receiver = sender + halfprocs;
```

![Figure 16.2: Time as a function of core count. Left: on node. Right: between nodes.](image)

The halfbandwidth is measured as the total number of bytes sent divided by the total time. Both numbers are measured outside a repeat loop that does each transaction 100 times.

```cpp
auto duration = myclock::now() - start_time;
auto microsec_duration = std::chrono::duration_cast<std::chrono::microseconds>(duration);
int total_ping_count;
MPI_Allreduce(&pingcount, &total_ping_count, 1, MPI_INT, MPI_SUM, comm);
long bytes = buffersize * sizeof(double) * total_ping_count;
float fsec = microsec_duration.count() * 1.e-6;
halfbandwidth = bytes / fsec;
```

In the left graph of figure 16.2 we see that the time for $P/2$ simultaneous pingpongs stays fairly constant. This reflects the fact that, on node, the ping pong operations are data copies, which proceed simultaneously. Thus, the time is independent of the number of cores that are moving data. The exception is the final data point: with all cores active we take up more than the available bandwidth on the node.

In the right graph, each pingpong is inter-node, going through the network. Here we see the runtime go up linearly with the number of pingpongs, or somewhat worse than that. This reflects the fact that network transfers are done sequentially. (Actually, message can be broken up in packets, as long as they satisfy MPI message semantics. This does not alter our argument.)
Next we explore the influence of the buffer size on performance. The right graph in figure 16.3 show that inter-node bandwidth is almost independent of the buffer size. This means that even our smallest buffer is large enough to overcome any MPI startup cost.

On other hand, the left graph shows a more complicated pattern. Initially, the bandwidth increases, possibly reflecting the decreasing importance of MPI startup. For the final data points, however, performance drops again. This is due to the fact that the data size overflows cache size, and we are dominated by bandwidth from memory, rather than cache.
16.2 Sources used in this chapter

16.2.1 Listing of code header
16. MPI Examples
PART II

OPENMP
This section of the book teaches OpenMP ('Open Multi Processing'), the dominant model for shared memory programming in science and engineering. It will instill the following competencies.

Basic level:

- Threading model: the student will understand the threading model of OpenMP, and the relation between threads and cores (chapter 17); the concept of a parallel region and private versus shared data (chapter 18).
- Loop parallelism: the student will be able to parallelize loops, and understand the impediments to parallelization, and iteration scheduling (chapter 19; reductions (chapter 20).
- The student will understand the concept of worksharing constructs, and its implications for synchronization (chapter 21).

Intermediate level:

- The student will understand the abstract notion of synchronization, its implementations in OpenMP, and implications for performance (chapter 23).
- The student will understand the task model as underlying the thread model, be able to write code that spawns tasks, and be able to distinguish when tasks are needed versus simpler worksharing constructs (chapter 24).
- The student will understand thread/code affinity, how to control it, and possible implications for performance (chapter 25).

Advanced level:

- The student will understand the OpenMP memory model, and sequential consistency (chapter 26).
- The student will understand SIMD processing, the extent to which compilers do this outside of OpenMP, and how OpenMP can specify further opportunities for SIMD-ization (chapter 27).
- The student will understand offloading to Graphics Processing Units (GPUs), and the OpenMP directives for effecting this (chapter 28).
Chapter 17

Getting started with OpenMP

This chapter explains the basic concepts of OpenMP, and helps you get started on running your first OpenMP program.

17.1 The OpenMP model

We start by establishing a mental picture of the hardware and software that OpenMP targets.

17.1.1 Target hardware

Modern computers have a multi-layered design. Maybe you have access to a cluster, and maybe you have learned how to use MPI to communicate between cluster nodes. OpenMP, the topic of this chapter, is concerned with a single cluster node or motherboard, and getting the most out of the available parallelism available there.

Figure 17.1: A node with two sockets and a co-processor

Figure 17.1 pictures a typical design of a node: within one enclosure you find two sockets: single processor chips. Your personal laptop of computer will probably have one socket, most supercomputers have nodes
17. Getting started with OpenMP

with two or four sockets (the picture is of a Stampede node with two sockets)\(^1\), although the recent Intel Knight’s Landing is again a single-socket design.

![Figure 17.2: Structure of an Intel Sandybridge eight-core socket](image)

To see where OpenMP operates we need to dig into the sockets. Figure 17.2 shows a picture of an Intel Sandybridge socket. You recognize a structure with eight cores: independent processing units, that all have access to the same memory. (In figure 17.1 you saw four memory banks attached to each of the two sockets; all of the sixteen cores have access to all that memory.)

To summarize the structure of the architecture that OpenMP targets:

- A node has up to four sockets;
- each socket has up to 60 cores;
- each core is an independent processing unit, with access to all the memory on the node.

17.1.2 Target software

OpenMP is based on two concepts: the use of threads and the fork/join model of parallelism. For now you can think of a thread as a sort of process: the computer executes a sequence of instructions. The fork/join model says that a thread can split itself (‘fork’) into a number of threads that are identical copies. At some point these copies go away and the original thread is left (‘join’), but while the team of threads created by

\(^1\) In that picture you also see a co-processor: OpenMP is increasingly targeting those too.
the fork exists, you have parallelism available to you. The part of the execution between fork and join is known as a parallel region.

Figure 17.3 gives a simple picture of this: a thread forks into a team of threads, and these threads themselves can fork again.

![Figure 17.3: Thread creation and deletion during parallel execution](image)

The threads that are forked are all copies of the master thread: they have access to all that was computed so far; this is their shared data. Of course, if the threads were completely identical the parallelism would be pointless, so they also have private data, and they can identify themselves: they know their thread number. This allows you to do meaningful parallel computations with threads.

This brings us to the third important concept: that of work sharing constructs. In a team of threads, initially there will be replicated execution; a work sharing construct divides available parallelism over the threads. So there you have it: OpenMP uses teams of threads, and inside a parallel region the work is distributed over the threads with a work sharing construct. Threads can access shared data, and they have some private data.

An important difference between OpenMP and MPI is that parallelism in OpenMP is dynamically activated by a thread spawning a team of threads. Furthermore, the number of threads used can differ between parallel regions, and threads can create threads recursively. This is known as as dynamic mode. By contrast, in an MPI program the number of running processes is (mostly) constant throughout the run, and determined by factors external to the program.

### 17.1.3 About threads and cores

OpenMP programming is typically done to take advantage of multicore processors. Thus, to get a good speedup you would typically let your number of threads be equal to the number of cores. However, there is nothing to prevent you from creating more threads: the operating system will use time slicing to let them all be executed. You just don’t get a speedup beyond the number of actually available cores.

On some modern processors there are hardware threads, meaning that a core can actually let more than thread be executed, with some speedup over the single thread. To use such a processor efficiently you would let the number of OpenMP threads be 2× or 4× the number of cores, depending on the hardware.

### 17.1.4 About thread data

In most programming languages, visibility of data is governed by rules on the scope of variables: a variable is declared in a block, and it is then visible to any statement in that block and blocks with a lexical scope
17. Getting started with OpenMP

contained in it, but not in surrounding blocks:

```c
main () {
    // no variable `x' define here
    int x = 5;
    if (somecondition) { x = 6; }
    printf("x=\%e\n",x); // prints 5 or 6
}
printf("x=\%e\n",x); // syntax error: `x' undefined
}
```

In C, you can redeclare a variable inside a nested scope:

```c
{
    int x;
    if (something) {
        double x; // same name, different entity
    }
    x = ... // this refers to the integer again
}
```

Doing so makes the outer variable inaccessible.

Fortran has simpler rules, since it does not have blocks inside blocks.

In OpenMP the situation is a bit more tricky because of the threads. When a team of threads is created they can all see the data of the master thread. However, they can also create data of their own. We will go into the details later.

### 17.2 Compiling and running an OpenMP program

#### 17.2.1 Compiling

Your file or Fortran module needs to contain

```c
#include "omp.h"
```

in C, and

```fortran
use omp_lib
```

or

```c
#include "omp_lib.h"
```

for Fortran.

OpenMP is handled by extensions to your regular compiler, typically by adding an option to your commandline:
# gcc
gcc -o foo foo.c -fopenmp
# Intel compiler
icc -o foo foo.c -qopenmp

If you have separate compile and link stages, you need that option in both.

When you use the openmp compiler option, the OpenMP macro, (or cpp macro) _OPENMP will be defined. Thus, you can have conditional compilation by writing

```
#ifdef _OPENMP
...
#else
...
#endif
```

The value of this macro is a decimal value yyyyymm denoting the OpenMP standard release that this compiler supports.

Fortran note 12: OpenMP version. The parameter openmp_version contains the version in yyyyymm format.

### 17.2.2 Running an OpenMP program

You run an OpenMP program by invoking it the regular way (for instance ./a.out), but its behavior is influenced by some OpenMP environment variables. The most important one is OMP_NUM_THREADS:

```
export OMP_NUM_THREADS=8
```

which sets the number of threads that a program will use. See section 29.1 for a list of all environment variables.

### 17.3 Your first OpenMP program

In this section you will see just enough of OpenMP to write a first program and to explore its behavior. For this we need to introduce a couple of OpenMP language constructs. They will all be discussed in much greater detail in later chapters.

#### 17.3.1 Directives

OpenMP is not magic, so you have to tell it when something can be done in parallel. This is mostly done through directives; additional specifications can be done through library calls.

In C/C++ the pragma mechanism is used: annotations for the benefit of the compiler that are otherwise not part of the language. This looks like:

```
#pragma omp somedirective clause(value,othervalue)
 parallel statement;
#pragma omp somedirective clause(value,othervalue)
{
```
17. Getting started with OpenMP

```c
parallel statement 1;
parallel statement 2;
}
```

with

- the `#pragma omp sentinel` to indicate that an OpenMP directive is coming;
- a directive, such as `parallel`;
- and possibly clauses with values.
- After the directive comes either a single statement or a block in `curly braces`.

Directives in C/C++ are case-sensitive. Directives can be broken over multiple lines by escaping the line end.

The sentinel in Fortran looks like a comment:

```fortran
!omp directive clause(value)
statements
!omp end directive
```

The difference with the C directive is that Fortran cannot have a block, so there is an explicit `end-of directive` line.

If you break a directive over more than one line, all but the last line need to have a continuation character, and each line needs to have the sentinel:

```fortran
!omp parallel do &
!%omp copyin(x),copyout(y)
```

The directives are case-insensitive. In Fortran fixed-form source files, `c$omp` and `*$omp` are allowed too.

### 17.3.2 Parallel regions

The simplest way to create parallelism in OpenMP is to use the `parallel` pragma. A block preceded by the `omp parallel` pragma is called a `parallel region`; it is executed by a newly created team of threads. This is an instance of the `Single Program Multiple Data (SPMD)` model: all threads execute (redundantly) the same segment of code.

```c
#pragma omp parallel
{
    // this is executed by a team of threads
}
```

**Exercise 17.1.** Write a ‘hello world’ program, where the print statement is in a parallel region. Compile and run.

Run your program with different values of the environment variable `OMP_NUM_THREADS`. If you know how many cores your machine has, can you set the value higher?

We will go into much more detail in section 18.
17.3.3 An actual OpenMP program!

Let’s start exploring how OpenMP handles parallelism, using the following functions:

- `omp_get_num_threads` reports how many threads are currently active, and
- `omp_get_thread_num` reports the number of the thread that makes the call.
- `omp_get_num_procs` reports the number of available cores.

**Exercise 17.2.** Take the hello world program of exercise 17.1 and insert the above functions, before, in, and after the parallel region. What are your observations?

**Exercise 17.3.** Extend the program from exercise 17.2. Make a complete program based on these lines:

```c
int tsum = 0;
#pragma omp parallel
  tsum += /* the thread number */
printf("Sum is %d\n", tsum);
```

Compile and run again. (In fact, run your program a number of times.) Do you see something unexpected? Can you think of an explanation?

If the above puzzles you, read about race conditions in section 9.3.7.

17.3.4 Code and execution structure

Here are a couple of important concepts:

- An OpenMP directive is followed by an **structured block**; in C this is a single statement, a compound statement, or a block in braces; In Fortran it is delimited by the directive and its matching ‘end’ directive.
  
  A structured block can not be jumped into, so it can not start with a labeled statement, or contain a jump statement leaving the block.

- An OpenMP **construct** is the section of code starting with a directive and spanning the following structured block, plus in Fortran the end-directive. This is a lexical concept: it contains the statements directly enclosed, and not any subroutines called from them.

- A **region of code** is defined as all statements that are dynamically encountered while executing the code of an OpenMP construct. This is a dynamic concept: unlike a ‘construct’, it does include any subroutines that are called from the code in the structured block.

17.4 Thread data

In most programming languages, visibility of data is governed by rules on the **scope of variables**: a variable is declared in a block, and it is then visible to any statement in that block and blocks with a **lexical scope** contained in it, but not in surrounding blocks:

```c
main () {
  // no variable 'x' define here
  { 
    int x = 5;
```
17. Getting started with OpenMP

```c
if (somecondition) { x = 6; }
    printf("x=%e\n",x); // prints 5 or 6
}
printf("x=%e\n",x); // syntax error: `x' undefined
```

Fortran has simpler rules, since it does not have blocks inside blocks.

OpenMP has similar rules concerning data in parallel regions and other OpenMP constructs. First of all, data is visible in enclosed scopes:

```c
main() {
    int x;
    #pragma omp parallel
    {
        // you can use and set `x' here
        printf("x=%e\n",x); // value depends on what
            // happened in the parallel region
    }
}
```

In C, you can redefine a variable inside a nested scope:

```c
{
    int x;
    if (something) {
        double x; // same name, different entity
    }
    x = ... // this refers to the integer again
}
```

Doing so makes the outer variable inaccessible.

OpenMP has a similar mechanism:

```c
{
    int x;
    #pragma omp parallel
    {
        double x;
    }
}
```

There is an important difference: each thread in the team gets its own instance of the enclosed variable. This is illustrated in figure 17.4.

In addition to such scoped variables, which live on a stack, there are variables on the heap, typically created by a call to malloc (in C) or new (in C++). Rules for them are more complicated.

Summarizing the above, there are

- `shared variables`, where each thread refers to the same data item, and
- `private variables`, where each thread has its own instance.
In addition to using scoping, OpenMP also uses options on the directives to control whether data is private or shared.

Many of the difficulties of parallel programming with OpenMP stem from the use of shared variables. For instance, if two threads update a shared variable, you not guarantee an order on the updates.

We will discuss all this in detail in section 22.

17.5 Creating parallelism

The *fork/join model* of OpenMP means that you need some way of indicating where an activity can be forked for independent execution. There are two ways of doing this:

1. You can declare a parallel region and split one thread into a whole team of threads. We will discuss this next in section 18. The division of the work over the threads is controlled by *work sharing construct* (section 21).
2. Alternatively, you can use tasks and indicating one parallel activity at a time. You will see this in section 24.

Note that OpenMP only indicates how much parallelism is present; whether independent activities are in fact executed in parallel is a runtime decision. The factors influencing this are discussed in section 17.5.

Declaring a parallel region tells OpenMP that a team of threads can be created. The actual size of the team depends on various factors (see section 29.1 for variables and functions mentioned in this section).

- The *environment variable* OMP_NUM_THREADS limits the number of threads that can be created.
- If you don’t set this variable, you can also set this limit dynamically with the *library routine* omp_set_num_threads. This routine takes precedence over the aforementioned environment variable if both are specified.
- A limit on the number of threads can also be set as a clause on a parallel region.

If you specify a greater amount of parallelism than the hardware supports, the runtime system will probably ignore your specification and choose a lower value. To ask how much parallelism is actually used in your parallel region, use omp_get_num_threads. To query these hardware limits, use omp_get_num_procs. You can query the maximum number of threads with omp_get_max_threads. This equals the value of OMP_NUM_THREADS, not the number of actually active threads in a parallel region.
Another limit on the number of threads is imposed when you use nested parallel regions. This can arise if you have a parallel region in a subprogram which is sometimes called sequentially, sometimes in parallel. The variable `OMP_NESTED` controls whether the inner region will create a team of more than one thread.
17.6 Sources used in this chapter

17.6.1 Listing of code header
Chapter 18

OpenMP topic: Parallel regions

18.1 Creating parallelism with parallel regions

The simplest way to create parallelism in OpenMP is to use the `parallel` pragma. A block preceded by the `omp parallel` pragma is called a parallel region; it is executed by a newly created team of threads. This is an instance of the SPMD model: all threads execute the same segment of code.

```
#pragma omp parallel
{
    // this is executed by a team of threads
}
```

It would be pointless to have the block be executed identically by all threads. One way to get a meaningful parallel code is to use the function `omp_get_thread_num`, to find out which thread you are, and execute work that is individual to that thread. There is also a function `omp_get_num_threads` to find out the total number of threads. Both these functions give a number relative to the current team; recall from figure 17.3 that new teams can be created recursively.

For instance, if you program computes

```
result = f(x) + g(x) + h(x)
```

you could parallelize this as

```
double result, fresult, gresult, hresult;
#pragma omp parallel
{
    int num = omp_get_thread_num();
    if (num==0) fresult = f(x);
    else if (num==1) gresult = g(x);
    else if (num==2) hresult = h(x);
    result = fresult + gresult + hresult;
}
```

The first thing we want to do is create a team of threads. This is done with a parallel region. Here is a very simple example:

```
// hello.c
#pragma omp parallel
{
```

450
18.1. Creating parallelism with parallel regions

```c
int t = omp_get_thread_num();
printf("Hello world from %d!\n", t);
}
```

For the full source of this example, see section 18.5.2

or in Fortran

```fortran
!! hellocount.F90
!$omp parallel
  nthreads = omp_get_num_threads()
  mythread = omp_get_thread_num()
  write(*,'("Hello from",i3," out of",i3)') mythread,nthreads
!$omp end parallel
```

For the full source of this example, see section 18.5.3

or in C++

```c++
// hello.cxx
#pragma omp parallel
{
  int t = omp_get_thread_num();
  stringstream proctext;
  proctext << "Hello world from " << t << endl;
  cerr << proctext.str();
}
```

For the full source of this example, see section 18.5.4

(Note the use of stringstream: without that the output lines from the various threads may get mixed up.)

This code corresponds to the model we just discussed:

- Immediately preceding the parallel block, one thread will be executing the code. In the main program this is the initial thread.
- At the start of the block, a new team of threads is created, and the thread that was active before the block becomes the master thread of that team.
- After the block only the master thread is active.
- Inside the block there is team of threads: each thread in the team executes the body of the block, and it will have access to all variables of the surrounding environment. How many threads there are can be determined in a number of ways; we will get to that later.

Remark 25 In future versions of OpenMP, the master thread will be called the primary thread. In 5.1 the master construct will be deprecated, and masked (with added functionality) will take its place. In 6.0 master will disappear from the Spec, including proc_bind master "variable" and combined master constructs (master taskloop, etc.)

Exercise 18.1. Make a full program based on this fragment. Insert different print statements before, inside, and after the parallel region. Run this example. How many times is each print statement executed?

You see that the parallel directive
18. OpenMP topic: Parallel regions

- Is preceded by a special marker: a `#pragma omp` for C/C++, and the `!$OMP` sentinel for Fortran;
- Is followed by a single statement or a block in C/C++, or followed by a block in Fortran which is delimited by an `!$omp end directive`.

Directives look like 
 cpp directives, but they are actually handled by the compiler, not the preprocessor.

**Exercise 18.2.** Take the 'hello world' program above, and modify it so that you get multiple messages to your screen, saying

```
Hello from thread 0 out of 4!
Hello from thread 1 out of 4!
```

and so on. (The messages may very well appear out of sequence.)

What happens if you set your number of threads larger than the available cores on your computer?

**Exercise 18.3.** What happens if you call `omp_get_thread_num` and `omp_get_num_threads` outside a parallel region?

See also `OMP_WAIT_POLICY` values: ACTIVE, PASSIVE

### 18.2 Nested parallelism

What happens if you call a function from inside a parallel region, and that function itself contains a parallel region?

```c
int main() {
    ...  
    #pragma omp parallel
    {
        ...  
        func(...)
        ...  
    }  // end of main
    void func(...) {
        #pragma omp parallel
        {
            ...  
        }
    }
}
```

By default, the nested parallel region will have only one thread. To allow nested thread creation, use the environment variable `OMP_MAX_ACTIVE_LEVELS` (default: 1) to set the number of levels of parallel nesting. Equivalently, there are functions `omp_set_max_active_levels` and `omp_get_max_active_levels`:

```
OMP_MAX_ACTIVE_LEVELS=3
or
void omp_set_max_active_levels(int);
int omp_get_max_active_levels(void);
```
Remark 26 A deprecated mechanism is to set OMP_NESTED (default: false):

OMP_NESTED=true

or

omp_set_nested(1)

Nested parallelism can happen with nested loops, but it’s also possible to have a sections construct and a loop nested. Example:

Code:

// sectionnest.c
#pragma omp parallel sections reduction(+:s)
{
    #pragma omp section
    {
        double s1=0;
        omp_set_num_threads(team);
        #pragma omp parallel for reduction(+:s1)
        for (int i=0; i<N; i++) {

Output:

Nesting: false
Threads: 2, speedup: 2.0
Threads: 4, speedup: 2.0
Threads: 8, speedup: 2.0
Threads: 12, speedup: 2.0
Nesting: true
Threads: 2, speedup: 1.8
Threads: 4, speedup: 3.7
Threads: 8, speedup: 6.9
Threads: 12, speedup: 10.4

18.2.1 Subprograms with parallel regions

A common application of nested parallelism is the case where you have a subprogram with a parallel region, which itself gets called from a parallel region.

Exercise 18.4. Test nested parallelism by writing an OpenMP program as follows:

1. Write a subprogram that contains a parallel region.
2. Write a main program with a parallel region; call the subprogram both inside and outside the parallel region.
3. Insert print statements
   (a) in the main program outside the parallel region,
   (b) in the parallel region in the main program,
   (c) in the subprogram outside the parallel region,
   (d) in the parallel region inside the subprogram.

Run your program and count how many print statements of each type you get.

Writing subprograms that are called in a parallel region illustrates the following point: directives are evaluated with respect to the dynamic scope of the parallel region, not just the lexical scope. In the following example:

```c
#pragma omp parallel
{
    f();
}
void f() {
    #pragma omp for
    for ( .... ) {
        ...
    }
```

Victor Eijkhout 453
the body of the function $f$ falls in the dynamic scope of the parallel region, so the for loop will be parallelized.

If the function may be called both from inside and outside parallel regions, you can test which is the case with `omp_in_parallel`.

### 18.2.2 Fine control

The amount of nested parallelism can be set:

- `OMP_NUM_THREADS=4,2` means that initially a parallel region will have four threads, and each thread can create two more threads.
- `OMP_MAX_ACTIVE_LEVELS=123`

```c
omp_set_max_active_levels( n )
n = omp_get_max_active_levels()
```

```c
OMP_THREAD_LIMIT=123
n = omp_get_thread_limit()
```

```c
omp_set_max_active_levels
omp_get_max_active_levels
omp_get_level
omp_get_active_level
omp_get_ancestor_thread_num
omp_get_team_size(level)
```

### 18.3 Cancel parallel construct

It is possible to terminate a parallel construct early with the `cancel` directive:

```
|| !$omp cancel construct [if (expr)]
```

where construct is `parallel, sections, do` or `taskgroup`.

See section 31.3 for an example.

Cancelling is disabled by default for performance reasons. To activate it, set the `OMP_CANCELLATION` variable to true.

The state of cancellation can be queried with `omp_get_cancellation`, but there is no function to set it.

Cancellation can happen at most obvious places where OpenMP is active, but additional cancellation points can be set with

```
|| #pragma omp cancellation point <construct>
```

where the construct is `parallel, sections, for, do, taskgroup.`
18.4 Review questions

Exercise 18.5. T/F? The function `omp_get_num_threads()` returns a number that is equal to the number of cores.

Exercise 18.6. T/F? The function `omp_set_num_threads()` can not be set to a higher number than the number of cores.

Exercise 18.7. What function can be used to detect the number of cores?
18. OpenMP topic: Parallel regions

18.5 Sources used in this chapter

18.5.1 Listing of code header

18.5.2 Listing of code examples/omp/c/hello.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc,char **argv) {
    #pragma omp parallel
    {
        int t = omp_get_thread_num();
        printf("Hello world from %d\n",t);
    }
    return 0;
}
```

18.5.3 Listing of code examples/omp/f/hellocount.F90

```fortran
Program Hello

    use omp_lib
    integer :: nthreads,mythread

    !$omp parallel
    nthreads = omp_get_num_threads()
    mythread = omp_get_thread_num()
    write(*,'("Hello from",i3," out of",i3)') mythread,nthreads
    !$omp end parallel

end Program Hello
```

18.5.4 Listing of code examples/omp/cxx/hello.cxx

```c++
#include <iostream>
using std::cerr;
using std::cout;
using std::endl;
#include <sstream>
using std::stringstream;
#include <omp.h>

int main(int argc,char **argv) {
    #pragma omp parallel
```
\{ 
int t = omp_get_thread_num();
stringstream proctext;
proctext << "Hello world from " << t << endl;
cerr << proctext.str();
\}

return 0;
\}
Chapter 19

OpenMP topic: Loop parallelism

19.1 Loop parallelism

Loop parallelism is a very common type of parallelism in scientific codes, so OpenMP has an easy mechanism for it. OpenMP parallel loops are a first example of OpenMP ‘worksharing’ constructs (see section 21 for the full list): constructs that take an amount of work and distribute it over the available threads in a parallel region, created with the parallel pragma.

The parallel execution of a loop can be handled a number of different ways. For instance, you can create a parallel region around the loop, and adjust the loop bounds:

```c
#pragma omp parallel
{
    int threadnum = omp_get_thread_num(),
    numthreads = omp_get_num_threads();
    int low = N*threadnum/numthreads,
    high = N*(threadnum+1)/numthreads;
    for (i=low; i<high; i++)
        // do something with i
}
```

In effect, this is how you would parallelize a loop in MPI.

**Exercise 19.1.** What is an important difference between the resulting OpenMP and MPI code?

A more natural option is to use the for pragma:

```c
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++)
    // do something with i
```

This has several advantages. For one, you don’t have to calculate the loop bounds for the threads yourself, but you can also tell OpenMP to assign the loop iterations according to different schedules (section 19.3).

*Fortran note 13: OMP do pragma.* The for pragma only exists in C; there is a correspondingly named do pragma in Fortran.

458
19.1. Loop parallelism

```c
#pragma omp parallel
#pragma omp do
  do i=1,N
    ! something with i
  end do
#pragma omp end do
#pragma omp end parallel
```

Figure 19.1 shows the execution on four threads of

```c
#pragma omp parallel
{  
  code1();  
#pragma omp for  
    for (i=1; i<=4*N; i++) {  
      code2();  
    }  
  code3();  
}
```

The code before and after the loop is executed identically in each thread; the loop iterations are spread over the four threads.

![Figure 19.1: Execution of parallel code inside and outside a loop](image)

Note that the `do` and `for` pragmas do not create a team of threads: they take the team of threads that is active, and divide the loop iterations over them. This means that the `omp for` or `omp do` directive needs to be inside a parallel region.

As an illustration:
Exercise 19.2. What would happen in the above example if you increase the number of threads to be larger than the number of cores?

It is also possible to have a combined `omp parallel for` or `omp parallel do` directive.

19.1.1 Loops are static

There are some restrictions on the loop: basically, OpenMP needs to be able to determine in advance how many iterations there will be.

- The loop cannot contain `break`, `return`, `exit` statements, or `goto` to a label outside the loop.
- The `continue` in OpenMP loops (for C/C++) or `cycle` in OpenMP loops (for Fortran) statement is allowed.
- The index update has to be an increment (or decrement) by a fixed amount.
- The loop index variable is automatically private, and not changes to it inside the loop are allowed. The following loop is not parallelizable in OpenMP:

```
for (int i=0; i<N; ) {
    // something
    if (something)
        i++;
    else
        i += 2;
}
```

Remark 27: The loop index needs to be an integer value for the loop to be parallelizable. Unsigned values are allowed as of OpenMP-3.

19.1.2 Exercises

Exercise 19.3. Compute π by numerical integration. We use the fact that π is the area of the unit circle, and we approximate this by computing the area of a quarter circle using…
19.2 An example

Riemann sums.

• Let \( f(x) = \sqrt{1 - x^2} \) be the function that describes the quarter circle for \( x = 0 \ldots 1 \);
• Then we compute

\[
\pi/4 \approx \sum_{i=0}^{N-1} \Delta x f(x_i) \quad \text{where } x_i = i\Delta x \text{ and } \Delta x = 1/N
\]

Write a program for this, and parallelize it using OpenMP parallel for directives.

1. Put a parallel directive around your loop. Does it still compute the right result? Does the time go down with the number of threads? (The answers should be no and no.)
2. Change the parallel to parallel for (or parallel do). Now is the result correct? Does execution speed up? (The answers should now be no and yes.)
3. Put a critical directive in front of the update. (Yes and very much no.)
4. Remove the critical and add a clause reduction(+:quarterpi) to the for directive. Now it should be correct and efficient.

Use different numbers of cores and compute the speedup you attain over the sequential computation. Is there a performance difference between the OpenMP code with 1 thread and the sequential code?

Remark 28 In this exercise you may have seen the runtime go up a couple of times where you weren’t expecting it. The issue here is false sharing; see HPSC-3.6.5 for more explanation.

19.2 An example

To illustrate the speedup of perfectly parallel calculations, we consider a simple code that applies the same calculation to each element of an array.

All tests are done on the TACC Frontera cluster, which has dual-socket Intel Cascade Lake nodes, with a total of 56 cores. We control affinity by setting OMP_PROC_BIND=true.

Here is the essential code fragment:

```c
// speedup.c
#pragma omp parallel for
for (int ip=0; ip<N; ip++) {
    for (int jp=0; jp<M; jp++) {
        double f = sin(values[ip]);
        values[ip] = f;
    }
}
```

Exercise 19.4. Verify that the outer loop is parallel, but the inner one is not.

Exercise 19.5. Compare the time for the sequential code and the single-threaded OpenMP code. Try different optimization levels, and different compilers if you have them.
Now we investigate the influence of two parameters:

1. the OpenMP thread count: while we have 56 cores, values larger than that are allowed; and
2. the size of the problem: the smaller the problem, the larger the relative overhead of creating and synchronizing the team of threads.

We execute the above computation several times to even out effects of cache loading.

The results are in figure 19.2:

- While the problem size is always larger than the number of threads, only for the largest problem, which has at least 400 points per thread, is the speedup essentially linear.
- OpenMP allows for the number of threads to be larger than the core count, but there is no performance improvement in doing so.

The above tests did not use hyperthreads, since that is disabled on Frontera. However, the Intel Knights Landing nodes of the TACC Stampede2 cluster have four hyperthreads per core. Table 19.3 shows that this will indeed give a modest speedup.
19.2. An example

For reference, the commandlines executed were:

```bash
# frontera
make localclean run_speedup EXTRA_OPTIONS=-DN=200 NDIV=8 NP=112
make localclean run_speedup EXTRA_OPTIONS=-DN=2000 NDIV=8 NP=112
make localclean run_speedup EXTRA_OPTIONS=-DN=20000 NDIV=8 NP=112

# stampede2
make localclean run_speedup NDIV=8 EXTRA_OPTIONS="-DN=200000 -DM=1000" NP=272
```

**C++ note 2: Range syntax.** Parallel loops in C++ can use range-based syntax as of OpenMP-5.0:

```cpp
// speedup.cxx
#pragma omp parallel for
for ( auto& v : values ) {
    for ( int jp=0; jp<M; jp++ ) {
        double f = sin( v );
        v = f;
    }
}
```

Tests not reported here show exactly the same speedup as the C code.
19. OpenMP topic: Loop parallelism

19.3 Loop schedules

Usually you will have many more iterations in a loop than there are threads. Thus, there are several ways you can assign your loop iterations to the threads. OpenMP lets you specify this with the `schedule` clause.

```
#pragma omp for schedule(....)
```

The first distinction we now have to make is between static and dynamic schedules. With static schedules, the iterations are assigned purely based on the number of iterations and the number of threads (and the `chunk` parameter; see later). In dynamic schedules, on the other hand, iterations are assigned to threads that are unoccupied. Dynamic schedules are a good idea if iterations take an unpredictable amount of time, so that load balancing is needed.

![Figure 19.4: Illustration static round-robin scheduling versus dynamic](image)

Figure 19.4 illustrates this: assume that each core gets assigned two (blocks of) iterations and these blocks take gradually less and less time. You see from the left picture that thread 1 gets two fairly long blocks, where as thread 4 gets two short blocks, thus finishing much earlier. (This phenomenon of threads having unequal amounts of work is known as load imbalance.) On the other hand, in the right figure thread 4 gets block 5, since it finishes the first set of blocks early. The effect is a perfect load balancing.

![Figure 19.5: Illustration of the scheduling strategies of loop iterations](image)

The default static schedule is to assign one consecutive block of iterations to each thread. If you want different sized blocks you can define a chunk size:

```
#pragma omp for schedule(static[,chunk])
```
19.3. Loop schedules

(where the square brackets indicate an optional argument). With static scheduling, the compiler will determine the assignment of loop iterations to the threads at compile time, so, provided the iterations take roughly the same amount of time, this is the most efficient at runtime.

The choice of a chunk size is often a balance between the low overhead of having only a few chunks, versus the load balancing effect of having smaller chunks.

**Exercise 19.6.** Why is a chunk size of 1 typically a bad idea? (Hint: think about cache lines, and read HPSC-1.4.1.2.)

In dynamic scheduling OpenMP will put blocks of iterations (the default chunk size is 1) in a task queue, and the threads take one of these tasks whenever they are finished with the previous.

```c
#pragma omp for schedule(static[,chunk])
```

While this schedule may give good load balancing if the iterations take very differing amounts of time to execute, it does carry runtime overhead for managing the queue of iteration tasks.

Finally, there is the guided schedule, which gradually decreases the chunk size. The thinking here is that large chunks carry the least overhead, but smaller chunks are better for load balancing. The various schedules are illustrated in figure 19.5.

If you don’t want to decide on a schedule in your code, you can specify the runtime schedule. The actual schedule will then at runtime be read from the OMP_SCHEDULE environment variable. You can even just leave it to the runtime library by specifying auto.

**Exercise 19.7.** We continue with exercise 19.3. We add ‘adaptive integration’: where needed, the program refines the step size. This means that the iterations no longer take a predictable amount of time.

```c
for (i=0; i<nsteps; i++) {
    double x = i*h, x2 = (i+1)*h,
    y = sqrt(1-x*x), y2 =
    sqrt(1-x2*x2),
    slope = (y-y2)/h;
    if (slope>15) slope = 15;
    int samples = 1+(int)slope,
    is;
    for (is=0; is<samples;
```
19. OpenMP topic: Loop parallelism

3. Since this code is somewhat dynamic, try `schedule(dYNAMIC)`. This will actually give a fairly bad result. Why? Use `schedule(dYNAMIC, $n$)` instead, and experiment with values for $n$.
4. Finally, use `schedule(gUIDED)`, where OpenMP uses a heuristic. What results does that give?

Exercise 19.8. Program the LU factorization algorithm without pivoting.

```c
for k=1,n:
    A[k,k] = 1./A[k,k]
for i=k+1,n:
    A[i,k] = A[i,k]/A[k,k]
for j=k+1,n:
```

1. Argue that it is not possible to parallelize the outer loop.
2. Argue that it is possible to parallelize both the $i$ and $j$ loops.
3. Parallelize the algorithm by focusing on the $i$ loop. Why is the algorithm as given here best for a matrix on row-storage? What would you do if the matrix was on column storage?
4. Argue that with the default schedule, if a row is updated by one thread in one iteration, it may very well be updated by another thread in another. Can you find a way to schedule loop iterations so that this does not happen? What practical reason is there for doing so?

The schedule can be declared explicitly, set at runtime through the `OMP_SCHEDULE` environment variable, or left up to the runtime system by specifying auto. Especially in the last two cases you may want to enquire what schedule is currently being used with `omp_get_schedule`.

```c
int omp_get_schedule(omp_sched_t * kind, int * modifier);
```

Its mirror call is `omp_set_schedule`, which sets the value that is used when schedule value runtime is used. It is in effect equivalent to setting the environment variable `OMP_SCHEDULE`.

```c
void omp_set_schedule (omp_sched_t kind, int modifier);
```

<table>
<thead>
<tr>
<th>Type</th>
<th>environment variable</th>
<th>clause</th>
<th>omp_sched_t name</th>
<th>omp_sched_t value</th>
<th>modifier default</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>static[n]</td>
<td>schedule( ... )</td>
<td>omp_sched_static</td>
<td>1</td>
<td>N/ntreads</td>
</tr>
<tr>
<td>dynamic</td>
<td>dynamic[n]</td>
<td>dynamic[n]</td>
<td>omp_sched_dynamic</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>guided</td>
<td>guided[n]</td>
<td>guided[n]</td>
<td>omp_sched_guided</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>auto</td>
<td>auto</td>
<td>auto</td>
<td>omp_sched_auto</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Here are the various schedules you can set with the `schedule` clause:

- **affinity** Set by using value `omp_sched_affinity`
- **auto** The schedule is left up to the implementation. Set by using value `omp_sched_auto`
- **static** value: 1. The modifier parameter is the chunk size. Can also be set by using value `omp_sched_static`
- **dynamic** value: 2. The modifier parameter is the chunk size; default 1. Can also be set by using value `omp_sched_dynamic`
19.4 Reductions

So far we have focused on loops with independent iterations. Reductions are a common type of loop with dependencies. There is an extended discussion of reductions in section 20.

19.5 Collapsing nested loops

In general, the more work there is to divide over a number of threads, the more efficient the parallelization will be. In the context of parallel loops, it is possible to increase the amount of work by parallelizing all levels of loops instead of just the outer one.

Example: in

```c
for ( i=0; i<N; i++ )
    for ( j=0; j<N; j++ )
        A[i][j] = B[i][j] + C[i][j]
```

all $N^2$ iterations are independent, but a regular `omp for` directive will only parallelize one level. The `collapse` clause will parallelize more than one level:

```c
#pragma omp for collapse(2)
for ( i=0; i<N; i++ )
    for ( j=0; j<N; j++ )
        A[i][j] = B[i][j] + C[i][j]
```

It is only possible to collapse perfectly nested loops, that is, the loop body of the outer loop can consist only of the inner loop; there can be no statements before or after the inner loop in the loop body of the outer loop. That is, the two loops in

```c
for ( i=0; i<N; i++ ) {
    y[i] = 0.;
    for ( j=0; j<N; j++ )
        y[i] += A[i][j] * x[j]
}
```

cannot be collapsed.

**Exercise 19.9.** You could rewrite the above code as

```c
for ( i=0; i<N; i++ )
    y[i] = 0.;
for ( i=0; i<N; i++ ) {
    for ( j=0; j<N; j++ )
        y[i] += A[i][j] * x[j]
}
```

Is it now correct to have the `collapse` directive on the nested loop?
Exercise 19.10. Consider this code for matrix transposition:

```c
void transposer(int n, int m, double *dst, const double *src) {
    int blocksize;
    for (int i = 0; i < n; i += blocksize) {
        for (int j = 0; j < m; j += blocksize) {
            // transpose the block beginning at [i,j]
            for (int k = i; k < i + blocksize; ++k) {
                for (int l = j; l < j + blocksize; ++l) {
                    dst[k + l*m] = src[l + k*m];
                }
            }
        }
    }
}
```

Assuming that the src and dst array are disjoint, which loops are parallel, and how many levels can you collapse?

19.6 Ordered iterations

Iterations in a parallel loop that are executed in parallel do not execute in lockstep. That means that in

```c
#pragma omp parallel for
for ( ... i ... ) {
    ... f(i) ...
    printf("something with %d\n", i);
}
```

it is not true that all function evaluations happen more or less at the same time, followed by all print statements. The print statements can really happen in any order. The ordered clause coupled with the ordered directive can force execution in the right order:

```c
#pragma omp parallel for ordered
for ( ... i ... ) {
    ... f(i) ...
    #pragma omp ordered
    printf("something with %d\n", i);
}
```

Example code structure:

```c
#pragma omp parallel for shared(y) ordered
for ( ... i ... ) {
    int x = f(i);
    #pragma omp ordered
    y[i] += f(x)
    #pragma omp ordered
    z[i] = g(y[i])
}
```

There is a limitation: each iteration can encounter only one ordered directive.
19.7 nowait

The implicit barrier at the end of a work sharing construct can be cancelled with a `nowait` clause. This has the effect that threads that are finished can continue with the next code in the parallel region:

```c
#pragma omp parallel
{
#pragma omp for nowait
 for (i=0; i<N; i++) { ... }
// more parallel code
}
```

In the following example, threads that are finished with the first loop can start on the second. Note that this requires both loops to have the same schedule. We specify the static schedule here to have an identical scheduling of iterations over threads:

```c
#pragma omp parallel
{
  x = local_computation()
#pragma omp for schedule(static) nowait
  for (i=0; i<N; i++) {
    x[i] = ...
  }
#pragma omp for schedule(static)
  for (i=0; i<N; i++) {
    y[i] = ... x[i] ...
  }
}
```

19.8 While loops

OpenMP can only handle ‘for’ loops: while loops can not be parallelized. So you have to find a way around that. While loops are for instance used to search through data:

```c
while ( a[i]!=0 && i<imax ) {
  i++; }
// now i is the first index for which \n{a[i]} is zero.
```

We replace the while loop by a for loop that examines all locations:

```c
result = -1;
#pragma omp parallel for
for (i=0; i<imax; i++) {
  if (a[i]!=0 && result<0) result = i;
}
```

**Exercise 19.11.** Show that this code has a race condition.

You can fix the race condition by making the condition into a critical section; section 23.2.2. In this particular example, with a very small amount of work per iteration, that is likely to be inefficient in this case (why?). A more efficient solution uses the `lastprivate` pragma:
result = -1;
#pragma omp parallel for lastprivate(result)
for (i=0; i<imax; i++) {
    if (a[i]!=0) result = i;
}

You have now solved a slightly different problem: the result variable contains the last location where a[i] is zero.

19.9 Scaling tests

// speedup.c
#pragma omp parallel for
for (int ip=0; ip<N; ip++) {
    for (int jp=0; jp<M; jp++) {
        double f = sin(values[ip]);
        values[ip] = f;
    }
}

19.9.1 Lonestar 6

Lonestar 6, dual socket *AMD Milan*, total 112 cores: figure 19.6.

19.9.2 Frontera

*Intel Cascade Lake*, dual socket, 56 cores total; figure 19.7.

For all core counts to half the total, performance for all binding strategies seems equal. After that, close and spread perform equally, but the speedup for the false value gives erratic numbers.

19.9.3 Stampede2 skylake

Dual 24-core *Intel Skylake*; figure 19.8.

We see that close binding gives worse performance than spread. Setting binding to false only gives bad performance for large core counts.

19.9.4 Stampede2 Knights Landing


Since this is a single socket design, we don’t distinguish between the close and spread binding. However, the binding value of true shows good speedup – in fact beyond the core count – while false gives worse performance than in other architectures.

Single 68 core
19.10. Review questions

Exercise 19.12. The following loop can be parallelized with a parallel for. Is it correct to add the directive collapse(2)?

```c
for (i=0; i<N; i++) {
    y[i] = 0.;
    for (j=0; j<N; j++)
        y[i] += A[i][j] * x[j]
}
```

Exercise 19.13. Same question for the nested loop here:

---

19.9.5 Longhorn

Dual 20-core IBM Power9, 4 hyperthreads; 19.10

Unlike the Intel processors, here we use the hyperthreads. Figure 19.10 shows dip in the speedup at 40 threads. For higher thread counts the speedup increases to well beyond the physical core count of 40.

---

Figure 19.6: Speedup as function of thread count, Lonestar 6 cluster, different binding parameters
for (i=0; i<N; i++)
    y[i] = 0.;
for (i=0; i<N; i++) {
    for (j=0; j<N; j++)
        y[i] += A[i][j] * x[j]
19.11 Sources used in this chapter

19.11.1 Listing of code header

Figure 19.8: Speedup as function of thread count, Stampede2 skylake cluster, different binding parameters
Figure 19.9: Speedup as function of thread count, Stampede2 Knights Landing cluster, different binding parameters
Figure 19.10: Speedup as function of thread count, Longhorn cluster, different binding parameters
Chapter 20

OpenMP topic: Reductions

20.1 Reductions: why, what, how?

Parallel tasks often produce some quantity that needs to be summed or otherwise combined. If you write:

```c
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < N; i++)
    sum += f(i);
```

you will find that the `sum` value depends on the number of threads, and is likely not the same as when you execute the code sequentially. The problem here is the race condition involving the `sum` variable, since this variable is shared between all threads.

We will discuss several strategies of dealing with this.

20.1.1 Reduction clause

The easiest way to effect a reduction is of course to use the `reduction` clause. Adding this to an `omp parallel region` has the following effect:

- OpenMP will make a copy of the reduction variable per thread, initialized to the identity of the reduction operator, for instance 1 for multiplication.
- Each thread will then reduce into its local variable;
- At the end of the parallel region, the local results are combined, again using the reduction operator, into the global variable.

The simplest case is a reduction over a parallel loop. Here we compute $\pi/4$ as a Riemann sum:

```c
// pi.c
#pragma omp parallel for reduction(+:pi4)
for (int isample = 0; isample < N; isample++) {
    float xsample = isample * h;
    float y = sqrt(1 - xsample * xsample);
    pi4 += h * y;
}
```

*For the full source of this example, see section 20.6.2*

You can also reduce over sections:
20.1. Reductions: why, what, how?

Another reduction, this time over a parallel region, without any work sharing:

```c
// reductpar.c
int m = INT_MIN;
#pragma omp parallel reduction(max:m) num_threads(ndata)
{
  int t = omp_get_thread_num();
  int d = data[t];
  m = d>m ? d : m;
}
```

*For the full source of this example, see section 20.6.2*

If you want to reduce multiple variables with the same operator, use

```c
| reduction(+:x,y,z)
```

For multiple reduction with different operators, use more than one clause.

A reduction is one of those cases where the parallel execution can have a slightly different value from the one that is computed sequentially, because floating point operations are not associative. See HPSC-3.6.5 for more explanation.

### 20.1.2 Code your own solution

The most immediate way is to eliminate the race condition by declaring a *critical section*:

```c
double result = 0;
#pragma omp parallel
{
  double local_result;
  int num = omp_get_thread_num();
  if (num==0)  local_result = f(x);
  else if (num==1) local_result = g(x);
  else if (num==2) local_result = h(x);
  #pragma omp critical
  result += local_result;
}
```

This is a good solution if the amount of serialization in the critical section is small compared to computing the functions \( f, g, h \). On the other hand, you may not want to do that in a loop:
Exercise 20.1. Can you think of a small modification of this code, that still uses a critical section, that is more efficient? Time both codes.

20.1.2.1 False sharing

If your code can not be easily structured as a reduction, you can realize the above scheme by hand by ‘duplicating’ the global variable and gather the contributions later. This example presumes three threads, and gives each a location of their own to store the result computed on that thread:

```c
double result, local_results[3];
#pragma omp parallel
{
    int num = omp_get_thread_num();
    if (num==0) local_results[num] = f(x)
    else if (num==1) local_results[num] = g(x)
    else if (num==2) local_results[num] = h(x)

    result = local_results[0]+local_results[1]+local_results[2]
}
```

While this code is correct, it may be inefficient because of a phenomenon called false sharing. Even though the threads write to separate variables, those variables are likely to be on the same cacheline (see HPSC-1.4.1.2 for an explanation). This means that the cores will be wasting a lot of time and bandwidth updating each other’s copy of this cacheline.

False sharing can be prevented by giving each thread its own cacheline:

```c
double result, local_results[3][8];
#pragma omp parallel
{
    int num = omp_get_thread_num();
    if (num==0) local_results[num][1] = f(x)

    // etc cetera
}
```

A more elegant solution gives each thread a true local variable, and uses a critical section to sum these, at the very end:

```c
double result = 0;
#pragma omp parallel
{
```

```c
    double local_result;
    #pragma omp for
    for (i=0; i<N; i++) {
        local_result = f(x,i);
        #pragma omp critical
        result += local_result;
    } // end of for loop
```
double local_result;
local_result = ......
#pragma omp critical
    result += local_result;

20.2 Built-in reduction

20.2.1 Operators

Arithmetic reductions: +, *, -, max, min.

Logical operator reductions in C: & & | || ^

Logical operator reductions in Fortran: .and. .or. .eqv. .neqv. .iand. .ior. .iort.

Exercise 20.2. The maximum and minimum reductions were not added to OpenMP until OpenMP-3.1. Write a parallel loop that computes the maximum and minimum values in an array without using the reduction directive. Discuss the various options. Do timings to evaluate the speedup that is attained and to find the best option.

20.2.2 Types

Reduction can be applied to any type for which the operator is defined. The types to which max/min are applicable are limited.

Starting with the OpenMP-4.5 standard, you can reduce on statically allocated arrays:

```c
int ar[N];
#pragma omp parallel for reduction(+:ar[:N])
```

20.3 Initial value for reductions

The treatment of initial values in reductions is slightly involved.

```c
x = init_x
#pragma omp parallel for reduction(min:x)
for (int i=0; i<N; i++)
    x = min(x, data[i]);
```

Each thread does a partial reduction, but its initial value is not the user-supplied init_x value, but a value dependent on the operator. In the end, the partial results will then be combined with the user initial value. The initialization values are mostly self-evident, such as zero for addition and one for multiplication. For min and max they are respectively the maximal and minimal representable value of the result type.

Figure 20.1 illustrates this, where 1, 2, 3, 4 are four data items, i is the OpenMP initialization, and u is the user initialization; each p stands for a partial reduction value. The figure is based on execution using two threads.

Exercise 20.3. Write a program to test the fact that the partial results are initialized to the unit of the reduction operator.
20. OpenMP topic: Reductions

20.4 User-defined reductions

In a loop that performs a reduction, most of the element-by-element reduction as done in user code. However, in a parallel version of that loop, OpenMP needs to perform that same reduction on the partial results from the threads. Thus, if you want to perform your own reduction, you need to declare this reduction to OpenMP.

With user-defined reductions, the programmer specifies the function that does the elementwise comparison. We discuss two strategies:

1. In non-Object-Oriented (OO) languages you can define a function, and declare that to be a reduction operator with the declare reduction construct.
2. In OO languages (C++ and Fortran2003) you can overload ordinary operators for a class.

20.4.1 Reduction functions

This takes two steps.

1. You need a function of two arguments that returns the result of the comparison. You can do this yourself, but, especially with the C++ standard library, you can use functions such as \texttt{std::vector::insert}.
2. Specifying how this function operates on two variables \texttt{omp\_out} and \texttt{omp\_in}, corresponding to the partially reduced result and the new operand respectively. The new partial result should be left in \texttt{omp\_out}.
3. Optionally, you can specify the value to which the reduction should be initialized.

This is the syntax of the definition of the reduction, which can then be used in multiple reduction clauses.

```
#pragma omp declare reduction
   ( identifier : typelist : combiner )
   [initializer(initializer-expression)]
```

where:
20.4. User-defined reductions

identifier is a name; this can be overloaded for different types, and redefined in inner scopes.
typelist is a list of types.
combiner is an expression that updates the internal variable omp_out as function of itself and omp_in.
initializer sets omp_priv to the identity of the reduction; this can be an expression or a brace initializer.

20.4.1.1 Explicit expressions

For very simple cases:

```c
for (i=0; i<N; i++) {
    if (abs(data[i]) < result) {
        result = abs(data[i]);
    }
}
```

you can declare the reduction through an expression:

```c
#pragma omp declare reduction(minabs:int : 
    omp_out = abs(omp_in) > omp_out ? omp_out : abs(omp_in) ) 
    initializer (omp_priv=LARGENUM)
```

and use that in the reduction clause:

```c
#pragma omp parallel for reduction(minabs:result)
```

20.4.1.2 Reduction functions

For instance, recreating the maximum reduction would look like this:

```c
int mymax(int r, int n) {
    // r is the already reduced value
    // n is the new value
    int m;
    if (n>r) {
        m = n;
    } else {
        m = r;
    }
    return m;
}
```

```c
#pragma omp declare reduction \
    (rwz:int:omp_out=mymax(omp_out,omp_in)) \
    initializer(omp_priv=INT_MIN)
```

```c
m = INT_MIN;
```

```c
#pragma omp parallel for reduction(rwz:m)
```

```c
for (int idata=0; idata<ndata; idata++)
    m = mymax(m, data[idata]);
```

For the full source of this example, see section 20.6.3
Exercise 20.4. Write a reduction routine that operates on an array of nonnegative integers, finding the smallest nonzero one. If the array has size zero, or entirely consists of zeros, return \(-1\).

C++ note 3: Reduction over iterators. Support for C++ iterators

```cpp
#pragma omp declare reduction (merge : std::vector<int> : omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
```

C++ note 4: Templated reductions. You can reduce with a templated function if you put both the declaration and the reduction in the same templated function:

```cpp
template<typename T>
T generic_reduction(vector<T> tdata) {
    #pragma omp declare reduction
    (rwzt:T:omp_out=reduce_without_zero<T>(omp_out,omp_in))
    initializer(omp_priv=-1.f)

    T tmin = -1;
    #pragma omp parallel for reduction(rwzt:tmin)
    for (int id=0; id<tdata.size(); id++)
        tmin = reduce_without_zero<T>(tmin,tdata[id]);
    return tmin;
}
```

which is then called with specific data:

```cpp
auto tmin = generic_reduction<float>(fdata);
```

20.4.2 Overloaded operators

Fortran note 14: Reductions on derived types. Reduction can be applied to any derived type that has the reduction operator defined.

```fortran
!! reducttype.F90
Type inttype
    integer :: value = 0
end type inttype
Interface operator(+)
    module procedure addints
end Interface operator(+)
```

C++ note 5: Reduction on class objects. Reduction can be applied to any class for which the reduction operator is defined as `operator+` or whichever operator the case may be.

```cpp
// reductcomplex.cxx
class Thing {
    private:
        float x;
        public:
            Thing() : Thing( 0.f ) {};
```
20.5 Reductions and floating-point math

The mechanisms that OpenMP uses to make a reduction parallel go against the strict rules for floating point expression evaluation in C; see HPSC-3.7.5. OpenMP ignores this issue: it is the programmer’s job to ensure proper rounding behavior.
20. OpenMP topic: Reductions

20.6 Sources used in this chapter

20.6.1 Listing of code header

20.6.2 Listing of code examples/omp/c/reductpar.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <omp.h>

int main(int argc,char **argv) {
    int m,ndata = 4, data[4] = {2,-3,0,5};
    m = INT_MIN;
    for (int idata=0; idata<ndata; idata++) {
        int d = data[idata];
        m = d>m ? d : m;
    }
    if (m!=5)
        printf("Sequential: wrong reduced value: %d, s/b %d\n",m,2);
    else
        printf("Sequential case succeeded\n");

    m = INT_MIN;
    #pragma omp parallel reduction(max:m) num_threads(ndata)
    {
        int t = omp_get_thread_num();
        int d = data[t];
        m = d>m ? d : m;
    };

    if (m!=5)
        printf("Parallel: wrong reduced value: %d, s/b %d\n",m,2);
    else
        printf("Finished with correct parallel result\n");

    return 0;
}
```

20.6.3 Listing of code examples/omp/c/ireduct.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <omp.h>

int mymax(int r,int n) {
```
// r is the already reduced value
// n is the new value
int m;
if (n>r) {
    m = n;
} else {
    m = r;
}
printf("combine %d %d : %d\n",r,n,m);
return m;

int main(int argc,char **argv) {
    int m,ndata = 4, data[4] = {2,-3,0,5};

    #pragma omp declare reduction
    (rwz:int:omp_out=mymax(omp_out,omp_in)) \
    initializer(omp_priv=INT_MIN)
    m = INT_MIN;
    for (int idata=0; idata<ndata; idata++)
        m = mymax(m,data[idata]);
    if (m!=5)
        printf("Sequential: wrong reduced value: %d, s/b %d\n",m,2);
    else
        printf("Sequential case succeeded\n");

    m = INT_MIN;
    #pragma omp parallel for reduction(rwz:m)
    for (int idata=0; idata<ndata; idata++)
        m = mymax(m,data[idata]);
    if (m!=5)
        printf("Parallel: wrong reduced value: %d, s/b %d\n",m,2);
    else
        printf("Finished\n");

    return 0;
}
Chapter 21

OpenMP topic: Work sharing

The declaration of a parallel region establishes a team of threads. This offers the possibility of parallelism, but to actually get meaningful parallel activity you need something more. OpenMP uses the concept of a work sharing construct: a way of dividing parallelizable work over a team of threads. The work sharing constructs are:

- **for** (for C) or **do** (for Fortran). The threads divide up the loop iterations among themselves; see section 19.1.
- **sections** The threads divide a fixed number of sections between themselves; see section 21.1.
- **single** The section is executed by a single thread; section 21.2.
- **task** See section 24.
- **workshare** Can parallelize Fortran array syntax; section 21.3.

### 21.1 Sections

A parallel loop is an example of independent work units that are numbered. If you have a pre-determined number of independent work units, the **sections** is more appropriate. In a **sections** construct can be any number of **section** constructs. These need to be independent, and they can be execute by any available thread in the current team, including having multiple sections done by the same thread.

```c
#pragma omp sections
{
#pragma omp section
  // one calculation
#pragma omp section
  // another calculation
}
```

This construct can be used to divide large blocks of independent work. Suppose that in the following line, both \( f(x) \) and \( g(x) \) are big calculations:

\[
\| y = f(x) + g(x)
\]

You could then write
21.2 Single/master

The `single` and `master` pragma limit the execution of a block to a single thread. This can for instance be used to print tracing information or doing I/O operations.

```c
#pragma omp parallel
{
    #pragma omp single
    printf("We are starting this section!\n");
    // parallel stuff
}
```

Another use of `single` is to perform initializations in a parallel region:

```c
int a;
#pragma omp parallel
{
    #pragma omp single
    a = f(); // some computation
    #pragma omp sections
    // various different computations using a
}
The point of the single directive in this last example is that the computation needs to be done only once, because of the shared memory. Since it’s a work sharing construct there is an implicit barrier after it, which guarantees that all threads have the correct value in their local memory (see section 26.3.

**Exercise 21.1.** What is the difference between this approach and how the same computation would be parallelized in MPI? The master directive, also enforces execution on a single thread, specifically the master thread of the team, but it does not have the synchronization through the implicit barrier.

**Exercise 21.2.** Modify the above code to read:

```c
int a;
#pragma omp parallel
{
    #pragma omp master
    a = f(); // some computation
    #pragma omp sections
    // various different computations using a
}
```

This code is no longer correct. Explain.

Above we motivated the single directive as a way of initializing shared variables. It is also possible to use single to initialize private variables. In that case you add the copyprivate clause. This is a good solution if setting the variable takes I/O.

**Exercise 21.3.** Give two other ways to initialize a private variable, with all threads receiving the same value. Can you give scenarios where each of the three strategies would be preferable?

### 21.3 Fortran array syntax parallelization

The parallel do directive is used to parallelize loops, and this applies to both C and Fortran. However, Fortran also has implied loops in its *array syntax*. To parallelize array syntax you can use the workshare directive.

The workshare directive exists only in Fortran. It can be used to parallelize the implied loops in *array syntax*, as well as forall loops.
21.4 Sources used in this chapter

21.4.1 Listing of code header
Chapter 22

OpenMP topic: Controlling thread data

In a parallel region there are two types of data: private and shared. In this sections we will see the various way you can control what category your data falls under; for private data items we also discuss how their values relate to shared data.

22.1 Shared data

In a parallel region, any data declared outside it will be shared: any thread using a variable \( x \) will access the same memory location associated with that variable.

Example:

```c
int x = 5;
#pragma omp parallel
{
    x = x+1;
    printf("shared: x is %d\n",x);
}
```

All threads increment the same variable, so after the loop it will have a value of five plus the number of threads; or maybe less because of the data races involved. See HPSC-2.6.1.5 for an explanation of the issues involved; see 23.2.2 for a solution in OpenMP.

Sometimes this global update is what you want; in other cases the variable is intended only for intermediate results in a computation. In that case there are various ways of creating data that is local to a thread, and therefore invisible to other threads.

22.2 Private data

In the C/C++ language it is possible to declare variables inside a *lexical scope*; roughly: inside curly braces. This concept extends to OpenMP parallel regions and directives: any variable declared in a block following an OpenMP directive will be local to the executing thread.

Example:
22.3 Data in dynamic scope

Functions that are called from a parallel region fall in the dynamic scope of that parallel region. The rules for variables in that function are as follows:

\[
\text{int } x = 5; \\
\text{#pragma omp parallel} \\
\{ \\
\quad \text{int } x; \ x = 3; \ \\
\quad \text{printf(“local: } x \text{ is } \%d\n”, x);} \\
\}
\]

After the parallel region the outer variable \(x\) will still have the value 5: there is no storage association between the private variable and global one.

Fortran note 15: Private variables in parallel region. The Fortran language does not have this concept of scope, so you have to use a private clause:

\[
\text{!$OMP parallel private(x)}
\]

The private directive declares data to have a separate copy in the memory of each thread. Such private variables are initialized as they would be in a main program. Any computed value goes away at the end of the parallel region. (However, see below.) Thus, you should not rely on any initial value, or on the value of the outer variable after the region.

\[
\text{int } x = 5; \\
\text{#pragma omp parallel private(x)} \\
\{ \\
\quad x = x+1; \ // \text{dangerous} \ \\
\quad \text{printf(“private: } x \text{ is } \%d\n”, x);} \\
\} \\
\text{printf(“after: } x \text{ is } \%d\n”, x); \ // \text{also dangerous}
\]

Data that is declared private with the private directive is put on a separate stack per thread. The OpenMP standard does not dictate the size of these stacks, but beware of stack overflow. A typical default is a few megabyte; you can control it with the environment variable OMP_STACKSIZE. Its values can be literal or with suffixes:

123 456k 567K 678m 789M 246g 357G

A normal Unix process also has a stack, but this is independent of the OpenMP stacks for private data. You can query or set the Unix stack with ulimit:

\[
[] \text{ulimit } -s 64000 \\
[] \text{ulimit } -s 8192 \\
[] \text{ulimit } -s 8192
\]

The Unix stack can grow dynamically as space is needed. This does not hold for the OpenMP stacks: they are immediately allocated at their requested size. Thus it is important not too make them too large.
22. OpenMP topic: Controlling thread data

- Any variables locally defined to the function are private.
- static variables in C and save variables in Fortran are shared.
- The function arguments inherit their status from the calling environment.

22.4 Temporary variables in a loop

It is common to have a variable that is set and used in each loop iteration:

```c
#pragma omp parallel for
for ( ... i ... ) {
    x = i*h;
    s = sin(x); c = cos(x);
    a[i] = s+c;
    b[i] = s-c;
}
```

By the above rules, the variables \(x, s, c\) are all shared variables. However, the values they receive in one iteration are not used in a next iteration, so they behave in fact like private variables to each iteration.

- In both C and Fortran you can declare these variables private in the parallel for directive.
- In C, you can also redefine the variables inside the loop.

Sometimes, even if you forget to declare these temporaries as private, the code may still give the correct output. That is because the compiler can sometimes eliminate them from the loop body, since it detects that their values are not otherwise used.

22.5 Default

- Loop variables in an omp for are private;
- Local variables in the parallel region are private.

You can alter this default behavior with the default clause:

```c
#pragma omp parallel default(shared) private(x)
{ ... }
#pragma omp parallel default(private) shared(matrix)
{ ... }
```

and if you want to play it safe:

```c
#pragma omp parallel default(none) private(x) shared(matrix)
{ ... }
```

- The shared clause means that all variables from the outer scope are shared in the parallel region; any private variables need to be declared explicitly. This is the default behavior.
- The private clause means that all outer variables become private in the parallel region. They are not initialized; see the next option. Any shared variables in the parallel region need to be declared explicitly. This value is not available in C.
22.6 Array data

The rules for arrays are slightly different from those for scalar data:

1. Statically allocated data, that is with a syntax like

   ```
   int array[100];
   integer, dimension(:) :: array(100)
   ```

   can be shared or private, depending on the clause you use.

2. Dynamically allocated data, that is, created with `malloc` or `allocate`, can only be shared.

Example of the first type:

```c
int array[nthreads];
#pragma omp parallel firstprivate(array)
{
   int t = omp_get_thread_num();
   array[t] = t;
}
```

For the full source of this example, see section 22.10.2

each thread gets a private copy of the array, properly initialized.

On the other hand, in

```c
// alloc.c
int *array = (int*) malloc(nthreads*sizeof(int));
#pragma omp parallel firstprivate(array)
{
   int t = omp_get_thread_num();
   array += t;
   array[0] = t;
}
```

For the full source of this example, see section 22.10.2

each thread gets a private pointer, but all pointers point to the same object.
22.7 First and last private

Above, you saw that private variables are completely separate from any variables by the same name in the surrounding scope. However, there are two cases where you may want some storage association between a private variable and a global counterpart.

First of all, private variables are created with an undefined value. You can force their initialization with firstprivate.

```c
int t=2;
#pragma omp parallel firstprivate(t)
{
    t += f(omp_get_thread_num());
g(t);
}
```

The variable `t` behaves like a private variable, except that it is initialized to the outside value.

Secondly, you may want a private value to be preserved to the environment outside the parallel region. This really only makes sense in one case, where you preserve a private variable from the last iteration of a parallel loop, or the last section in an sections construct. This is done with lastprivate:

```c
#pragma omp parallel for \
    lastprivate(tmp)
for (i=0; i<N; i++) {
    tmp = ......
    x[i] = .... tmp ....
}
```

22.8 Persistent data through threadprivate

Most data in OpenMP parallel regions is either inherited from the master thread and therefore shared, or temporary within the scope of the region and fully private. There is also a mechanism for thread-private data, which is not limited in lifetime to one parallel region. The threadprivate pragma is used to declare that each thread is to have a private copy of a variable:

```c
#pragma omp threadprivate(var)
```

The variable needs be:

- a file or static variable in C,
- a static class member in C++, or
- a program variable or common block in Fortran.

22.8.1 Thread private initialization

If each thread needs a different value in its threadprivate variable, the initialization needs to happen in a parallel region.
In the following example a team of 7 threads is created, all of which set their thread-private variable. Later, this variable is read by a larger team: the variables that have not been set are undefined, though often simply zero:

```c
// threadprivate.c
static int tp;

int main(int argc, char **argv) {
    #pragma omp threadprivate(tp)
    #pragma omp parallel num_threads(7)
    tp = omp_get_thread_num();
    #pragma omp parallel num_threads(9)
    printf("Thread %d has %d
", omp_get_thread_num(), tp);
}
```

For the full source of this example, see section 22.10.3

Fortran note 16: Private common blocks. Named common blocks can be made thread-private with the syntax

```fortran
#pragma threadprivate(/blockname/)
```

Example:

```
!! threadprivate.F90
common /threaddata/tp
integer :: tp
!$omp threadprivate(/threaddata/)
!$omp parallel num_threads(7)
    tp = omp_get_thread_num()
!$omp end parallel
!$omp parallel num_threads(9)
print *,"Thread",omp_get_thread_num(),"has","tp
!$omp end parallel
```

On the other hand, if the thread private data starts out identical in all threads, the `copyin` clause can be used:

```
#pragma omp threadprivate(private_var)

private_var = 1;
#pragma omp parallel copyin(private_var)
    private_var += omp_get_thread_num()
```

If one thread needs to set all thread private data to its value, the `copyprivate` clause can be used:

```
#pragma omp parallel
{
    ...
```
Threadprivate variables require `OMP_DYNAMIC` to be switched off.

### 22.8.2 Thread private example

The typical application for thread-private variables is in random number generators. A random number generator needs saved state, since it computes each next value from the current one. To have a parallel generator, each thread will create and initialize a private ‘current value’ variable. This will persist even when the execution is not in a parallel region; it gets updated only in a parallel region.

**Exercise 22.1.** Calculate the area of the Mandelbrot set by random sampling. Initialize the random number generator separately for each thread; then use a parallel loop to evaluate the points. Explore performance implications of the different loop scheduling strategies.

### 22.9 Allocators

The OpenMP was initially designed for shared memory. With accelerators (see chapter 28, non-coherent memory was added to this. In the OpenMP-5 standard, the story is further complicated, to account for new memory types such as high-bandwidth memory and non-volatile memory.

There are several ways of using the OpenMP memory allocators.

- First, in a directory on a static array:

  ```c
  float A[N], B[N];
  #pragma omp allocate(A) allocator(omp_large_cap_mem_alloc)
  ```

- As a clause on private variables:

  ```c
  #pragma omp task private(B) allocate(omp_const_mem_alloc: B)
  ```

- With `omp_alloc`, using a (possibly user-defined) allocator.

Next, there are memory spaces. The binding between OpenMP identifiers and hardware is implementation defined.

#### 22.9.1 Pre-defined types

Allocators: `omp_default_mem_alloc`, `omp_large_cap_mem_alloc`, `omp_const_mem_alloc`, `omp_high_bw_mem_alloc`, `omp_low_lat_mem_alloc`, `omp_cgroup_mem_alloc`, `omp_pteam_mem_alloc`, `omp_thread_mem_alloc`.

Memory spaces: `omp_default_mem_space`, `omp_large_cap_mem_space`, `omp_const_mem_space`, `omp_high_bw_mem_space`, `omp_low_lat_mem_space`. 
22.10 Sources used in this chapter

22.10.1 Listing of code header

22.10.2 Listing of code examples/omp/c/alloc.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

  int nthreads;
  #pragma omp parallel
  #pragma omp master
  nthreads = omp_get_num_threads();

  int *array = (int*) malloc(nthreads*sizeof(int));
  for (int i=0; i<nthreads; i++)
    array[i] = 0;

  #pragma omp parallel firstprivate(array)
  {
    int t = omp_get_thread_num();
    array += t;
    array[0] = t;
  }

  printf("Array result: \n");
  for (int i=0; i<nthreads; i++)
    printf("%d:%d, ",i,array[i]);
  printf("\n");
}

{ int nthreads;
  #pragma omp parallel
  #pragma omp master
  nthreads = omp_get_num_threads();

  int array[nthreads];
  for (int i=0; i<nthreads; i++)
    array[i] = 0;

  #pragma omp parallel firstprivate(array)
  {
    int t = omp_get_thread_num();
    array[t] = t;
  }

  printf("Array result: \n");
```
for (int i=0; i<nthreads; i++)
    printf("%d:%d, ",i,array[i]);
printf("\n");//

int nthreads=4;
int array[nthreads];
for (int i=0; i<nthreads; i++)
    array[i] = 0;

int t = 2;
array += t;
array[0] = t;

printf("Array result:\n");
for (int i=0; i<nthreads; i++)
    printf("%d:%d, ",i,array[i]);
printf("\n");
return 0;

22.10.3 Listing of code examples/omp/c/threadprivate.c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

static int tp;

int main(int argc,char **argv) {

    #pragma omp threadprivate(tp)
    #pragma omp parallel num_threads(7)
        tp = omp_get_thread_num();
    #pragma omp parallel num_threads(9)
        printf("Thread %d has %d\n",omp_get_thread_num(),tp);

    return 0;
}
Chapter 23

OpenMP topic: Synchronization

In the constructs for declaring parallel regions above, you had little control over in what order threads executed the work they were assigned. This section will discuss synchronization constructs: ways of telling threads to bring a certain order to the sequence in which they do things.

- **critical**: a section of code can only be executed by one thread at a time; see 23.2.2.
- **atomic**: Atomic update of a single memory location. Only certain specified syntax patterns are supported. This was added in order to be able to use hardware support for atomic updates.
- **barrier**: section 23.1.
- **ordered**: section 19.6.
- **locks**: section 23.3.
- **flush**: section 26.3.
- **nowait**: section 19.7.

### 23.1 Barrier

A barrier defines a point in the code where all active threads will stop until all threads have arrived at that point. With this, you can guarantee that certain calculations are finished. For instance, in this code snippet, computation of $y$ can not proceed until another thread has computed its value of $x$.

```c
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  x[mytid] = some_calculation();
  y[mytid] = x[mytid]+x[mytid+1];
}
```

This can be guaranteed with a barrier pragma:

```c
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  x[mytid] = some_calculation();
  #pragma omp barrier
  y[mytid] = x[mytid]+x[mytid+1];
}
```
Apart from the barrier directive, which inserts an explicit barrier, OpenMP has *implicit barriers* after a load sharing construct. Thus the following code is well defined:

```c
#pragma omp parallel
{
    #pragma omp for
    for (int mytid=0; mytid<number_of_threads; mytid++)
        x[mytid] = some_calculation();
    #pragma omp for
    for (int mytid=0; mytid<number_of_threads-1; mytid++)
        y[mytid] = x[mytid]+x[mytid+1];
}
```

You can also put each parallel loop in a parallel region of its own, but there is some overhead associated with creating and deleting the team of threads in between the regions.

### 23.1.1 Implicit barriers

At the end of a parallel region the team of threads is dissolved and only the master thread continues. Therefore, there is an *implicit barrier at the end of a parallel region*.

There is some *barrier behavior* associated with *omp for* loops and other *worksharing constructs* (see section 21.3). For instance, there is an *implicit barrier* at the end of the loop. This barrier behavior can be cancelled with the *nowait* clause.

You will often see the idiom

```c
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=0; i<N; i++)
        a[i] = // some expression
    #pragma omp for
    for (i=0; i<N; i++)
        b[i] = ....... a[i] .......
```

Here the *nowait* clause implies that threads can start on the second loop while other threads are still working on the first. Since the two loops use the same schedule here, an iteration that uses a[i] can indeed rely on it that that value has been computed.

### 23.2 Mutual exclusion

Sometimes it is necessary to limit a piece of code so that it can be executed by only one thread at a time. Such a piece of code is called a *critical section*, and OpenMP has several mechanisms for realizing this.
23.2. Mutual exclusion

23.2.1 Why critical sections?

The most common use of critical sections is to update a variable. Since updating involves reading the old value, and writing back the new, this has the possibility for a race condition: another thread reads the current value before the first can update it; the second thread the updates to the wrong value.

Suppose that two processes both try to increment an integer variable I:

process 1: \( I = I + 2 \)

process 2: \( I = I + 3 \)

This is a legitimate activity if the variable is an accumulator for values computed by independent processes. The result of these two updates depends on the sequence in which the processors read and write the variable.

<table>
<thead>
<tr>
<th>scenario 1.</th>
<th>scenario 2.</th>
<th>scenario 3.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( I = 0 )</td>
<td>( I = 0 )</td>
</tr>
<tr>
<td>read ( I = 0 )</td>
<td>read ( I = 0 )</td>
<td>read ( I = 0 )</td>
</tr>
<tr>
<td>compute ( I = 2 )</td>
<td>compute ( I = 2 )</td>
<td>compute ( I = 2 )</td>
</tr>
<tr>
<td>write ( I = 2 )</td>
<td>write ( I = 3 )</td>
<td>write ( I = 3 )</td>
</tr>
<tr>
<td>write ( I = 3 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( I = 3 )</td>
<td>( I = 2 )</td>
<td>( I = 5 )</td>
</tr>
</tbody>
</table>

Figure 23.1: Three executions of a data race scenario

Figure 23.1 illustrates three scenarios. Such a scenario, where the final result depends on ‘micro-timing’ of the actions of a thread, is known as a race condition or data race. A formal definition would be:

We talk of a a data race if there are two statements \( S_1, S_2 \),

- that are not causally related;
- that both access a location \( L \); and
- at least one access is a write.

Enclosing the update statement in a critical section, or making it atomic by some other mechanism, enforces scenario 3 of the above figure.

23.2.2 critical and atomic

There are two pragmas for critical sections: critical and atomic. Both denote atomic operations in a technical sense. The first one is general and can contain an arbitrary sequence of instructions; the second one is more limited but has performance advantages.

The typical application of a critical section is to update a variable:

```c
#pragma omp parallel
{
    int mytid = omp_get_thread_num();
    double tmp = some_function(mytid);
}
```
Exercise 23.1. Consider a loop where each iteration updates a variable.

```c
#pragma omp parallel for shared(result)
for (i) {
    result += some_function_of(i);
}
```

Discuss qualitatively the difference between:
- turning the update statement into a critical section, versus
- letting the threads accumulate into a private variable `tmp` as above, and summing these after the loop.

Do an Ahmdal-style quantitative analysis of the first case, assuming that you do $n$ iterations on $p$ threads, and each iteration has a critical section that takes a fraction $f$. Assume the number of iterations $n$ is a multiple of the number of threads $p$. Also assume the default static distribution of loop iterations over the threads.

A critical section works by acquiring a lock, which carries a substantial overhead. Furthermore, if your code has multiple critical sections, they are all mutually exclusive: if a thread is in one critical section, the other ones are all blocked.

On the other hand, the syntax for atomic sections is limited to the update of a single memory location, but such sections are not exclusive and they can be more efficient, since they assume that there is a hardware mechanism for making them critical.

The problem with critical sections being mutually exclusive can be mitigated by naming them:

```c
#pragma omp critical (optional_name_in_parens)
```

Critical sections are an easy way to turn an existing code into a correct parallel code. However, there are performance disadvantages to critical sections, and sometimes a more drastic rewrite is called for.

### 23.3 Locks

OpenMP also has the traditional mechanism of a lock. A lock is somewhat similar to a critical section: it guarantees that some instructions can only be performed by one process at a time. However, a critical section is indeed about code; a lock is about data. With a lock you make sure that some data elements can only be touched by one process at a time.

One simple example of the use of locks is generation of a histogram. A histogram consists of a number of bins, that get updated depending on some data. Here is the basic structure of such a code:

```c
int count[100];
float x = some_function();
int ix = (int)x;
if (ix>=100)
```
23.3. Locks

```c
error();
else
count[ix]++;
```

It would be possible to guard the last line:

```c
#pragma omp critical
count[ix]++;
```

but that is unnecessarily restrictive. If there are enough bins in the histogram, and if the `some_function` takes enough time, there are unlikely to be conflicting writes. The solution then is to create an array of locks, with one lock for each count location.

Create/destroy:

```c
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Set and release:

```c
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

Since the set call is blocking, there is also

```c
omp_test_lock();
```

Unsetting a lock needs to be done by the thread that set it.

Lock operations implicitly have a `flush`.

**Exercise 23.2.** In the following code, one process sets array A and then uses it to update B; the other process sets array B and then uses it to update A. Argue that this code can deadlock. How could you fix this?

```c
#pragma omp parallel shared(a, b, nthreads, locka, lockb)
#pragma omp sections nowait
{
#pragma omp section
{
omp_set_lock(&locka);
for (i=0; i<N; i++)
a[i] = ..
omp_set_lock(&lockb);
for (i=0; i<N; i++)
b[i] = .. a[i]..
omp_unset_lock(&lockb);
omp_unset_lock(&locka);

#pragma omp section
{
omp_set_lock(&lockb);
```
23. OpenMP topic: Synchronization

```c
for (i=0; i<N; i++)
    b[i] = ...
omp_set_lock(&locka);
for (i=0; i<N; i++)
    a[i] = .. b[i] ..
omp_unset_lock(&locka);
omp_unset_lock(&lockb);
}
/* end of sections */
} /* end of parallel region */
```

### 23.3.1 Nested locks

A lock as explained above can not be locked if it is already locked. A *nested lock* can be locked multiple times by the same thread before being unlocked.

- `omp_init_nest_lock`
- `omp_destroy_nest_lock`
- `omp_set_nest_lock`
- `omp_unset_nest_lock`
- `omp_test_nest_lock`

### 23.3.2 Example: object with atomic update

OO languages such as C++ allow for syntactic simplification, for instance building the locking and unlocking actions into the update operator.

*C++ note 6: Lock inside overloaded operator.*

```c
// lockobject.cxx
class object {
private:
    omp_lock_t the_lock;
    int _value{0};
public:
    object() {
        omp_init_lock(&the_lock);
    }
    ~object() {
        omp_destroy_lock(&the_lock);
    }
    int operator+=( int i ) {
        // atomic increment
        omp_set_lock(&the_lock);
        _value += (i>0); int rv = _value;
        omp_unset_lock(&the_lock);
        return rv;
    }
    auto value() { return _value; }
};
```
23.4 Example: Fibonacci computation

The Fibonacci sequence is recursively defined as

\[ F(0) = 1, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2. \]

We start by sketching the basic single-threaded solution. The naive code looks like:

```c
int main() {
    value = new int[nmax+1];
    value[0] = 1;
    value[1] = 1;
    fib(10);
}

int fib(int n) {
    int i, j, result;
    if (n >= 2) {
        i = fib(n-1); j = fib(n-2);
        value[n] = i + j;
    }
    return value[n];
}
```

However, this is inefficient, since most intermediate values will be computed more than once. We solve this by keeping track of which results are known:

```c
done = new int[nmax+1];
for (i=0; i<nmax; i++)
    done[i] = 0;
done[0] = 1;
done[1] = 1;
...```

For the full source of this example, see section 23.7.2

Running this:

```c
for (int ithread=0; ithread<NTHREADS; ithread++) {
    threads.push_back
    ( thread(
        [&my_object] () {
            for (int iop=0; iop<NOPS; iop++)
                my_object += iop;
        })
    );
}
```

For the full source of this example, see section 23.7.2

Victor Eijkhout 505
The OpenMP parallel solution calls for two different ideas. First of all, we parallelize the recursion by using tasks (section 24):

```c
int fib(int n) {
    int i, j;
    if (!done[n]) {
        i = fib(n-1); j = fib(n-2);
        value[n] = i+j; done[n] = 1;
    }
    return value[n];
}
```

This computes the right solution, but, as in the naive single-threaded solution, it recomputes many of the intermediate values.

A naive addition of the done array leads to data races, and probably an incorrect solution:

```c
int fib(int n) {
    int i, j, result;
    if (!done[n]) {
        #pragma omp task shared(i) firstprivate(n)
        i=fib(n-1);
        #pragma omp task shared(j) firstprivate(n)
        j=fib(n-2);
        #pragma omp taskwait
        value[n] = i+j;
        done[n] = 1;
    }
    return value[n];
}
```

For instance, there is no guarantee that the done array is updated later than the value array, so a thread can think that done[n-1] is true, but value[n-1] does not have the right value yet.

One solution to this problem is to use a lock, and make sure that, for a given index n, the values done[n] and value[n] are never touched by more than one thread at a time:

```c
int fib(int n) {
    int i, j;
```
omp_set_lock( &dolock[n] );
if (!done[n]) {
  #pragma omp task shared(i) firstprivate(n)
  i = fib(n-1);
  #pragma omp task shared(j) firstprivate(n)
  j = fib(n-2);
  #pragma omp taskwait
  value[n] = i+j;
  done[n] = 1;
}
omp_unset_lock( &dolock[n] );
return value[n];

This solution is correct, optimally efficient in the sense that it does not recompute anything, and it uses tasks to obtain a parallel execution.

However, the efficiency of this solution is only up to a constant. A lock is still being set, even if a value is already computed and therefore will only be read. This can be solved with a complicated use of critical sections, but we will forego this.

23.5 Review

Exercise 23.3. C++ only.
Write a class for a scalar that can be atommically updated:

```cpp
object my_object;
#pragma omp parallel
#pragma omp single
#pragma omp taskgroup
for (int ithread=0; ithread<NTHREADS; ithread++) {
  #pragma omp task shared(my_object)
  for (int iop=0; iop<NOPS; iop++)
    my_object.inc(iop);
}
```

23.6 Review questions

Exercise 23.4. C++ only: write a class with an atomic update method, so as to make the following code work:

```cpp
object my_object;
#pragma omp parallel
#pragma omp single
#pragma omp taskgroup
for (int ithread=0; ithread<NTHREADS; ithread++) {
  #pragma omp task shared(my_object)
  for (int iop=0; iop<NOPS; iop++)
    my_object.inc(iop);
```
} 
    cout << "Did " << NTHREADS * NOPS << " updates, over " << NTHREADS 
        << " threads, resulting in " << my_object.value() << endl;
23.7 Sources used in this chapter

23.7.1 Listing of code header

23.7.2 Listing of code examples/omp/cxx/lockobject.cxx

```cpp
#include <iostream>
#include <thread>
#include <vector>
using namespace std;

#include <cmath>
#include <omp.h>

#ifdef NOLOCK
#define omp_set_lock(x)
#define omp_unset_lock(x)
#endif

class object {
  private:
    omp_lock_t the_lock;
    int _value{0};
  public:
    object() {
      omp_init_lock(&the_lock);
    }
    ~object() {
      omp_destroy_lock(&the_lock);
    }
    int operator +=( int i ) {
      // let's waste a little time,
      // otherwise the threads finish before they start
      float s = i;
      for (int i=0; i<1000; i++)
        s += sin(i)*sin(i);

      // atomic increment
      omp_set_lock(&the_lock);
      _value += (s>0); int rv = _value;
      omp_unset_lock(&the_lock);
      return rv;
    }
    auto value() { return _value; }
  }

#define NTHREADS 50
#define NOPS 100

int main() {

Victor Eijkhout
509
```
/ * Create a bunch of threads, that
 * each do a bunch of updates
 */

object my_object;
vector<thread> threads;
for (int ithread=0; ithread<NTHREADS; ithread++) {
    threads.push_back
        ( thread(
            [&my_object] () {
                for (int iop=0; iop<NOPS; iop++)
                    my_object += iop; } ) );
}
for ( auto &t : threads )
    t.join();

/*
 * Check that no updates have gone lost
 */

cout << "Did " << NTHREADS * NOPS << " updates, over " << threads.size()
    << " threads, resulting in " << my_object.value() << endl;

return 0;
}
Chapter 24

OpenMP topic: Tasks

Tasks are a mechanism that OpenMP uses under the cover: if you specify something as being a task, OpenMP will create a ‘block of work’: a section of code plus the data environment in which it occurred. This block is set aside for execution at some later point.

Let’s look at a simple example using the `task` directive.

<table>
<thead>
<tr>
<th>Code</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = f();</code></td>
<td>the variable x gets a value</td>
</tr>
<tr>
<td><code>#pragma omp task</code></td>
<td>a task is created with the current value of x</td>
</tr>
<tr>
<td><code>{ y = g(x); }</code></td>
<td></td>
</tr>
<tr>
<td><code>z = h();</code></td>
<td>the variable z gets a value</td>
</tr>
</tbody>
</table>

The thread that executes this code segment creates a task, which will later be executed, probably by a different thread (from the same team; we will not mention this from now on). The exact timing of the execution of the task is up to a task scheduler, which operates invisible to the user.

The task mechanism allows you to do things that are hard or impossible with the loop and section constructs. For instance, a `while loop` traversing a `linked list` can be implemented with tasks:

<table>
<thead>
<tr>
<th>Code</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p = head_of_list();</code></td>
<td>one thread traverses the list</td>
</tr>
<tr>
<td><code>while (!end_of_list(p))</code> {</td>
<td></td>
</tr>
<tr>
<td><code>#pragma omp task</code></td>
<td>a task is created,</td>
</tr>
<tr>
<td><code>process(p);</code></td>
<td>one for each element</td>
</tr>
<tr>
<td><code>p = next_element(p);</code></td>
<td>the generating thread goes on without waiting</td>
</tr>
<tr>
<td>}</td>
<td>the tasks are executed while</td>
</tr>
<tr>
<td></td>
<td>more are being generated.</td>
</tr>
</tbody>
</table>

The way tasks and threads interact is different from the worksharing constructs you’ve seen so far. Typically, one thread will generate the tasks, adding them to a queue, from which all threads can take and execute them. This leads to the following idiom:

```c
#pragma omp parallel
#pragma omp single
{ ... 
#pragma omp task
```
24. OpenMP topic: Tasks

1. A parallel region creates a team of threads;
2. a single thread then creates the tasks, adding them to a queue that belongs to the team,
3. and all the threads in that team (possibly including the one that generated the tasks)

With tasks it becomes possible to parallelize processes that did not fit the earlier OpenMP constructs. For instance, if a certain operation needs to be applied to all elements of a linked list, you can have one thread go down the list, generating a task for each element of the list.

Another concept that was hard to parallelize earlier is the ‘while loop’. This does not fit the requirement for OpenMP parallel loops that the loop bound needs to be known before the loop executes.

Exercise 24.1. Use tasks to find the smallest factor of a large number (using $2999 \cdot 3001$ as test case): generate a task for each trial factor. Start with this code:

```c
int factor=0;
#pragma omp parallel
#pragma omp single
  for (int f=2; f<4000; f++) {
    // see if `f` is a factor
    if (n%f==0) { // found factor!
      factor = f;
    }
  }
if (factor>0)
  break;
if (factor>0)
  printf("Found a factor: %d \n",factor);
```

- Turn the factor finding block into a task.
- Run your program a number of times:
  ```sh
  for i in `seq 1 1000` ; do ./taskfactor ; done | grep -v 2999
  ```
  Does it find the wrong factor? Why? Try to fix this.
- Once a factor has been found, you should stop generating tasks. Let tasks that should not have been generated, meaning that they test a candidate larger than the factor found, print out a message.

24.1 Task data

Treatment of data in a task is somewhat subtle. The basic problem is that a task gets created at one time, and executed at some later time. Thus, if shared data is accessed, does the task see the value at creation time or at execution time? In fact, both possibilities make sense depending on the application, so we need to discuss the rules when which possibility applies.

The first rule is that shared data is shared in the task, but private data becomes `firstprivate`. To see the distinction, consider two code fragments.
24.2. Task synchronization

Even though the above segment looks like a linear set of statements, it is impossible to say when the code after the `task` directive will be executed. This means that the following code is incorrect:

```c
x = f();
#pragma omp task
{ y = g(x); }
#pragma omp task
{ y2 = g2(x); }
```

Explanation: when the statement computing z is executed, the task computing y has only been scheduled; it has not necessarily been executed yet.

In order to have a guarantee that a task is finished, you need the `taskwait` directive. The following creates two tasks, which can be executed in parallel, and then waits for the results:

<table>
<thead>
<tr>
<th>Code</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = f();</code></td>
<td>the variable x gets a value</td>
</tr>
<tr>
<td><code>#pragma omp task</code></td>
<td>two tasks are created with the current value of x</td>
</tr>
<tr>
<td><code>{ y1 = g1(x); }</code></td>
<td></td>
</tr>
<tr>
<td><code>#pragma omp task</code></td>
<td>the thread waits until the tasks are finished</td>
</tr>
<tr>
<td><code>{ y2 = g2(x); }</code></td>
<td></td>
</tr>
<tr>
<td><code>#pragma omp taskwait</code></td>
<td>the variable z is computed using the task results</td>
</tr>
<tr>
<td><code>z = h(y1)+h(y2);</code></td>
<td></td>
</tr>
</tbody>
</table>

In the first example, the variable `count` is declared outside the parallel region and is therefore shared. When the print statement is executed, all tasks will have been generated, and so `count` will be zero. Thus, the output will likely be 0,50.

In the second example, the `count` variable is private to the thread creating the tasks, and so it will be `firstprivate` in the task, preserving the value that was current when the task was created.
The `task` pragma is followed by a structured block. Each time the structured block is encountered, a new task is generated. On the other hand `taskwait` is a standalone directive; the code that follows is just code, it is not a structured block belonging to the directive.

Another aspect of the distinction between generating tasks and executing them: usually the tasks are generated by one thread, but executed by many threads. Thus, the typical idiom is:

```c
#pragma omp parallel
#pragma omp single
{
   // code that generates tasks
}
```

This makes it possible to execute loops in parallel that do not have the right kind of iteration structure for a `omp parallel for`. As an example, you could traverse and process a linked list:

```c
#pragma omp parallel
#pragma omp single
{
   while (!tail(p)) {
      p = p->next();
      //omp task process(p)
   }
   #pragma omp taskwait
}
```

One task traverses the linked list creating an independent task for each element in the list. These tasks are then executed in parallel; their assignment to threads is done by the task scheduler.

You can indicate task dependencies in several ways:

1. Using the ‘task wait’ directive you can explicitly indicate the join of the forked tasks. The instruction after the wait directive will therefore be dependent on the spawned tasks.
2. The `taskgroup` directive, followed by a structured block, ensures completion of all tasks created in the block, even if recursively created.
3. Each OpenMP task can have a `depend` clause, indicating what data dependency of the task. By indicating what data is produced or absorbed by the tasks, the scheduler can construct the dependency graph for you.

Another mechanism for dealing with tasks is the `taskgroup`: a task group is a code block that can contain `task` directives; all these tasks need to be finished before any statement after the block is executed.

A task group is somewhat similar to having a `taskwait` directive after the block. The big difference is that that `taskwait` directive does not wait for tasks that are recursively generated, while a `taskgroup` does.

### 24.3 Task dependencies

It is possible to put a partial ordering on tasks through use of the `depend` clause. For example, in
it is conceivable that the second task is executed before the first, possibly leading to an incorrect result. This is remedied by specifying:

```c
#pragma omp task depend(out:x)
    x = f()
#pragma omp task depend(in:x)
    y = g(x)
```

Exercise 24.2. Consider the following code:

```c
for i in [1:N]:
    x[0,i] = some_function_of(i)
    x[i,0] = some_function_of(i)
for i in [1:N]:
    for j in [1:N]:
        x[i,j] = x[i-1,j]+x[i,j-1]
```

- Observe that the second loop nest is not amenable to OpenMP loop parallelism.
- Can you think of a way to realize the computation with OpenMP loop parallelism? Hint: you need to rewrite the code so that the same operations are done in a different order.
- Use tasks with dependencies to make this code parallel without any rewriting: the only change is to add OpenMP directives.

Tasks dependencies are used to indicate how two uses of one data item relate to each other. Since either use can be a read or a write, there are four types of dependencies.

**RaW (Read after Write)** The second task reads an item that the first task writes. The second task has to be executed after the first:

```c
... #pragma omp task depend(OUT:x)
    foo(x)
... #pragma omp task depend(IN:x)
    foo(x)
```

**WaR (Write after Read)** The first task reads an item, and the second task overwrites it. The second task has to be executed second to prevent overwriting the initial value:

```c
... #pragma omp task depend(IN:x)
    foo(x)
... #pragma omp task depend(OUT:x)
    foo(x)
```

**WaW (Write after Write)** Both tasks set the same variable. Since the variable can be used by an intermediate task, the two writes have to be executed in this order.

```c
... #pragma omp task depend(OUT:x)
    foo(x)
... #pragma omp task depend(OUT:x)
    foo(x)
```
24. OpenMP topic: Tasks

**RaR (Read after Read)** Both tasks read a variable. Since neither tasks has an ‘out’ declaration, they can run in either order.

```
... omp task depend(IN:x)
  foo(x)
... omp task depend(IN:x)
  foo(x)
```

### 24.4 Task reduction

The *reduction* only pertains to ordinary parallel loops, not to *taskgroup* loops of tasks. To do a reduction over computations in tasks you need the *task_reduction* clause (a OpenMP-5.0 feature):

```
#pragma omp taskgroup task_reduction(+:sum)
```

The task group can contain both task that contribute to the reduction, and ones that don’t. The former type needs a clause *in_reduction*:

```
#pragma omp task in_reduction(+:sum)
```

As an example, here the sum \(\sum_{i=1}^{100} i\) is computed with tasks:

```
// taskreduct.c
#pragma omp parallel
#pragma omp single
{
  #pragma omp taskgroup task_reduction(+:sum)
  for (int itask=1; itask<=bound; itask++) {
    #pragma omp task in_reduction(+:sum)
    sum += itask;
  }
}
```

### 24.5 More

#### 24.5.1 Scheduling points

Normally, a task stays tied to the thread that first executes it. However, at a *task scheduling point* the thread may switch to the execution of another task created by the same team.

- There is a scheduling point after explicit task creation. This means that, in the above examples, the thread creating the tasks can also participate in executing them.
- There is a scheduling point at *taskwait* and *taskyield*.

On the other hand a task created with them *untied* clause on the task pragma is never tied to one thread. This means that after suspension at a scheduling point any thread can resume execution of the task. If you do this, beware that the value of a thread-id does not stay fixed. Also locks become a problem.

Example: if a thread is waiting for a lock, with a scheduling point it can suspend the task and work on another task.
24.6 Examples

24.6.1 Fibonacci

As an example of the use of tasks, consider computing an array of Fibonacci values:

```c
// taskgroup0.c
for (int i=2; i<N; i++)
{
    fibo_values[i] = fibo_values[i-1] + fibo_values[i-2];
}
```

For the full source of this example, see section 24.7.2

If you simply turn each calculation into a task, results will be unpredictable (confirm this!) since tasks can be executed in any sequence. To solve this, we put dependencies on the tasks:

```c
while (!omp_test_lock(lock))
    #pragma omp taskyield
;
```
24. OpenMP topic: Tasks

```c
#include <stdio.h>

int main()
{
    int N = 10;
    int fibo_values[N] = {0};
    fibo_values[0] = 0;
    fibo_values[1] = 1;
    for (int i = 2; i < N; i++)
    {
        fibo_values[i] = fibo_values[i-1] + fibo_values[i-2];
    }
    return 0;
}
```

For the full source of this example, see section 24.7.3

### 24.6.2 Binomial coefficients

**Exercise 24.4.** An array of binomial coefficients can be computed as follows:

```c
#include <stdio.h>

int main()
{
    int n = 10;
    int row, col;
    int array[n][n];
    for (row = 0; row <= n; row++)
    {
        for (col = 0; col <= row; col++)
        {
            if (row == 1 || col == 1 || col == row)
                array[row][col] = 1;
            else
                array[row][col] = array[row-1][col] + array[row-1][col-1];
        }
    }
    return 0;
}
```

For the full source of this example, see section 24.7.4

Putting a single task group around the double loop, and use `depend` clauses to make the execution satisfy the proper dependencies.
24.7 Sources used in this chapter

24.7.1 Listing of code header

24.7.2 Listing of code examples/omp/c/taskgroup0.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#include <string.h>

int main(int argc,char **argv) {
    int N = 1000;
    if (argc>1) {
        if (!strcmp(argv[1],"-h")) {
            printf("usage: %s [nnn]\n",argv[0]);
            return 0;
        }
        N = atoi(argv[1]);
        if (N>99) {
            printf("Sorry, this overflows: setting N=99\n");
            N = 99;
        }
    }
    long int *fibo_values = (long int*)malloc(N*sizeof(long int));

    fibo_values[0] = 1; fibo_values[1] = 1;
    for (int i=2; i<N; i++) {
        fibo_values[i] = fibo_values[i-1]+fibo_values[i-2];
    }
    printf("F(%d) = %ld\n",N,fibo_values[N-1]);
    return 0;
}
```

24.7.3 Listing of code examples/omp/c/taskgroup2.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#include <string.h>

int main(int argc,char **argv) {
    int N = 1000;
    if (argc>1) {
        if (!strcmp(argv[1],"-h")) {
            printf("usage: %s [nnn]\n",argv[0]);
```
24. OpenMP topic: Tasks

```c
... return 0; }
N = atoi(argv[1]);
if (N>99) {
    printf("Sorry, this overflows: setting N=99\n");
    N = 99;
}
}

long int *fibo_values = (long int*)malloc(N*sizeof(long int));

fibo_values[0] = 1; fibo_values[1] = 1;
#pragma omp parallel
#pragma omp single
#pragma omp taskgroup
{
    for (int i=2; i<N; i++)
        #pragma omp task \
        depend(out:fibo_values[i]) \
        depend(in:fibo_values[i-1],fibo_values[i-2])
        {
            fibo_values[i] = fibo_values[i-1]+fibo_values[i-2];
        }
    printf("F(%d) = %ld\n",N,fibo_values[N-1]);
}
return 0;
}
```

24.7.4 Listing of code code/omp/c/binomial1.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define NMAX 30000/sizeof(int)
long int array[NMAX][NMAX];

long int coeff(int row,int col) {
    if (row==1) return 1;
    else if (col==1) return 1;
    else if (col==row) return 1;
    else return array[row-1][col-1]+array[row-1][col];
}

int main(int argc,char **argv) {
    int n;
    if (argc>1)
        n = atoi(argv[1]);
    else n = 100;
    if (n>NMAX) {
...```
24.7. Sources used in this chapter

```c
printf("N too large: keep under \%d\n",NMAX);
return 0; }

for (int row=1; row<=n; row++)
    for (int col=1; col<=row; col++)
        if (row==1 || col==1 || col==row)  
            array[row][col] = 1;
    else
        array[row][col] = array[row-1][col-1] + array[row-1][col];

for (int col=1; col<=n; col++) {
    long int binom = array[n][col];
    printf("%ld ",binom);
} printf("\n");

return 0;
}
```
Chapter 25

OpenMP topic: Affinity

25.1 OpenMP thread affinity control

The matter of thread affinity becomes important on *multi-socket nodes*; see the example in section 25.2.

Thread placement can be controlled with two environment variables:

- the environment variable `OMP_PROC_BIND` describes how threads are bound to *OpenMP places*;
- the variable `OMP_PLACES` describes these places in terms of the available hardware.

When you’re experimenting with these variables it is a good idea to set `OMP_DISPLAY_ENV` to true, so that OpenMP will print out at runtime how it has interpreted your specification. The examples in the following sections will display this output.

25.1.1 Thread binding

The variable `OMP_PLACES` defines a series of places to which the threads are assigned.

Example: if you have two sockets and you define

```
OMP_PLACES=sockets
```

then

- thread 0 goes to socket 0,
- thread 1 goes to socket 1,
- thread 2 goes to socket 0 again,
- and so on.

On the other hand, if the two sockets have a total of sixteen cores and you define

```
OMP_PLACES=cores
OMP_PROC_BIND=close
```

then

- thread 0 goes to core 0, which is on socket 0,
- thread 1 goes to core 1, which is on socket 0,
- thread 2 goes to core 2, which is on socket 0,
- and so on, until thread 7 goes to core 7 on socket 0, and
25.1. OpenMP thread affinity control

- thread 8 goes to core 8, which is on socket 1,
- et cetera.

The value OMP_PROC_BIND=close means that the assignment goes successively through the available places. The variable OMP_PROC_BIND can also be set to spread, which spreads the threads over the places. With
OMP_PLACES=cores
OMP_PROC_BIND=spread
you find that
- thread 0 goes to core 0, which is on socket 0,
- thread 1 goes to core 8, which is on socket 1,
- thread 2 goes to core 1, which is on socket 0,
- thread 3 goes to core 9, which is on socket 1,
- and so on, until thread 14 goes to core 7 on socket 0, and
- thread 15 goes to core 15, which is on socket 1.

So you see that OMP_PLACES=cores and OMP_PROC_BIND=spread very similar to OMP_PLACES=sockets. The difference is that the latter choice does not bind a thread to a specific core, so the operating system can move threads about, and it can put more than one thread on the same core, even if there is another core still unused.

The value OMP_PROC_BIND=master puts the threads in the same place as the master of the team. This is convenient if you create teams recursively. In that case you would use the dproc clause rather than the environment variable, set to spread for the initial team, and to master for the recursively created team.

25.1.2 Effects of thread binding

Let’s consider two example program. First we consider the program for computing $\pi$, which is purely compute-bound.

<table>
<thead>
<tr>
<th>#threads</th>
<th>close/cores</th>
<th>spread/sockets</th>
<th>spread/cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.359</td>
<td>0.354</td>
<td>0.353</td>
</tr>
<tr>
<td>2</td>
<td>0.177</td>
<td>0.177</td>
<td>0.177</td>
</tr>
<tr>
<td>4</td>
<td>0.088</td>
<td>0.088</td>
<td>0.088</td>
</tr>
<tr>
<td>6</td>
<td>0.059</td>
<td>0.059</td>
<td>0.059</td>
</tr>
<tr>
<td>8</td>
<td>0.044</td>
<td>0.044</td>
<td>0.044</td>
</tr>
<tr>
<td>12</td>
<td>0.029</td>
<td>0.045</td>
<td>0.029</td>
</tr>
<tr>
<td>16</td>
<td>0.022</td>
<td>0.050</td>
<td>0.022</td>
</tr>
</tbody>
</table>

We see pretty much perfect speedup for the OMP_PLACES=cores strategy; with OMP_PLACES=sockets we probably get occasional collisions where two threads wind up on the same core.

Next we take a program for computing the time evolution of the heat equation:

$$t = 0, 1, 2, \ldots \quad \forall i : x_i^{(t+1)} = 2x_i^{(t)} - x_{i-1}^{(t)} - x_{i+1}^{(t)}$$

This is a bandwidth-bound operation because the amount of computation per data item is low.
Again we see that `OMP_PLACES=sockets` gives worse performance for high core counts, probably because of threads winding up on the same core. The thing to observe in this example is that with 6 or 8 cores the `OMP_PROC_BIND=spread` strategy gives twice the performance of `OMP_PROC_BIND=close`.

The reason for this is that a single socket does not have enough bandwidth for all eight cores on the socket. Therefore, dividing the eight threads over two sockets gives each thread a higher available bandwidth than putting all threads on one socket.

### 25.1.3 Place definition

There are three predefined values for the `OMP_PLACES` variable: sockets, cores, threads. You have already seen the first two; the `threads` value becomes relevant on processors that have hardware threads. In that case, `OMP_PLACES=cores` does not tie a thread to a specific hardware thread, leading again to possible collisions as in the above example. Setting `OMP_PLACES=threads` ties each OpenMP thread to a specific hardware thread.

There is also a very general syntax for defining places that uses a `location:number:stride` syntax. Examples:

- `OMP_PLACES="{0:8:1}, {8:8:1}"` is equivalent to sockets on a two-socket design with eight cores per socket: it defines two places, each having eight consecutive cores. The threads are then places alternating between the two places, but not further specified inside the place.
- The setting cores is equivalent to `OMP_PLACES="{0}, {1}, {2}, ..., {15}"`
- On a four-socket design, the specification `OMP_PLACES="{0:4:8}:4:1"` states that the place 0, 8, 16, 24 needs to be repeated four times, with a stride of one. In other words, thread 0 winds up on core 0 of some socket, the thread 1 winds up on core 1 of some socket, et cetera.

### 25.1.4 Binding possibilities

Values for `OMP_PROC_BIND` are: false, true, master, close, spread.

- false: set no binding
25.1. OpenMP thread affinity control

- true: lock threads to a core
- master: collocate threads with the master thread
- close: place threads close to the master in the places list
- spread: spread out threads as much as possible

This effect can be made local by giving the `a proc` clause in the `parallel` directive.

A safe default setting is

```bash
export OMP_PROC_BIND=true
```

which prevents the operating system from migrating a thread. This prevents many scaling problems.


As an example, consider a code where two threads write to a shared location.

```c
// sharing.c
#pragma omp parallel
  { // not a parallel for: just a bunch of reps
    for (int j = 0; j < reps; j++) {
      #pragma omp for schedule(static,1)
      for (int i = 0; i < N; i++){
        #pragma omp atomic
        a++;
      }
    }

  }
```

For the full source of this example, see section 25.4.2

There is now a big difference in runtime depending on how close the threads are. We test this on a processor with both cores and hyperthreads. First we bind the OpenMP threads to the cores:

```bash
OMP_NUM_THREADS=2 OMP_PLACES=cores OMP_PROC_BIND=close ./sharing
run time = 4752.231836usec
sum = 80000000.0
```

Next we force the OpenMP threads to bind to hyperthreads inside one core:

```bash
OMP_PLACES=threads OMP_PROC_BIND=close ./sharing
run time = 941.970110usec
sum = 80000000.0
```

Of course in this example the inner loop is pretty much meaningless and parallelism does not speed up anything:

```bash
OMP_NUM_THREADS=1 OMP_PLACES=cores OMP_PROC_BIND=close ./sharing
run time = 806.669950usec
sum = 80000000.0
```

However, we see that the two-thread result is almost as fast, meaning that there is very little parallelization overhead.

Victor Eijkhout
25.2 First-touch

The affinity issue shows up in the first-touch phenomenon.

A little background knowledge. Memory is organized in memory pages, and what we think of as ‘addresses’ really virtual addresses, mapped to physical addresses, through a page table.

This means that data in your program can be anywhere in physical memory. In particular, on a dual socket node, the memory can be mapped to either of the sockets.

The next thing to know is that memory allocated with malloc and like routines is not immediately mapped; that only happens when data is written to it. In light of this, consider the following OpenMP code:

```c
#include <stdlib.h>

// heat.c
double *x = (double*) malloc(N*sizeof(double));
for (i=0; i<N; i++)
  x[i] = 0;

#pragma omp parallel for
for (i=0; i<N; i++)
  .... something with x[i] ...
```

Since the initialization loop is not parallel it is executed by the master thread, making all the memory associated with the socket of that thread. Subsequent access by the other socket will then access data from memory not attached to that socket.

Let’s consider an example. We make the initialization parallel subject to an option:

```c
// heat.c
#pragma omp parallel if (init>0)
{
  #pragma omp for
  for (int i=0; i<N; i++)
    y[i] = x[i] = 0.;
  x[0] = 0; x[N-1] = 1.;
}
```

If the initialization is not parallel, the array will be mapped to the socket of the master thread; if it is parallel, it may be mapped to different sockets, depending on where the threads run.

As a simple application we run a heat equation, which is parallel, though not embarrassingly so:

```c
for (int it=0; it<1000; it++) {
  #pragma omp parallel for
  for (int i=1; i<N-1; i++)
    y[i] = ( x[i-1]+x[i]+x[i+1] )/3.;
  #pragma omp parallel for
  for (int i=1; i<N-1; i++)
    x[i] = y[i];
}
```

On the TACC Frontera machine, with dual 28-core Intel Cascade Lake processors, we use the following settings:
25.2. First-touch

```
export OMP_PLACES=cores
export OMP_PROC_BIND=close
# no parallel initialization
make heat & OMP_NUM_THREADS=56 ./heat
# yes parallel initialization
make heat & OMP_NUM_THREADS=56 ./heat 1
```

This gives us a remarkable difference in runtime:

- Sequential init: avg=2.089, stddev=0.1083
- Parallel init: avg=1.006, stddev=0.0216

This large difference will be mitigated for algorithms with higher arithmetic intensity.

**Exercise 25.1.** How do the OpenMP dynamic schedules relate to this issue?

### 25.2.1 C++

The problem with realizing first-touch in C++ is that `std::vector` fills its allocation with default values. This makes

```
vector<double> x(N);
```

equivalent to the non-parallel allocation and initialization above.

Here is a solution.

**C++ note 7: Uninitialized containers.** We make a template for uninitialized types:

```
// heatalloc.cxx
template<typename T>
struct uninitialized {
    uninitialized() {};
    T val;
    constexpr operator T() const {return val;};
    double operator=( const T& v ) { val = v; return val; };
};
```

so that we can create vectors that behave normally:

```
vector<uninitialized<double>> x(N), y(N);

#pragma omp parallel for
for ( int i=0; i<N; i++)
    y[i] = x[i] = 0.;
x[0] = 0.; x[N-1] = 1.;
```

Running the code with the regular definition of a vector, and the above modification, reproduces the runtimes of the C variant above.

Another option is to wrap memory allocated with `new` in a `unique_ptr`:
25. OpenMP topic: Affinity

```cpp
// heatptr.cxx
unique_ptr<double[]> x(new double[N]);
unique_ptr<double[]> y(new double[N]);

#pragma omp parallel for
for (int i=0; i<N; i++) {
    y[i] = x[i] = 0.0;
}
x[0] = 0.; x[N-1] = 1.;
```

Note that this gives fairly elegant code, since square bracket indexing is overloaded for `unique_ptr`. The only disadvantage is that we cannot query the size of these arrays. Or do bound checking with `at`, but in high performance contexts that is usually not appropriate anyway.

25.2.2 Remarks

You could move pages with `move_pages`.

By regarding affinity, in effect you are adopting an SPMD style of programming. You could make this explicit by having each thread allocate its part of the arrays separately, and storing a private pointer as `threadprivate[18]`. However, this makes it impossible for threads to access each other’s parts of the distributed array, so this is only suitable for total data parallel or embarrassingly parallel applications.

25.3 Affinity control outside OpenMP

There are various utilities to control process and thread placement.

Process placement can be controlled on the Operating system level by `numactl` (the TACC utility `tacc_affinity` is a wrapper around this) on Linux (also `taskset`); Windows `start/affinity`.

Corresponding system calls: `pbintr` on Solaris, `sched_setaffinity` on Linux, `SetThreadAffinityMask` on Windows.

Corresponding environment variables: `SUNW_MP_PROCBIND` on Solaris, `KMP_AFFINITY` on Intel.

The Intel compiler has an environment variable for affinity control:

```
export KMP_AFFINITY=verbose,scatter
```

values: none,scatter,compact

For gcc:

```
export GOMP_CPU_AFFINITY=0,8,1,9
```

For the Sun compiler:

`SUNW_MP_PROCBIND`
25.4 Sources used in this chapter

25.4.1 Listing of code header

25.4.2 Listing of code examples/omp/c/sharing.c

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int i,j;
    int reps = 1000;
    int N = 8*10000;
    double start, stop, delta;
    double a;

    #pragma omp parallel
    a = 0;
    start = omp_get_wtime();
    #pragma omp parallel
    { // not a parallel for: just a bunch of reps
        for (int j = 0; j < reps; j++) {
            #pragma omp for schedule(static,1)
            for (int i = 0; i < N; i++) {
                #pragma omp atomic
                a++;
            }
        }
    }
    stop = omp_get_wtime();
    delta = ((double)(stop - start))/reps;
    printf("run time = %fusec\n", 1.0e6*delta);
    printf("sum = %.1f\n", a);
    return 0;
}
```
Chapter 26

OpenMP topic: Memory model

26.1 Thread synchronization

Let’s do a producer-consumer model\(^1\). This can be implemented with sections, where one section, the producer, sets a flag when data is available, and the other, the consumer, waits until the flag is set.

```c
#pragma omp parallel sections
{
    // the producer
    #pragma omp section
    {
        ... do some producing work ...
        flag = 1;
    }
    // the consumer
    #pragma omp section
    {
        while (flag==0) { }
        ... do some consuming work ...
    }
}
```

One reason this doesn’t work, is that the compiler will see that the flag is never used in the producing section, and that is never changed in the consuming section, so it may optimize these statements, to the point of optimizing them away.

The producer then needs to do:

```c
... do some producing work ...
#pragma omp flush
#pragma atomic write
flag = 1;
#pragma omp flush(flag)
```

and the consumer does:

---

1. This example is from Intel’s excellent OMP course by Tim Mattson
26.2 Data races

OpenMP, being based on shared memory, has a potential for race conditions. These happen when two threads access the same data item. The problem with race conditions is that programmer convenience runs counter to efficient execution. For this reason, OpenMP simply does not allow some things that would be desirable.

For a simple example:

```c
// race.c
#pragma omp parallel for shared(counter)
for (int i=0; i<count; i++)
    counter++;
printf("Counter should be %d, is %d\n", count, counter);
```

For the full source of this example, see section 26.4.2

The basic rule about multiple-thread access of a single data item is:

Any memory location that is written by one thread, cannot be read by another thread in the same parallel region, if no synchronization is done.

To start with that last clause: any workshare construct ends with an implicit barrier, so data written before that barrier can safely be read after it.
26.2.1 Dekker’s algorithm

A standard illustration of the weak memory model is *Dekker’s algorithm*. We model that in OpenMP as follows:

```c
// weak1.c
int a=0,b=0,r1,r2;
#pragma omp parallel sections shared(a, b, r1, r2)
{
    #pragma omp section
    {
        a = 1;
        r1 = b;
        tasks++;
    }
    #pragma omp section
    {
        b = 1;
        r2 = a;
        tasks++;
    }
}
```

Under any reasonable interpretation of parallel execution, the possible values for \( r_1, r_2 \) are 1, 1, 0, 1 or 1, 0. This is known as *sequential consistency*: the parallel outcome is consistent with a sequential execution that interleaves the parallel computations, respecting their local statement orderings. (See also HPSC-2.6.1.6.)

However, running this, we get a small number of cases where \( r_1 = r_2 = 0 \). There are two possible explanations:

1. The compiler is allowed to interchange the first and second statements, since there is no dependence between them; or
2. The thread is allowed to have a local copy of the variable that is not coherent with the value in memory.

We fix this by flushing both \( a, b \):

```c
// weak2.c
int a=0,b=0,r1,r2;
#pragma omp parallel sections shared(a, b, r1, r2)
{
    #pragma omp section
    {
        a = 1;
        #pragma omp flush (a,b)
        r1 = b;
        tasks++;
    }
    #pragma omp section
    {
        b = 1;
        #pragma omp flush (a,b)
        r2 = a;
        tasks++;
    }
}
```
26.3 Relaxed memory model

There is an implicit flush of all variables at the start and end of a parallel region.
There is a flush at each barrier, whether explicit or implicit, such as at the end of a work sharing.
At entry and exit of a critical section
When a lock is set or unset.
26.4 Sources used in this chapter

26.4.1 Listing of code header

26.4.2 Listing of code examples/omp/c/race.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <omp.h>

int main(int argc, char **argv) {
    int count = 100000;
    int counter = 0;

    #pragma omp parallel for shared(counter)
    for (int i=0; i<count; i++)
        counter++;
    printf("Counter should be %d, is %d\n", count, counter);

    return 0;
}
```
Chapter 27

OpenMP topic: SIMD processing

You can declare a loop to be executable with vector instructions with `simd`.

**Remark 29** Depending on your compiler, it may be necessary to give an extra option enabling SIMD:

- `-fopenmp-simd` for GCC / Clang, and
- `-qopenmp-simd` for ICC.

The `simd` pragma has the following clauses:

- `safelen($n$)`: limits the number of iterations in a SIMD chunk. Presumably useful if you combine `parallel` for `simd`.
- `linear`: lists variables that have a linear relation to the iteration parameter.
- `aligned`: specifies alignment of variables.

If your SIMD loop includes a function call, you can declare that the function can be turned into vector instructions with `declare simd`.

If a loop is both multi-threadable and vectorizable, you can combine directives as `pragma omp parallel for simd`.

Compilers can be made to report whether a loop was vectorized:

```
LOOP BEGIN at simdf.c(61,15)
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
LOOP END
```

with such options as `-Qvec-report=3` for the Intel compiler.

Performance improvements of these directives need not be immediately obvious. In cases where the operation is bandwidth-limited, using `simd` parallelism may give the same or worse performance as thread parallelism.

The following function can be vectorized:

```
// simdfunctions.c
#pragma omp declare simd
double cs(double x1,double x2,double y1,double y2) {
  double
```

535
27. OpenMP topic: SIMD processing

```c
inprod = x1*x2+y1*y2,
    xnorm = sqrt(x1*x1 + x2*x2),
    ynorm = sqrt(y1*y1 + y2*y2);
return inprod / (xnorm*ynorm);
}
#pragma omp declare simd uniform(x1,x2,y1,y2) linear(i)
double csa(double *x1,double *x2,double *y1,double *y2, int i) {
    double
        inprod  = x1[i]*x2[i]+y1[i]*y2[i],
        xnorm  = sqrt(x1[i]*x1[i] + x2[i]*x2[i]),
        ynorm  = sqrt(y1[i]*y1[i] + y2[i]*y2[i]);
    return inprod / (xnorm*ynorm);
}
```

For the full source of this example, see section 27.1.2

Compiling this the regular way

```
# parameter 1(x1): %xmm0
# parameter 2(x2): %xmm1
# parameter 3(y1): %xmm2
# parameter 4(y2): %xmm3

movaps  %xmm0, %xmm5 5 <- x1
movaps  %xmm2, %xmm4 4 <- y1
mulsd   %xmm1, %xmm5 5 <- 5 * x2 = x1 * x2
mulsd   %xmm3, %xmm4 4 <- 4 * y2 = y1 * y2
mulsd   %xmm0, %xmm0 0 <- 0 * 0 = x1 * x1
mulsd   %xmm1, %xmm1 1 <- 1 * 1 = x2 * x2
addsd  %xmm4, %xmm5 5 <- 5 + 4 = x1*x2 + y1*y2
mulsd   %xmm2, %xmm2 2 <- 2 * 2 = y1 * y1
mulsd   %xmm3, %xmm3 3 <- 3 * 3 = y2 * y2
addsd  %xmm1, %xmm0 0 <- 0 + 1 = x1*x1 + x2*x2
addsd  %xmm3, %xmm2 2 <- 2 + 3 = y1*y1 + y2*y2
sqrtsd %xmm0, %xmm0 0 <- sqrt(0) = sqrt( x1*x1 + x2*x2 )
sqrtsd %xmm2, %xmm2 2 <- sqrt(2) = sqrt( y1*y1 + y2*y2 )
```

which uses the scalar instruction `mulsd`: multiply scalar double precision.

With a `declare simd` directive:

```
movaps  %xmm0, %xmm7
movaps  %xmm2, %xmm4
mulpd   %xmm1, %xmm7
mulpd   %xmm3, %xmm4
```

which uses the vector instruction `mulpd`: multiply packed double precision, operating on 128-bit SSE2 registers.

Compiling for the Intel Knight's Landing gives more complicated code:

```
# parameter 1(x1): %xmm0
```

Parallel Computing – r428
# parameter 2(x2): %xmm1
# parameter 3(y1): %xmm2
# parameter 4(y2): %xmm3

vmulpd %xmm3, %xmm2, %xmm4 4 <- y1*y2
vmulpd %xmm1, %xmm1, %xmm5 5 <- x1*x2
vbroadcastsd .L_2i10floatpacket.0(%rip), %zmm21
movl $3, %eax set accumulator EAX
vbroadcastsd .L_2i10floatpacket.5(%rip), %zmm24
kmovw %eax, %k3 set mask k3
vmulpd %xmm3, %xmm3, %xmm6 6 <- y1*y1 (stall)
vfmadd213pd %xmm0, %xmm1, %xmm4 4 <- 4 + x1*x2 (no reuse!)
vfmadd213pd %xmm5, %xmm0, %xmm0 0 <- 0 + 0*5 = x1 + x1*(x1*x2)
vmovaps %zmm21, %zmm18 #25.26 c7
vmovaps %zmm0, %zmm3{%k3}{z} #25.26 c11
vmovaps %zmm2, %zmm2,%%k2{%k3} #24.29 c13
vpcmpgtq %zmm0, %zmm21, %k1{%k3} #25.26 c13
vfmadd213pd %xmm6, %xmm2, %xmm2 #25.26 c19
vpcmpgtq %zmm0, %zmm21, %k1{%k3} #25.26 c17
vrsqrt28pd %zmm3, %zmm16{%k3}{z} #25.26 c19
vpxorq %zmm4, %zmm4, %zmm26{%k3} #25.26 c19
vrsqrt28pd %zmm7, %zmm20{%k3}{z} #25.26 c21
vmulpd {rn-sae}, %zmm3, %zmm16, %zmm19{%k3}{z} #25.26 c27 stall 2
vfmadd213pd {rn-sae}, %zmm17, %zmm19, %zmm18{%k3} #25.26 c35
vfmadd213pd {rn-sae}, %zmm22, %zmm23, %zmm21{%k3} #25.26 c35
vfmadd213pd {rn-sae}, %zmm19, %zmm18, %zmm19{%k3} #25.26 c41
vfmadd213pd {rn-sae}, %zmm23, %zmm21, %zmm23{%k3} #25.26 c45 stall 1
vfmadd213pd {rn-sae}, %zmm19, %zmm19, %zmm3{%k3} #25.26 c47
vfmadd213pd {rn-sae}, %zmm22, %zmm22, %zmm21{%k3} #25.26 c51 stall 1
vfmadd213pd {rn-sae}, %zmm23, %zmm23, %zmm7{%k3} #25.26 c53
vfmadd213pd %zmm19, %zmm18, %zmm3{%k3} #25.26 c57 stall 1
vfmadd213pd %zmm23, %zmm21, %zmm7{%k3} #25.26 c59
vscalefpd .L_2i10floatpacket.3(%rip){1to8}, %zmm3, %zmm3{%k1} #25.26 c63 stall 1
vscalefpd .L_2i10floatpacket.3(%rip){1to8}, %zmm7, %zmm7{%k2} #25.26 c65
vfixupimmpd $112, .L_2i10floatpacket.4(%rip){1to8}, %zmm0, %zmm3{%k3} #25.26 c65
vfixupimmpd $112, .L_2i10floatpacket.4(%rip){1to8}, %zmm2, %zmm7{%k3} #25.26 c67
27. OpenMP topic: SIMD processing

```c
#pragma omp declare simd uniform(op1) linear(k) notinbranch
double SqrtMul(double *op1, double op2, int k) {
    return (sqrt(op1[k]) * sqrt(op2));
}
```
27.1 Sources used in this chapter

27.1.1 Listing of code header

27.1.2 Listing of code code/omp/c/simd/tools.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#pragma omp declare simd
double cs(double x1,double x2,double y1,double y2) {
    double
        inprod = x1*x2+y1*y2,
        xnorm = sqrt(x1*x1 + x2*x2),
        ynorm = sqrt(y1*y1 + y2*y2);
    return inprod / (xnorm*ynorm);
}

#pragma omp declare simd uniform(x1,x2,y1,y2) linear(i)
double csa(double *x1,double *x2,double *y1,double *y2, int i) {
    double
        inprod = x1[i]*x2[i]+y1[i]*y2[i],
        xnorm = sqrt(x1[i]*x1[i] + x2[i]*x2[i]),
        ynorm = sqrt(y1[i]*y1[i] + y2[i]*y2[i]);
    return inprod / (xnorm*ynorm);
}
```

Victor Eijkhout
Chapter 28

OpenMP topic: Offloading

This chapter explains the mechanisms for offloading work to a Graphics Processing Unit (GPU). The memory of a processor and that of an attached GPU are not coherent: there are separate memory spaces and writing data in one is not automatically reflected in the other. OpenMP transfers data (or maps it) when you enter an target construct.

```
#pragma omp target
{
    // do stuff on the GPU
}
```

You can test whether the target region is indeed executed on a device with omp_is_initial_device:

```
#pragma omp target
if (omp_is_initial_device()) printf("Offloading failed\n");
```

28.0.1 Targets and tasks

The target clause causes OpenMP to create a target task. This is a task running on the host, dedicated to managing the offloaded region.

The target region is executed by a new initial task. This is distinct from the initial task that executes the main program.

The task that created the target task is called the generating task.

By default, the generating task is blocked while the task on the device is running, but adding the targetnowait clause makes it asynchronous. This requires a taskwait directive to synchronize host and device.

28.1 Data on the device

- Scalars are treated as firstprivate, that is, they are copied in but not out.
- Stack arrays tofrom.
- Heap arrays are not mapped by default.
For explicit mapping with `map`:

```c
#pragma omp target map(...) {
    // do stuff on the GPU
}
```

The following map options exist:

- `map(to: x,y,z)` copy from host to device when entering the target region.
- `map(from: x,y,z)` copy from device to host when exiting the target region.
- `map(tofrom: x,y,z)` is equivalent to combining the previous two.
- `map(allo: x,y,z)` allocates data on the device.

*Fortran note 17: Array sizes in map clause.* If the compiler can deduce the array bounds and size, it is not necessary to specify them in the 'map' clause.

Data transfer to a device is probably slow, so mapping the data at the start of an offloaded section of code is probably not the best idea. Additionally, in many cases data will stay resident on the device throughout several iterations of, for instance, a time-stepping Partial Differential Equation (PDE) solver. For such reasons, it is possible to move data onto, and off from, the device explicitly, using the `enter data` and `exit data` directives.

```
#pragma omp target enter data map(to: x,y)
#pragma omp target {
    // do something
}
#pragma omp target exit data map(from: x,y)
```

Also update `to` (synchronize data from host to device), `update from` (synchronize data to host from device).

### 28.2 Execution on the device

For parallel execution of a loop on the device use the `teams` clause:

```
#pragma omp target teams distribute parallel do
```

On GPU devices and the like, there is a structure to threads:

- threads are grouped in `teams`, and they can be synchronized only within these teams;
- teams are groups in `leagues`, and no synchronization between leagues is possible inside a `target` region.

The combination `teams distribute` splits the iteration space over teams. By default a static schedule is used, but the option `dist_schedule` can be used to specify a different one. However, this combination only gives the chunk of space to the master thread in each team. Next we need `parallel for` or `parallel do` to spread the chunk over the threads in the team.
28. OpenMP topic: Offloading

28.3 Sources used in this chapter

28.3.1 Listing of code header
Chapter 29

OpenMP remaining topics

29.1 Runtime functions and internal control variables

OpenMP has a number of settings that can be set through environment variables, and both queried and set through library routines. These settings are called Internal Control Variables (ICVs): an OpenMP implementation behaves as if there is an internal variable storing this setting.

The runtime functions are:

- Counting threads and cores: `omp_set_num_threads`, `omp_get_num_threads`, `omp_get_max_threads`, `omp_get_num_procs`; see section 17.5.
- Querying the current thread: `omp_get_thread_num`, `omp_in_parallel`
- `omp_set_dynamic`
- `omp_get_dynamic`
- `omp_set_nested`
- `omp_get_nested`
- `omp_get_wtime`
- `omp_get_wtick`
- `omp_set_schedule`
- `omp_get_schedule`
- `omp_set_max_active_levels`
- `omp_get_max_active_levels`
- `omp_get_thread_limit`
- `omp_get_level`
- `omp_get_active_level`
- `omp_get_ancestor_thread_num`
- `omp_get_team_size`

Here are the OpenMP environment variables:

- `OMP_CANCELLATION` Set whether cancellation is activated; see section 18.3.
- `OMP_DISPLAY_ENV` Show OpenMP version and environment variables
- `OMP_DEFAULT_DEVICE` Set the device used in target regions
- `OMP_DYNAMIC` Dynamic adjustment of threads
- `OMP_MAX_ACTIVE_LEVELS` Set the maximum number of nested parallel regions; section 18.2.
29. OpenMP remaining topics

- **OMP_MAX_TASK_PRIORITY** Set the maximum task priority value; section 24.5.2.
- **OMP_NESTED** Nested parallel regions
- **OMP_NUM_THREADS** Specifies the number of threads to use
- **OMP_PROC_BIND** Whether threads may be moved between CPUs; section 25.1.
- **OMP_PLACES** Specifies on which CPUs the threads should be placed; section 25.1.
- **OMP_STACKSIZE** Set default thread stack size
- **OMP_SCHEDULE** How threads are scheduled
- **OMP_THREAD_LIMIT** Set the maximum number of threads
- **OMP_WAIT_POLICY** How waiting threads are handled; ICV `wait-policy-var`. Values: ACTIVE for keeping threads spinning, PASSIVE for possibly yielding the processor when threads are waiting.

There are 4 ICVs that behave as if each thread has its own copy of them. The default is implementation-defined unless otherwise noted.

- It may be possible to adjust dynamically the number of threads for a parallel region. Variable: **OMP_DYNAMIC**; routines: `omp_set_dynamic`, `omp_get_dynamic`.
- If a code contains nested parallel regions, the inner regions may create new teams, or they may be executed by the single thread that encounters them. Variable: **OMP_NESTED**; routines `omp_set_nested`, `omp_get_nested`. Allowed values are TRUE and FALSE; the default is false.
- The number of threads used for an encountered parallel region can be controlled. Variable: **OMP_NUM_THREADS**; routines `omp_set_num_threads`, `omp_get_max_threads`.
- The schedule for a parallel loop can be set. Variable: **OMP_SCHEDULE**; routines `omp_set_schedule`, `omp_get_schedule`.

Nonobvious syntax:

```bash
export OMP_SCHEDULE="static,100"
```

Other settings:

- `omp_get_num_threads`: query the number of threads active at the current place in the code; this can be lower than what was set with `omp_set_num_threads`. For a meaningful answer, this should be done in a parallel region.
- `omp_get_thread_num`
- `omp_in_parallel`: test if you are in a parallel region (see for instance section 18).
- `omp_get_num_procs`: query the physical number of cores available.

Other environment variables:

- **OMP_STACKSIZE** controls the amount of space that is allocated as per-thread stack; the space for private variables.
- **OMP_WAIT_POLICY** determines the behavior of threads that wait, for instance for critical section:
  - ACTIVE puts the thread in a spin-lock, where it actively checks whether it can continue;
  - PASSIVE puts the thread to sleep until the Operating System (OS) wakes it up.

The 'active' strategy uses CPU while the thread is waiting; on the other hand, activating it after the wait is instantaneous. With the 'passive' strategy, the thread does not use any CPU while waiting, but activating it again is expensive. Thus, the passive strategy only makes sense if threads will be waiting for a (relatively) long time.
• **OMP_PROC_BIND** with values TRUE and FALSE can bind threads to a processor. On the one hand, doing so can minimize data movement; on the other hand, it may increase load imbalance.

## 29.2 Timing

OpenMP has a wall clock timer routine `omp_get_wtime`

\[
\text{double omp_get_wtime(void);} \quad \text{double omp_get_wtick.}
\]

The starting point is arbitrary and is different for each program run; however, in one run it is identical for all threads. This timer has a resolution given by `omp_get_wtick`.

**Exercise 29.1.** Use the timing routines to demonstrate speedup from using multiple threads.

- Write a code segment that takes a measurable amount of time, that is, it should take a multiple of the tick time.
- Write a parallel loop and measure the speedup. You can for instance do this

```c
for (int use_threads=1; use_threads<=nthreads; use_threads++) {
    #pragma omp parallel for num_threads(use_threads)
    for (int i=0; i<nthreads; i++) {
        ....
    }
    if (use_threads==1)
        time1 = tend-tstart;
    else // compute speedup
        ....
}
```

- In order to prevent the compiler from optimizing your loop away, let the body compute a result and use a reduction to preserve these results.

## 29.3 Thread safety

With OpenMP it is relatively easy to take existing code and make it parallel by introducing parallel sections. If you’re careful to declare the appropriate variables shared and private, this may work fine. However, your code may include calls to library routines that include a race condition; such code is said not to be thread-safe.

For example a routine

```c
static int isave;
int next_one() {
    int i = isave;
    isave += 1;
    return i;
}
```

... for ( .... ) {
    int ivalue = next_one();
}
has a clear race condition, as the iterations of the loop may get different next_one values, as they are supposed to, or not. This can be solved by using an critical pragma for the next_one call; another solution is to use an threadprivate declaration for isave. This is for instance the right solution if the next_one routine implements a random number generator.

29.4 Performance and tuning

The performance of an OpenMP code can be influenced by the following.

**Amdahl effects** Your code needs to have enough parts that are parallel (see HPSC-2.2.3). Sequential parts may be sped up by having them executed redundantly on each thread, since that keeps data locally.

**Dynamism** Creating a thread team takes time. In practice, a team is not created and deleted for each parallel region, but creating teams of different sizes, or recurse size thread creation, may introduce overhead.

**Load imbalance** Even if your program is parallel, you need to worry about load balance. In the case of a parallel loop you can set the schedule clause to dynamic, which evens out the work, but may cause increased communication.

**Communication** Cache coherence causes communication. Threads should, as much as possible, refer to their own data.

- Threads are likely to read from each other’s data. That is largely unavoidable.
- Threads writing to each other’s data should be avoided: it may require synchronization, and it causes coherence traffic.
- If threads can migrate, data that was local at one time is no longer local after migration.
- Reading data from one socket that was allocated on another socket is inefficient; see section 25.2.

**Affinity** Both data and execution threads can be bound to a specific locale to some extent. Using local data is more efficient than remote data, so you want to use local data, and minimize the extent to which data or execution can move.

- See the above points about phenomena that cause communication.
- Section 25.1.1 describes how you can specify the binding of threads to places. There can, but does not need, to be an effect on affinity. For instance, if an OpenMP thread can migrate between hardware threads, cached data will stay local. Leaving an OpenMP thread completely free to migrate can be advantageous for load balancing, but you should only do that if data affinity is of lesser importance.
- Static loop schedules have a higher chance of using data that has affinity with the place of execution, but they are worse for load balancing. On the other hand, the nowait clause can alleviate some of the problems with static loop schedules.

**Binding** You can choose to put OpenMP threads close together or to spread them apart. Having them close together makes sense if they use lots of shared data. Spreading them apart may increase bandwidth. (See the examples in section 25.1.2.)

**Synchronization** Barriers are a form of synchronization. They are expensive by themselves, and they expose load imbalance. Implicit barriers happen at the end of worksharing constructs; they can be removed with nowait.
Critical sections imply a loss of parallelism, but they are also slow as they are realized through operating system functions. These are often quite costly, taking many thousands of cycles. Critical sections should be used only if the parallel work far outweighs it.

29.5 Accelerators

In OpenMP-4.0 there is support for offloading work to an accelerator or co-processor:

```c
#pragma omp target [clauses]
```

with clauses such as

- data: place data
- update: make data consistent between host and device

29.6 Tools interface

The OpenMP-5.0 defines a tools interface. This means that routines can be defined that get called by the OpenMP runtime. For instance, the following example defines callback that are evaluated when OpenMP is initialized and finalized, thereby giving the runtime for the application.

```c
int ompt_initialize(ompt_function_lookup_t lookup, int initial_device_num,
                    ompt_data_t *tool_data) {
    printf("libomp init time: %f\n",
            omp_get_wtime() - *(double *)(tool_data->ptr));
    *(double *)(tool_data->ptr) = omp_get_wtime();
    return 1; // success: activates tool
}

void ompt_finalize(ompt_data_t *tool_data) {
    printf("application runtime: %f\n",
            omp_get_wtime() - *(double *)(tool_data->ptr));
}

ompt_start_tool_result_t *ompt_start_tool(unsigned int omp_version,
                                          const char *runtime_version) {
    static double time = 0; // static definition needs constant assignment
    time = omp_get_wtime();
    static ompt_start_tool_result_t ompt_start_tool_result = {
        &ompt_initialize, &ompt_finalize, {.ptr = &time}};
    return &ompt_start_tool_result; // success: registers tool
}
```

(Example courtesy of https://git.rwth-aachen.de/OpenMPTools/OMPT-Examples.)

Victor Eijkhout
29. OpenMP remaining topics

29.7 OpenMP standards

Here is the correspondence between the value of OpenMP versions (given by the \_OPENMP macro) and the standard versions:

- 201511 OpenMP-4.5,
- 201611 Technical report 4: information about the OpenMP-5.0 but not yet mandated.
- 201811 OpenMP-5.0
- 202011 OpenMP-5.1,
- 202111 OpenMP-5.2.
29.8 Sources used in this chapter

29.8.1 Listing of code header
Chapter 30

OpenMP Review

30.1 Concepts review

30.1.1 Basic concepts
- process / thread / thread team
- threads / cores / tasks
- directives / library functions / environment variables

30.1.2 Parallel regions
execution by a team

30.1.3 Work sharing
- loop / sections / single / workshare
- implied barrier
- loop scheduling, reduction
- sections
- single vs master
- (F) workshare

30.1.4 Data scope
- shared vs private, C vs F
- loop variables and reduction variables
- default declaration
- firstprivate, lastprivate

30.1.5 Synchronization
- barriers, implied and explicit
- nowait
- critical sections
- locks, difference with critical

30.1.6 Tasks
- generation vs execution
- dependencies
30.2 Review questions

30.2.1 Directives

What do the following program output?

```c
int main() {
    printf("procs %d\n", omp_get_num_procs());
    printf("threads %d\n", omp_get_num_threads());
    printf("num %d\n", omp_get_thread_num());
    return 0;
}
```

```c
int main() {
    #pragma omp parallel
    {
        printf("procs %d\n", omp_get_num_procs());
        printf("threads %d\n", omp_get_num_threads());
        printf("num %d\n", omp_get_thread_num());
    }
    return 0;
}
```

Program main
use omp_lib
print *,"Procs: ",&
omp_get_num_procs()
print *,"Threads: ",&
omp_get_num_threads()
print *,"Num: ",&
omp_get_thread_num()
End Program

Program main
use omp_lib
!$OMP parallel
    print *,"Procs: ",&
    omp_get_num_procs()
    print *,"Threads: ",&
    omp_get_num_threads()
    print *,"Num: ",&
    omp_get_thread_num()
!$OMP end parallel
End Program
**30.2.2 Parallelism**

Can the following loops be parallelized? If so, how? (Assume that all arrays are already filled in, and that there are no out-of-bounds errors.)

```
// variant #1
for (i=0; i<N; i++) {
    x[i] = a[i]+b[i+1];
    a[i] = 2*x[i] + c[i+1];
}

// variant #2
for (i=0; i<N; i++) {
    x[i] = a[i]+b[i+1];
    a[i] = 2*x[i+1] + c[i+1];
}

! variant #1
do i=1,N
    x(i) = a(i)+b(i+1)
    a(i) = 2*x(i) + c(i+1)
end do

! variant #2
do i=1,N
    x(i) = a(i)+b(i+1)
    a(i) = 2*x(i+1) + c(i+1)
end do
```

```
// variant #3
for (i=1; i<N; i++) {
    x[i] = a[i]+b[i+1];
    a[i] = 2*x[i-1] + c[i+1];
}

// variant #4
for (i=1; i<N; i++) {
    x[i] = a[i]+b[i+1];
    a[i+1] = 2*x[i-1] + c[i+1];
}

! variant #3
do i=2,N
    x(i) = a(i)+b(i+1)
    a(i) = 2*x(i-1) + c(i+1)
end do

! variant #4
do i=2,N
    x(i) = a(i)+b(i+1)
    a(i+1) = 2*x(i-1) + c(i+1)
end do
```
30.2.3 Data and synchronization

30.2.3.1

What is the output of the following fragments? Assume that there are four threads.

```c
// variant #1
ing t nt;
#pragma omp parallel
{ nt = omp_get_thread_num();
printf("thread number: %d\n",nt);
}
```

```c
// variant #2
int nt;
#pragma omp parallel private(nt)
{ nt = omp_get_thread_num();
printf("thread number: %d\n",nt);
}
```

```c
// variant #3
int nt;
#pragma omp parallel
#pragma omp single
{ nt = omp_get_thread_num();
printf("thread number: %d\n",nt);
}
```

```c
!$OMP single
integer nt
!$OMP parallel
 nt = omp_get_thread_num()
 print *,"thread number: ",nt
!$OMP end parallel
```

```c
!$OMP parallel private(nt)
 nt = omp_get_thread_num()
 print *,"thread number: ",nt
!$OMP end parallel
```

```c
!$OMP parallel
 nt = omp_get_thread_num()
 print *,"thread number: ",nt
!$OMP end parallel
```

---

Victor Eijkhout
30. OpenMP Review

30.2.3.2

The following is an attempt to parallelize a serial code. Assume that all variables and arrays are defined. What errors and potential problems do you see in this code? How would you fix them?

```
! variant #4
integer nt
!$OMP parallel
!$OMP master
   nt = omp_get_thread_num()
   print *, "thread number: ", nt
!$OMP end master
!$OMP end parallel

! variant #5
integer nt
!$OMP parallel
!$OMP critical
   nt = omp_get_thread_num()
   print *, "thread number: ", nt
!$OMP end critical
!$OMP end parallel

#pragma omp parallel
{
    x = f();
#pragma omp for
    for (i=0; i<N; i++)
        y[i] = g(x,i);
    z = h(y);
}
```

```
!$OMP parallel
x = f()
!$OMP do
   do i=1,N
      y(i) = g(x,i)
   end do
!$OMP end do
!$OMP end parallel
z = h(y)
!$OMP end parallel
```
30.2.3.3
Assume two threads. What does the following program output?

```c
int a;
#pragma omp parallel private(a) {
    ...
    a = 0;
    #pragma omp for
    for (int i = 0; i < 10; i++)
    {
        #pragma omp atomic
        a++;
    }
    #pragma omp single
    printf("a=\%e\n",a);
}
```

30.2.4  Reductions
30.2.4.1
Is the following code correct? Is it efficient? If not, can you improve it?

```c
#pragma omp parallel shared(r)
{
    int x;
    x = f(omp_get_thread_num());
    #pragma omp critical
    r += f(x);
}
```

30.2.4.2
Compare two fragments:

```c
// variant 1
#pragma omp parallel reduction(+:s)
#pragma omp for
    for (i=0; i<N; i++)
    s += f(i);

// variant 2
#pragma omp parallel reduction(+:s)
#pragma omp for reduction(+:s)
    for (i=0; i<N; i++)
    s += f(i);
```

```c
! variant 1
!$OMP parallel reduction(+:s)
!$OMP do
    do i=1,N
    s += f(i);
!$OMP end do

!$OMP end parallel
```
Do they compute the same thing?
30.2.5 Barriers

Are the following two code fragments well defined?

```c
#pragma omp parallel
{
#pragma omp for
for (mytid=0; mytid<nthreads; mytid++)
x[mytid] = some_calculation();
#pragma omp for
for (mytid=0; mytid<nthreads-1; mytid++)
y[mytid] = x[mytid]+x[mytid+1];
}
```

```c
#pragma omp parallel
{
#pragma omp for
for (mytid=0; mytid<nthreads-1; mytid++)
nowait
y[mytid] = x[mytid]+x[mytid+1];
}
```

30.2.6 Data scope

The following program is supposed to initialize as many rows of the array as there are threads.

```c
int main() {
    int i, icount, iarray[100][100];
    icount = -1;
    #pragma omp parallel private(i)
    {
        #pragma omp critical
        {
            icount++;
            for (i=0; i<100; i++)
                iarray[icount][i] = 1;
        }
        return 0;
    }
}
```

Describe the behavior of the program, with argumentation,

• as given;
• if you add a clause `private(icount)` to the `parallel` directive;
• if you add a clause `firstprivate(icount)`.

What do you think of this solution:

```c
#pragma omp parallel private(i)
    shared(icount)
{
#pragma omp critical
    {
        icount++;
        for (i=0; i<100; i++)
            iarray[icount][i] = 1;
    }
    return 0;
}
```

Program main

```c
integer :: i, icount, iarray(100,100)
icount = 0
$OMP parallel private(i)
$OMP critical
icount = icount + 1
$OMP end critical
do i=1,100
    iarray(icount,i) = 1
end do
$OMP end parallel
End program
```
30. OpenMP Review

```c
!$OMP parallel private(i)
    $\rightarrow$ shared(icount)
!$OMP critical
    icount = icount + 1
do i = 1, 100
    iarray(icount, i) = 1
end do
!$OMP critical
!$OMP end parallel
```

### 30.2.7 Tasks
Fix two things in the following example:

```c
#pragma omp parallel
#pragma omp single
{
    int x, y, z;
    #pragma omp task
    x = f();
    #pragma omp task
    y = g();
    #pragma omp task
    z = h();
    printf("sum = \%d\n", x+y+z);
}
```

### 30.2.8 Scheduling
Compare these two fragments. Do they compute the same result? What can you say about their efficiency?

```c
#pragma omp parallel
#pragma omp single
{
    for (i = 0; i < N; i++) {
        #pragma omp task
        x[i] = f(i)
    }
    #pragma omp taskwait
}
```

```
#pragma omp parallel
#pragma omp for schedule(dynamic)
{
    for (i = 0; i < N; i++) {
        x[i] = f(i)
    }
}
```

How would you make the second loop more efficient? Can you do something similar for the first loop?
30.3 Sources used in this chapter

30.3.1 Listing of code header
Chapter 31

OpenMP Examples

31.1 N-body problems

So-called *N-body problems* come up with we describe the interactions between a, probably large, number of entities under a force such as gravity. Examples are molecular dynamics and star clusters.

While clever algorithms exist that take into account the decay of the force over distance, we here consider the naive algorithm that explicitly computes all interactions.

A particle has *x*, *y* coordinates and a mass *c*. For two particles \((x_1, y_1, c_1), (x_2, y_2, c_2)\) the force on particle 1 from particle 2 is:

\[
F_{12} = \frac{c_1 \cdot c_2}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \cdot \hat{r}_{12}
\]

where \(\hat{r}_{12}\) is the unit vector pointing from particle 2 to 1. With *n* particles, each particle *i* feels a force

\[
\vec{F}_i = \sum_{j\neq i} \vec{F}_{ij}.
\]

Let’s start with a couple of building blocks.

```c
// molecularstruct.c
struct point{ double x,y; double c; }; 
struct force{ double x,y; double f; };

/* Force on p1 from p2 */
struct force force_calc( struct point p1,struct point p2 ) {
  double dx = p2.x - p1.x, dy = p2.y - p1.y;
  double f = p1.c * p2.c / sqrt( dx*dx + dy*dy );
  struct force exert = {dx,dy,f};
  return exert;
}
```

Force accumulation:

```c
void add_force( struct force *f,struct force g ) {
  f->x += g.x; f->y += g.y; f->f += g.f;
}
void sub_force( struct force *f,struct force g ) {
```
### 31.1. N-body problems

\[
\begin{align*}
&f\to x := g.x; \quad f\to y := g.y; \quad f\to f += g.f;
\end{align*}
\]

In C++ we can have a class with an addition operator and such:

```cpp
// molecular.cxx
class force {
private:
    double _x{0.}, _y{0.}; double _f{0.};
public:
    force() {};
    force(double x, double y, double f)
    : _x(x), _y(y), _f(f) {};

    force operator+( const force& g ) {
        return { _x+g._x, _y+g._y, _f+g._f };}
};
```

For reference, this is the sequential code:

```cpp
for (int ip=0; ip<N; ip++) {
    for (int jp=ip+1; jp<N; jp++) {
        struct force f = force_calc(points[ip],points[jp]);
        add_force( forces[ip], f );
        sub_force( forces[jp], f );
    }
}
```

Here \( \vec{F}_{ij} \) is only computed for \( j > i \), and then added to both \( \vec{F}_i \) and \( \vec{F}_j \).

In C++ we use the overloaded operators:

```cpp
for (int ip=0; ip<N; ip++) {
    for (int jp=ip+1; jp<N; jp++) {
        force f = points[ip].force_calc(points[jp]);
        forces[ip] += f;
        forces[jp] -= f;
    }
}
```

**Exercise 31.1.** Argue that both the outer loop and the inner are not directly parallelizable.

We will now explore a number of different strategies for parallelization. All tests are done on the TACC Frontera cluster, which has dual-socket Intel Cascade Lake nodes, with a total of 56 cores. Our code uses 10 thousand particles, and each interaction evaluation is repeated 10 times to eliminate cache loading effects.

### 31.1.1 Solution 1: no conflicting writes

In our first attempt at an efficient parallel code, we compute the full \( N^2 \) interactions. One solution would be to compute the \( \vec{F}_{ij} \) interactions for all \( i, j \), so that there are no conflicting writes.
for (int ip=0; ip<N; ip++) {
    struct force sumforce;
    sumforce.x=0.; sumforce.y=0.; sumforce.f=0.;
    #pragma omp parallel for reduction(+:sumforce)
    for (int jp=0; jp<N; jp++) {
        if (ip==jp) continue;
        struct force f = force_calc(points[ip],points[jp]);
        sumforce.x += f.x; sumforce.y += f.y; sumforce.f += f.f;
    }
    // end parallel jp loop
    add_force( forces+ip, sumforce );
} // end ip loop

In C++ we use the fact that we can reduce on any class that has an addition operator:

for (int ip=0; ip<N; ip++) {
    force sumforce;
    #pragma omp parallel for reduction(+:sumforce)
    for (int jp=0; jp<N; jp++) {
        if (ip==jp) continue;
        force f = points[ip].force_calc(points[jp]);
        sumforce += f;
    }
    // end parallel jp loop
    forces[ip] += sumforce;
} // end ip loop

This increases the scalar work by a factor of two, but surprisingly, on a single thread the run time improves: we measure a speedup of 6.51 over the supposedly ’optimal’ code.

Exercise 31.2. What would be an explanation?

However, increasing the number of threads has limited benefits for this strategy. Figure 31.1 shows that the speedup is not only sublinear: it actually decreases with increasing core count.

Exercise 31.3. What would be an explanation?

31.1.2 Solution 2: using atomics

Next we try to parallelize the outer loop.

    #pragma omp parallel for schedule(guided,4)
    for (int ip=0; ip<N; ip++) {
        for (int jp=ip+1; jp<N; jp++) {
            struct force f = force_calc(points[ip],points[jp]);
            add_force( forces+ip,f );
            sub_force( forces+jp,f );
        }
    }

To deal with the conflicting jp writes, we make the writes atomic:

    void sub_force( struct force *f,struct force g ) {
    #pragma omp atomic
    f->x -= g.x;
    #pragma omp atomic

74x726
31.1. N-body problems

![Graph showing Speedup of reduction variant over sequential]

Figure 31.1: Speedup of reduction variant over sequential

```c
f->y -= g.y;
#pragma omp atomic
f->f += g.f;
```

This works fairly well, as figure 31.2 shows.

### 31.1.3 Solution 3: all interactions atomic

But if we decide to use atomic updates, we can take the full square loop, collapse the two loops, and make every write atomic.

```c
#pragma omp parallel for collapse(2)
for (int ip=0; ip<N; ip++) {
    for (int jp=0; jp<N; jp++) {
        if (ip==jp) continue;
        struct force f = force_calc(points[ip],points[jp]);
        add_force(forces+ip, f);
    } // end parallel jp loop
} // end ip loop
```
Figure 31.2: Speedup of triangular loop with atomic update

Figure 31.3 shows that this is pretty close to perfect.

Everything in one plot in figure 31.4.

### 31.2 Tree traversal

OpenMP tasks are a great way of handling trees.

In *post-order tree traversal* you visit the subtrees before visiting the root. This is the traversal that you use to find summary information about a tree, for instance the sum of all nodes, and the sums of nodes of all subtrees:

```plaintext
for all children c do
    compute the sum $s_c$

$s \leftarrow \sum_c s_c$
```

Another example is matrix factorization:

$$S = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}$$
where the two inverses $A_{11}^{-1}, A_{22}^{-1}$ can be computed independently and recursively.

### 31.3  Depth-first search

In this section we look at the ‘eight queens’ problem, as an example of Depth First Search (DFS): is it possible to put eight queens on a chess board so that none of them threaten each other? With DFS, the search space of possibilities is organized as a tree – each partial solution leads to several possibilities for the next steps – which is traversed in a particular manner: a chain of possibilities is extended as far as feasible, after which the search backtracks to the next chain.

The sequential implementation is easy enough. The main program fires off:

```cpp
placement initial; initial.fill(empty);
auto solution = place_queen(0, initial);
```
where I hope you can take the details on trust.

The recursive call then has this structure:

```cpp
optional<placement> place_queen(int iqueen, const placement& current) {
    for (int col=0; col<N; col++) {
        placement next = current;
        next.at(iqueen) = col;
        if (feasible(next)) {
            if (iqueen==N-1)
                return next;
            auto attempt = place_queen(iqueen+1, next);
            if (attempt.has_value())
                return attempt;
        } // end if(feasible)
    }
    return {};
};
```

(This uses the C++17 `optional` header.) At each `iqueen` level we
- go through a loop of all column positions;
filter out positions that are not feasible;
• report success if this was the last level; or
• recursively continue the next level otherwise.

This problem seems a prime candidate for OpenMP tasks, so we start with the usual idiom for the main program:

```cpp
placement initial; initial.fill(empty);
optional<placement> eightqueens;
#pragma omp parallel
#pragma omp single
eightqueens = place_queen(0,initial);
```

We create a task for each column, and since they are in a loop we use `taskgroup` rather than `taskwait`.

```cpp
#pragma omp taskgroup
for (int col=0; col<N; col++) {
    placement next = current;
    next.at(iqueen) = col;
    #pragma omp task firstprivate(next)
    if (feasible(next)) {
        // stuff
    } // end if(feasible)
}
```

However, the sequential program had `return` and `break` statements in the loop, which is not allowed in workshare constructs such as `taskgroup`. Therefore we introduce a return variable, declared as shared:

```cpp
// queens0.cxx
optional<placement> result = {};
#pragma omp taskgroup
for (int col=0; col<N; col++) {
    placement next = current;
    next.at(iqueen) = col;
    #pragma omp task firstprivate(next) shared(result)
    if (feasible(next)) {
        if (iqueen==N-1) {
            result = next;
        } else { // do next level
            auto attempt = place_queen(iqueen+1,next);
            if (attempt.has_value()) {
                result = attempt;
            }
        }
    } // end if(iqueen==N-1)
    } // end if(feasible)

return result;
```

So that was easy, this computes the right solution, and it uses OpenMP tasks. Done?

Actually this runs very slowly because, now that we’ve dispensed with all early breaks from the loop, we in effect traverse the whole search tree. (It’s not quite breadth-first, though.) Figure 31.5 shows this for \(N = 12\) with the Intel compiler (version 2019) in the left panel, and the GNU compiler (version 9.1) in the
middle. In both cases, the blue bars give the result for the code with only the \texttt{taskgroup} directive, with time plotted as function of core count.

We see that, for the Intel compiler, running time indeed goes down with core count. So, while we compute too much (the whole search space), at least parallelization helps. With a number of threads greater than the problem size, the benefit of parallelization disappears, which makes some sort of sense.

We also see that the GCC compiler is really bad at OpenMP tasks: the running time actually increases with the number of threads.

Fortunately, with OpenMP-4 we can break out of the loop with a \texttt{cancel} of the task group:

```cxx
// queenfinal.cxx
if (feasible(next)) {
  if (iqueen==N-1) {
    result = next;
    #pragma omp cancel taskgroup
  } else { // do next level
    auto attempt = place_queen(iqueen+1,next);
    if (attempt.has_value()) {
      result = attempt;
      #pragma omp cancel taskgroup
    }
  } // end if (iqueen==N-1)
} // end if (feasible)
```

Surprisingly, this does not immediately give a performance improvement. The reason for this is that
cancellation is disabled by default, and we have to set the environment variable

\texttt{OMP\_CANCELLATION=true}

With that, we get very good performance, as figure 31.6 shows, which lists sequential time, and multicore running time on the code with \texttt{cancel} directives. Running time is now approximately the same as the sequential time. Some questions are still left:

- Why does the time go up with core count?
- Why is the multicore code slower than the sequential code, and would the parallel code be faster than sequential if the amount of scalar work (for instance in the \texttt{feasible} function) would be larger?

One observation not reported here is that the GNU compiler has basically the same running time with and without cancellation. This is again shows that the GNU compiler is really bad at OpenMP tasks.
31.4 Sources used in this chapter

31.4.1 Listing of code header
PART III

PETSC
Chapter 32

PETSc basics

32.1 What is PETSc and why?

PETSc is a library with a great many uses, but for now let’s say that it’s primarily a library for dealing with the sort of linear algebra that comes from discretized PDEs. On a single processor, the basics of such computations can be coded out by a grad student during a semester course in numerical analysis, but on large scale issues get much more complicated and a library becomes indispensible.

PETSc’s prime justification is then that it helps you realize scientific computations at large scales, meaning large problem sizes on large numbers of processors.

There are two points to emphasize here:

- Linear algebra with dense matrices is relatively simple to formulate. For sparse matrices the amount of logistics in dealing with nonzero patterns increases greatly. PETSc does most of that for you.
- Linear algebra on a single processor, even a multicore one, is manageable; distributed memory parallelism is much harder, and distributed memory sparse linear algebra operations are doubly so. Using PETSc will save you many, many, Many! hours of coding over developing everything yourself from scratch.

Remark 30 The PETSc library has hundreds of routines. In this chapter and the next few we will only touch on a basic subset of these. The full list of man pages can be found at https://petsc.org/release/docs/manualpages/singleindex.html. Each man page comes with links to related routines, as well as (usually) example codes for that routine.

32.1.1 What is in PETSc?

The routines in PETSc (of which there are hundreds) can roughly be divided into these classes:

- Basic linear algebra tools: dense and sparse matrices, both sequential and parallel, their construction and simple operations.
- Solvers for linear systems, and to a lesser extent nonlinear systems; also time-stepping methods.
- Profiling and tracing: after a successful run, timing for various routines can be given. In case of failure, there are traceback and memory tracing facilities.
32.1.2 Programming model

PETSc, being based on MPI, uses the SPMD programming model (section 2.1), where all processes execute the same executable. Even more than in regular MPI codes, this makes sense here, since most PETSc objects are collectively created on some communicator, often `MPI_COMM_WORLD`. With the object-oriented design (section 32.1.3) this means that a PETSc program almost looks like a sequential program.

```c
MatMult(A, x, y); // y <- Ax
VecCopy(y, res); // r <- y
VecAXPY(res, -1., b); // r <- r - b
```

This is sometimes called **sequential semantics**.

32.1.3 Design philosophy

PETSc has an object-oriented design, even though it is written in C. There are classes of objects, such as `Mat` for matrices and `Vec` for Vectors, but there is also the `KSP` (for "Krylov SPace solver") class of linear system solvers, and `PetscViewer` for outputting matrices and vectors to screen or file.

Part of the object-oriented design is the polymorphism of objects: after you have created a `Mat` matrix as sparse or dense, all methods such as MatMult (for the matrix-vector product) take the same arguments: the matrix, and an input and output vector.

This design where the programmer manipulates a 'handle' also means that the internal of the object, the actual storage of the elements, is hidden from the programmer. This hiding goes so far that even filling in elements is not done directly but through function calls:

```c
VecSetValue(i, j, v, mode)
MatSetValue(i, j, v, mode)
MatSetValues(ni, is, nj, js, v, mode)
```

32.1.4 Language support

32.1.4.1 C/C++

PETSc is implemented in C, so there is a natural interface to C. There is no separate C++ interface.

32.1.4.2 Fortran

A `Fortran90` interface exists. The `Fortran77` interface is only of interest for historical reasons.

To use Fortran, include both a module and a cpp header file:

```c
#include "petsc/finclude/petscXXX.h"
use petscXXX
```

(here `XXX` stands for one of the PETSc types, but including `petsc.h` and using `use petsc` gives inclusion of the whole library.)

Variables can be declared with their type (`Vec, Mat, KSP` et cetera), but internally they are Fortran `Type` objects so they can be declared as such.
32. PETSc basics

Example:

```c
#include "petsc/include/petscvec.h"
use petscvec
Vec b
type(tVec) x
```

The output arguments of many query routines are optional in PETSc. While in C a generic `NULL` can be passed, Fortran has type-specific nulls, such as `PETSC_NULL_INTEGER`, `PETSC_NULL_OBJECT`.

32.1.4.3 Python

A python interface was written by Lisandro Dalcin. It can be added to PETSc at installation time; section 32.3.

This book discusses the Python interface in short remarks in the appropriate sections.

32.1.5 Documentation

PETSc comes with a manual in pdf form and web pages with the documentation for every routine. The starting point is the web page https://petsc.org/release/documentation/.

There is also a mailing list with excellent support for questions and bug reports.

**TACC note.** For questions specific to using PETSc on TACC resources, submit tickets to the TACC or XSEDE portal.

32.2 Basics of running a PETSc program

32.2.1 Compilation

A PETSc compilation needs a number of include and library paths, probably too many to specify interactively. The easiest solution is to create a makefile and load the standard variables and compilation rules. (You can use `$PETSC_DIR/share/petsc/Makefile.user` for inspiration.)

Throughout, we will assume that variables `PETSC_DIR` and `PETSC_ARCH` have been set. These depend on your local installation; see section 32.3.

In the easiest setup, you leave the compilation to PETSc and your make rules only do the link step, using `CLINKER` or `FLINKER` for C/Fortran respectively:

```make
include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules
program : program.o
    ${CLINKER} -o $@ $^ ${PETSC_LIB}
```

The two include lines provide the compilation rule and the library variable.

You can use these rules:
32.2. Basics of running a PETSc program

```bash
% : %.F90
   $(LINK.F) -o $@ $^ $(LDLIBS)
%: %.F90
   $(COMPILE.F) $(OUTPUT_OPTION) $<
%
% : %.cxx
   $(LINK.cc) -o $@ $^ $(LDLIBS)
%: %.cxx
   $(COMPILE.cc) $(OUTPUT_OPTION) $<
```

```bash
## example link rule:
# app : a.o b.o c.o
#       $(LINK.F) -o $@ $^ $(LDLIBS)
```

(The `PETSC_CC_INCLUDES` variable contains all paths for compilation of C programs; correspondingly there is `PETSC_FC_INCLUDES` for Fortran source.)

If don’t want to include those configuration files, you can find out the include options by:

```bash
cd $PETSC_DIR
make getincdirs
make getlinklibs
```

and copying the results into your compilation script.

There is an example makefile `$PETSC_DIR/share/petsc/Makefile.user` you can take for inspiration. Invoked without arguments it prints out the relevant variables:

```bash
[c:246] make -f ! $PETSC_DIR/share/petsc/Makefile.user
CC=/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/bin/mpicc
CXX=/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/bin/mpicxx
FC=/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/bin/mpif90
CFLAGS=-Wall -Wwrite-strings -Wno-strict-aliasing -Wno-unknown-pragmas -fstack-protector -Qunoptimizedifiantions -g3
CXXFLAGS=-Wall -Wwrite-strings -Wno-strict-aliasing -Wno-unknown-pragmas -fstack-protector -fvisibility=hidden -g
FFLAGS=-m64 -g
LDLIBS=-lpetsc -lm
```

**TACC note.** On TACC clusters, a petsc installation is loaded by commands such as

```bash
module load petsc/3.16
```

Use `module avail petsc` to see what configurations exist. The basic versions are

```bash
# development
module load petsc/3.11-debug
# production
module load petsc/3.11
```

Other installations are real versus complex, or 64bit integers instead of the default 32. The command
32. PETSc basics

Figure 32.1 PetscInitialize
C:

PetscErrorCode PetscInitialize
(int *argc, char ***args, const char file[], const char help[])

Input Parameters:
argv - count of number of command line arguments
args - the command line arguments
file - [optional] PETSc database file.
help - [optional] Help message to print, use NULL for no message

Fortran:
call PetscInitialize(file,ierr)

Input parameters:
ierr - error return code
file - [optional] PETSc database file,
    use PETSC_NULL_CHARACTER to not check for code specific file.

module spider petsc
tells you all the available petsc versions. The listed modules have a naming convention such as
petsc/3.11-i64debug where the 3.11 is the PETSc release (minor patches are not included
in this version; TACC aims to install only the latest patch, but generally several versions are
available), and i64debug describes the debug version of the installation with 64bit integers.

32.2.2 Running

PETSc programs use MPI for parallelism, so they are started like any other MPI program:

mpiexec -n 5 -machinefile mf \
    your_petsc_program option1 option2 option3

TACC note. On TACC clusters, use ibrun.

32.2.3 Initialization and finalization

PETSc has an call that initializes both PETSc and MPI, so normally you would replace MPI_Init by PetscInitialize
(figure 32.1). Unlike with MPI, you do not want to use a NULL value for the argc, argv arguments, since
PETSc makes extensive use of commandline options; see section 39.3.

```c
// init.c
ierr = PetscInitialize
    (&argc, &argv, (char*)0, help);
int flag;
MPI_Initialized(&flag);
if (flag)
    printf("MPI was initialized by PETSc\n");
else
    printf("MPI not yet initialized\n");
```
32.2. Basics of running a PETSc program

For the full source of this example, see section 32.4.2

There are two further arguments to PetscInitialize:

1. the name of an options database file; and
2. a help string, that is displayed if you run your program with the -h option.

Fortran note 18: Petsc Initialization.

- The Fortran version has no arguments for commandline options; it also doesn’t take a help string.
- The first argument is the name of a database file of options; if none is specified, give PETSC_NULL_CHARACTER as argument.
- If your main program is in C, but some of your PETSc calls are in Fortran files, it is necessary to call PetscInitializeFortran after PetscInitialize.

```fortran
!! init.F90
   call PetscInitialize(PETSC_NULL_CHARACTER,ierr)    
   CHKERRA(ierr)
   call MPI_Initialized(flag,ierr)                    
   CHKERRA(ierr)
   if (flag) then                                      
      print *,"MPI was initialized by PETSc"
```

For the full source of this example, see section 32.4.3

Python note 32: Init, and with commandline options. The following works if you don’t need commandline options.

```python
from petsc4py import PETSc

to pass commandline arguments to PETSc, do:

import sys
from petsc4py import init
init(sys.argv)
from petsc4py import PETSc

After initialization, you can use MPI_COMM_WORLD or PETSC_COMM_WORLD (which is created by MPI_Comm_dup and used internally by PETSc):

```python
MPI_Comm  comm = PETSC_COMM_WORLD;
MPI_Comm_rank(comm,&mytid);
MPI_Comm_size(comm,&ntids);
```

Python note 33: Communicator object.

```python
comm = PETSc.COMM_WORLD
nprocs = comm.getSize(self)
procno = comm.getRank(self)
```

The corresponding call to replace MPI_Finalize is PetscFinalize. You can elegantly capture and return the error code by the idiom

```python
return PetscFinalize();
```

at the end of your main program.

Victor Eijkhout
32.3 PETSc installation

PETSc has a large number of installation options. These can roughly be divided into:

1. Options to describe the environment in which PETSc is being installed, such as the names of the compilers or the location of the MPI library;
2. Options to specify the type of PETSc installation: real versus complex, 32 versus 64-bit integers, et cetera;
3. Options to specify additional packages to download.

For an existing installation, you can find the options used, and other aspects of the build history, in the configure.log/make.log files:

$PETSC_DIR/$PETSC_ARCH/lib/petsc/conf/configure.log
$PETSC_DIR/$PETSC_ARCH/lib/petsc/conf/make.log

32.3.1 Debug

For any set of options, you will typically make two installations: one with --with-debugging=yes and once no. See section 39.1.1 for more detail on the differences between debug and non-debug mode.

32.3.2 Environment options

Compilers, compiler options, MPI.

While it is possible to specify --download_mpich, this should only be done on machines that you are certain do not already have an MPI library, such as your personal laptop. Supercomputer clusters are likely to have an optimized MPI library, and letting PETSc download its own will lead to degraded performance.

32.3.3 Variants

- Scalars: the option --with-scalar-type has values real, complex; --with-precision has values single, double, __float128, __fp16.

32.3.4 External packages

PETSc can extend its functionality through external packages such as mumps, Hypre, fftw. These can be specified in two ways:

1. Referring to an installation already on your system:
   --with-hdf5-include=${TACC_HDF5_INC}
   --with-hf5_lib=${TACC_HDF5_LIB}
2. By letting petsc download and install them itself:
   --with-parmetis=1 --download-parmetis=1

Python note 34: petsc4py interface. The Python interface (section 32.1.4.3) can be installed with the option
--download-petsc4py=<no,yes,filename,url>

This is easiest if your python already includes mpi4py; see section 1.5.4.
Remark 31  There are two packages that PETSc is capable of downloading and install, but that you may want to avoid:

- **fblaslapack**: this gives you BLAS/LAPACK through the Fortran ‘reference implementation’. If you have an optimized version, such as Intel’s mkl available, this will give much higher performance.
- **mpich**: this installs a MPI implementation, which may be required for your laptop. However, supercomputer clusters will already have an MPI implementation that uses the high-speed network. PETSc’s downloaded version does not do that. Again, finding and using the already installed software may greatly improve your performance.
32.4 Sources used in this chapter

32.4.1 Listing of code header

32.4.2 Listing of code examples/petsc/c/init.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <petscsys.h>

int main(int argc,char **argv)
{
    PetscErrorCode ierr;
    char help[] = "\nInit example.\n\n"
    ierr = PetscInitialize
    (argc,&argv,(char*)0,help); CHKERRQ(ierr);
    int flag;
    MPI_Init(&flag);
    if (flag)
        printf("MPI was initialized by PETSc\n");
    else
        printf("MPI not yet initialized\n");
    return PetscFinalize();
}
```

32.4.3 Listing of code examples/petsc/f/init.F90

```fortran
Program PetscInit

#include <petsc/finclude/petscsys.h>
use petsc
implicit none

logical :: flag
PetscErrorCode ierr;
character*80 :: help = "\nInit example.\n\n"
call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
CHKERRA(ierr)
call MPI_Init(flag,ierr)
CHKERRA(ierr)
if (flag) then
    print *,"MPI was initialized by PETSc"
else
    print *,"MPI not yet initialized"
end if
call PetscFinalize(ierr); CHKERRA(ierr);

End Program PetscInit
```
Chapter 33

PETSc objects

33.1 Distributed objects

PETSc is based on the SPMD model, and all its objects act like they exist in parallel, spread out over all the processes. Therefore, prior to discussing specific objects in detail, we briefly discuss how PETSc treats distributed objects.

For a matrix or vector you need to specify the size. This can be done two ways:

- you specify the global size and PETSc distributes the object over the processes, or
- you specify on each process the local size

If you specify both the global size and the local sizes, PETSc will check for consistency.

For example, if you have a vector of \( N \) components, or a matrix of \( N \) rows, and you have \( P \) processes, each process will receive \( N/P \) components or rows if \( P \) divides evenly in \( N \). If \( P \) does not divide evenly, the excess is spread over the processes.

The way the distribution is done is by contiguous blocks: with 10 processes and 1000 components in a vector, process 0 gets the range 0 … 99, process 1 gets 1 … 199, et cetera. This simple scheme suffices for many cases, but PETSc has facilities for more sophisticated load balancing.

33.1.1 Support for distributions

Once an object has been created and distributed, you do not need to remember the size or the distribution yourself: you can query these with calls such as \( \text{VecGetSize}, \text{VecGetLocalSize} \).

The corresponding matrix routines \( \text{MatGetSize}, \text{MatGetLocalSize} \) give both information for the distributions in \( i \) and \( j \) direction, which can be independent. Since a matrix is distributed by rows, \( \text{MatGetOwnershipRange} \) only gives a row range.


33. PETSc objects

Figure 33.1 PetscSplitOwnership

PetscSplitOwnership

Synopsis

#include "petscsys.h"

PetscErrorCode PetscSplitOwnership

    (MPI_Comm comm, PetscInt *n, PetscInt *N)

Collective (if n or N is PETSC_DECIDE)

Input Parameters

comm - MPI communicator that shares the object being divided
n - local length (or PETSC_DECIDE to have it set)
N - global length (or PETSC_DECIDE)

// split.c
N = 100; n = PETSC_DECIDE;
PetcSplitOwnership(comm,&n,&N);
PetcPrintf(comm,"Global %d, local %d\n",N,n);

N = PETSC_DECIDE; n = 10;
PetcSplitOwnership(comm,&n,&N);
PetcPrintf(comm,"Global %d, local %d\n",N,n);

For the full source of this example, see section 33.8.2

While PETSc objects are implemented using local memory on each process, conceptually they act like
global objects, with a global indexing scheme. Thus, each process can query which elements out of the
global object are stored locally. For vectors, the relevant routine is VecGetOwnershipRange, which returns
two parameters, low and high, respectively the first element index stored, and one-more-than-the-last
index stored.

This gives the idiom:

VecGetOwnershipRange(myvector,&low,&high);
for (int myidx=low; myidx<high; myidx++)
    // do something at index myidx

These conversions between local and global size can also be done explicitly, using the PetscSplitOwnership
(figure 33.1) routine. This routine takes two parameter, for the local and global size, and whichever one is
initialized to PETSC_DECIDE gets computed from the other.

33.2 Scalars

Unlike programming languages that explicitly distinguish between single and double precision numbers,
PETSc has only a single scalar type: PetscScalar. The precision of this is determined at installation time.
In fact, a PetscScalar can even be a complex number if the installation specified that the scalar type is
complex.
Even in applications that use complex numbers there can be quantities that are real: for instance, the norm of a complex vector is a real number. For that reason, PETSc also has the type `PetscReal`. There is also an explicit `PetscComplex`.

Furthermore, there is

```
#define PETSC_BINARY_INT_SIZE  (32/8)
#define PETSC_BINARY_FLOAT_SIZE (32/8)
#define PETSC_BINARY_CHAR_SIZE  (8/8)
#define PETSC_BINARY_SHORT_SIZE (16/8)
#define PETSC_BINARY_DOUBLE_SIZE (64/8)
#define PETSC_BINARY_SCALAR_SIZE sizeof(PetscScalar)
```

### 33.2.1 Integers

Integers in PETSc are likewise of a size determined at installation time: `PetscInt` can be 32 or 64 bits. The latter possibility is useful for indexing into large vectors and matrices. Furthermore, there is a `PetscErrorCode` type for catching the return code of PETSc routines; see section 39.1.2.

For compatibility with other packages there are two more integer types:

- `PetscBLASInt` is the integer type used by the Basic Linear Algebra Subprograms (BLAS) / Linear Algebra Package (LAPACK) library. This is 32-bits if the `-download- blas-lapack` option is used, but it can be 64-bit if MKL is used. The routine `PetscBLASIntCast` casts a `PetscInt` to `PetscBLASInt`, or returns `PETSC_ERR_ARG_OUTOFRANGE` if it is too large.
- `PetscMPIInt` is the integer type of the MPI library, which is always 32-bits. The routine `PetscMPIIntCast` casts a `PetscInt` to `PetscMPIInt`, or returns `PETSC_ERR_ARG_OUTOFRANGE` if it is too large.

Many external packages do not support 64-bit integers.

### 33.2.2 Complex

Numbers of type `PetscComplex` have a precision matching `PetscReal`.

Form a complex number using `PETSC_i`:

```
PetscComplex x = 1.0 + 2.0 * PETSC_i;
```

The real and imaginary part can be extract with the functions `PetscRealPart` and `PetscImaginaryPart` which return a `PetscReal`.

There are also routines `VecRealPart` and `VecImaginaryPart` that replace a vector with its real or imaginary part respectively. Likewise `MatRealPart` and `MatImaginaryPart`.

### 33.2.3 MPI Scalars

For MPI calls, `MPIU_REAL` is the MPI type corresponding to the current `PetscReal`.

For MPI calls, `MPIU_SCALAR` is the MPI type corresponding to the current `PetscScalar`.

For MPI calls, `MPIU_COMPLEX` is the MPI type corresponding to the current `PetscComplex`.
33. PETSc objects

Figure 33.2 VecCreate
C:
PetscErrorCode VecCreate(MPI_Comm comm,Vec *v);

F:
VecCreate( comm,v,ierr )
MPI_Comm :: comm
Vec :: v
PetscErrorCode :: ierr

Python:
vec = PETSc.Vec()
vec.create()
# or:
vec = PETSc.Vec().create()

33.2.4 Booleans
There is a PetscBool datatype with values PETSC_TRUE and PETSC_FALSE.

33.3 Vec: Vectors
Vectors are objects with a linear index. The elements of a vector are floating point numbers or complex numbers (see section 33.2), but not integers: for that see section 33.5.1.

33.3.1 Vector construction
Constructing a vector takes a number of steps. First of all, the vector object needs to be created on a communicator with VecCreate (figure 33.2)

Python note 35: Vector creation. In python, PETSc.Vec() creates an object with null handle, so a subsequent create() call is needed. In C and Fortran, the vector type is a keyword; in Python it is a member of PETSc.Vec.Type.

```python
# setvalues.py
comm = PETSc.COMM_WORLD
x = PETSc.Vec().create(comm=comm)
x.setType(PETSc.Vec.Type.MPI)
```

The corresponding routine VecDestroy (figure 33.3) deallocates data and zeros the pointer. (This and all other Destroy routines are collective because of underlying MPI technicalities.)

The vector type needs to be set with VecSetType (figure 33.4).

The most common vector types are:
- VECSEQ for sequential vectors, that is, living on a single process; This is typically created on the MPI_COMM_SELF or PETSC_COMM_SELF communicator.
33.3. Vec: Vectors

Figure 33.3 VecDestroy
Synopsis
#include "petscvec.h"
PetscErrorCode VecDestroy(Vec *v)

Collective on Vec

Input Parameters:
v - the vector

Figure 33.4 VecSetType
Synopsis:
#include "petscvec.h"
PetscErrorCode VecSetType(Vec vec, VecType method)

Collective on Vec

Input Parameters:
vec - The vector object
method - The name of the vector type

Options Database Key
-vec_type <type> - Sets the vector type; use -help for a list of available types

- VECMPI for a vector distributed over the communicator. This is typically created on the MPI_COMM_WORLD or PETSC_COMM_WORLD communicator, or one derived from it.
- VECSTANDARD is VECSEQ when used on a single process, or VECMPI on multiple.

You may wonder why these types exist: you could have just one type, which would be as parallel as possible. The reason is that in a parallel run you may occasionally have a separate linear system on each process, which would require a sequential vector (and matrix) on each process, not part of a larger linear system.

Once you have created one vector, you can make more like it by VecDuplicate,

|| VecDuplicate(Vec old, Vec *new);

or VecDuplicateVecs

|| VecDuplicateVecs(Vec old, PetscInt n, Vec **new);

for multiple vectors. For the latter, there is a joint destroy call VecDestroyVecs:

|| VecDestroyVecs(PetscInt n, Vec **vecs);

(which is different in Fortran).

33.3.2 Vector layout

Next in the creation process the vector size is set with VecSetSizes (figure 33.5). Since a vector is typically distributed, this involves the global size and the sizes on the processors. Setting both is redundant, so it
33. **PETSc objects**

---

**Figure 33.5 VecSetSizes**

C:
```c
#include "petscvec.h"
PetscErrorCode VecSetSizes(Vec v, PetscInt n, PetscInt N)
Collective on Vec

Input Parameters
v : the vector
n : the local size (or PETSC_DECIDE to have it set)
N : the global size (or PETSC_DECIDE)

Python:
```
PETSc.Vec.setSizes(self, size, bsize=None)
size is a tuple of local/global
```

---

**Figure 33.6 VecGetSize**

VecGetSize / VecGetLocalSize

C:
```c
#include "petscvec.h"
PetscErrorCode VecGetSize(Vec x,PetscInt *gsize)
PetscErrorCode VecGetLocalSize(Vec x,PetscInt *lsize)

Input Parameter
x - the vector

Output Parameters
gsize - the global length of the vector
lsize - the local length of the vector

Python:
```
PETSc.Vec.getLocalSize(self)
PETSc.Vec.getSize(self)
PETSc.Vec.getSizes(self)
```

---

is possible to specify one and let the other be computed by the library. This is indicated by setting it to PETSC_DECIDE.

**Python note 36: Vector size.** Use `PETSc.DECIDE` for the parameter not specified:

```
||x.setSizes([2,PETSc.DECIDE])
```

The size is queried with `VecGetSize` (figure 33.6) for the global size and `VecGetLocalSize` (figure 33.6) for the local size.

Each processor gets a contiguous part of the vector. Use `VecGetOwnershipRange` (figure 33.7) to query the first index on this process, and the first one of the next process.

In general it is best to let PETSc take care of memory management of matrix and vector objects, including allocating and freeing the memory. However, in cases where PETSc interfaces to other applications it maybe desirable to create a `Vec` object from an already allocated array: `VecCreateSeqWithArray` and
33.3. Vec: Vectors

**Figure 33.7 VecGetOwnershipRange**

```c
#include "petscvec.h"
PetscErrorCode VecGetOwnershipRange(Vec x, PetscInt *low, PetscInt *high)
```

Input parameter:

* x - the vector

Output parameters:

* low - the first local element, pass in NULL if not interested
* high - one more than the last local element, pass in NULL if not interested

Fortran note:

use PETSC_NULL_INTEGER for NULL.

**Figure 33.8 VecAXPY**

Synopsis:

```c
#include "petscvec.h"
PetscErrorCode VecAXPY(Vec y, PetscScalar alpha, Vec x)
```

Not collective on Vec

Input Parameters:

* alpha - the scalar
* x, y - the vectors

Output Parameter:

* y - output vector

**VecCreateMPIWithArray.**

```c
VecCreateSeqWithArray
(MPI_Comm comm, PetscInt bs,
 PetscInt n, PetscScalar *array, Vec *V);
```

**VecCreateMPIWithArray**

```c
(MPI_Comm comm, PetscInt bs,
 PetscInt n, PetscInt N, PetscScalar *array, Vec *vv);
```

As you will see in section 33.4.1, you can also create vectors based on the layout of a matrix, using **MatCreateVecs.**

**33.3.3 Vector operations**

There are many routines operating on vectors that you need to write scientific applications. Examples are: norms, vector addition (including BLAS-type ‘AXPY’ routines: **VecAXPY** (figure 33.8)), pointwise scaling, inner products. A large number of such operations are available in PETSc through single function calls to VecXYZ routines.

For debugging purposes, the **VecView** (figure 33.9) routine can be used to display vectors on screen as ascii output,
33. PETSc objects

Figure 33.9 VecView
C:
#include "petscvec.h"
PetscErrorCode VecView(Vec vec,PetscViewer viewer)

for ascii output use:
PETSC_VIEWER_STDOUT_WORLD

Python:
PETSc.Vec.view(self, Viewer viewer=None)

ascii output is default or use:
PETSc.Viewer.STDOUT(type cls, comm=None)

Figure 33.10 VecDot
Synopsis:
#include "petscvec.h"
PetscErrorCode VecDot(Vec x,Vec y,PetscScalar *val)

Collective on Vec

Input Parameters:
x, y - the vectors

Output Parameter:
val - the dot product

// fftsine.c
ierr = VecView(signal,PETSC_VIEWER_STDOUT_WORLD);
ierr = MatMult(transform,signal,frequencies);
ierr = VecScale(frequencies,1./Nglobal);
ierr = VecView(frequencies,PETSC_VIEWER_STDOUT_WORLD);

For the full source of this example, see section 33.8.3

but the routine call also use more general PetscViewer objects, for instance to dump a vector to file.

Here are a couple of representative vector routines:

PetscReal lambda;
ierr = VecNorm(y,NORM_2,&lambda); CHKERRQ(ierr);
ierr = VecScale(y,1./lambda); CHKERRQ(ierr);

Exercise 33.1. Create a vector where the values are a single sine wave. using VecGetSize, VecGetLocalSize, VecGetOwnershipRange. Quick visual inspection:

  ibrun vec -n 12 -vec_view
(There is a skeleton for this exercise under the name vec.)

Exercise 33.2. Use the routines VecDot (figure 33.10), VecScale (figure 33.11) and VecNorm (figure 33.12) to compute the inner product of vectors x, y, scale the vector x, and
Figure 33.11 VecScale
Synopsis:
#include "petscvec.h"
PetscErrorCode VecScale(Vec x, PetscScalar alpha)

Not collective on Vec

Input Parameters:
x - the vector
alpha - the scalar

Output Parameter:
x - the scaled vector

Figure 33.12 VecNorm
C:
#include "petscvec.h"
PetscErrorCode VecNorm(Vec x,NormType type,PetscReal *val)
where type is
   NORM_1, NORM_2, NORM_FROBENIUS, NORM_INFINITY

Python:
PETScl.Vec.norm(self, norm_type=None)
where norm is variable in PETSc.NormType:
   NORM_1, NORM_2, NORM_FROBENIUS, NORM_INFINITY or
   N1, N2, FRB, INF

check its norm:
   \[ p \leftarrow x^T y \]
   \[ x \leftarrow x / p \]
   \[ n \leftarrow \|x\|_2 \]

Python note 37: Vector operations. The plus operator is overloaded so that
   \[ x + y \]
is defined.
   \[ x . \text{sum()} \ # \text{max}, \text{min}, \ldots \]
   \[ x . \text{dot}(y) \]
   \[ x . \text{norm(PETSc.NormType.NORM_INFINITY)} \]

33.3.3.1 Split collectives

MPI is capable (in principle) of 'overlapping computation and communication', or latency hiding. PETSc supports this by splitting norms and inner products into two phases.

- Start inner product / norm with VecDotBegin / VecNormBegin;
33. PETSc objects

Figure 33.13 VecSetValue
Synopsis
#include <petscvec.h>
PetscErrorCode VecSetValue
(Vec v, PetscInt row, PetscScalar value, InsertMode mode);

Not Collective
Input Parameters
v - the vector
row - the row location of the entry
value - the value to insert
mode - either INSERT_VALUES or ADD_VALUES

Figure 33.14 VecSetValues
Synopsis
#include "petscvec.h"
PetscErrorCode VecSetValues
(Vec x, PetscInt ni, const PetscInt ix[], const PetscScalar y[], InsertMode iora);

Not Collective
Input Parameters:
x - vector to insert in
ni - number of elements to add
ix - indices where to add
y - array of values
iora - either INSERT_VALUES or ADD_VALUES, where
      ADD_VALUES adds values to any existing entries, and
      INSERT_VALUES replaces existing entries with new values

- Conclude inner product / norm with VecDotEnd / VecNormEnd;

Even if you achieve no overlap, it is possible to use these calls to combine a number of 'collectives': do the
Begin calls of one inner product and one norm; then do (in the same sequence) the End calls. This means
that only a single reduction is performed on a two-word package, rather than two separate reductions on
a single word.

33.3.4 Vector elements

Setting elements of a traditional array is simple. Setting elements of a distributed array is harder. First of
all, VecSet sets the vector to a constant value:

ierr = VecSet(x, 1.); CHKERRQ(ierr);

In the general case, setting elements in a PETSc vector is done through a function VecSetValue (figure 33.13)
for setting elements that uses global numbering; any process can set any elements in the vector. There is
also a routine VecSetValues (figure 33.14) for setting multiple elements. This is mostly useful for setting
dense subblocks of a block matrix.
33.3. Vec: Vectors

Figure 33.15 VecAssemblyBegin

```c
#include "petscvec.h"
PetscErrorCode VecAssemblyBegin(Vec vec)
PetscErrorCode VecAssemblyEnd(Vec vec)
```

Collective on Vec

Input Parameter
vec - the vector

We illustrate both routines by setting a single element with `VecSetValue`, and two elements with `VecSetValues`. In the latter case we need an array of length two for both the indices and values. The indices need not be successive.

```c
i = 1; v = 3.14;  
VecSetValue(x,i,v,INSERT_VALUES);  
ii[0] = 1; ii[1] = 2; vv[0] = 2.7; vv[1] = 3.1;  
VecSetValues(x,2,ii,vv,INSERT_VALUES);
```

```python
i = 1; v = 3.14;  
VecSetValue(x,i,v,INSERT_VALUES);  
ii[0] = 1; ii[1] = 2; vv[0] = 2.7; vv[1] = 3.1
```

**Python note 38: Setting vector values.** Single element:

```python
x.setValue(0,1.)
```

Multiple elements:

```python
x.setValues( [2*procno,2*procno+1], [2.,3. ] )
```

Using `VecSetValue` for specifying a local vector element corresponds to simple insertion in the local array. However, an element that belongs to another process needs to be transferred. This done in two calls: `VecAssemblyBegin` (figure 33.15) and `VecAssemblyEnd`.

```c
if (myrank==0) then  
do vecidx=0,globalsize-1
  veclt = vecidx
  call VecSetValue(vector,vecidx,veclt,INSERT_VALUES,ierr)
end do
end if
```

```c
call VecAssemblyBegin(vector,ierr)
call VecAssemblyEnd(vector,ierr)
```

For the full source of this example, see section 33.8.4

(If you know the MPI library, you’ll recognize that the first call corresponds to posting nonblocking send and receive calls; the second then contains the wait calls. Thus, the existence of these separate calls make latency hiding possible.)

```c
VecAssemblyBegin(myvec);
// do work that does not need the vector myvec
VecAssemblyEnd(myvec);
```
Figure 33.16 VecGetArray

C:
#include "petscvec.h"
PetscErrorCode VecGetArray(Vec x,PetscScalar **a)
PetscErrorCode VecGetArrayRead(Vec x,const PetscScalar **a)

Input Parameter
x : the vector

Output Parameter
a : location to put pointer to the array

PetscErrorCode VecRestoreArray(Vec x,PetscScalar **a)
PetscErrorCode VecRestoreArrayRead(Vec x,const PetscScalar **a)

Input Parameters
x : the vector
a : location of pointer to array obtained from VecGetArray()

Fortran90:
#include <petsc/include/petscvec.h>
use petscvec
VecGetArrayF90(Vec x,{Scalar, pointer :: xx_v(:)},integer ierr)
(Vec x,{Real, pointer :: xx_v(:)},integer ierr)
VecRestoreArrayF90(Vec x,{Scalar, pointer :: xx_v(:)},integer ierr)

Python:
PETSc.Vec.getArray(self, readonly=False)
?? PETSc.Vec.resetArray(self, force=False)

Elements can either be inserted with INSERT_VALUES, or added with ADD_VALUES in the VecSetValue / VecSetValues call. You can not immediately mix these modes; to do so you need to call VecAssemblyBegin / VecAssemblyEnd in between add/insert phases.

33.3.4.1 Explicit element access

Since the vector routines cover a large repertoire of operations, you hardly ever need to access the actual elements. Should you still need those elements, you can use VecGetArray (figure 33.16) for general access or VecGetArrayRead (figure 33.16) for read-only.

PETSc insists that you properly release this pointer again with VecRestoreArray (figure 33.17) or VecRestoreArrayRead (figure 33.17).

In the following example, a vector is scaled through direct array access. Note the differing calls for the source and target vector, and note the const qualifier on the source array:

C:
#include "petscvec.h"
PetscErrorCodeVecGetArray(Vec x,PetscScalar **a)
PetscErrorCode VecGetArrayRead(Vec x,const PetscScalar **a)
33.3. Vec: Vectors

Figure 33.17 VecRestoreArray

C:
#include "petscvec.h"
PetscErrorCode VecRestoreArray(Vec x, PetscScalar **a)

Logically Collective on Vec

Input Parameters:
x - the vector
a - location of pointer to array obtained from VecGetArray()

Fortran90:
#include <petsc/finclude/petscvec.h>
use petscvec
VecRestoreArrayF90(Vec x, {Scalar, pointer :: xx_v(:)}, integer ierr)

Input Parameters:
x - vector
xx_v - the Fortran90 pointer to the array

---

<table>
<thead>
<tr>
<th>Input Parameter</th>
<th>x : the vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Parameter</td>
<td>a : location to put pointer to the array</td>
</tr>
</tbody>
</table>

PetscErrorCode VecRestoreArray(Vec x, PetscScalar **a)
PetscErrorCode VecRestoreArrayRead(Vec x, const PetscScalar **a)

Input Parameters
x : the vector
a : location of pointer to array obtained from VecGetArray()

Fortran90:
#include <petsc/finclude/petscvec.h>
use petscvec
VecGetArrayF90(Vec x, {Scalar, pointer :: xx_v(:)}, integer ierr)
(VecRestoreArrayF90(Vec x, {Scalar, pointer :: xx_v(:)}, integer ierr)

Python:
PETSc.Vec.getArray(self, readonly=False)
?? PETSc.Vec.resetArray(self, force=False)

This example also uses VecGetLocalSize to determine the size of the data accessed. Even running in a distributed context you can only get the array of local elements. Accessing the elements from another process requires explicit communication; see section 33.5.2.

There are some variants to the VecGetArray operation:

- VecReplaceArray (figure 33.18) frees the memory of the Vec object, and replaces it with a different array. That latter array needs to be allocated with PetscMalloc.
**PETSc objects**

**Figure 33.18 VecPlaceArray**  
Replace the storage of a vector by another array

**Synopsis**

```c
#include "petscvec.h"
PetscErrorCode VecPlaceArray(Vec vec,const PetscScalar array[])
PetscErrorCode VecReplaceArray(Vec vec,const PetscScalar array[])
```

**Input Parameters**

- `vec` - the vector
- `array` - the array

- **VecPlaceArray** (figure 33.18) also installs a new array in the vector, but it keeps the original array; this can be restored with `VecResetArray`.

Putting the array of one vector into another has a common application, where you have a distributed vector, but want to apply PETSc operations to its local section as if it were a sequential vector. In that case you would create a sequential vector, and **VecPlaceArray** the contents of the distributed vector into it.

**Fortran note 19:** F90 array access through pointer. There are routines such as `VecGetArrayF90` (with corresponding `VecRestoreArrayF90`) that return a (Fortran) pointer to a one-dimensional array.

```fortran
!! vecset.F90
Vec :: vector
PetscScalar, dimension(:), pointer :: elements
call VecGetArrayF90(vector,elements,ierr)
write (msg,10) myrank,elements(1)
10 format("First element on process",i3,",","f7.4","n")
call PetscSynchronizedPrintf(comm,msg,ierr)
call PetscSynchronizedFlush(comm,PETSC_STDOUT,ierr)
call VecRestoreArrayF90(vector,elements,ierr)
```

*For the full source of this example, see section 33.8.4*

```fortran
!! vecarray.F90
PetscScalar, dimension(:), Pointer :: k
in_array,out_array
call VecGetArrayReadF90(x,in_array,ierr)
call VecGetArrayF90(y,out_array,ierr)
call VecGetLocalSize(x,localsize,ierr)
do index=1,localsize
   out_array(index) = 2*in_array(index)
end do
call VecRestoreArrayReadF90(x,in_array,ierr)
call VecRestoreArrayF90(y,out_array,ierr)
```

*For the full source of this example, see section 33.8.5*

**Python note 39:** Vector access.

```python
x.getArray()
x.getValues(3)
x.getValues([1, 2])
```
Figure 33.19 MatCreate

C:
PetscErrorCode MatCreate(MPI_Comm comm,Mat *v);

Python:
mat = PETSc.Mat()
mat.create()
# or:
mat = PETSc.Mat().create()

33.3.5 File I/O

As mentioned above, VecView can be used for displaying a vector on the terminal screen. However, viewers are actually much more general. As explained in section 39.2.2, they can also be used to export vector data, for instance to file.

The converse operation, to load a vector that was exported in this manner, is VecLoad.

Since these operations are each other’s inverses, usually you don’t need to know the file format. But just in case:

| PetscInt   | VEC_FILE_CLASSID |
| PetscInt   | number of rows   |
| PetscScalar| *values of all entries |

That is, the file starts with a magic number, then the number of vector elements, and subsequently all scalar values.

33.4 Mat: Matrices

PETSc matrices come in a number of types, sparse and dense being the most important ones. Another possibility is to have the matrix in operation form, where only the action $y \leftarrow Ax$ is defined.

33.4.1 Matrix creation

Creating a matrix also starts by specifying a communicator on which the matrix lives collectively: MatCreate (figure 33.19)

Set the matrix type with MatsetType (figure 33.20). The main choices are between sequential versus distributed and dense versus sparse, giving types: MATMPIDENSE, MATMPIAIJ, MATSEQDENSE, MATSEQAIJ.

Distributed matrices are partitioned by block rows: each process stores a block row, that is, a contiguous set of matrix rows. It stores all elements in that block row. In order for a matrix-vector product to be executable, both the input and output vector need to be partitioned conforming to the matrix.

While for dense matrices the block row scheme is not scalable, for matrices from PDEs it makes sense. There, a subdivision by matrix blocks would lead to many empty blocks.
33. PETSc objects

Figure 33.20 MatSetType
#include "petscmat.h"
PetscErrorCode MatSetType(Mat mat, MatType matype)

Collective on Mat

Input Parameters:
mat- the matrix object
matype- matrix type

Options Database Key
-mat_type <method> -Sets the type; use -help for a list of available methods (for instance, seqaij)

![Figure 33.1: Matrix partitioning by block rows](image)

Just as with vectors, there is a local and global size; except that that now applies to rows and columns. Set sizes with MatSetSizes (figure 33.21) and subsequently query them with MatSizes (figure 33.22). The concept of local column size is tricky: since a process stores a full block row you may expect the local column size to be the full matrix size, but that is not true. The exact definition will be discussed later, but for square matrices it is a safe strategy to let the local row and column size to be equal.

Instead of querying a matrix size and creating vectors accordingly, the routine MatCreateVecs (figure 33.23) can be used. (Sometimes this is even required; see section 33.4.9.)

33.4.2 Nonzero structure

In case of a dense matrix, once you have specified the size and the number of MPI processes, it is simple to determine how much space PETSc needs to allocate for the matrix. For a sparse matrix this is more complicated, since the matrix can be anywhere between completely empty and completely filled in. It would be possible to have a dynamic approach where, as elements are specified, the space grows; however,
33.4. Mat: Matrices

Figure 33.21 MatSetSizes
C:
#include "petscmat.h"
PetscErrorCode MatSetSizes(Mat A,
    PetscInt m, PetscInt n, PetscInt M, PetscInt N)

Input Parameters
A : the matrix
m : number of local rows (or PETSC_DECIDE)
n : number of local columns (or PETSC_DECIDE)
M : number of global rows (or PETSC_DETERMINE)
N : number of global columns (or PETSC_DETERMINE)

Python:
PETS.C.Mat.setSizes(self, size, bsize=None)
where 'size' is a tuple of 2 global sizes
or a tuple of 2 local/global pairs

Figure 33.22 MatSizes
C:
#include "petscmat.h"
PetscErrorCode MatGetSize(Mat mat,PetscInt *m,PetscInt *n)
PetscErrorCode MatGetLocalSize(Mat mat,PetscInt *m,PetscInt *n)

Python:
PETS.C.Mat.getSize(self) # tuple of global sizes
PETS.C.Mat.getLocalSize(self) # tuple of local sizes
PETS.C.Mat.getSizes(self) # tuple of local/global size tuples

Figure 33.23 MatCreateVecs
Synopsis
Get vector(s) compatible with the matrix, i.e. with the same parallel layout

#include "petscmat.h"
PetscErrorCode MatCreateVecs(Mat mat,Vec *right,Vec *left)

Collective on Mat

Input Parameter
mat - the matrix

Output Parameter;
right - (optional) vector that the matrix can be multiplied against
left - (optional) vector that the matrix vector product can be stored in
33. PETSc objects

**Figure 33.24 MatSeqAIJSetPreallocation**

```c
#include "petscmat.h"
PetscErrorCode MatSeqAIJSetPreallocation
  (Mat B,PetscInt nz,const PetscInt nnz[])
PetscErrorCode MatMPIAIJSetPreallocation
  (Mat B,PetscInt d_nz,const PetscInt d_nnz[],
   PetscInt o_nz,const PetscInt o_nnz[])
```

Input Parameters

- **B** - the matrix
- **nz/d_nz/o_nz** - number of nonzeros per row in matrix or diagonal/off-diagonal portion of local submatrix
- **nnz/d_nnz/o_nnz** - array containing the number of nonzeros in the various rows of the sequential matrix / diagonal / offdiagonal part of the local submatrix or NULL (PETSC_NULL_INTEGER in Fortran) if nz/d_nz/o_nz is used.

Python:

```python
PETSc.Mat.setPreallocationNNZ(self, [nnz_d,nnz_o])
PETSc.Mat.setPreallocationCSR(self, csr)
PETSc.Mat.setPreallocationDense(self, array)
```

Repeated allocations and re-allocations are inefficient. For this reason PETSc puts a small burden on the programmer: you need to specify a bound on how many elements the matrix will contain.

We explain this by looking at some cases. First we consider a matrix that only lives on a single process. You would then use `MatSeqAIJSetPreallocation` (figure 33.24). In the case of a tridiagonal matrix you would specify that each row has three elements:

```c
MatSeqAIJSetPreallocation(A,3, NULL);
```

If the matrix is less regular you can use the third argument to give an array of explicit row lengths:

```c
int *rowlengths;
// allocate, and then:
for (int row=0; row<nrows; row++)
  rowlengths[row] = // calculation of row length
MatSeqAIJSetPreallocation(A,NULL,rowlengths);
```

In case of a distributed matrix you need to specify this bound with respect to the block structure of the matrix. As illustrated in figure 33.2, a matrix has a diagonal part and an off-diagonal part. The diagonal part describes the matrix elements that couple elements of the input and output vector that live on this process. The off-diagonal part contains the matrix elements that are multiplied with elements not on this process, in order to compute elements that do live on this process.

The preallocation specification now has separate parameters for these diagonal and off-diagonal parts: with `MatMPIAIJSetPreallocation` (figure 33.24), you specify for both either a global upper bound on the number of nonzeros, or a detailed listing of row lengths. For the matrix of the Laplace equation, this specification would seem to be:

```c
MatMPIAIJSetPreallocation(A, 3, NULL, 2, NULL);
```
33.4. Mat: Matrices

Figure 33.2: The diagonal and off-diagonal parts of a matrix

Figure 33.25 MatSetValue
C:
#include <petscmat.h>
PetscErrorCode MatSetValue(
    Mat m, PetscInt row, PetscInt col, PetscScalar value, InsertMode mode)

Input Parameters
m : the matrix
row : the row location of the entry
col : the column location of the entry
value : the value to insert
mode : either INSERT_VALUES or ADD_VALUES

Python:
PETSc.Mat.setValue(self, row, col, value, addv=None)
also supported:
A[row, col] = value

However, this is only correct if the block structure from the parallel division equals that from the lines in the domain. In general it may be necessary to use values that are an overestimate. It is then possible to contract the storage by copying the matrix.

Specifying bounds on the number of nonzeros is often enough, and not too wasteful. However, if many rows have fewer nonzeros than these bounds, a lot of space is wasted. In that case you can replace the NULL arguments by an array that lists for each row the number of nonzeros in that row.

33.4.3 Matrix elements

You can set a single matrix element with MatSetValue (figure 33.25) or a block of them, where you supply a set of i and j indices, using MatSetValues.

After setting matrix elements, the matrix needs to be assembled. This is where PETSc moves matrix elements to the right processor, if they were specified elsewhere. As with vectors this takes two calls:
33. PETSc objects

Figure 33.26 MatAssemblyBegin
C:
#include "petscmat.h"
PetscErrorCode MatAssemblyBegin(Mat mat,MatAssemblyType type)
PetscErrorCode MatAssemblyEnd(Mat mat,MatAssemblyType type)

Input Parameters
mat- the matrix
type- type of assembly, either MAT_FLUSH_ASSEMBLY
  or MAT_FINAL_ASSEMBLY

Python:
assemble(self, assembly=None)
assemblyBegin(self, assembly=None)
assemblyEnd(self, assembly=None)

there is a class PETSc.Mat.AssemblyType:
FINAL = FINAL_ASSEMBLY = 0
FLUSH = FLUSH_ASSEMBLY = 1

MatAssemblyBegin (figure 33.26) and MatAssemblyEnd (figure 33.26) which can be used to achieve latency hiding.

Elements can either be inserted (INSERT_VALUES) or added (ADD_VALUES). You can not immediately mix these modes; to do so you need to call MatAssemblyBegin / MatAssemblyEnd with a value of MAT_FLUSH_ASSEMBLY.

PETSc sparse matrices are very flexible: you can create them empty and then start adding elements. However, this is very inefficient in execution since the OS needs to reallocate the matrix every time it grows a little. Therefore, PETSc has calls for the user to indicate how many elements the matrix will ultimately contain.

    ||MatSetOption(A, MAT_NEW_NONZERO_ALLOCATION_ERR, PETSC_FALSE)

33.4.3.1 Element access

If you absolutely need access to the matrix elements, there are routines such as MatGetRow (figure 33.27). With this, any process can request, using global row numbering, the contents of a row that it owns. (Requesting elements that are not local requires the different mechanism of taking submatrices; section 33.4.6.)

Since PETSc is geared towards sparse matrices, this returns not only the element values, but also the column numbers, as well as the mere number of stored columns. If any of these three return values are not needed, they can be unrequested by setting the parameter passed to NULL.

PETSc insists that you properly release the row again with MatRestoreRow (figure 33.27).

It is also possible to retrieve the full Compressed Row Storage (CRS) contents of the local matrix with MatDenseGetArray, MatDenseRestoreArray, MatSeqAIJGetArray, MatSeqAIJRestoreArray. (Routines MatGetArray / MatRestoreArray are deprecated.)
33.4. Mat: Matrices

Figure 33.27 MatGetRow

Synopsis:

#include "petscmat.h"
PetscErrorCode MatGetRow
(Mat mat, PetscInt row,
 PetscInt *ncols, const PetscInt *cols[], const PetscScalar *vals[])

PetscErrorCode MatRestoreRow
(Mat mat, PetscInt row,
 PetscInt *ncols, const PetscInt *cols[], const PetscScalar *vals[])

Input Parameters:
mat - the matrix
row - the row to get

Output Parameters
ncols - if not NULL, the number of nonzeros in the row
cols - if not NULL, the column numbers
vals - if not NULL, the values

33.4.4 Matrix viewers

Matrices can be 'viewed' (see section 39.2.2 for a discussion of the PetscViewer mechanism) in a variety of ways, starting with the MatView call. However, often it is more convenient to use online options such as

yourprogram -mat_view
yourprogram -mat_view draw
yourprogram -ksp_mat_view draw

where -mat_view is activated by the assembly routine, while -ksp_mat_view shows only the matrix used as operator for a KSP object. Without further option refinements this will display the matrix elements inside the sparsity pattern. Using a sub-option draw will cause the sparsity pattern to be displayed in an X11 window.

33.4.5 Matrix operations

33.4.5.1 Matrix-vector operations

In the typical application of PETSc, solving large sparse linear systems of equations with iterative methods, matrix-vector operations are most important. Foremost there is the matrix-vector product MatMult (figure 33.28) and the transpose product MatMultTranspose (figure 33.28). (In the complex case, the transpose product is not the Hermitian matrix product; for that use MatMultHermitianTranspose.)

For the BLAS gemv semantics $y \leftarrow \alpha A x + \beta y$, MatMultAdd (figure 33.29) computes $z \leftarrow A x + y$.

33.4.5.2 Matrix-matrix operations

There is a number of matrix-matrix routines such as MatMatMult.
33. PETSc objects

Figure 33.28 MatMult
Synopsis
#include "petscmat.h"
PetscErrorCode MatMult(Mat mat, Vec x, Vec y)
PetscErrorCode MatMultTranspose(Mat mat, Vec x, Vec y)

Neighbor-wise Collective on Mat

Input Parameters
mat - the matrix
x - the vector to be multiplied

Output Parameters
y - the result

Figure 33.29 MatMultAdd
Synopsis
#include "petscmat.h"
PetscErrorCode MatMultAdd(Mat mat, Vec x, Vec y, Vec z)

Neighbor-wise Collective on Mat

Input Parameters
mat - the matrix
x, y - the vectors

Output Parameters
z - the result

Notes
The vectors x and z cannot be the same.

33.4.6 Submatrices

Given a parallel matrix, there are two routines for extracting submatrices:
- MatCreateSubMatrix creates a single parallel submatrix.
- MatCreateSubMatrices creates a sequential submatrix on each process.

33.4.7 Shell matrices

In many scientific applications, a matrix stands for some operator, and we are not intrinsically interested in the matrix elements, but only in the action of the matrix on a vector. In fact, under certain circumstances it is more convenient to implement a routine that computes the matrix action than to construct the matrix explicitly.

Maybe surprisingly, solving a linear system of equations can be handled this way. The reason is that PETSc’s iterative solvers (section 36.1) only need the matrix-times-vector (and perhaps the matrix-transpose-times-vector) product.

PETSc supports this mode of working. The routine MatCreateShell (figure 33.30) declares the argument to
33.4. Mat: Matrices

Figure 33.30 MatCreateShell

```c
#include "petscmat.h"
PetscErrorCode MatCreateShell
    (MPI_Comm comm,
     PetscInt m,PetscInt n,PetscInt M,PetscInt N,
     void *ctx,Mat *A)
```

Collective

Input Parameters:
- `comm` MPI communicator
- `m` number of local rows (must be given)
- `n` number of local columns (must be given)
- `M` number of global rows (may be PETSC_DETERMINE)
- `N` number of global columns (may be PETSC_DETERMINE)
- `ctx` pointer to data needed by the shell matrix routines

Output Parameter:
- `A` the matrix

Figure 33.31 MatShellSetOperation

```c
#include "petscmat.h"
PetscErrorCode MatShellSetOperation
    (Mat mat,MatOperation op,void (*g)(void))
```

Logically Collective on Mat

Input Parameters:
- `mat` the shell matrix
- `op` the name of the operation
- `g` the function that provides the operation.

be a matrix given in operator form.

33.4.7.1 Shell operations

The next step is then to add the custom multiplication routine, which will be invoked by `MatMult: MatShellSetOperation` (figure 33.31)

The routine that implements the actual product should have the same signature as `MatMult`, accepting a matrix and two vectors. The key to realizing your own product routine lies in the 'context' argument to the create routine. With `MatShellSetContext` (figure 33.32) you pass a pointer to some structure that contains all contextual information you need. In your multiplication routine you then retrieve this with `MatShellGetContext` (figure 33.33).

What operation is specified is determined by a keyword `MATOP_<OP>` where `<OP>` is the name of the matrix routine, minus the `Mat` part, in all caps.

```c
MatCreate(comm,&A);
MatSetSizes(A,localsize,localsize,matrix_size,matrix_size);
MatSetType(A,MATSHELL);
```
33. PETSc objects

Figure 33.32 MatShellSetContext
Synopsis
#include "petscmat.h"
PetscErrorCode MatShellSetContext(Mat mat, void *ctx)

Input Parameters
mat - the shell matrix
ctx - the context

Figure 33.33 MatShellGetContext
#include "petscmat.h"
PetscErrorCode MatShellGetContext(Mat mat, void *ctx)

Not Collective

Input Parameter:
mat - the matrix, should have been created with MatCreateShell()

Output Parameter:
ctx - the user provided context

| MatSetFromOptions(A);
| MatShellSetOperation(A, MATOP_MULT, (void*)mymatmult);
| MatShellSetContext(A, (void*)Diag);
| MatSetUp(A); |

(The call to MatSetSizes needs to come before MatSetType.)

33.4.7.2 Shell context

Setting the context means passing a pointer (really: an address) to some allocated structure

```c
struct matrix_data mystruct;
MatShellSetContext(A, &mystruct);
MatSetUp(A);
```

The routine signature has this argument as a void* but it’s not necessary to cast it to that. Getting the context means that a pointer to your structure needs to be set

```c
struct matrix_data *mystruct;
MatShellGetContext(A, &mystruct);
```

Somewhat confusingly, the Get routine also has a void* argument, even though it’s really a pointer variable.

33.4.8 Multi-component matrices

For multi-component physics problems there are essentially two ways of storing the linear system

1. Grouping the physics equations together, or
2. grouping the domain nodes together.
In both cases this corresponds to a block matrix, but for a problem of $N$ nodes and 3 equations, the respective structures are:

1. $3 \times 3$ blocks of size $N$, versus
2. $N \times N$ blocks of size 3.

The first case can be pictured as

$$
\begin{pmatrix}
A_{00} & A_{01} & A_{02} \\
A_{10} & A_{11} & A_{12} \\
A_{20} & A_{21} & A_{22} \\
\end{pmatrix}
$$

and while it looks natural, there is a computational problem with it. Preconditioners for such problems often look like

$$
\begin{pmatrix}
A_{00} & A_{11} \\
A_{22} \\
\end{pmatrix}
\quad \text{or} \quad
\begin{pmatrix}
A_{00} & A_{10} & A_{11} \\
A_{20} & A_{21} & A_{22} \\
\end{pmatrix}
$$

With the block-row partitioning of PETSc’s matrices, this means at most a 50% efficiency for the preconditioner solve.

It is better to use the second scheme, which requires the `MATMPIBIJ` format, and use so-called field-split preconditioners; see section 36.1.7.3.5.

### 33.4.9 Fourier transform

The Fast Fourier Transform (FFT) can be considered a matrix-vector multiplication. PETSc supports this by letting you create a matrix with `MatCreateFFT`. This requires that you add an FFT library, such as `fftw`, at configuration time; see section 32.3.4.

FFT libraries may use padding, so vectors should be created with `MatCreateVecsFFTW`, not with an independent `VecSetSizes`.

The `fftw` library does not scale the output vector, so a forward followed by a backward pass gives a result that is too large by the vector size.

```c
// fftsine.c
ierr = VecView(signal, PETSC_VIEWER_STDOUT_WORLD);
ierr = MatMult(transform, signal, frequencies);
ierr = VecScale(frequencies, 1./Nglobal);
ierr = VecView(frequencies, PETSC_VIEWER_STDOUT_WORLD);
```

*For the full source of this example, see section 33.8.3*

One full cosine wave:

- $0.809017 + 0.587785 i$
- $-0.309017 - 0.951057 i$
- $0.309017 - 0.951057 i$
- $0.809017 - 0.587785 i$
- $-1. + 1.22465e-16 i$
- $0.809017 + 0.587785 i$
- $0.309017 + 0.951057 i$
- $-0.309017 + 0.951057 i$
- $-0.809017 + 0.587785 i$
- $-1. + 1.22465e-16 i$
Frequency $n = 1$ amplitude $\equiv 1$:

- $-3.35828e-19 + 3.26458e-18$ i
- $0. - 1.22465e-17$ i
- $-1.33873e-17 + 3.26458e-18$ i
- $7.40494e-18 + 1.56772e-17$ i
- $0. + 1.8215e-17$ i

Strangely enough, the backward pass does not need to be scaled:

```c
Vec confirm;
ierr = VecDuplicate(signal,&confirm);
ierr = MatMultTranspose(transform,frequencies,confirm);
ierr = VecAXPY(confirm,-1,signal);
PetscReal nrm;
ierr = VecNorm(confirm,NORM_2,&nrm);
PetscPrintf(MPI_COMM_WORLD,"FFT accuracy %e\n",nrm);
ierr = VecDestroy(&confirm);
```

For the full source of this example, see section 33.8.3

### 33.5 Index sets and Vector Scatters

In the PDE type of applications that PETSc was originally intended for, vector data can only be real or complex: there are no vector of integers. On the other hand, integers are used for indexing into vector, for instance for gathering boundary elements into a *halo region*, or for doing the *data transpose* of an *FFT* operation.

To support this, PETSc has the following object types:

- An **IS** object describes a set of integer indices;
- A **VecScatter** object describes the correspondence between a group of indices in an input vector and a group of indices in an output vector.

#### 33.5.1 IS: index sets

An **IS** object contains a set of **PetscInt** values. It can be created with:

- **ISCreate** for creating an empty set;
- **ISCreateStride** for a strided set;
- **ISCreateBlock** for a set of contiguous blocks, placed at an explicitly given list of starting indices;
- **ISCreateGeneral** for an explicitly given list of indices.

For example, to describe odd and even indices (on two processes):

```c
// oddeven.c
IS oddeven;
if (procid==0) {
ierr = ISCreateStride(comm,Nglobal/2,0,2,&oddeven);
} else {
ierr = ISCreateStride(comm,Nglobal/2,1,2,&oddeven);
}
```
33.5. Index sets and Vector Scatters

Figure 33.34 VecScatterCreate

Synopsis
Creates a vector scatter context. Collective on Vec

```c
#include "petscvec.h"
PetscErrorCode VecScatterCreate(Vec xin, IS ix, Vec yin, IS iy, VecScatter *newctx)
```

Input Parameters:
- `xin`: a vector that defines the layout of vectors from which we scatter
- `yin`: a vector that defines the layout of vectors to which we scatter
- `ix`: the indices of `xin` to scatter (if NULL scatters all values)
- `iy`: the indices of `yin` to hold results (if NULL fills entire vector `yin`)

Output Parameter
- `newctx`: location to store the new scatter context

For the full source of this example, see section 33.8.6

After this, there are various query and set operations on index sets.
You can read out the indices of a set by `ISGetIndices` and `ISRestoreIndices`.

33.5.2 VecScatter: all-to-all operations

A VecScatter object is a generalization of an all-to-all operation. However, unlike MPI `MPI_Alltoall`, which formulates everything in terms of local buffers, a VecScatter is more implicit in only describing indices in the input and output vectors.

The `VecScatterCreate` (figure 33.34) call has as arguments:

- An input vector. From this, the parallel layout is used; any vector being scattered from should have this same layout.
- An IS object describing what indices are being scattered; if the whole vector is rearranged, `NULL` (Fortran: `PETSC_NULL_IS`) can be given.
- An output vector. From this, the parallel layout is used; any vector being scattered into should have this same layout.
- An IS object describing what indices are being scattered into; if the whole vector is a target, `NULL` can be given.

As a simple example, the odd/even sets defined above can be used to move all components with even index to process zero, and the ones with odd index to process one:

```c
VecScatter separate;
 ierr = VecScatterCreate (in, oddeven, out, NULL, &separate);
 ierr = VecScatterBegin (separate, in, out, INSERT_VALUES, SCATTER_FORWARD);
 ierr = VecScatterEnd (separate, in, out, INSERT_VALUES, SCATTER_FORWARD);
```

For the full source of this example, see section 33.8.6

Victor Eijkhout
33. PETSc objects

Note that the index set is applied to the input vector, since it describes the components to be moved. The output vector uses \textit{NULL} since these components are placed in sequence.

\textbf{Exercise 33.3.} Modify this example so that the components are still separated odd/even, but now placed in descending order on each process.

\textbf{Exercise 33.4.} Can you extend this example so that process \( p \) receives all indices that are multiples of \( p \)? Is your solution correct if \( N_{\text{global}} \) is not a multiple of \( n_{\text{procs}} \)?

33.5.2.1 More VecScatter modes

There is an added complication, in that a \texttt{VecScatter} can have both sequential and parallel input or output vectors. Scattering onto process zero is also a popular option.

33.6 AO: Application Orderings

PETSc’s decision to partition a matrix by contiguous block rows may be a limitation in the sense an application can have a natural ordering that is different. For such cases the \texttt{AO} type can translate between the two schemes.

33.7 Partitionings

By default, PETSc uses partitioning of matrices and vectors based on consecutive blocks of variables. In regular cases that is not a bad strategy. However, for some matrices a permutation and re-division can be advantageous. For instance, one could look at the \textit{adjacency graph}, and minimize the number of \textit{edge cuts} or the sum of the \textit{edge weights}.

This functionality is not built into PETSc, but can be provided by \textit{graph partitioning packages} such as \texttt{ParMetis} or \texttt{Zoltan}. The basic object is the \texttt{MatPartitioning}, with routines for

- Create and destroy: \texttt{MatPartitioningCreate}, \texttt{MatPartitioningDestroy};
- Setting the type \texttt{MatPartitioningSetType} to an explicit partitioner, or something generated as the dual or a refinement of the current matrix;
- Apply with \texttt{MatPartitioningApply}, giving a distributed \texttt{IS} object, which can then be used in \texttt{MatCreateSubMatrix} to repartition.

Illustrative example:

```c
MatPartitioning part;
MatPartitioningCreate(comm,&part);
MatPartitioningSetType(part,MATPARTITIONINGPARMETIS);
MatPartitioningApply(part,is);
/* get new global number of each old global number */
ISPartitioningToNumbering(is,&isn);
ISBuildTwoSided(is,NULL,&isrows);
MatCreateSubMatrix(A,isrows,isrows,MAT_INITIAL_MATRIX,&perA);
```

Other scenario:
MatPartitioningSetAdjacency(part,A);
MatPartitioningSetType(part,MATPARTITIONINGHIERARCH);
MatPartitioningHierarchicalSetNcoarseparts(part,2);
MatPartitioningHierarchicalSetNfineparts(part,2);
33.8 Sources used in this chapter

33.8.1 Listing of code header

33.8.2 Listing of code examples/petsc/c/split.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

static char help[] = "Ownership example."

#include <petscvec.h>

int main(int argc,char **argv)
{
    Vec x,y; /* vectors */
    int nprocs,procid;
    PetscErrorCode ierr;

    PetscInitialize(&argc,&argv,(char*)0,help);
    MPI_Comm comm = PETSC_COMM_WORLD;
    MPI_Comm_size(comm,&nprocs);
    MPI_Comm_rank(comm,&procid);

    PetscInt N,n;
    N = 100; n = PETSC_DECIDE;
    PetscSplitOwnership(comm,&n,&N);
    PetscPrintf(comm,"Global %d, local %d\n",N,n);

    N = PETSC_DECIDE; n = 10;
    PetscSplitOwnership(comm,&n,&N);
    PetscPrintf(comm,"Global %d, local %d\n",N,n);

    return PetscFinalize();
}
```

33.8.3 Listing of code examples/petsc/c/fftsine.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

static char help[] = "FFT example."

#include <petscmat.h>

int main(int argc,char **argv)
{
    Vec x,y; /* vectors */
    int nprocs,procid;
```
PetscErrorCode ierr;

PetscInitialize(&argc,&argv,(char*)0,help);
MPI_Comm comm = PETSC_COMM_WORLD;
MPI_Comm_size(comm,&nprocs);
MPI_Comm_rank(comm,&procid);

PetscInt Nlocal = 10, Nglobal = Nlocal*nprocs;
PetscPrintf(comm,"FFT examples on N=%d n=%d\n",Nglobal,Nlocal);

Mat transform;
int dimensionality=1;
PetscInt dimensions[dimensionality]; dimensions[0] = Nglobal;
PetscPrintf(comm,\"Creating fft D=%d, dim=%d\n\",dimensionality,dimensions[0]);
ierr = MatCreateFFT(comm,dimensionality,dimensions,MATFFTW,&transform); CHKERRQ(ierr);
{
  PetscInt fft_i,fft_j;
  ierr = MatGetSize(transform,&fft_i,&fft_j); CHKERRQ(ierr);
  PetscPrintf(comm,"FFT global size %d x %d\n",fft_i,fft_j);
}

Vec signal,frequencies;
ierr = MatCreateVecsFFTW(transform,&frequencies,&signal,PETSC_NULL); CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject)signal,"signal"); CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject)frequencies,"frequencies"); CHKERRQ(ierr);
ierr = VecAssemblyBegin(signal); CHKERRQ(ierr);
ierr = VecAssemblyEnd(signal); CHKERRQ(ierr);
{
  PetscInt nlocal,nglobal;
  ierr = VecGetLocalSize(signal,&nlocal); CHKERRQ(ierr);
  ierr = VecGetSize(signal,&nglobal); CHKERRQ(ierr);
  ierr = PetscPrintf(comm,"Signal local=%d global=%d\n",nlocal,nglobal); CHKERRQ(ierr);
}

PetscInt myfirst,mylast;
ierr = VecGetOwnershipRange(signal,&myfirst,&mylast); CHKERRQ(ierr);
printf("Setting %d -- %d\n",myfirst,mylast);
for (PetscInt vecindex=0; vecindex<Nglobal; vecindex++) {
  PetscScalar
    pi = 4. * atan(1.0),
    h = 1./Nglobal,
    phi = 2* pi * vecindex * h,
    puresine = cos( phi )
#if defined(PETSC_USE_COMPLEX)
  + PETSC_i * sin(phi)
#endif
  ierr = VecSetValue(signal,vecindex,puresine,INSERT_VALUES); CHKERRQ(ierr);
}

ierr = VecAssemblyBegin(signal); CHKERRQ(ierr);
ierr = VecAssemblyEnd(signal); CHKERRQ(ierr);
ierr = VecView(signal,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
ierr = MatMult(transform,signal,frequencies); CHKERRQ(ierr);
33. PETSc objects

```c
ierr = VecScale(frequencies,1./Nglobal); CHKERRQ(ierr);
ierr = VecView(frequencies,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
{
    Vec confirm;
    ierr = VecDuplicate(signal,&confirm); CHKERRQ(ierr);
    ierr = MatMultTranspose(transform,frequencies,confirm); CHKERRQ(ierr);
    ierr = VecAXPY(confirm,-1,signal); CHKERRQ(ierr);
    PetscReal nrm;
    ierr = VecNorm(confirm,NORM_2,&nrm); CHKERRQ(ierr);
    PetscPrintf(MPI_COMM_WORLD,"FFT accuracy %e\n",nrm);
    ierr = VecDestroy(&confirm); CHKERRQ(ierr);
}
ierr = MatDestroy(&transform); CHKERRQ(ierr);
ierr = VecDestroy(&signal); CHKERRQ(ierr);
ierr = VecDestroy(&frequencies); CHKERRQ(ierr);
return PetscFinalize();
}
```

33.8.4 Listing of code examples/petsc/f/vecset.F90

Program VecSetF90

```fortran
#include <petsc/finclude/petsc.h>
use petsc
implicit none

Vec :: vector
PetscScalar,dimension(:),pointer :: elements
PetscErrorCode :: ierr
PetscInt :: globalsize
integer :: myrank,vecidx,comm
PetscScalar :: vecelt
character*80 :: msg

call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
CHKERRA(ierr)
comm = MPI_COMM_WORLD
call MPI_Comm_rank(comm,myrank,ierr)

call VecCreate(comm,vector,ierr)
call VecSetType(vector,VECMPI,ierr)
call VecSetSizes(vector,2,PETSC_DECIDE,ierr)
call VecGetSize(vector,globalsize,ierr)
if (myrank==0) then
    do vecidx=0,globalsize-1
        vecelt = vecidx
        call VecSetValue(vector,vecidx,vecelt,INSERT_VALUES,ierr)
    end do
end if
```

Parallel Computing – r428
33.8. Sources used in this chapter

Listing of code examples/petsc/f08/vecarray.F90

33.8.5 Listing of code examples/petsc/f08/vecarray.F90

program VecArray
#include <petsc/finclude/petsc.h>
use petsc
implicit none

MPI_Comm :: comm
Vec :: x,y
PetscInt :: n=1, procno
PetscScalar :: one=1.0, two=2.0, value, inprod, scaling, xnorm, ynorm
PetscScalar, dimension(:), Pointer :: &
in_array, out_array
PetscInt :: globalsize, localsize, myfirst, mylast, index
Character*80 :: message
PetscBool :: flag
PetscErrorCode :: ierr

call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
CHKERRA(ierr)
comm = PETSC_COMM_WORLD
call MPI_Comm_rank(comm,procno,ierr)

!!
!! Get a commandline argument for the size of the problem
!!
call PetscOptionsGetInt( &
    PETSC_NULL_OPTIONS,PETSC_NULL_CHARACTER, &
    "-n",n,flag,ierr)
CHKERRA(ierr)

!!
!! Create vector 'x' with a default layout
!!
call VecCreate(comm,x,ierr); CHKERRA(ierr)
call VecSetSizes(x,n,PETSC_DECIDE,ierr); CHKERRA(ierr)
call VecSetFromOptions(x,ierr); CHKERRA(ierr)
!! Set x,y to constant values
!!
call VecSet(x,one,ierr); CHKERRA(ierr)

!!
!! Make another vector, just like x
!!
call VecDuplicate(x,y,ierr); CHKERRA(ierr)

!!
!! Get arrays and operate on them
!!
call VecGetArrayReadF90(x,in_array,ierr)
call VecGetArrayF90(y,out_array,ierr)
call VecGetLocalSize(x,localsize,ierr)
do index=1,localsize
   out_array(index) = 2*in_array(index)
end do

call VecRestoreArrayReadF90(x,in_array,ierr)
call VecRestoreArrayF90(y,out_array,ierr)

!!
!! Sanity check printout
!!
call VecNorm(x,NORM_2,xnorm,ierr)
call VecNorm(y,NORM_2,ynorm,ierr)
write(message,10) xnorm,ynorm
10 format("Norm x: ",f6.3,", y: ",f6.3,"\n")
call PetscPrintf(comm,message,ierr)

!!
!! Free work space. All PETSc objects should be destroyed when they
!! are no longer needed
!!
call VecDestroy(x,ierr)
call VecDestroy(y,ierr)

call PetscFinalize(ierr);
CHKERRA(ierr)

END program VecArray

33.8.6 Listing of code examples/petsc/c/oddeven.c

#include "petscksp.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
   PetscErrorCode ierr;

   ...
MPI_Comm comm;

PetscFunctionBegin;
PetscInitialize(&argc,&args,0,0);

comm = MPI_COMM_WORLD;

int nprocs, procid;
MPI_Comm_rank(comm,&procid);
MPI_Comm_size(comm,&nprocs);
if (nprocs!=2) {
    PetscPrintf(comm,"This example only works on 2 processes, not %d\n",nprocs);
    PetscFunctionReturn(-1); }

PetscInt Nglobal = 2*nprocs;
{
    PetscInt x=1;
    ierr = PetscOptionsGetInt(PETSC_NULL,PETSC_NULL,"-x", &x,NULL); CHKERRQ(ierr);
    Nglobal *= x;
}

Vec in, out;
ierr = VecCreate(comm,&in); CHKERRQ(ierr);
ierr = VecSetType(in,VECMPI); CHKERRQ(ierr);
ierr = VecSetSizes(in,PETSC_DECIDE,Nglobal); CHKERRQ(ierr);
ierr = VecDuplicate(in,&out); CHKERRQ(ierr);
{
    PetscInt myfirst, mylast;
    ierr = VecGetOwnershipRange(in,&myfirst,&mylast); CHKERRQ(ierr);
    for (PetscInt index=myfirst; index<mylast; index++) {
        PetscScalar v = index;
        ierr = VecSetValue(in,index,v,INSERT_VALUES); CHKERRQ(ierr);
    }
    ierr = VecAssemblyBegin(in); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(in); CHKERRQ(ierr);
}

IS oddeven;
if (procid==0) {
    ierr = ISCreateStride(comm,Nglobal/2,0,2,&oddeven); CHKERRQ(ierr);
} else {
    ierr = ISCreateStride(comm,Nglobal/2,1,2,&oddeven); CHKERRQ(ierr);
}
ISView(oddeven,0);

VecScatter separate;
ierr = VecScatterCreate
    (in,oddeven,out,NULL,&separate); CHKERRQ(ierr);
ierr = VecScatterBegin
    (separate,in,out,INSERT_VALUES,SCATTER_FORWARD); CHKERRQ(ierr);
ierr = VecScatterEnd
    (separate,in,out,INSERT_VALUES,SCATTER_FORWARD); CHKERRQ(ierr);}
33. PETSc objects

```c
ierr = ISDestroy(&oddeven); CHKERRQ(ierr);
ierr = VecScatterDestroy(&separate); CHKERRQ(ierr);

ierr = VecView(in,0); CHKERRQ(ierr);
ierr = VecView(out,0); CHKERRQ(ierr);

ierr = VecDestroy(&in); CHKERRQ(ierr);
ierr = VecDestroy(&out); CHKERRQ(ierr);

return PetscFinalize();
```
Chapter 34

Grid support

PETSc’s DM objects raise the abstraction level from the linear algebra problem to the physics problem: they allow for a more direct expression of operators in terms of their domain of definition. In this section we look at the DMDA ‘distributed array’ objects, which correspond to problems defined on Cartesian grids. Distributed arrays make it easier to construct the coefficient matrix of an operator that is defined as a stencil on a 1/2/3-dimensional Cartesian grid.

The main creation routine exists in three variants that mostly differ their number of parameters. For instance, DMDACreate2d has parameters along the x,y axes. However, DMDACreate1d has no parameter for the stencil type, since in 1D those are all the same, or for the process distribution.

34.1 Grid definition

A two-dimensional grid is created with DMDACreate2d (figure 34.1)

```
DMDACreate2d( communicator, 
x_boundary,y_boundary, 
stenciltype, 
gridx,gridy, procx,procy, dof,width, 
partitionx,partitiony, 
grid);
```

- Boundary type is a value of type DMBoundaryType. Values are:
  - DM_BOUNDARY_NONE
  - DM_BOUNDARY_GHOSTED,
  - DM_BOUNDARY_PERIODIC,
- The stencil type is of type DMStencilType, with values
  - DMDA_STENCIL_BOX,
  - DMDA_STENCIL_STAR.
(See figure 34.1.)
- The gridx,gridy values are the global grid size. This can be set with commandline options -da_grid_x/y/z.
- The procx,procy variables are an explicit specification of the processor grid. Failing this specification, PETSc will try to find a distribution similar to the domain grid.
Grid support

Figure 34.1 DMDACreate2d

```c
#include "petscdmda.h"
PetscErrorCode DMDACreate2d(MPI_Comm comm,
    DMBoundaryType bx,DMBoundaryType by,DMDAStencilType stencil_type,
    PetscInt M,PetscInt N,PetscInt m,PetscInt n,PetscInt dof,
    PetscInt s,const PetscInt lx[],const PetscInt ly[],
    DM *da)
```

Input Parameters

- `comm` - MPI communicator
- `bx`, `by` - type of ghost nodes: DM_BOUNDARY_NONE, DM_BOUNDARY_GHOSTED, DM_BOUNDARY_PERIODIC.
- `stencil_type` - stencil type: DMDA_STENCIL_BOX or DMDA_STENCIL_STAR.
- `M,N` - global dimension in each direction of
- `m,n` - corresponding number of processors in each dimension (or PETSC_DECIDE)
- `dof` - number of degrees of freedom per node
- `s` - stencil width
- `lx`, `ly` - arrays containing the number of nodes in each cell along the x and y coordinates, or NULL.

Output Parameter

- `da` - the resulting distributed array object

- `dof` indicates the number of ‘degrees of freedom’, where 1 corresponds to a scalar problem.
- `width` indicates the extent of the stencil: 1 for a 5-point stencil or more general a 2nd order stencil for 2nd order PDEs, 2 for 2nd order discretizations of a 4th order PDE, et cetera.
- `partitionx`, `partitiony` are arrays giving explicit partitionings of the grid over the processors, or PETSC_NULL for default distributions.

Code:

```c
// dmrhs.c
DM grid;
 ierr = DMDACreate2d( comm,
    DM_BOUNDARY_NONE, DM_BOUNDARY_NONE,
    DMDA_STENCIL_STAR,
    100,100,
    PETSC_DECIDE, PETSC_DECIDE,
    1,
    1,
    NULL, NULL,
    &grid );
 ierr = DMSetFromOptions(grid);
 ierr = DMSetUp(grid);
 ierr = DMViewFromOptions(grid, NULL, "-dm_view");
```

After you define a DM object, each process has a contiguous subdomain out of the total grid. You can query its size and location with DMDAGetCorners, or query that and all other information with DMDAGetLocalInfo (figure 34.2), which returns an DMDALocalInfo (figure 34.3) structure.
34.1. Grid definition

Figure 34.2 DMDAGetLocalInfo

```c
#include "petscdmda.h"
PetscErrorCode DMDAGetLocalInfo(DM da, DMDALocalInfo *info)
```

Figure 34.3 DMDALocalInfo
typedef struct {
    PetscInt dim, dof, sw;
    PetscInt mx, my, mz; /* global number of grid points in each direction */
    PetscInt xs, ys, zs; /* starting point of this processor, excluding ghosts */
    PetscInt xm, ym, zm; /* number of grid points on this processor, excluding ghosts */
    PetscInt gxs, gys, gzs; /* starting point of this processor including ghosts */
    PetscInt gxm, gym, gzm; /* number of grid points on this processor including ghosts */
    DMBoundaryType bx, by, bz; /* type of ghost nodes at boundary */
    DMDAStencilType st;
    DM da;
} DMDALocalInfo;

Fortran Notes - This should be declared as

```fortran
DMDALocalInfo :: info(DMDA_LOCAL_INFO_SIZE)
```

and the entries accessed via

```fortran
info(DMDA_LOCAL_INFO_DIM)
info(DMDA_LOCAL_INFO_DOF) etc.
```

The entries bx, by, bz, st, and da are not accessible from Fortran.

(A `DMDALocalInfo` struct is the same for 1/2/3 dimensions, so certain fields may not be applicable to your specific PDE.)

34.1.1 Associated vectors

Using the fields in this structure, each process can now iterate over its own subdomain. For instance, the 'top left' corner of the owned subdomain is at xs, ys and the number of points is xm, ym (see figure 34.2), so we can iterate over the subdomain as:

```c
for (int j=info.ys; j<info.ys+info.ym; j++) {
    for (int i=info.xs; i<info.xs+info>xm; i++) {
        // actions on point i,j
    }
}
```

On each point of the domain, we describe the stencil at that point. First of all, we now have the information to compute the x, y coordinates of the domain points:

```c
PetscReal **xyarray;
DMDAVecGetArray(grid, xy, &xyarray);
for (int j=info.ys; j<info.ys+info>ym; j++) {
    for (int i=info.xs; i<info.xs+info>xm; i++) {
```
In some circumstances, we want to perform stencil operations on the vector of a DMDA grid. This requires having the *halo region*. Above, you already saw the \( gxs, gys \) and other quantities relating to the halo of each process’ subdomain.

What we need is a way to make vectors that contain these halo points.

- You can make a traditional vector corresponding to a grid with \texttt{DMCreateGlobalVector}; if you need this vector only for a short while, use \texttt{DMGetGlobalVector} and \texttt{DMRestoreGlobalVector}.
- You can make a vector including halo points with \texttt{DMCreateLocalVector}; if you need this vector only for a short while, use \texttt{DMGetLocalVector} and \texttt{DMRestoreLocalVector}.
- If you have a 'global' vector, you can make the corresponding 'local' vector, filling in its halo points, with \texttt{DMGlobalToLocal}; after operating on a local vector, you can copy its non-halo part back to a global vector with \texttt{DMLocalToGlobal}.

Here we set up a local vector for operations:

```c
Vec ghostvector;
ierr = DMGetLocalVector(grid,&ghostvector);
ierr = DMGlobalToLocal(grid,xy,INSERT_VALUES,ghostvector);
PetscReal **xyarray,**gh;
ierr = DMDAVecGetArray(grid,xy,xyarray);
ierr = DMDAVecGetArray(grid,ghostvector,gh);
// computation on the arrays
ierr = DMDAVecRestoreArray(grid,xy,xyarray);
ierr = DMDAVecRestoreArray(grid,ghostvector,gh);
ierr = DMLocalToGlobal(grid,ghostvector,INSERT_VALUES,xy);
ierr = DMLocalToGlobal(grid,ghostvector);
```

The actual operations involve some tests for the actual presence of the halo:

```c
for (int j=info.ys; j<info.ys+info.ym; j++) {
    for (int i=info.xs; i<info.xs+info.xm; i++) {
        if (info.gxs<info.xs && info.gys<info.ys)
            if (i-1>=info.gxs && i+1<info.gxs+info.gxm
                && j-1>=info.gys && j+1<info.gys+info.gym)
                xyarray[j][i] = ( gh[j-1][i] + gh[j][i-1] + gh[j][i+1] + gh[j+1][i] )
                /4. ;
```

For the full source of this example, see section 34.6.2

### 34.1.2 Associated matrix

We construct a matrix on a DMDA by constructing a stencil on every \((i,j)\) coordinate on a process:
Each matrix element row, col is a combination of two MatStencil objects. Technically, this is a struct with members i, j, k, s for the domain coordinates and the number of the field.

We could construct the columns in this row one by one, but MatSetValuesStencil can set multiple rows or columns at a time, so we construct all columns at the same time:

```c
MatStencil col[5];
PetscScalar v[5];
PetscInt ncols = 0;
/*** diagonal element ****/
  col[ncols].i = i; col[ncols].j = j;
  v[ncols++] = 4.;
/*** off diagonal elements ****/
....
```

The other 'legs' of the stencil need to be set conditionally: the connection to \((i-1, j)\) is missing on the top row of the domain, and the connection to \((i, j-1)\) is missing on the left column. In all:

```c
// grid2d.c
for (int j=info.ys; j<info.ys+info.ym; j++) {
  for (int i=info.xs; i<info.xs+info.xm; i++) {
    MatStencil row, col[5];
    PetscScalar v[5];
    PetscInt ncols = 0;
    //**** local connection: diagonal element ****/
    col[ncols].j = j; col[ncols].i = i; v[ncols++] = 4.;
    /* boundaries: top and bottom row */
    if (i>0)  {col[ncols].j = j; col[ncols].i = i-1; v[ncols++] = -1.;}
    if (i<info.mx-1) {col[ncols].j = j; col[ncols].i = i+1; v[ncols++] = -1.;}
    /* boundary left and right */
    if (j>0)  {col[ncols].j = j-1; col[ncols].i = i; v[ncols++] = -1.;}
    if (j<info.my-1) {col[ncols].j = j+1; col[ncols].i = i; v[ncols++] = -1.;}
    ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES);
  }
}
```
34. Grid support

```c
ierr = DMCreate(PETSC_COMM_WORLD, &dm); CHKERRQ(ierr);
ierr = DMSetType(dm, DMPLEX); CHKERRQ(ierr);
```

A `DMPLEX` is by default two-dimensional. Use

`plexprogram -dm_plex_dim k`

for other dimensions. In two dimensions there are three levels of cells:

- 0-cells are vertices,
- 1-cells are edges, and
- 2-cells are triangles.

The default $2 \times 2$ grid has:

```
[code/petsc/c 1087] ./plex7 -dm_view
DM Object: Sphere 1 MPI processes
type: plex
Sphere in 2 dimensions:
  0-cells: 9
  1-cells: 16
  2-cells: 8
```

For larger grids:

`plexprogram -dm_plex_box_faces 4,4`

Graphics output from

```
plexprogram -dm_view draw -draw_pause 20
plexprogram -dm_view :outputfile.tex:ascii_latex 
     -dm_plex_view_scale 4
```
34.2. General Data Management

34.2.1 Matrix from dmplex

Loop over batch of elements (e):
  Loop over element matrix entries (f, g, gc -> i, j):
    Loop over quadrature points (q):
      Make \( u_q \) and \( \text{grad} U_q \) (loops over fields, Nb, Ncomp)
      \[
      \begin{align*}
      \text{elemMat}[i,j] &+ = \psi_f^{fc}(q) g_{fc,gc}(u, \text{grad} u) \phi_g^{gc}(q) \\
      &+ \psi_f^{fc}(q) \cdot g^{(1)} w_{fc,gc,df}(u, \text{grad} u) \phi_g^{gc}(q) \\
      &+ \psi_f^{fc}(q) \cdot g^{(2)} w_{fc,gc,df}(u, \text{grad} u) \phi_g^{gc}(q) \\
      &+ \psi_f^{fc}(q) \cdot g^{(3)} w_{fc,gc,df}(u, \text{grad} u) \phi_g^{gc}(q)
      \end{align*}
      \]

\[
\text{ierr} = \text{DMPlexGetDepthStratum}(dm, 0, &vStart, &vEnd); \text{CHKERRQ} (\text{ierr});
\]
\[
\text{ierr} = \text{PetscSectionCreate}(PetscObjectComm((PetscObject) dm), &s); \text{CHKERRQ} (\text{ierr});
\text{// setup the section}
\]
\[
\text{ierr} = \text{DMSetLocalSection}(dm, s); \text{CHKERRQ} (\text{ierr});
\]
\[
\text{ierr} = \text{PetscSectionDestroy}(s); \text{CHKERRQ} (\text{ierr});
\]
\[
\text{ierr} = \text{PetscSectionSetNumFields}(s, 1); \text{CHKERRQ} (\text{ierr});
\]
\[
\text{ierr} = \text{PetscSectionSetFieldComponents}(s, 0, 1); \text{CHKERRQ} (\text{ierr});
\]
\[
\text{ierr} = \text{PetscSectionSetChart}(s, vStart, vEnd); \text{CHKERRQ} (\text{ierr});
\[
\text{// printf("start-end: %d -- %d\n", vStart, vEnd);
for (v = vStart; v < vEnd; ++v) {
  \text{ierr} = \text{PetscSectionSetDof}(s, v, 1); \text{CHKERRQ} (\text{ierr});
  \text{ierr} = \text{PetscSectionSetFieldDof}(s, v, 0, 1); \text{CHKERRQ} (\text{ierr});
}
\]
\[
\text{ierr} = \text{PetscSectionSetUp}(s); \text{CHKERRQ} (\text{ierr});
\]

Victor Eijkhout
34. Grid support

34.3 Constructing a vector on a grid

A DMDA object is a description of a grid, so we now need to concern how to construct a linear system defined on that grid.

We start with vectors: we need a solution vector and a right-hand side. Here we have two options:

1. we can build a vector from scratch that has the right structure; or
2. we can use the fact that a grid object has a vector that can be extracted.

34.3.1 Create confirming vector

If we create a vector with VecCreate and VecSetSizes, it is easy to get the global size right, but the default partitioning will probably not be conformal to the grid distribution. Also, getting the indexing scheme right is not trivial.

First of all, the local size needs to be set explicitly, using information from the DMDALocalInfo object:

```c
Vec xy;
ierr = VecCreate(comm,&xy);
ierr = VecSetType(xy,VECMPI);
PetscInt nlocal = info.xm*info.ym, nglobal = info.mx*info.my;
ierr = VecSetSizes(xy,nlocal,nglobal);
```

After this, you don’t use VecSetValues, but set elements directly in the raw array, obtained by DMDAVecGetArray:

```c
PetscReal **xyarray;
DMDAVecGetArray(grid,xy,&xyarray);
for (int j=info.ys; j<info.ys+info.ym; j++) {
    for (int i=info.xs; i<info.xs+info.xm; i++) {
        PetscReal x = i*hx, y = j*hy;
        xyarray[j][i] = x*y;
    }
}
DMDAVecRestoreArray(grid,xy,&xyarray);
```

34.3.2 Extract vector from DMDA

34.3.3 Refinement

The routine DMDASetRefinementFactor can be activated with the options -da_refine or separately -da_refine_x/y/z for the directions.

34.4 Vectors of a distributed array

A distributed array is similar to a distributed vector, so there are routines of extracting the values of the array in the form of a vector. This can be done in two ways: of ways. (The routines here actually pertain to the more general DM 'Data Management' object, but we will for now discuss them in the context of DMDA.)
1. You can create a ‘global’ vector, defined on the same communicator as the array, and which is
disjointly partitioned in the same manner. This is done with \texttt{DMCreateGlobalVector}:
\begin{verbatim}
PetscErrorCode DMCreateGlobalVector(DM dm, Vec *vec)
\end{verbatim}

2. You can create a ‘local’ vector, which is sequential and defined on \texttt{PETSC_COMM_SELF}, that has not
only the points local to the process, but also the ‘halo’ region with the extent specified in the
definition of the \texttt{DMDACreate} call. For this, use \texttt{DMCreateLocalVector}:
\begin{verbatim}
PetscErrorCode DMCreateLocalVector(DM dm, Vec *vec)
\end{verbatim}

Values can be moved between local and global vectors by:
\begin{itemize}
\item \texttt{DMGlobalToLocal}: this establishes a local vector, including ghost/halo points from a disjointly dis-
tributed global vector. (For overlapping communication and computation, use \texttt{DMGlobalToLocalBegin}
and \texttt{DMGlobalToLocalEnd}.)
\item \texttt{DMLocalToGlobal}: this copies the disjoint parts of a local vector back into a global vector. (For
overlapping communication and computation use \texttt{DMLocalToGlobalBegin} and \texttt{DMLocalToGlobalEnd}.)
\end{itemize}

### 34.5 Matrices of a distributed array

Once you have a grid, can create its associated matrix:
\begin{verbatim}
DMSetUp(grid);
DMCreateMatrix(grid,&A)
\end{verbatim}

With this subdomain information you can then start to create the coefficient matrix:
\begin{verbatim}
DM grid;
PetscInt i_first, j_first, i_local, j_local;
DMDAGetCorners(grid,&i_first,&j_first,NULL,&i_local,&j_local,NULL);
for ( PetscInt i_index=i_first; i_index<i_first+i_local; i_index++) {
  for ( PetscInt j_index=j_first; j_index<j_first+j_local; j_index++) {
    // construct coefficients for domain point (i_index,j_index)
  }
}
\end{verbatim}

Note that indexing here is in terms of the grid, not in terms of the matrix.

For a simple example, consider 1-dimensional smoothing. From \texttt{DMDAGetCorners} we need only the param-
eters in \texttt{i}-direction:
\begin{verbatim}
// grid1d.c
PetscInt i_first, i_local;
enerr = DMDAGetCorners(grid,&i_first,NULL,NULL,&i_local,NULL NULL);
for ( PetscInt i_index=i_first; i_index<i_first+i_local; i_index++) {
\end{verbatim}

For the full source of this example, see section 34.6.3

We then use a single loop to set elements for the local range in \texttt{i}-direction:
MatStencil  row = {{0},col[3] = {{0}};
PetscScalar v[3];
PetscInt  ncols = 0;
  row.i = i_index;
col[ncols].i = i_index;  v[ncols] = 2.;
  ncols++;
  if (i_index>0) {  col[ncols].i = i_index-1;  v[ncols] = 1.;  ncols++;
  }  
  if (i_index<i_global-1) {  col[ncols].i = i_index+1;  v[ncols] = 1.;  ncols++;
  }  
  ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES);

For the full source of this example, see section 34.6.3
34.6 Sources used in this chapter

34.6.1 Listing of code header

34.6.2 Listing of code examples/petsc/c/dmrhs.c

```c
#include <stdlib.h>
#include <stdio.h>
#include "petsc.h"
#include "petscdmda.h"

int main(int argc,char **argv) {

  PetscErrorCode ierr;
  ierr = PetscInitialize(&argc,&argv,0,0); CHKERRQ(ierr);

  MPI_Comm comm = MPI_COMM_WORLD;
  int nprocs,procno;
  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_rank(comm,&procno);

  /*
   * Create a 2d grid and a matrix on that grid.
   */
  DM grid;
  ierr = DMDACreate2d
    ( comm,
      DM_BOUNDARY_NONE,DM_BOUNDARY_NONE,
      DMDA_STENCIL_STAR,
      100,100,
      PETSC_DECIDE,PETSC_DECIDE,
      1,
      1,
      NULL,NULL,
      &grid
    ); CHKERRQ(ierr);
  ierr = DMSetFromOptions(grid); CHKERRQ(ierr);
  ierr = DMSetUp(grid); CHKERRQ(ierr);
  ierr = DMViewFromOptions(grid,NULL,"-dm_view"); CHKERRQ(ierr);

  /*
   * Print out how the grid is distributed over processors
   */
  DMDALocalInfo info;
  ierr = DMDAGetLocalInfo(grid,&info); CHKERRQ(ierr);
  ierr = PetscPrintf(comm,"[x%d]
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"

Victor Eijkhout

627
34. Grid support

); CHKERRQ(ierr);
 ierr = PetscSynchronizedFlush(comm,stdout); CHKERRQ(ierr);
 ierr = PetscPrintf(comm,"create
"); CHKERRQ(ierr);

Vec xy;
 ierr = VecCreate(comm,&xy); CHKERRQ(ierr);
 ierr = VecSetType(xy,VECMPI); CHKERRQ(ierr);
 PetscInt nlocal = info.xm*info.ym, nglobal = info.mx*info.my;
 ierr = VecSetSizes(xy,nlocal,nglobal); CHKERRQ(ierr);
{
  PetscReal
    hx = 1. / ( info.mx-1 ),
    hy = 1. / ( info.my-1 );
  PetscReal **xyarray;
  DMDAVecGetArray(grid,xy,&xyarray); CHKERRQ(ierr);
  for (int j=info.ys; j<info.ys+info.ym; j++) {
    for (int i=info.xs; i<info.xs+info.xm; i++) {
      PetscReal x = i*hx, y = j*hy;
      xyarray[j][i] = x*y;
    }
  }
  DMDAVecRestoreArray(grid,xy,&xyarray); CHKERRQ(ierr);
}
ierr = VecAssemblyBegin(xy); CHKERRQ(ierr);
 ierr = VecAssemblyEnd(xy); CHKERRQ(ierr);
  // ierr = VecView(xy,0); CHKERRQ(ierr);
{
  Vec ghostvector;
  ierr = DMGetLocalVector(grid,&ghostvector); CHKERRQ(ierr);
  ierr = DMGlobalToLocal(grid,xy,INSERT_VALUES,ghostvector); CHKERRQ(ierr);
  PetscReal **gh;
  ierr = DMDAVecGetArray(grid,ghostvector,&gh); CHKERRQ(ierr);
  // computation on the arrays
  for (int j=info.ys; j<info.ys+info.ym; j++) {
    for (int i=info.xs; i<info.xs+info.xm; i++) {
      PetscReal x = i*hx, y = j*hy;
      xyarray[j][i] = ( gh[j-1][i] + gh[j][i-1] + gh[j][i+1] + gh[j+1][i] )
                      /4.2;
      }  // inner loop
    }  // outer loop
  }  // ghost loop
DMDAVecRestoreArray(grid,xy,&xyarray); CHKERRQ(ierr);
DMDAVecRestoreArray(grid,ghostvector,&gh); CHKERRQ(ierr);
  ierr = DMLocalToGlobal(grid,ghostvector,INSERT_VALUES,xy); CHKERRQ(ierr);
  ierr = DMRestoreLocalVector(grid,&ghostvector); CHKERRQ(ierr);
  /* printf("x: %d+%d in %d+%d, y: %d+%d in %d+%d
", */
  /* info.xs,info.xm,info.gxs,info.gxm, */
  /* info.ys,info.ym,info.gys,info.gym */
  /* ); CHKERRQ(ierr);
  ierr = PetscSynchronizedFlush(comm,stdout); CHKERRQ(ierr);
  ierr = PetscPrintf(comm,"create
"); CHKERRQ(ierr);
}

}; CHKERRQ(ierr);
34.6. Sources used in this chapter

```c
/* info.ys,info.ym,info.gys,info.gym */
/* */
// if (! (info.ys==info.gys || info.ys==info.gys ) ) goto exit;
}

ierr = VecDestroy(&xy); CHKERRQ(ierr);
ierr = DMDestroy(&grid); CHKERRQ(ierr);

return PetscFinalize();
}

34.6.3 Listing of code examples/petsc/c/grid1d.c

#include <stdlib.h>
#include <stdio.h>
#include "petsc.h"
#include "petscdmda.h"

int main(int argc,char **argv) {

PetscErrorCode ierr;
ierr = PetscInitialize(&argc,&argv,0,0); CHKERRQ(ierr);

MPI_Comm comm = MPI_COMM_WORLD;
int nprocs,procno;
MPI_Comm_size(comm,&nprocs);
MPI_Comm_rank(comm,&procno);

PetscInt i_global = 10*nprocs;

/*
* Create a 2d grid and a matrix on that grid.
*/
DM grid;
ierr = DMDACreate1d // IN:
( comm, // collective on this communicator
  DM_BOUNDARY_NONE, // no periodicity and such
  i_global, // global size 100x100; can be changed with options
  1, // degree of freedom per node
  1, // stencil width
  NULL, // arrays of local sizes in each direction
  &grid // OUT: resulting object
); CHKERRQ(ierr);
ierr = DMDASetUp(grid); CHKERRQ(ierr);

Mat A;
ierr = DMCreateMatrix(grid,&A); CHKERRQ(ierr);

/*
* Print out how the grid is distributed over processors
*/
PetscInt i_first,i_local;
ierr = DMDAGetCorners(grid,&i_first,NULL,NULL,&i_local,NULL,NULL);CHKERRQ(ierr);
```

Victor Eijkhout
/* ierr = PetscSynchronizedPrintf */
/* (comm, */
/* "%d Local part = %d-%d x %d-%d\n", */
/* procno,info.xm,info.xs+info.xm,info.ys,info.ys+info.ym); CHKERRQ(ierr); */
/* ierr = PetscSynchronizedFlush(comm,stdout); CHKERRQ(ierr); */

/* Fill in the elements of the matrix */
for (PetscInt i_index=i_first; i_index<i_first+i_local; i_index++) {
  MatStencil row = {0}, col[3] = {{0}};
  PetscScalar v[3];
  PetscInt ncols = 0;
  row.i = i_index;
  col[ncols].i = i_index; v[ncols] = 2.;
  ncols++;
  if (i_index>0) { col[ncols].i = i_index-1; v[ncols] = 1.; ncols++; }
  if (i_index<i_global-1) { col[ncols].i = i_index+1; v[ncols] = 1.; ncols++; }
  ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES);CHKERRQ(ierr);
}

ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);

/* Create vectors on the grid */
Vec x,y;
CHKERRQ(DMCreateGlobalVector(grid,&x)); CHKERRQ(ierr);
CHKERRQ(VecDuplicate(x,&y)); CHKERRQ(ierr);

/* Set vector values: first locally, then global */
PetscReal one = 1.;
{ Vec xlocal;
  ierr = DMCreateLocalVector(grid,&xlocal); CHKERRQ(ierr);
  ierr = VecSet(xlocal,one); CHKERRQ(ierr);
  ierr = DMLocalToGlobalBegin(grid,xlocal,INSERT_VALUES,x); CHKERRQ(ierr);
  ierr = DMLocalToGlobalEnd(grid,xlocal,INSERT_VALUES,x); CHKERRQ(ierr);
  ierr = VecDestroy(&xlocal); CHKERRQ(ierr);
}

/* Solve a linear system on the grid */
KSP solver;
CHKERRQ(KSPCreate(comm,&solver)); CHKERRQ(ierr);
CHKERRQ(KSPSetType(solver,KSPBCGS)); CHKERRQ(ierr);
CHKERRQ(KSPSetOperators(solver,A,A)); CHKERRQ(ierr);
CHKERRQ(KSPSetFromOptions(solver)); CHKERRQ(ierr);
CHKERRQ(KSPSolve(solver,x,y)); CHKERRQ(ierr);
34.6. Sources used in this chapter

/************************************************************************
 * Report on success of the solver, or lack thereof
 */
{
    PetscInt its; KSPConvergedReason reason;
    ierr = KSPGetConvergedReason(solver,&reason);
    ierr = KSPGetIterationNumber(solver,&its); CHKERRQ(ierr);
    if (reason<0) {
        PetscPrintf(comm,"Failure to converge after %d iterations; reason %s\n",
        its,KSPConvergedReasons[reason]);
    } else {
        PetscPrintf(comm,"Number of iterations to convergence: %d\n",its);
    }
}

/************************************************************************
 * Clean up
 */
    ierr = KSPDestroy(&solver); CHKERRQ(ierr);
    ierr = VecDestroy(&x); CHKERRQ(ierr);
    ierr = VecDestroy(&y); CHKERRQ(ierr);
    ierr = MatDestroy(&A); CHKERRQ(ierr);
    ierr = DMDestroy(&grid); CHKERRQ(ierr);

    return PetscFinalize();
}
Star stencil
34.6. Sources used in this chapter

Figure 34.2: Illustration of various fields of the DMDALocalInfo structure
Chapter 35

Finite Elements support
35.1 Sources used in this chapter

35.1.1 Listing of code header
Chapter 36

PETSc solvers

Probably the most important activity in PETSc is solving a linear system. This is done through a solver object: an object of the class \texttt{KSP}. (This stands for Krylov SPace solver.) The solution routine \texttt{KSPSolve} takes a matrix and a right-hand-side and gives a solution; however, before you can call this some amount of setup is needed.

There two very different ways of solving a linear system: through a direct method, essentially a variant of Gaussian elimination; or through an iterative method that makes successive approximations to the solution. In PETSc there are only iterative methods. We will show how to achieve direct methods later. The default linear system solver in PETSc is fully parallel, and will work on many linear systems, but there are many settings and customizations to tailor the solver to your specific problem.

36.1 KSP: linear system solvers

36.1.1 Math background

Many scientific applications boil down to the solution of a system of linear equations at some point:

\[ Ax = b \]

The elementary textbook way of solving this is through an \textit{LU factorization}, also known as \textit{Gaussian elimination}:

\[ LU \leftarrow A, \quad Lz = b, \quad Ux = z. \]

While PETSc has support for this, its basic design is geared towards so-called iterative solution methods. Instead of directly computing the solution to the system, they compute a sequence of approximations that, with luck, converges to the true solution:

\[
\text{while not converged} \quad x_{i+1} \leftarrow f(x_i) 
\]

The interesting thing about iterative methods is that the iterative step only involves the \textit{matrix-vector product}:
36.1. KSP: linear system solvers

Figure 36.1 KSPCreate

C:
PetscErrorCode KSPCreate(MPI_Comm comm,KSP *v);

Python:
ksp = PETSc.KSP()
ksp.create()
# or:
ksp = PETSc.KSP().create()

while not converged
\[ r_i = Ax_i - b \]
\[ x_{i+1} \leftarrow f(r_i) \]

This residual is also crucial in determining whether to stop the iteration: since we (clearly) cannot measure the distance to the true solution, we use the size of the residual as a proxy measurement.

The remaining point to know is that iterative methods feature a preconditioner. Mathematically this is equivalent to transforming the linear system to

\[ M^{-1}Ax = M^{-1}b \]

so conceivably we could iterate on the transformed matrix and right-hand side. However, in practice we apply the preconditioner in each iteration:

while not converged
\[ r_i = Ax_i - b \]
\[ z_i = M^{-1}r_i \]
\[ x_{i+1} \leftarrow f(z_i) \]

In this schematic presentation we have left the nature of the \(f()\) update function unspecified. Here, many possibilities exist; the primary choice here is of the iterative method type, such as ‘conjugate gradients’, ‘generalized minimum residual’, or ‘bi-conjugate gradients stabilized’. (We will go into direct solvers in section 36.2.)

36.1.2 Solver objects

First we create a KSP object, which contains the coefficient matrix, and various parameters such as the desired accuracy, as well as method specific parameters: KSPCreate (figure 36.1).

After this, the basic scenario is:

```c
Vec rhs, sol;
KSP solver;
KSPCreate(comm, &solver);
KSPSetOperators(solver, A, A);
KSPSetFromOptions(solver);
KSPSolve(solver, rhs, sol);
KSPDestroy(&solver);
```
36. PETSc solvers

Figure 36.2 KSPSetTolerances

```c
#include "petscksp.h"
PetscErrorCode KSPSetTolerances
  (KSP ksp,PetscReal rtol,PetscReal abstol,PetscReal dtol,PetscInt maxits)
```

Logically Collective on ksp

Input Parameters:
- ksp - the Krylov subspace context
- rtol - the relative convergence tolerance, relative decrease in the
  (possibly preconditioned) residual norm
- abstol - the absolute convergence tolerance absolute size of the
  (possibly preconditioned) residual norm
- dtol - the divergence tolerance, amount (possibly preconditioned)
  residual norm can increase before KSPConvergedDefault() concludes that
  the method is diverging
- maxits - maximum number of iterations to use

Options Database Keys
- `-ksp_atol <abstol>` - Sets abstol
- `-ksp_rtol <rtol>` - Sets rtol
- `-ksp_divtol <dtol>` - Sets dtol
- `-ksp_max_it <maxits>` - Sets maxits

using various default settings. The vectors and the matrix have to be conformly partitioned. The `KSPSetOperators` call takes two operators: one is the actual coefficient matrix, and the second the one that the preconditioner is derived from. In some cases it makes sense to specify a different matrix here. (You can retrieve the operators with `KSPGetOperators`.) The call `KSPSetFromOptions` can cover almost all of the settings discussed next.

KSP objects have many options to control them, so it is convenient to call `KSPView` (or use the commandline option `-ksp_view`) to get a listing of all the settings.

36.1.3 Tolerances

Since neither solution nor solution speed is guaranteed, an iterative solver is subject to some tolerances:
- a relative tolerance for when the residual has been reduced enough;
- an absolute tolerance for when the residual is objectively small;
- a divergence tolerance that stops the iteration if the residual grows by too much; and
- a bound on the number of iterations, regardless any progress the process may still be making.

These tolerances are set with `KSPSetTolerances` (figure 36.2), or options `-ksp_atol`, `-ksp_rtol`, `-ksp_divtol`, `-ksp_max_it`. Specify to `PETSC_DEFAULT` to leave a value unaltered.

In the next section we will see how you can determine which of these tolerances caused the solver to stop.

36.1.4 Why did my solver stop? Did it work?

On return of the `KSPSolve` routine there is no guarantee that the system was successfully solved. Therefore, you need to invoke `KSPGetConvergedReason` (figure 36.3) to get a `KSPConvergedReason` parameter that
Figure 36.3 KSP::GetConvergedReason

C:
PetscErrorCode KSPGetConvergedReason
    (KSP ksp,KSPConvergedReason *reason)
Not Collective

Input Parameter
ksp -the KSP context

Output Parameter
reason -negative value indicates diverged, positive value converged,
see KSPConvergedReason

Python:
    r = KSP.getConvergedReason(self)
where r in PETSc.KSP.ConvergedReason

indicates what state the solver stopped in:

- The iteration can have successfully converged; this corresponds to reason> 0;
- the iteration can have diverged, or otherwise failed: reason< 0;
- or the iteration may have stopped at the maximum number of iterations while still making
  progress; reason= 0.

For more detail, KSPConvergedReasonView (before version 3.14: KSPReasonView) can print out the reason in
readable form; for instance

    KSPConvergedReasonView(solver,PETSC_VIEWER_STDOUT_WORLD);
    // before 3.14:
    KSPReasonView(solver,PETSC_VIEWER_STDOUT_WORLD);

(This can also be activated with the -ksp_converged_reason commandline option.)

In case of successful convergence, you can use KSP::GetIterationNumber to report how many iterations were
taken.

The following snippet analyzes the status of a KSP object that has stopped iterating:

    // shellvector.c
    PetscInt its; KSPConvergedReason reason;
    Vec Res; PetscReal norm;
    ierr = KSPGetConvergedReason(Solve,&reason);
    ierr = KSPConvergedReasonView(Solve,PETSC_VIEWER_STDOUT_WORLD);
    if (reason<0) {
        PetscPrintf(comm,"Failure to converge: reason=%d\n",reason);
    } else {
        ierr = KSPGetIterationNumber(Solve,&its);
        PetscPrintf(comm,"Number of iterations: %d\n",its);
    }

For the full source of this example, see section 36.4.2
36. PETSc solvers

Figure 36.4 KSPSetType
#include "petscksp.h"
PetscErrorCode KSPSetType(KSP ksp, KSPType type)

Logically Collective on ksp

Input Parameters:
ksp : the Krylov space context
type : a known method

Figure 36.5 KSPMatSolve
PetscErrorCode KSPMatSolve(KSP ksp, Mat B, Mat X)

Input Parameters
ksp - iterative context
B - block of right-hand sides

Output Parameter
X - block of solutions

36.1.5 Choice of iterator

There are many iterative methods, and it may take a few function calls to fully specify them. The basic routine is KSPSetType (figure 36.4), or use the option -ksp_type.

Here are some values (the full list is in petscksp.h):

- **KSPCG**: only for symmetric positive definite systems. It has a cost of both work and storage that is constant in the number of iterations. There are variants such as KSPPIPECG that are mathematically equivalent, but possibly higher performing at large scale.
- **KSPGMRES**: a minimization method that works for nonsymmetric and indefinite systems. However, to satisfy this theoretical property it needs to store the full residual history to orthogonalize each compute residual to, implying that storage is linear, and work quadratic, in the number of iterations. For this reason, GMRES is always used in a truncated variant, that regularly restarts the orthogonalization. The restart length can be set with the routine KSPGMRESSetRestart or the option -ksp_gmres_restart.
- **KSPBCGS**: a quasi-minimization method; uses less memory than GMRES.

Depending on the iterative method, there can be several routines to tune its workings. Especially if you’re still experimenting with what method to choose, it may be more convenient to specify these choices through commandline options, rather than explicitly coded routines. In that case, a single call to KSPSetFromOptions is enough to incorporate those.

36.1.6 Multiple right-hand sides

For the case of multiple right-hand sides, use KSPMatSolve (figure 36.5).
36.1.7 Preconditioners

Another part of an iterative solver is the preconditioner. The mathematical background of this is given in section 36.1.1. The preconditioner acts to make the coefficient matrix better conditioned, which will improve the convergence speed; it can even be that without a suitable preconditioner a solver will not converge at all.

36.1.7.1 Background

The mathematical requirement that the preconditioner $M$ satisfy $M \approx A$ can take two forms:

1. We form an explicit approximation to $A^{-1}$; this is known as a sparse approximate inverse.
2. We form an operator $M$ (often given in factored or other implicit) form, such that $M \approx A$, and solving a system $Mx = y$ for $x$ can be done relatively quickly.

In deciding on a preconditioner, we now have to balance the following factors.

1. What is the cost of constructing the preconditioner? This should not be more than the gain in solution time of the iterative method.
2. What is the cost per iteration of applying the preconditioner? There is clearly no point in using a preconditioner that decreases the number of iterations by a certain amount, but increases the cost per iteration much more.
3. Many preconditioners have parameter settings that make these considerations even more complicated: low parameter values may give a preconditioner that is cheaply to apply but does not improve convergence much, while large parameter values make the application more costly but decrease the number of iterations.

36.1.7.2 Usage

Unlike most of the other PETSc object types, a PC object is typically not explicitly created. Instead, it is created as part of the KSP object, and can be retrieved from it.

```c
PC prec;
KSPGetPC(solver,&prec);
PCSetType(prec,PCILU);
```

Beyond setting the type of the preconditioner, there are various type-specific routines for setting various parameters. Some of these can get quite tedious, and it is more convenient to set them through command-line options.
36. PETSc solvers

### 36.1.7.3 Types

<table>
<thead>
<tr>
<th>Method</th>
<th>PCType</th>
<th>Options Database Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>PCJACOBI</td>
<td>jacobi</td>
</tr>
<tr>
<td>Block Jacobi</td>
<td>PCBJACOBI</td>
<td>bjacobi</td>
</tr>
<tr>
<td>SOR (and SSOR)</td>
<td>PCSOR</td>
<td>sorr</td>
</tr>
<tr>
<td>SOR with Eisenstat trick</td>
<td>PCEISENSTAT</td>
<td>eisenstat</td>
</tr>
<tr>
<td>Incomplete Cholesky</td>
<td>PCICC</td>
<td>icc</td>
</tr>
<tr>
<td>Incomplete LU</td>
<td>PCILU</td>
<td>ilu</td>
</tr>
<tr>
<td>Additive Schwarz</td>
<td>PCASM</td>
<td>asm</td>
</tr>
<tr>
<td>Generalized Additive Schwarz</td>
<td>PCGASM</td>
<td>gasm</td>
</tr>
<tr>
<td>Algebraic Multigrid</td>
<td>PCGAMG</td>
<td>gamg</td>
</tr>
<tr>
<td>Balancing Domain Decomposition by Constraints Linear solver</td>
<td>PCBDDC</td>
<td>bddc</td>
</tr>
<tr>
<td>Use iterative method</td>
<td>PCKSP</td>
<td>ksp</td>
</tr>
<tr>
<td>Combination of preconditioners</td>
<td>PCCOMPOSITE</td>
<td>composite</td>
</tr>
<tr>
<td>LU</td>
<td>PCLU</td>
<td>lu</td>
</tr>
<tr>
<td>Cholesky</td>
<td>PCCHOLESKY</td>
<td>cholesky</td>
</tr>
<tr>
<td>No preconditioning</td>
<td>PCNONE</td>
<td>none</td>
</tr>
<tr>
<td>Shell for user-defined PC</td>
<td>PCSHELL</td>
<td>shell</td>
</tr>
</tbody>
</table>

Here are some of the available preconditioner types.

The Hypre package (which needs to be installed during configuration time) contains itself several preconditioners. In your code, you can set the preconditioner to PCHYPRE, and use PCHYPRESetType to one of: euclid, pilut, parasails, boomeramg, ams, ads. However, since these preconditioners themselves have options, it is usually more convenient to use commandline options:

```
-pc_type hypre -pc_hypre_type xxxx
```

#### 36.1.7.3.1 Sparse approximate inverses

The inverse of a sparse matrix (at least, those from PDEs) is typically dense. Therefore, we aim to construct a *sparse approximate inverse*.

PETSc offers two such preconditioners, both of which require an external package.

- **PCSPAI**. This is a preconditioner that can only be used in single-processor runs, or as local solver in a block preconditioner; section 36.1.7.3.3.
- As part of the PCHYPRE package, the parallel variant *parasails* is available.

```
-pc_type hypre -pc_hypre_type parasails
```

#### 36.1.7.3.2 Incomplete factorizations

The LU factorization of a matrix stemming from PDEs problems has several practical problems:

- It takes (considerably) more storage space than the coefficient matrix, and
- it correspondingly takes more time to apply.

For instance, for a three-dimensional PDE in $N$ variables, the coefficient matrix can take storage space $7N$, while the LU factorization takes $O(N^{5/3})$. 

---

Parallel Computing – r428
For this reason, often incompletely \textit{LU} factorizations are popular.

- PETSc has of itself a \texttt{PCILU} type, but this can only be used sequentially. This may sound like a limitation, but in parallel it can still be used as the subdomain solver in a block methods; section 36.1.7.3.3.
- As part of \textit{Hypre}, \texttt{pilut} is a parallel ILU.

There are many options for the ILU type, such as \texttt{PCFactorSetLevels} (option \texttt{-pc_factor_levels}), which sets the number of levels of fill-in allowed.

36.1.7.3.3 \textit{Block methods}  
Certain preconditioners seem almost intrinsically sequential. For instance, an ILU solution is sequential between the variables. There is a modest amount of parallelism, but that is hard to explore.

Taking a step back, one of the problems with parallel preconditioners lies in the cross-process connections in the matrix. If only those were not present, we could solve the linear system on each process independently. Well, since a preconditioner is an approximate solution to begin with, ignoring those connections only introduces an extra degree of approxomaticity.

There are two preconditioners that operate on this notion:

- \texttt{PCBJACOBI}: block Jacobi. Here each process solves locally the system consisting of the matrix coefficients that couple the local variables. In effect, each process solves an independent system on a subdomain.
The next question is then what solver is used on the subdomains. Here any preconditioner can be used, in particular the ones that only existed in a sequential version. Specifying all this in code gets tedious, and it is usually easier to specify such a complicated solver through commandline options:

\begin{verbatim}
-pc_type jacobi -sub_ksp_type preonly \
  -sub_pc_type ilu -sub_pc_factor_levels 1
\end{verbatim}
(Note that this also talks about a sub \texttt{ksp}: the subdomain solver is in fact a \texttt{KSP} object. By setting its type to \texttt{preonly} we state that the solver should consist of solely applying its preconditioner.)
The block Jacobi preconditioner can asymptotically only speed up the system solution by a factor relating to the number of subdomains, but in practice it can be quite valuable.

- \texttt{PCASM}: additive Schwarz method. Here each process solves locally a slightly larger system, based on the local variables, and one (or a few) levels of connections to neighboring processes. In effect, the processes solve system on overlapping subdomains. This preconditioner can asymptotically reduce the number of iterations to \(O(1)\), but that requires exact solutions on the subdomains, and in practice it may not happen anyway.

Figure 36.1 illustrates these preconditioners both in matrix and subdomain terms.

36.1.7.3.4 \textit{Multigrid preconditioners}  
- There is a Algebraic MultiGrid (AMG) type built into PETSc: \texttt{PCGAMG};
- the external packages \textit{Hypre} and \textit{ML} have AMG methods.
- There is a general Multi-Grid (MG) type: \texttt{PCMG}.
36. PETSc solvers

Figure 36.1: Illustration of block Jacobi and Additive Schwarz preconditioners: left domains and subdomains, right the corresponding submatrices

36.1.7.3.5 Field split preconditioners  For background refer to section 33.4.8.

36.1.8  Customization: monitoring and convergence tests

PETSc solvers can do various callbacks to user functions.

36.1.8.0.1 Shell preconditioners  You already saw that, in an iterative methods, the coefficient matrix can be given operationally as a shell matrix; section 33.4.7. Similarly, the preconditioner matrix can be specified operationally by specifying type PCSHELL.

This needs specification of the application routine through PCShellSetApply:

\[
\text{PCShellSetApply}(\text{PC } pc, \text{PetscErrorCode } (*\text{apply})(\text{PC}, \text{Vec}, \text{Vec}));
\]

and probably specification of a context pointer through PCShellSetContext:

\[
\text{PCShellSetContext}(\text{PC } pc, \text{void } *\text{ctx});
\]

The application function then retrieves this context with PCShellGetContext:

\[
\text{PCShellGetContext}(\text{PC } pc, \text{void } **\text{ctx});
\]

If the shell preconditioner requires setup, a routine for this can be specified with PCShellSetUp:

\[
\text{PCShellSetUp}(\text{PC } pc, \text{PetscErrorCode } (*\text{setup})(\text{PC}));
\]
36.1.8.0.2 Combining preconditioners

It is possible to combine preconditioners with PCCOMPOSITE.

```c
PCSetType(pc, PCCOMPOSITE);
PCCompositeAddPC(pc, type1);
PCCompositeAddPC(pc, type2);
```

By default, the preconditioners are applied additively; for multiplicative application

```c
PCCompositeSetType(pc, PC_COMPOSITE_MULTIPLICATIVE);
```

36.1.8.1 Convergence tests

For instance, you can set your own convergence test with KSPSetConvergenceTest.

```c
KSPSetConvergenceTest
(KSP ksp,
 PetscErrorCode (*test)(
   KSP ksp, PetscInt it, PetscReal rnorm,
   KSPConvergedReason *reason, void *ctx),
   void *ctx,
   PetscErrorCode (*destroy)(void *ctx));
```

This routines accepts

- the custom stopping test function,
- a ‘context’ void pointer to pass information to the tester, and
- optionally a custom destructor for the context information.

By default, PETSc behaves as if this function has been called with KSPConvergedDefault as argument.

36.1.8.2 Convergence monitoring

There is also a callback for monitoring each iteration. It can be set with KSPMonitorSet.

```c
KSPMonitorSet
(KSP ksp,
 PetscErrorCode (*mon)(
   KSP ksp, PetscInt it, PetscReal rnorm, void *ctx),
   void *ctx, PetscErrorCode (*mondestroy)(void **));
```

By default no monitor is set, meaning that the iteration process runs without output. The option `-ksp_monitor` activates printing a norm of the residual. This corresponds to setting KSPMonitorDefault as the monitor.

This actually outputs the ‘preconditined norm’ of the residual, which is not the L2 norm, but the square root of $r^T M^{-1} r$, a quantity that is computed in the course of the iteration process. Specifying KSPMonitorTrueResidualNorm (with corresponding option `-ksp_monitor_true_residual`) as the monitor prints the actual norm $\sqrt{r^T r}$. However, to compute this involves extra computation, since this quantity is not normally computed.
36. PETSc solvers

36.1.8.3 Auxiliary routines

KSPGetSolution KSPGetRhs KSPBuildSolution KSPBuildResidual

KSPGetSolution(KSP ksp, Vec *x);
KSPGetRhs(KSP ksp, Vec *rhs);
KSPBuildSolution(KSP ksp, Vec u, Vec *v);
KSPBuildResidual(KSP ksp, Vec t, Vec u, Vec *v);

36.2 Direct solvers

PETSc has some support for direct solvers, that is, variants of LU decomposition. In a sequential context, the PCLU preconditioner can be used for this: a direct solver is equivalent to an iterative method that stops after one preconditioner application. This can be forced by specifying a KSP type of KSPPREONLY.

Distributed direct solvers are more complicated. PETSc does not have this implemented in its basic code, but it becomes available by configuring PETSc with the scalapack library.

You need to specify which package provides the LU factorization:

PCFactorSetMatSolverType(pc, MatSolverType solver)

where the solver variable is of type MatSolverType, and can be MATSOLVERMUMPS and such when specified in source:

// direct.c
ierr = KSPCreate(comm,&Solver);
ierr = KSPSetOperators(Solver,A,A);
ierr = KSPSetType(Solver,KSPPREONLY);
{
    PC Prec;
ierr = KSPGetPC(Solver,&Prec);
ierr = PCSetType(Prec,PCLU);
ierr = PCFactorSetMatSolverType(Prec,MATSOLVERMUMPS);
}

As specified on the commandline

yourprog -ksp_type preonly -pc_type lu -pc_factor_mat_solver_type mumps

the choices are mumps, superlu, umfpack, or a number of others. Note that availability of these packages depends on how PETSc was installed on your system.

36.3 Control through command line options

From the above you may get the impression that there are lots of calls to be made to set up a PETSc linear system and solver. And what if you want to experiment with different solvers, does that mean that you have to edit a whole bunch of code? Fortunately, there is an easier way to do things. If you call the routine KSPSetFromOptions (figure 36.6) with the solver as argument, PETSc will look at your command line options and take those into account in defining the solver. Thus, you can either omit setting options in your source code, or use this as a way of quickly experimenting with different possibilities. Example:
36.3. Control through command line options

Figure 36.6 \texttt{KSPSetFromOptions}

Synopsis

\begin{verbatim}
#include "petscksp.h"
PetscErrorCode KSPSetFromOptions(KSP ksp)
\end{verbatim}

Collective on \texttt{ksp}

Input Parameters

\texttt{ksp} - the Krylov space context

\begin{verbatim}
myprogram -ksp_max_it 200 \\
 -ksp_type gmres -ksp_type_gmres_restart 20 \\
 -pc_type ilu -pc_type_ilu_levels 3
\end{verbatim}
36.4 **Sources used in this chapter**

36.4.1 **Listing of code header**

36.4.2 **Listing of code code/petsc/c**
Chapter 37

PETSC nonlinear solvers

37.1 Nonlinear systems

Nonlinear system solving means finding the zero of a general nonlinear function, that is:

\[ x : f(x) = 0 \]

with \( f : \mathbb{R}^n \rightarrow \mathbb{R}^n \). In the special case of a linear function,

\[ f(x) = Ax - b, \]

we solve this by any of the methods in chapter 36.

The general case can be solved by a number of methods, foremost Newton’s method, which iterates

\[ x_{n+1} = x_n - F(x_n)^{-1} f(x_n) \]

where \( F \) is the Hessian \( F_{ij} = \partial f_i / \partial x_j \).

You see that you need to specify two functions that are dependent on your specific problem: the objective function itself, and its Hessian.

37.1.1 Basic setup

The PETSc nonlinear solver object is of type \texttt{SNES}: ‘simple nonlinear equation solver’. As with linear solvers, we create this solver on a communicator, set its type, incorporate options, and call the solution routine \texttt{SNESSolve} (figure 37.1):

```c
Vec value_vector, solution_vector;
/* vector creation code missing */
SNES solver;
SNESCreate( comm, &solver );
SNESSetFunction( solver, value_vector, formfunction, NULL );
SNESSetFromOptions( solver );
SNESSolve( solver, NULL, solution_vector );
```

The function has the type
37. PETSC nonlinear solvers

Figure 37.1 SNESolve

```c
#include "petscsnes.h"
PetscErrorCode SNESolve(SNES snes, Vec b, Vec x)
```

Collective on SNES

Input Parameters

- **snes** - the SNES context
- **b** - the constant part of the equation \( F(x) = b \), or NULL to use zero.
- **x** - the solution vector.

```c
PetscErrorCode formfunction(SNES, Vec, Vec, void*)
```

where the parameters are:

- the solver object, so that you can access to its internal parameters
- the \( x \) value at which to evaluate the function
- the result vector \( f(x) \) for the given input
- a context pointer for further application-specific information.

Example:

```c
PetscErrorCode evaluation_function(SNES solver, Vec x, Vec fx, void *ctx) {
    const PetscReal *x_array;
    PetscReal *fx_array;
    VecGetArrayRead(fx,&fx_array);
    VecGetArray(x,&x_array);
    for (int i=0; i<localsize; i++)
        fx_array[i] = pointfunction( x_array[i] );
    VecRestoreArrayRead(fx,&fx_array);
    VecRestoreArray(x,&x_array);
}
```

Comparing the above to the introductory description you see that the Hessian is not specified here. An analytic Hessian can be dispensed with if you instruct PETSc to approximate it by finite differences:

\[
H(x)y \approx \frac{f(x + hy) - f(x)}{h}
\]

with \( h \) some finite difference. The commandline option `-snes_fd` forces the use of this finite difference approximation. However, it may lead to a large number of function evaluations. The option `-snes_fd_color` applies a coloring to the variables, leading to a drastic reduction in the number of function evaluations.

If you can form the analytic Jacobian / Hessian, you can specify it with `SNESSetJacobian` (figure 37.2), where the Jacobian is a function of type `SNESJacobianFunction` (figure 37.3).

Specifying the Jacobian:

```c
Mat J;
CHKERRQ(MatCreate(comm,&J)); CHKERRQ(ierr);
CHKERRQ(MatSetType(J,MATSEQDENSE)); CHKERRQ(ierr);
```
37.2 Time-stepping

For cases

\[ \dot{u} = G(t, u) \]

call \texttt{TSSetRHSFunction}.

\begin{verbatim}
#include "petscts.h"
PetscErrorCode TSSetRHSFunction(TS ts, Vec r,
  PetscErrorCode (*f)(TS,PetscReal,Vec,Vec,void*),
  void *ctx);
\end{verbatim}

For implicit cases

\[ F(t, u, \dot{u}) = 0 \]

call \texttt{TSSetIFunction}
#include "petscts.h"

PetscErrorCode TSSetIFunction

(TS ts, Vec r, TSIFunction f, void *ctx)
37.3 Sources used in this chapter

37.3.1 Listing of code header
Chapter 38

PETSc GPU support

38.1 Installation with GPUs

PETSc can be configured with options

--with-cuda --with-cudac=nvcc?

You can test the presence of CUDA with:

```c
// cudainstalled.c
#ifndef PETSC_HAVE_CUDA
#error "CUDA is not installed in this version of PETSC"
#endif
```

Some GPUs can accommodate MPI by being directly connected to the network through GPUDirect Remote Memory Access (RMA). If not, use this runtime option:

```c
-use_gpu_aware_mpi 0
```

More conveniently, add this to your .petsrc file; section 39.3.3.

38.2 Setup for GPU

GPUs need to be initialized. This can be done implicitly when a GPU object is created, or explicitly through PetscCUDAInitialize.

```c
// cudainit.c
ierr = PetscCUDAInitialize(comm,PETSC_DECIDE);
ierr = PetscCUDAInitializeCheck();
```

38.3 Distributed objects

Objects such as matrices and vectors need to be create explicitly with a CUDA type. After that, most PETSc calls are independent of the presence of GPUs.

Should you need to test, there is a C PreProcessor (CPP) macro PETSC_HAVE_CUDA.
38.3.1 Vectors

Analogous to vector creation as before, there are specific create calls `VecCreateSeqCUDA`, `VecCreateMPICUDAWithArray`, or the type can be set in `VecSetType`:

```c
// kspcu.c
#define PETSC_HAVE_CUDA
    ierr = VecCreateMPICUDA(comm,localsize,PETSC_DECIDE,&Rhs);
#else
    ierr = VecCreateMPI(comm,localsize,PETSC_DECIDE,&Rhs);
#endif
```

The type `VECCUDA` is sequential or parallel dependent on the run; specific types are `VECSEQCUDA`, `VECMPICUDA`.

38.3.2 Matrices

```c
    ierr = MatCreate(comm,&A);
    #ifdef PETSC_HAVE_CUDA
        ierr = MatSetType(A,MATMPIAIJCUSPARSE);
    #else
        ierr = MatSetType(A,MATMPIAIJ);
    #endif
```

Dense matrices can be created with specific calls `MatCreateDenseCUDA`, `MatCreateSeqDenseCUDA`, or by setting types `MATTDENSECUDA`, `MATTSEQDENSECUDA`, `MATMPIDENSECUDA`.

Sparse matrices: `MATAIJCUSPARSE` which is sequential or distributed depending on how the program is started. Specific types are: `MATMPIAIJCUSPARSE`, `MATSEQAIJCUSPARSE`.

38.3.3 Array access

All sorts of 'array' operations such as `MatDenseCUDAGetArray`, `VecCUDAGetArray`, `PetscMalloc` to use the GPU: `PetscMallocSetCUDAHost`, and switch back with `PetscMallocResetCUDAHost`.

38.4 Other

The memories of a CPU and GPU are not coherent. This means that routines such as `PetscMalloc1` can not immediately be used for GPU allocation. Use the routines `PetscMallocSetCUDAHost` and `PetscMallocResetCUDAHost` to switch the allocator to GPU memory and back.

```c
#define cudamatsel.c
Mat cuda_matrix;
PetscScalar *matdata;
    ierr = PetscMallocSetCUDAHost();
    ierr = PetscMalloc1(global_size*global_size,&matdata);
    ierr = PetscMallocResetCUDAHost();
    ierr = MatCreateDenseCUDA
              (comm,
               global_size,global_size,global_size,global_size,matdata,
               &cuda_matrix);
```
38.5 Sources used in this chapter

38.5.1 Listing of code header
Chapter 39

PETSc tools

39.1 Error checking and debugging

39.1.1 Debug mode

During installation (see section 32.3), there is an option of turning on debug mode. An installation with debug turned on:

- Does more runtime checks on numerics, or array indices;
- Does a memory analysis when you insert the CHKMEMQ macro (section 39.1.3);
- Has the macro PETSC_USE_DEBUG set to 1.

39.1.2 Error codes

PETSc performs a good amount of runtime error checking. Some of this is for internal consistency, but it can also detect certain mathematical errors. To facilitate error reporting, the following scheme is used.

1. Every PETSc routine is a function returning a parameter of type PetscErrorCode.
2. For a good traceback, surround the executable part of any subprogram with PetscFunctionBeginUser and PetscFunctionReturn, where the latter has the return value as parameter. (The routine PetscFunctionBegin does the same, but should only be used for PETSc library routines.)
3. Calling the macro CHKERRQ on the error code will cause an error to be printed and the current routine to be terminated. Recursively this gives a traceback of where the error occurred.

\[
\text{PetscErrorCode ierr;}
\]
\[
ierr = \text{AnyPetscRoutine( arguments ); CHKERRQ(ierr);}
\]

4. Other error checking macros are CHKERRABORT which aborts immediately, and CHKERRMPI.
5. You can effect your own error return by using SETERRQ (figure 39.1) SETERRQ1 (figure 39.1), SETERRQ2 (figure 39.1).

Fortran note 20: Error code handling. In the main program, use CHKERRA and SETERRA. Also beware that these error 'commands' are macros, and after expansion may interfere with Fortran line length, so they should only be used in .F90 files.

Example. We write a routine that sets an error:
Figure 39.1 SETERRQ

```c
#include <petscsys.h>
PetscErrorCode SETERRQ (MPI_Comm comm, PetscErrorCode ierr, char *message)
PetscErrorCode SETERRQ1(MPI_Comm comm, PetscErrorCode ierr, char *formatmessage, arg1)
PetscErrorCode SETERRQ2(MPI_Comm comm, PetscErrorCode ierr, char *formatmessage, arg1, arg2)
PetscErrorCode SETERRQ3(MPI_Comm comm, PetscErrorCode ierr, char *formatmessage, arg1, arg2, arg3)
```

Input Parameters:
- `comm` - A communicator, so that the error can be collective
- `ierr` - nonzero error code, see the list of standard error codes in include/petscerror.h
- `message` - error message in the printf format
- `arg1`, `arg2`, `arg3` - argument (for example an integer, string or double)

```c
// backtrace.c
PetscErrorCode this_function_bombs() {
    PetscFunctionBegin;
    SETERRQ(PETSC_COMM_SELF, 1, "We cannot go on like this");
    PetscFunctionReturn(0);
}
```

For the full source of this example, see section 39.6.2

Running this gives, in process zero, the output

```
[0]PETSC ERROR: We cannot go on like this
[0]PETSC ERROR: backtrace on a [computer name]
[0]PETSC ERROR: Configure options [all options]
[0]PETSC ERROR: #1 this_function_bombs() line 20 in backtrace.c
[0]PETSC ERROR: #2 main() line 30 in backtrace.c
```

Fortran note 21: Backtrace on error. In Fortran the backtrace is not quite as elegant.

```fortran
!! backtrace.F90
Subroutine this_function_bombs(ierr)
    implicit none
    integer, intent(out) :: ierr

    SETERRQ(PETSC_COMM_SELF, 1, "We cannot go on like this")
    ierr = -1

end Subroutine this_function_bombs
```

For the full source of this example, see section 39.6.3

```
[0]PETSC ERROR: ----- Error Message ------------------------------
[0]PETSC ERROR: We cannot go on like this
[.. . .]
[0]PETSC ERROR: #1 User provided function() line 0 in User file
```

Remark 32 In this example, the use of `PETSC_COMM_SELF` indicates that this error is individually generated
39.2. Program output

on a process; use `PETSC_COMM_WORLD` only if the same error would be detected everywhere.

Exercise 39.1. Look up the definition of `SETERRQ1`. Write a routine to compute square roots that is used as follows:

```c
x = 1.5; ierr = square_root(x,&rootx); CHKERRQ(ierr);
PetscPrintf(PETSC_COMM_WORLD,"Root of %f is %fn",x,rootx);
```

This should give as output:

```
Root of 1.500000 is 1.224745
[0]PETSC ERROR: ----- Error Message ----------------------------------------------
[0]PETSC ERROR: Cannot compute the root of -2.600000
[...]  
[0]PETSC ERROR: #1 square_root() line 23 in root.c 
[0]PETSC ERROR: #2 main() line 39 in root.c
```

### 39.1.3 Memory corruption

PETSc has its own memory management (section 39.5) and this facilitates finding memory corruption errors. The macro `CHKMEMQ` (in void functions) checks all memory that was allocated by PETSc, either internally or through the allocation routines, for corruption. Sprinkling this macro through your code can detect memory problems before they lead to a `segfault`.

This testing is only done if the commandline argument `-malloc_debug` (`-malloc_test` in debug mode) is supplied, so it carries no overhead for production runs.

#### 39.1.3.1 Valgrind

Valgrind is rather verbose in its output. To limit the number of processes that run under valgrind:

```
mpiexec -n 3 valgrind --track-origins=yes ./app -args : -n 5 ./app -args
```

### 39.2 Program output

PETSc has a variety of mechanisms to export or visualize program data. We will consider a few possibilities here.

#### 39.2.1 Screen I/O

Printing screen output in parallel is tricky. If two processes execute a print statement at more or less the same time there is no guarantee as to in what order they may appear on screen. (Even attempts to have them print one after the other may not result in the right ordering.) Furthermore, lines from multi-line print actions on two processes may wind up on the screen interleaved.
39. PETSc tools

Figure 39.2 PetscPrintf
C:
PetscErrorCode PetscPrintf(MPI_Comm comm, const char format[], ...)

Fortran:
PetscPrintf(MPI_Comm, character(*), PetscErrorCode ierr)

Python:
PETSc.Sys.Print(type cls, *args, **kwargs)
kwargs:
comm : communicator object

Figure 39.3 PetscSynchronizedPrintf
C:
PetscErrorCode PetscSynchronizedPrintf(
    MPI_Comm comm, const char format[], ...)

Fortran:
PetscSynchronizedPrintf(MPI_Comm, character(*), PetscErrorCode ierr)

Python:
PETSc.Sys.syncPrint(type cls, *args, **kwargs)
kwargs:
comm : communicator object
flush : if True, do synchronizedFlush
other keyword args as for python3 print function

39.2.1.1 printf replacements

PETSc has two routines that fix this problem. First of all, often the information printed is the same on all processes, so it is enough if only one process, for instance process 0, prints it. This is done with PetscPrintf (figure 39.2).

If all processes need to print, you can use PetscSynchronizedPrintf (figure 39.3) that forces the output to appear in process order.

To make sure that output is properly flushed from all system buffers use PetscSynchronizedFlush (figure 39.4) where for ordinary screen output you would use stdout for the file.

Fortran note 22: Print string construction. Fortran does not have the variable-number-of-arguments mechanism from C, so you can only use PetscPrintf on a buffer that you construct with a Write statement:

```fortran
Character*80 :: message
write(message,10) xnorm,ynorm
10 format("Norm x: ",f6.3," ",y: ",f6.3,"\n")
call PetscPrintf(comm,message,ierr)
```

Fortran note 23: Printing and newlines. The Fortran calls are only wrappers around C routines, so you can use \n newline characters in the Fortran string argument to PetscPrintf.

The file to flush is typically PETSC_STDOUT.
Figure 39.4 PetscSynchronizedFlush

C:
PetscErrorCode PetscSynchronizedFlush(MPI_Comm comm, FILE *fd)
fd : output file pointer, needs to be valid on process zero

Fortran:
PetscSynchronizedFlush(comm,fd,err)
Integer :: comm
fd is usually PETSC_STDOUT
PetscErrorCode :: err

python:
PETSc.Sys.syncFlush(type cls, comm=None)

Figure 39.5 PetscViewerRead

Synopsis

#include "petscviewer.h"
PetscErrorCode PetscViewerRead(PetscViewer viewer, void *data, PetscInt num, PetscInt *count, PetscDataType dtype)

Collective

Input Parameters
viewer - The viewer
data - Location to write the data
num - Number of items of data to read
datatype - Type of data to read

Output Parameters
count - number of items of data actually read, or NULL

Python note 40: Petsc print and python print. Since the print routines use the python print call, they automatically include the trailing newline. You don’t have to specify it as in the C calls.

39.2.1.2 scanf replacement

Using scanf in Petsc is tricky, since integers and real numbers can be of different sizes, depending on the installation. Instead, use PetscViewerRead (figure 39.5), which operates in terms of PetscDataType.

39.2.2 Viewers

In order to export PETSc matrix or vector data structures there is a PetscViewer object type. This is a quite general concept of viewing: it encompasses ascii output to screen, binary dump to file, or communication to a running Matlab process. Calls such as MatView or KSPView accept a PetscViewer argument.

In cases where this makes sense, there is also an inverse ‘load’ operation. See section 33.3.5 for vectors.

Some viewers are predefined, such as PETSC_VIEWER_STDOUT_WORLD for ascii rendering to standard out. (In C, specifying zero or NULL also uses this default viewer; for Fortran use PETSC_NULL_VIEWER.)
39. PETSc tools

39.2.2.1 Viewer types

For activities such as dumping to file you first need create the viewer with `PetscViewerCreate` and set its type with `PetscViewerSetType`.

```c
PetscViewerCreate(comm,&viewer);
PetscViewerSetType(viewer,PETSCVIEWERBINARY);
```

Popular types include PETSCVIEWERASCII, PETSCVIEWERBINARY, PETSCVIEWERSTRING, PETSCVIEWERDRAW, PETSCVIEWERSOCKET, PETSCVIEWERHDF5, PETSCVIEWERVTK; the full list can be found in include/petscviewer.h.

39.2.2.2 Viewer formats

Viewers can take further format specifications by using `PetscViewerPushFormat`:

```c
PetscViewerPushFormat
  (PETSC_VIEWER_STDOUT_WORLD,
   PETSC_VIEWER_ASCII_INFO_DETAIL);
```

and afterwards a corresponding `PetscViewerPopFormat`

Python note 41: HDF5 file generation.

```python
# hdf5.py
file_name = "hdf5.dat"
viewer = PETSc.Viewer().createHDF5(file_name, 'w', comm)
x.view(viewer)
viewer = PETSc.Viewer().createHDF5(file_name, 'r', comm)
x.load(viewer)
```

39.2.2.3 Commandline option for viewers

PETSc objects viewers can be activated by calls such as `MatView`, but often it is more convenient to do this through commandline options, such as `-mat_view`, `-vec_view`, or `-ksp_view`. By default, these output to stdout in ascii form, but this can be controlled by further option values:

Program `-mat_view binary:matrix.dat`

where binary forces a binary dump (ascii is the default) and a file name is explicitly given.

Binary dump may not be supported for all datatypes, in particular DM. For that case, do

Program `-dm_view draw`

```bash
-draw_pause 20
```

which pops up an X11 window, for the duration of the indicated pause.

If a viewer needs to be triggered at a specific location, calls such as `VecViewFromOptions` can be used. These routines all have a similar calling sequence:

```c
#include "petscsys.h"
PetscErrorCode  PetscObjectViewFromOptions(PetscObject obj, PetscObject bobj, const char *optionname[])
PetscErrorCode  VecViewFromOptions(Vec A, PetscObject obj, const char name[])
```
39.3. Commandline options

PETSc has a large number of commandline options, most of which we will discuss later. For now we only mention 
\texttt{-log_summary} which will print out profile of the time taken in various routines. For these options
to be parsed, it is necessary to pass \texttt{argc,argv} to the \texttt{PetscInitialize} call.

39.3.1 Adding your own options

You can add custom commandline options to your program. Various routines such as \texttt{PetscOptionsGetInt} scan the commandline for options and set parameters accordingly. For instance,

\begin{verbatim}
// ksp.c
PetscBool flag;
int domain_size = 100;
ierr = PetscOptionsGetInt(NULL,PETSC_NULL,\"-n\",&domain_size,&flag);
PetscPrintf(comm,\"Using domain size %d\n\",domain_size);
\end{verbatim}

39.2.2.4 Naming objects

A helpful facility for viewing is to name an object: that name will then be displayed when the object is viewed.

\begin{verbatim}
Vec i_local;
ierr = VecCreate(comm,&i_local); CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject)i_local,\"space local\"); CHKERRQ(ierr);
\end{verbatim}

giving:

Vec Object: space local 4 MPI processes
  type: mpi
Process [0]
[ ... et cetera ... ]

Victor Eijkhout 663
For the full source of this example, see section 39.6.4

declarates the existence of an option \texttt{\textasciitilde n} to be followed by an integer.

Now executing

\texttt{mpiexec yourprogram \textasciitilde n 5}

will

1. set the \texttt{flag} to true, and
2. set the parameter \texttt{domain\_size} to the value on the commandline.

Omitting the \texttt{\textasciitilde n} option will leave the default value of \texttt{domain\_size} unaltered.

For flags, use \texttt{PetscOptionsHasName}.

\textbf{Python note 42: Petsc options.} In Python, do not specify the initial hyphen of an option name. Also, the functions such as \texttt{getInt} do not return the boolean flag; if you need to test for the existence of the commandline option, use:

\begin{verbatim}
hasn = PETSc.Options().hasName("n")
\end{verbatim}

There is a related mechanism using \texttt{PetscOptionsBegin / PetscOptionsEnd}:

\begin{verbatim}
// optionsbegin.c
ierr = PetscOptionsBegin(comm,NULL,"Parameters",NULL);
ierr = PetscOptionsInt("\textasciitilde i","i value",__FILE__,__LINE__,i_value,&i_value,&i_flag);
ierr = PetscOptionsInt("\textasciitilde j","j value",__FILE__,__LINE__,j_value,&j_value,&j_flag);
ierr = PetscOptionsEnd();
if (i_flag)
   PetscPrintf(comm,"Option \textasciitilde i was used\n");
if (j_flag)
   PetscPrintf(comm,"Option \textasciitilde j was used\n");
\end{verbatim}

For the full source of this example, see section 39.6.5

The selling point for this approach is that running your code with

\texttt{mpiexec yourprogram \textasciitilde help}

will display these options as a block. Together with a ton of other options, unfortunately.

\subsection*{39.3.2 Options prefix}

In many cases, your code will have only one \texttt{KSP} solver object, so specifying \texttt{\textasciitilde ksp\_view} or \texttt{\textasciitilde ksp\_monitor} will display / trace that one. However, you may have multiple solvers, or nested solvers. You may then not want to display all of them.

As an example of the nest solver case, consider the case of a \texttt{block jacobi preconditioner}, where the block itself solved with an iterative method. You can trace that one with \texttt{\textasciitilde sub\_ksp\_monitor}.

The \texttt{sub\_} is an \textit{option prefix}, and you can defined your own with \texttt{KSPSetOptionsPrefix}. (There are similar routines for other PETSc object types.)

Example:
39.4 Timing and profiling

PETSc has a number of timing routines that make it unnecessary to use system routines such as getrusage or MPI routines such as MPI_Wtime. The main (wall clock) timer is PetscTime (figure 39.6). Note the return type of PetscLogDouble which can have a different precision from PetscReal.

The routine PetscGetCPUPtime is less useful, since it measures only time spent in computation, and ignores things such as communication.
39. **PETSc tools**

**Figure 39.7 PetscMalloc**

Synopsis
Allocates an array of memory aligned to PETSC_MEMALIGN

C:
#include <petscsys.h>
PetscErrorCode PetscMalloc1(size_t m1,type **r1)

Input Parameter:
m1 - number of elements to allocate (may be zero)

Output Parameter:
r1 - memory allocated

**Figure 39.8 PetscFree**

Synopsis
Frees memory, not collective

C:
#include <petscsys.h>
PetscErrorCode PetscFree(void *memory)

Input Parameter:
memory - memory to free (the pointer is ALWAYS set to NULL upon success)

### 39.4.1 Logging

Petsc does a lot of logging on its own operations. Additionally, you can introduce your own routines into this log.

The simplest way to display statistics is to run with an option `-log_view`. This takes an optional file name argument:

mpiexec -n 10 yourprogram -log_view :statistics.txt

The corresponding routine is **PetscLogView**.

### 39.5 Memory management

Allocate the memory for a given pointer: **PetscNew**, allocate arbitrary memory with **PetscMalloc**, allocate a number of objects with **PetscMalloc1** (figure 39.7) (this does not zero the memory allocated, use **PetscCalloc1** to obtain memory that has been zeroed); use **PetscFree** (figure 39.8) to free.

```c
PetscInt *idxs;
PetscMalloc1(10,&idxs);
// better than:
// PetscMalloc(10*sizeof(PetscInt),&idxs);
for (PetscInt i=0; i<10; i++)
    idxs[i] = f(i);
PetscFree(idxs);
```
Allocated memory is aligned to `PETSC_MEMALIGN`.

The state of memory allocation can be written to file or standard out with `PetscMallocDump`. The command-line option `-malloc_dump` outputs all not-freed memory during `PetscFinalize`.

### 39.5.1 GPU allocation

The memories of a CPU and GPU are not coherent. This means that routines such as `PetscMalloc1` can not immediately be used for GPU allocation. See section 38.4 for details.
39.6 Sources used in this chapter

39.6.1 Listing of code header

39.6.2 Listing of code examples/petsc/c/backtrace.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <petsc.h>

PetscErrorCode this_function_bombs() {
  PetscFunctionBegin;
  SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this");
  PetscFunctionReturn(0);
}

int main(int argc,char **argv)
{
  PetscErrorCode ierr;

  char help[] = "\nInit example.\n\n";
  ierr = PetscInitialize(&argc,&argv,(char*)0,help); CHKERRQ(ierr);
  ierr = this_function_bombs(); CHKERRQ(ierr);
  return PetscFinalize();
}
```

39.6.3 Listing of code examples/petsc/f/backtrace.F90

Program BackTrace

```fortran
#include <petsc/include/petsc.h>
use petsc
implicit none

PetscErrorCode :: ierr

call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
  CHKERRA(ierr)
call this_function_bombs(ierr)
  CHKERRA(ierr)
call PetscFinalize(ierr)
  CHKERRA(ierr)
contains

Subroutine this_function_bombs(ierr)
  implicit none
  integer,intent(out) :: ierr

  SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this")
```

668 Parallel Computing – r428
#include "petscksp.h"

#define __FUNCTION__ "main"

int main(int argc,char **args)
{
  PetscErrorCode ierr;
  MPI_Comm comm;
  KSP solver;
  Mat A;
  Vec Rhs,Sol;
  PetscScalar one = 1.0;

  PetscFunctionBegin;
  PetscInitialize(&argc,&args,0,0);
  comm = PETSC_COMM_SELF;

  /*
   * Read the domain size, and square it to get the matrix size
   */
  PetscBool flag;
  int matrix_size = 100;
  ierr = PetscOptionsGetInt(NULL,PETSC_NULL,"-n",&matrix_size,&flag); CHKERRQ(ierr);
  PetscPrintf(comm,"Using matrix size %d\n",matrix_size);

  /*
   * Create the five-point laplacian matrix
   */
  ierr = MatCreate(comm,&A); CHKERRQ(ierr);
  ierr = MatSetType(A,MATSEQAIJ); CHKERRQ(ierr);
  ierr = MatSetSizes(A,matrix_size,matrix_size,matrix_size,matrix_size); CHKERRQ(ierr);
  ierr = MatSeqAIJSetPreallocation(A,3,PETSC_NULL); CHKERRQ(ierr);
  ierr = MatCreateVecs(A,&Rhs,PETSC_NULL); CHKERRQ(ierr);
  for (int i=0; i<matrix_size; i++) {
    PetscScalar h = 1./(matrix_size+1), pi = 3.1415926,
    sin1 = i * pi * h, sin2 = 2 * i * pi * h, sin3 = 3 * i * pi * h,
    coefs[3] = {-1,2,-1};
    PetscInt cols[3] = {i-1,i+1};
    ierr = VecSetValue(Rhs,i,sin1 + .5 * sin2 + .3 * sin3, INSERT_VALUES); CHKERRQ(ierr);
    if (i==0) {
      ierr = MatSetValues(A,1,&i,2,cols+1,coefs+1,INSERT_VALUES); CHKERRQ(ierr);
    } else if (i!=matrix_size-1) {

  }
```c
ierr = MatSetValues(A,1,&i,2,cols,coefs,INSERT_VALUES); CHKERRQ(ierr);
} else {
  ierr = MatSetValues(A,1,&i,2,cols,coefs,INSERT_VALUES); CHKERRQ(ierr);
}
}
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
//MatView(A,PETSC_VIEWER_STDOUT_WORLD);

/*
 * Create right hand side and solution vectors
 */
ierr = VecDuplicate(Rhs,&Sol); CHKERRQ(ierr);
ierr = VecSet(Rhs,one); CHKERRQ(ierr);

/*
 * Create iterative method and preconditioner
 */
ierr = KSPCreate(comm,&solver);
ierr = KSPSetOperators(solver,A,A); CHKERRQ(ierr);
ierr = KSPSetType(solver,KSPCG); CHKERRQ(ierr);
{
  PC prec;
  ierr = KSPGetPC(solver,&prec); CHKERRQ(ierr);
ierr = PCSetType(prec,PCNONE); CHKERRQ(ierr);
}

/*
 * Incorporate any commandline options for the KSP
 */
ierr = KSPSetFromOptions(solver); CHKERRQ(ierr);

/*
 * Solve the system and analyze the outcome
 */
ierr = KSPSolve(solver,Rhs,Sol); CHKERRQ(ierr);
{
  PetscInt its; KSPConvergedReason reason;
  ierr = KSPGetConvergedReason(solver,&reason);
ierr = KSPGetIterationNumber(solver,&its); CHKERRQ(ierr);
  if (reason<0) {
    PetscPrintf
      (comm,"Failure to converge after %d iterations; reason %s\n",
        its,KSPConvergedReasons[reason]);
  } else {
    PetscPrintf
      (comm,"Number of iterations to convergence: %d\n",
        its);
  }
}

ierr = MatDestroy(&A); CHKERRQ(ierr);
ierr = KSPDestroy(&solver); CHKERRQ(ierr);
```
39.6.5 Listing of code examples/petsc/c/optionsbegin.c

```c
#include "petsc.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
    PetscErrorCode ierr;
    MPI_Comm comm;

    PetscFunctionBegin;
    PetscInitialize(&argc,&args,0,0);
    comm = MPI_COMM_WORLD;

    PetscInt i_value = 1, j_value = 1;
    PetscBool i_flag = 0, j_flag = 0;

    ierr = PetscOptionsBegin(comm,NULL,"Parameters",NULL); CHKERRQ(ierr);
    ierr = PetscOptionsInt("-i","i value",__FILE__,i_value,&i_value,&i_flag); CHKERRQ(ierr);
    ierr = PetscOptionsInt("-j","j value",__FILE__,j_value,&j_value,&j_flag); CHKERRQ(ierr);
    ierr = PetscOptionsEnd(); CHKERRQ(ierr);
    if (i_flag)
        PetscPrintf(comm,"Option `-i' was used\n");
    if (j_flag)
        PetscPrintf(comm,"Option `-j' was used\n");

    PetscPrintf(comm,"i=%d, j=%d\n",i_value,j_value);

    return PetscFinalize();
}
```
Chapter 40

PETSc topics

40.1 Communicators

PETSc has a ‘world’ communicator, which by default equals MPI_COMM_WORLD. If you want to run PETSc on a subset of processes, you can assign a subcommunicator to the variable PETSC_COMM_WORLD in between the calls to MPI_Init and PetscInitialize. Petsc communicators are of type PetscComm.
40.2 Sources used in this chapter

40.2.1 Listing of code header
PART IV

OTHER PROGRAMMING MODELS
Chapter 41

Co-array Fortran

This chapter explains the basic concepts of Co-array Fortran (CAF), and helps you get started on running your first program.

41.1 History and design

https://en.wikipedia.org/wiki/Coarray_Fortran

41.2 Compiling and running

CAF is built on the same SPMD design as MPI. Where MPI talks about processes or ranks, CAF calls the running instances of your program images.

The Intel compiler uses the flag -coarray=xxx with values single, shared, distributed gpu.

It is possible to bake the number of 'images' into the executable, but by default this is not done, and it is determined at runtime by the variable FOR_COARRAY_NUM_IMAGES.

CAF can not be mixed with OpenMP.

41.3 Basics

Co-arrays are defined by giving them, in addition to the Dimension, a Codimension

\[
\begin{align*}
\text{Complex, codimension(*) :} & \text{ number} \\
\text{Integer, dimension(,:,:), codimension[-1:1,*] :} & \text{ grid}
\end{align*}
\]

This means we are respectively declaring an array with a single number on each image, or a three-dimensional grid spread over a two-dimensional processor grid.

Traditional-like syntax can also be used:

\[
\begin{align*}
\text{Complex :} & \text{ number[*]} \\
\text{Integer :} & \text{ grid(10,20,30)[-1:1,*]}
\end{align*}
\]
Unlike Message Passing Interface (MPI), which normally only supports a linear process numbering, CAF allows for multi-dimensional process grids. The last dimension is always specified as *, meaning it is determined at runtime.

### 41.3.1 Image identification

As in other models, in CAF one can ask how many images/processes there are, and what the number of the current one is, with num_images and this_image respectively.

```f90
!! hello.F90
write(*,*) "Hello from image ", this_image(), &
"out of ", num_images()," total images"
```

For the full source of this example, see section 41.4.2.

If you call this_image with a co-array as argument, it will return the image index, as a tuple of cosubscripts, rather than a linear index. Given such a set of subscripts, image_index will return the linear index.

The functions lcobound and ucobound give the lower and upper bound on the image subscripts, as a linear index, or a tuple if called with a co-array variable.

### 41.3.2 Remote operations

The appeal of CAF is that moving data between images looks (almost) like an ordinary copy operation:

```f90
real :: x(2)[*]
integer :: p
p = this_image()
x(1)[ p+1 ] = x(2)[ p ]
```

Exchanging grid boundaries is elegantly done with array syntax:

```f90
real,Dimension( 0:N+1,0:N+1 )[*] :: grid
grid( N+1,:) [p] = grid( 0,:)[p+1]
grid( 0,:) [p] = grid( N,:)[p-1]
```

### 41.3.3 Synchronization

The fortran standard forbids race conditions:

If a variable is defined on an image in a segment, it shall not be referenced, defined or become undefined in a segment on another image unless the segments are ordered.

That is, you should not cause them to happen. The language and runtime are certainly not going to help you with that.

Well, a little. After remote updates you can synchronize images with the sync call. The easiest variant is a global synchronization:

```f90
sync all
```
Compare this to a wait call after MPI nonblocking calls.

More fine-grained, one can synchronize with specific images:

\[
\text{sync images( } / p-1,p,p+1 / \text{ )}
\]

While remote operations in CAF are nicely one-sided, synchronization is not: if image \( p \) issues a call

\[
\text{sync}(q)
\]

then \( q \) also needs to issue a mirroring call to synchronize with \( p \).

As an illustration, the following code is not a correct implementation of a ping-pong:

```fortran
!! pingpong.F90
sync all
if (procid==1) then
   number[procid+1] = number[procid]
else if (procid==2) then
   number[procid-1] = 2*number[procid]
end if
sync all
```

We can solve this with a global synchronization:

```fortran
sync all
if (procid==1) then
   number[procid+1] = number[procid]
sync all
if (procid==2) then
   number[procid-1] = 2*number[procid]
sync all
```

or a local one:

```fortran
if (procid==1) then
   number[procid+1] = number[procid]
if (procid<=2) sync images( /1,2/ )
if (procid==2) then
   number[procid-1] = 2*number[procid]
if (procid<=2) sync images( /2,1/ )
```

Note that the local sync call is done on both images involved.

Example of how you would synchronize a collective:

```fortran
if (this_image() .eq. 1 ) sync images( * )
if (this_image() .ne. 1 ) sync images( 1 )
```

Here image 1 synchronizes with all others, but the others don’t synchronize with each other.
else
    sync images( (/procid-1, procid+1/) )
end if

For the full source of this example, see section 41.4.3

41.3.4 Collectives

Collectives are not part of CAF as of the 2008 Fortran standard.
41. Co-array Fortran

41.4 Sources used in this chapter

41.4.1 Listing of code header

41.4.2 Listing of code examples/caf/f08/hello.F90

program hello_image

    write(*,*) "Hello from image ", this_image(), &
    "out of ", num_images()," total images"

end program hello_image

41.4.3 Listing of code examples/caf/f08/rightcopy.F90

program RightCopy

    integer,dimension(2),codimension[*] :: numbers
    integer :: procid,nprocs

    procid = this_image()
    nprocs = num_images()
    numbers(:,:,procid] = procid
    if (procid<nprocs) then
        numbers(1,procid+1] = procid
    end if
    if (procid==1) then
        sync images( (/procid+1/) )
    else if (procid==nprocs) then
        sync images( (/procid-1/) )
    else
        sync images( (/procid-1,procid+1/) )
    end if

    write(*,*) "After shift,,procid," has",numbers(:,:,procid]"

end program RightCopy
Chapter 42

Sycl, OneAPI, DPC++

This chapter explains the basic concepts of Sycl/Dpc++, and helps you get started on running your first program.

- **SYCL** is a C++-based language for portable parallel programming.
- **Data Parallel C++ (DPCPP)** is Intel’s extension of Sycl.
- **OneAPI** is Intel’s compiler suite, which contains the DPCPP compiler.

**Intel DPC++ extension.** The various Intel extensions are listed here: https://spec.oneapi.com/versions/latest/elements/dpcpp/source/index.html#extensions-table

42.1 Logistics

Headers:

```cpp
#include <CL/sycl.hpp>
```

You can now include namespace, but with care! If you use

```cpp
using namespace cl;
```

you have to prefix all SYCL class with `sycl::`, which is a bit of a bother. However, if you use

```cpp
using namespace cl::sycl;
```

you run into the fact that SYCL has its own versions of many Standard Template Library (STL) commands, and so you will get name collisions. The most obvious example is that the `cl::sycl` name space has its own versions of `cout` and `endl`. Therefore you have to use explicitly `std::cout` and `std::endl`. Using the wrong I/O will cause tons of inscrutable error messages. Additionally, SYCL has its own version of `free`, and of several math routines.

**Intel DPC++ extension.**

```cpp
using namespace sycl;
```
42. Platforms and devices

Since DPCPP is cross-platform, we first need to discover the devices.

First we list the platforms:

```cpp
// devices.cxx
std::vector<sycl::platform> platforms = sycl::platform::get_platforms();
for (const auto &plat : platforms) {
    // get_info is a template. So we pass the type as an `arguments`.
    std::cout << "Platform: " << plat.get_info<sycl::info::platform::name>() << " "
              << plat.get_info<sycl::info::platform::vendor>() << " "
              << plat.get_info<sycl::info::platform::version>() << std::endl;
}
```

For the full source of this example, see section 42.11.2

Then for each platform we list the devices:

```cpp
std::vector<sycl::device> devices = plat.get_devices();
for (const auto &dev : devices) {
    std::cout << "-- Device: "
              << dev.get_info<sycl::info::device::name>()
              << (dev.is_host() ? " is the host" : "")
              << (dev.is_cpu() ? " is a cpu" : "")
              << (dev.is_gpu() ? " is a gpu" : "")
              << std::endl;
}
```

For the full source of this example, see section 42.11.2

You can query what type of device you are dealing with by `is_host`, `is_cpu`, `is_gpu`.

42.3 Queues

The execution mechanism of SYCL is the *queue*: a sequence of actions that will be executed on a selected device. The only user action is submitting actions to a queue; the queue is executed at the end of the scope where it is declared.

Queue execution is asynchronous with host code.

42.3.1 Device selectors

You need to select a device on which to execute the queue. A single queue can only dispatch to a single device.

A queue is coupled to one specific device, so it can not spread work over multiple devices. You can find a default device for the queue with

```cpp
sycl::queue myqueue;
```

The following example explicitly assigns the queue to the CPU device using the `sycl::cpu_selector`.

---
42.4. Kernels

```cpp
// cpuname.cxx
sycl::queue myqueue( sycl::cpu_selector{} );

For the full source of this example, see section 42.11.3
```

The `sycl::host_selector` bypasses any devices and makes the code run on the host.

It is good for your sanity to print the name of the device you are running on:

```cpp
// devname.cxx
std::cout << myqueue.get_device().get_info<sycl::info::device::name>() << std::endl;
```

For the full source of this example, see section 42.11.4

If you try to select a device that is not available, a `sycl::runtime_error` exception will be thrown.

Intel DPC++ extension.

```cpp
#include "CL/sycl/intel/fpga_extensions.hpp"
```

42.3.2 Queue execution

It seems that queue kernels will also be executed when only they go out of scope, but not the queue:

```cpp
cpu_selector selector;
queue q(selector);
{
    q.submit( /* some kernel */ );
} // here the kernel executes
```

42.3.3 Kernel ordering

Kernels are not necessarily executed in the order in which they are submitted. You can enforce this by specifying an in-order queue:

```cpp
sycl::queue myqueue{property::queue::inorder();}
```

42.4 Kernels

One kernel per submit.

```cpp
myqueue.submit( [&] ( handler &commandgroup ) {
    commandgroup.parallel_for<uniquename>
        ( range<1>{N},
          [=] ( id<1> idx ) { ... idx }
        )
    );
```

Victor Eijkhout 683
Note that the lambda in the kernel captures by value. Capturing by reference makes no sense, since the kernel is executed on a device.

```cpp
cgh.single_task(
    [=] () {
        // kernel function is executed EXACTLY once on a SINGLE work-item
    });
```

The `submit` call results in an event object:

```cpp
auto myevent = myqueue.submit(/* stuff */);
```

This can be used for two purposes:

1. It becomes possible to wait for this specific event:

   ```cpp
   myevent.wait();
   ```

2. It can be used to indicate kernel dependencies:

   ```cpp
   myqueue.submit([=] (handler &h) {
       h.depends_on(myevent);
       /* stuff */
   });
   ```

### 42.5 Parallel operations

#### 42.5.1 Loops

```cpp
cgh.parallel_for(
    range3<3>(1024,1024,1024),
    // using 3D in this example
    [=](id<3> myID) {
        // kernel function is executed on an n-dimensional range (NDrange)
    });
```

```cpp
cgh.parallel_for(
    nd_range3<3>( {1024,1024,1024},{16,16,16} ),
    // using 3D in this example
    [=](nd_item<3> myID) {
        // kernel function is executed on an n-dimensional range (NDrange)
    });
```

```cpp
cgh.parallel_for_work_group(
    range2<2>(1024,1024),
    // using 2D in this example
    [=](group<2> myGroup) {
        // kernel function is executed once per work-group
    });
```

```cpp
grp.parallel_for_work_item(
    range1<1>(1024),
    // using 1D in this example
```

```cpp
cgh.parallel_for_work_item(
    range1<1>(1024),
    // using 1D in this example
```
42.5. Parallel operations

\[
\begin{align*}
\{&=\}(h_{\text{item}}<1> \text{myItem}) \{
// \text{kernel function is executed once per work-item}
\}\); \\
\end{align*}
\]

42.5.1 Loop bounds: ranges

SYCL adopts the modern C++ philosophy that one does not iterate over by explicitly enumerating indices, but by indicating their range. This is realized by the \texttt{range} class, which is templated over the number of space dimensions.

\[
\texttt{sycl::range<2> matrix(10,10)};
\]

Some compilers are sensitive to the type of the integer arguments:

\[
\texttt{sycl::range<1> array\{ static\_cast\_size\_t\texttt{\{}size\texttt{\}}\} ;}
\]

42.5.2 Loop indices

Kernels such as \texttt{parallel\_for} expects two arguments:
- a range over which to index; and
- a lambda of one argument: an index.

There are several ways of indexing. The \texttt{id<nd>} class of multi-dimensional indices.

\[
\begin{align*}
\texttt{myHandle.parallel\_for<\texttt{class uniqueID}>}
( \texttt{mySize},
\{&=\}( \texttt{id<1> index} \} \{ \\
\quad \texttt{float} \ x = \texttt{index.get(0)} * \texttt{h};
\quad \texttt{deviceAccessA[index]} \texttt{\} = 2.;
\} \\
) \\
\texttt{cgh.parallel\_for<\texttt{class foo}>}
( \texttt{range<1>{D\*D\*D}},
\{&=\}( \texttt{id<1> item} \} \{ \\
\quad \texttt{xx[ item[0] ] = 2 \* item[0] + 1;}
\} \\
)
\end{align*}
\]

While the C++ vectors remain one-dimensional, DPCPP allows you to make multi-dimensional buffers:

\[
\begin{align*}
\texttt{std::vector<int> y(D\*D\*D)};
\texttt{buffer<int,1> y\_buf(y.data(), range<1>{D\*D\*D});}
\texttt{cgh.parallel\_for<\texttt{class foo2D}>}
( \texttt{range<2>{D,D\*D}},
\{&=\}( \texttt{id<2> item} \} \{ \\
\quad \texttt{yy[ item[0] + D\*item[1] ] = 2;} \\
\} \\
)
\end{align*}
\]

\textit{Intel DPC++ extension.} There is an implicit conversion from the one-dimensional \texttt{sycl::id<1>} to \texttt{size\_t}, so

\textit{Victor Eijkhout}
42. Sycl, OneAPI, DPC++

```cpp
[=](sycl::id<1> i) {
    data[i] = i;
}
```

is legal, which in SYCL requires
```
||data[i[0]] = i[0];
```

### 42.5.1.3 Multi-dimensional indexing

```cpp
// stencil2d.cxx
sycl::range<2> stencil_range(N, M);
sycl::range<2> alloc_range(N + 2, M + 2);
std::vector<float>
    input(alloc_range.size()),
    output(alloc_range.size());
sycl::buffer<float, 2> input_buf(input.data(), alloc_range);
sycl::buffer<float, 2> output_buf(output.data(), alloc_range);
```

```cpp
constexpr size_t B = 4;
sycl::range<2> local_range(B, B);
sycl::range<2> tile_range = local_range + sycl::range<2>(2, 2); // Includes boundary
```

We first copy global data into an array local to the work group:
```
sycl::id<2> offset(1, 1);
h.parallel_for( // Load this tile into work-group local memory
    [=] (sycl::nd_range<2>(stencil_range, local_range, offset),
    [=] (sycl::nd_item<2> it) {
        // Load this tile into work-group local memory
        sycl::id<2> lid = it.get_local_id();
        sycl::range<2> lrange = it.get_local_range();
        for (int ti = lid[0]; ti < B + 2; ti += lrange[0]) {
            for (int tj = lid[1]; tj < B + 2; tj += lrange[1]) {
                int gi = ti + B * it.get_group(0);
                int gj = tj + B * it.get_group(1);
                tile[ti][tj] = input[gi][gj];
            }
        }
    });
```

Global coordinates in the input are computed from the nd_item’s coordinate and group:
```
[=] (sycl::nd_item<2> it) {
    for (int ti ...) {
        for (int tj ...) {
            int gi = ti + B * it.get_group(0);
            int gj = tj + B * it.get_group(1);
            ... = input[gi][gj];
        }
    }
```

Local coordinates in the tile, including boundary, I DON’T QUITE GET THIS YET.
42.5. Parallel operations

```cpp
[=] ( sycl::nd_item<2> it ) {
    sycl::id<2> lid = it.get_local_id();
    sycl::range<2> lrange = it.get_local_range();
    for (int ti = lid[0]; ti < B + 2; ti += lrange[0]) {
        for (int tj = lid[1]; tj < B + 2; tj += lrange[1]) {
            tile[ti][tj] = ..
        }
    }
```

### 42.5.2 Task dependencies

Each `submit` call can be said to correspond to a ‘task’. Since it returns a token, it becomes possible to specify task dependencies by referring to a token as a dependency in a later specified task.

```cpp
queue myQueue;
auto myTokA = myQueue.submit
    ( [&](handler& h) {
        h.parallel_for<class taskA>(...);
    } );
auto myTokB = myQueue.submit
    ( [&](handler& h) {
        h.depends_on(myTokA);
        h.parallel_for<class taskB>(...);
    } );
```

### 42.5.3 Race conditions

Sycl has the same problems with race conditions that other shared memory system have:

```cpp
// sum1d.cxx
auto array_accessor =
    array_buffer.get_access<sycl::access::mode::read>(h);
auto scalar_accessor =
    scalar_buffer.get_access<sycl::access::mode::read_write>(h);

h.parallel_for<class uniqueID>(
    array_range,
    [=](sycl::id<1> index) {
        scalar_accessor[0] += array_accessor[index];
    } ); // end of parallel for
```

To get this working correctly would need either a reduction primitive or atomics on the accumulator. The 2020 proposed standard has improved atomics.

```cpp
// reduct1d.cxx
auto input_values = array_buffer.get_access<sycl::access::mode::read>(h);
auto sum_reduction = sycl::reduction( scalar_buffer, h, std::plus<>() );

h.parallel_for
    ( array_range, sum_reduction,
      [=]( sycl::id<1> index, auto& sum )
```
42.5.4 Reductions

Reduction operations were added in the the SYCL 2020 Provisional Standard, meaning that they are not yet finalized.

Here is one approach, which works in hipsycl:

```cpp
// reducscalar.cxx
auto reduce_to_sum =
    sycl::reduction( sum_array, static_cast<float>(0.), std::plus<float>() );
myqueue.parallel_for// parallel_for<reduction_kernel<T,BinaryOp,__LINE__>>
( array_range, // sycl::range(1)(input_size),
  reduce_to_sum, // sycl::reduction(output, identity, op),
  [=] (sycl::id<1> idx, auto& reducer) { // type of reducer is impl-dependent, so use
    auto
    reducer.combine(shared_array[idx[0]]); // (input[idx[0]]);
    //reducer += shared_array[idx[0]]; // see line 216: add_reducer += input0[idx[0]];
  } ).wait();
```

Here a `sycl::reduction` object is created from the target data and the reduction operator. This is then passed to the `parallel_for` and its `combine` method is called.

### 42.6 Memory access

Memory treatment in SYCL is a little complicated, because is (at the very least) host memory and device memory, which are not necessarily coherent.

There are also three mechanisms:

- Unified Shared Memory, based on ordinary C/C++ ‘star’-pointers.
- Buffers, using the `buffer` class; this needs the `accessor` class to access the data.
- Images.

<table>
<thead>
<tr>
<th>Location</th>
<th>allocation</th>
<th>coherence</th>
<th>copy to/from device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td><code>malloc</code></td>
<td>explicit transfer</td>
<td><code>queue::memcpy</code></td>
</tr>
<tr>
<td></td>
<td><code>malloc_host</code></td>
<td>coherent host/device</td>
<td></td>
</tr>
<tr>
<td>Device</td>
<td><code>malloc_device</code></td>
<td>explicit transfer</td>
<td><code>queue::memcpy</code></td>
</tr>
<tr>
<td>Shared</td>
<td><code>malloc_shared</code></td>
<td>coherent host/device</td>
<td></td>
</tr>
</tbody>
</table>

Table 42.1: Memory types and treatments
42.6. Memory access

42.6.1 Unified shared memory

Memory allocated with `malloc_host` is visible on the host:

```c
// outshared.cxx
floattype
  *host_float = (floattype*)malloc_host(sizeof(floattype), ctx ),
  *shar_float = (floattype*)malloc_shared(sizeof(floattype), dev, ctx );
  cgh.single_task
    ( [=] () {
      shar_float[0] = 2 * host_float[0];
      sout << "Device sets " << shar_float[0] << sycl::endl;
    });
```

For the full source of this example, see section 42.11.5

Device memory is allocated with `malloc_device`, passing the queue as parameter:

```c
// reductimpl.cxx
floattype
  *host_float = (floattype*)malloc(sizeof(floattype) ),
  *devc_float = (floattype*)malloc_device(sizeof(floattype), dev, ctx );
  [&](sycl::handler &cgh) {
    cgh.memcpy(devc_float, host_float, sizeof(floattype));
  }
```

Note the corresponding `free` call that also has the queue as parameter.

Note that you need to be in a parallel task. The following gives a segmentation error:

```c
  [&](sycl::handler &cgh) {
    shar_float[0] = host_float[0];
  }
```

Ordinary memory, for instance from `malloc`, has to be copied in a kernel:

```c
  [&](sycl::handler &cgh) {
    cgh.memcpy(devc_float, host_float, sizeof(floattype));
  }
  [&](sycl::handler &cgh) {
    sycl::stream sout(1024, 256, cgh);
    cgh.single_task
      ( [=] () {
        sout << "Number " << devc_float[0] << sycl::endl;
      });
  } // end of submitted lambda
free(host_float);
sycl::free(devc_float, myqueue);
```

For the full source of this example, see section 42.11.6

42.6.2 Buffers and accessors

Arrays need to be declared in a way such that they can be access from any device.
42. Sycl, OneAPI, DPC++

```cpp
// forloop.cxx
std::vector<int> myArray(SIZE);
range<1> mySize{myArray.size()};
buffer<int, 1> bufferA(myArray.data(), myArray.size());
```

For the full source of this example, see section 42.11.7

Inside the kernel, the array is then unpacked from the buffer:

```cpp
myqueue.submit( [&] (handler &h) {
    auto deviceAccessA = bufferA.get_access<access::mode::read_write>(h);
```

For the full source of this example, see section 42.11.7

However, the `get_access` function results in a `sycl::accessor`, not a pointer to a simple type. The precise type is templated and complicated, so this is a good place to use `auto`.

Accessors can have a mode associated: `sycl::access::mode::read` `sycl::access::mode::write`

**Intel DPC++ extension.**

```cpp
array<floattype,1> leftsum{0.};
#ifdef __INTEL_CLANG_COMPILER
sycl::buffer leftbuf(leftsum);
#else
sycl::range<1> scalar{1};
sycl::buffer<floattype,1> leftbuf(leftsum.data(), scalar);
#endif
```

**Intel DPC++ extension.** there are modes

```cpp
// standard
sycl::accessor acc = buffer.get_access<sycl::access::mode::write>(h);
// dpcpp extension
sycl::accessor acc( buffer,h,sycl::read_only );
sycl::accessor acc( buffer,h,sycl::write_only );
```

### 42.6.3 Querying

The function `get_range` can query the size of either a buffer or an accessor:

```cpp
// range2.cxx
sycl::buffer<int, 2>
a_buf(a.data(), sycl::range<2>(N, M)),
b_buf(b.data(), sycl::range<2>(N, M)),
c_buf(c.data(), sycl::range<2>(N, M));
sycl::range<2>
a_range = a_buf.get_range(),
b_range = b_buf.get_range();
if (a_range==b_range) {
```

For the full source of this example, see section 42.11.8
42.7 Parallel output

There is a `sycl::cout` and `sycl::endl`.

```cpp
sycl::accessor c = c_buf.get_access<sycl::access::mode::write>(h);

sycl::range<2> c_range = c.get_range();
if (a_range==c_range) {
    h.parallel_for
        ( a_range,
          [=]( sycl::id<2> idx ) {
                c[idx] = a[idx] + b[idx];
          } );
}
```

For the full source of this example, see section 42.11.8

42.8 DPCPP extensions

Intel has made some extensions to SYCL:
- Unified Shared Memory,
- Ordered queues.

42.9 Intel devcloud notes

`qsub -I` for interactive session.
`gdb-oneapi` for debugging.

VICTOR EIJKHOUT
42.10 Examples

42.10.1 Kernels in a loop

The following idiom works:

```cpp
sycl::event last_event = queue.submit([&] (sycl::handler &h) {
    for (int iteration=0; iteration<N; iteration++) {
        last_event = queue.submit([&] (sycl::handler &h) {
            h.depends_on(last_event);
        });
    }
});
```

*For the full source of this example, see section 42.11.10*

42.10.2 Stencil computations

The problem with stencil computations is that only interior points are updated. Translated to SYCL: we need to iterate over a subrange of the range over which the buffer is defined. First let us define these ranges:

```cpp
// jacobi1d.cxx
sycl::range<1> unknowns(N);
sycl::range<1> with_boundary(N + 2);
std::vector<float> old_values(with_boundary.size(), 0.);
std::vector<float> new_values(with_boundary.size(), 0.);
old_values.back() = 1.;
new_values.back() = 1.;
```

*For the full source of this example, see section 42.11.10*

Note the boundary value 1. on the right boundary.

Restricting the iteration to the interior points is done through the `offset` parameter of the `parallel_for`:

```cpp
sycl::id<1> offset(1);
h.parallel_for(
    unknowns, offset,
    [=] (sycl::id<1> idx) {
        int i = idx[0];
        float self = old_array[i];
        float left = old_array[i - 1];
        float right = old_array[i + 1];
        new_array[i] = (self + left + right) / 3.0f;
    });
```

*For the full source of this example, see section 42.11.10*
42.11 Sources used in this chapter

42.11.1 Listing of code header

42.11.2 Listing of code code/dpcpp/cxx/devices.cxx

```cpp
#include <CL/sycl.hpp>
#include <vector>
namespace sycl = cl::sycl;

int main() {

    std::cout << "List Platforms and Devices" << std::endl;
    std::vector<sycl::platform> platforms = sycl::platform::get_platforms();
    for (const auto &plat : platforms) {
        // get_info is a template. So we pass the type as an 'arguments'.
        std::cout << "Platform: "
            << plat.get_info<sycl::info::platform::name>() << " 
            << plat.get_info<sycl::info::platform::vendor>() << " 
            << plat.get_info<sycl::info::platform::version>() << std::endl;
    }
    std::vector<sycl::device> devices = plat.get_devices();
    for (const auto &dev : devices) {
        std::cout << "Device: "
            << dev.get_info<sycl::info::device::name>()
            << (dev.is_host() ? " is the host" : "")
            << (dev.is_cpu() ? " is a cpu" : "")
            << (dev.is_gpu() ? " is a gpu" : "")
            << std::endl;
    }
}
```

42.11.3 Listing of code code/dpcpp/cxx/cpuname.cxx

```cpp
#include <CL/sycl.hpp>
#include <iostream>
#include <array>
#include <cstdio>
using namespace cl;

int main() {

    sycl::queue myqueue( sycl::cpu_selector{} );
    std::cout << myqueue.get_device().get_info<sycl::info::device::name>() << std::endl;
}
```
```cpp
#include <CL/sycl.hpp>
#include <iostream>
#include <array>
#include <cstdio>

using namespace cl;

int main() {

    #if 0
        sycl::cpu_selector selector;
        sycl::queue myqueue(selector);
    #else
        sycl::queue myqueue;
    #endif

    std::cout << myqueue.get_device().get_info<sycl::info::device::name>() << std::endl;

    return 0;
}
```

```cpp
#include <CL/sycl.hpp>
#include <vector>

namespace sycl = cl::sycl;
using floattype = float;

int main(int argc, char **argv) {

    sycl::queue myqueue;
    std::cout << "Hello example running on "
    << myqueue.get_device().get_info<sycl::info::device::name>() << std::endl;

    auto ctx = myqueue.get_context();
    auto dev = myqueue.get_device();
    floattype
        *host_float = (floattype*)malloc_host( sizeof(floattype), ctx ),
        *shar_float = (floattype*)malloc_shared( sizeof(floattype), dev, ctx );
    host_float[0] = 3.14;

    myqueue.submit
        ( [&]()(sycl::handler &cgh) {
            // WRONG shar_float[0] = host_float[0];
        });
```

42.11.4 Listing of code code/dpcpp/cxx/devname.cxx

42.11.5 Listing of code code/dpcpp/cxx/outshared.cxx
42.11. Sources used in this chapter

```cpp
sycl::stream sout(1024, 256, cgh);
cgh.single_task
( [=] () {
  shar_float[0] = 2 * host_float[0];
sout << "Device sets " << shar_float[0] << sycl::endl;
} );
} ); // end of lambda submission
myqueue.wait();

std::cout << "Host receives " << shar_float[0] << std::endl;

return 0;
}

42.11.6 Listing of code code/dpcpp/cxx/outdevice.cxx

```cpp
#include <CL/sycl.hpp>
#include <vector>
namespace sycl = cl::sycl;
using floattype = float;

int main(int argc, char **argv) {

  sycl::queue myqueue;
  std::cout << "Hello example running on " <<
    myqueue.get_device().get_info<sycl::info::device::name>() << std::endl;

  auto ctx = myqueue.get_context();
  auto dev = myqueue.get_device();
  floattype
    *host_float = (floattype*)malloc( sizeof(floattype) ),
    *devc_float = (floattype*)malloc_device( sizeof(floattype), myqueue);
  host_float[0] = 3.14;

  myqueue.submit
    ( [&](sycl::handler &cgh) {
      cgh.memcpy(devc_float, host_float, sizeof(floattype));
    } );
  myqueue.wait();

  myqueue.submit
    ( [&](sycl::handler &cgh) {
      sycl::stream sout(1024, 256, cgh);
      cgh.single_task
        ( [=] () {
          sout << "Number " << devc_float[0] << sycl::endl;
        } );
    } );
  myqueue.wait();

  // print out device id
  std::cout
```
```cpp
42.11.7 Listing of code/code/dpcpp/cxx/forloop.cxx

#include <CL/sycl.hpp>
#include <iostream>
#include <array>
#include <vector>
#include <cstdio>
#define SIZE 1024

using namespace cl::sycl;

int main() {
    std::vector<int> myArray(SIZE);
    for (int i = 0; i<SIZE; ++i)
        myArray[i] = i;
    printf("Value at start: myArray[42] is %d\n", myArray[42]);
    {
        cpu_selector selector;
        queue myqueue(selector);
        range<1> mySize{myArray.size()};
        buffer<int, 1> bufferA(myArray.data(), myArray.size());

        myqueue.submit([&] (handler &h) {
            auto deviceAccessorA = bufferA.get_access<access::mode::read_write>(h);
            parallel_for<class uniqueID>
                (mySize, [=](id<1> index) {
                    deviceAccessorA[index] *= 2;
                });
        });
    }
    printf("Value after submit: myArray[42] is %d\n", myArray[42]);
    auto hostAccessorA = bufferA.get_access<access::mode::read>();
    printf("Value through host access: myArray[42] is %d\n", myArray[42]);
}
printf("Value at finish: myArray[42] is %d\n", myArray[42]);
return 0;
```
42.11. Sources used in this chapter

42.11.8 Listing of code code/dpcpp/cxx/range2.cxx

```cpp
#include <CL/sycl.hpp>
#include <algorithm>
#include <iostream>
namespace sycl = cl::sycl;

int main() {
    // Set up queue on any available device
    sycl::queue Q;

    // Initialize input and output memory on the host
    constexpr size_t N = 256;
    constexpr size_t M = 256;
    std::vector<int> a(N * M), b(N * M), c(N * M);
    std::fill(a.begin(), a.end(), 1);
    std::fill(b.begin(), b.end(), 2);
    std::fill(c.begin(), c.end(), 0);

    {
        // Create buffers associated with inputs and output
        sycl::buffer<int, 2> a_buf(a.data(), sycl::range<2>(N, M)),
                                b_buf(b.data(), sycl::range<2>(N, M)),
                                c_buf(c.data(), sycl::range<2>(N, M));

        sycl::range<2> a_range = a_buf.get_range(),
                        b_range = b_buf.get_range();
        if (a_range==b_range) {
            // Submit the kernel to the queue
            Q.submit( [&] ( sycl::handler &h ) {
                sycl::accessor a = a_buf.get_access<sycl::access::mode::read>(h);
                sycl::accessor b = b_buf.get_access<sycl::access::mode::read>(h);
                sycl::accessor c = c_buf.get_access<sycl::access::mode::write>(h);

                sycl::range<2> c_range = c.get_range();
                if (a_range==c_range) {
                    h.parallel_for
                        ( a_range,
                          [=]( sycl::id<2> idx ) {
                            c[idx] = a[idx] + b[idx];
                        });
            });
        }
    }
}
```

Victor Eijkhout 697
42. Sycl, OneAPI, DPC++

```cpp
    } });
    }
} );
else
    throw(std::out_of_range("array size mismatch");

    } });

    // Check that all outputs match expected value
    bool passed = std::all_of(c.begin(), c.end(), [] (int i) {
        return (i == 3);
    });
    std::cout << ((passed) ? "SUCCESS" : "FAILURE") << std::endl;
    return (passed) ? 0 : 1;
```

42.11.9 Listing of code code/dpcpp/cxx/hello.cxx

```cpp
#include <CL/sycl.hpp>
#include <vector>
namespace sycl = cl::sycl;

int main(int argc, char **argv) {
    const auto global_range = 4;
    sycl::queue myQueue;

    std::cout << "Hello example running on 
      " << myQueue.get_device().get_info<sycl::info::device::name>() << std::endl;
    // Create a command_group to issue command to the group
    myQueue.submit(
        [&] (sycl::handler &cgh) {
            sycl::stream sout(1024, 256, cgh);
            cgh.parallel_for<class hello_world>
                (sycl::range<1>(global_range), [=](sycl::id<1> idx) {
                    sout << "Hello, World: World rank " << idx << sycl::endl;
                }); // End of the kernel function
        }, // End of the command_group
    )]; // End of the queue commands.
    myQueue.wait();

    return 0;
```
42.11.10 Listing of code code/dpcpp/cxx/jacobi1d.cxx

```cpp
#include <algorithm>
#include <iostream>
#include <iomanip>
#include <numeric>
#include <random>
#include <vector>
#include <CL/sycl.hpp>

namespace sycl = cl::sycl;

int main() {
    sycl::queue queue;

    const size_t N = 16;
    sycl::range<1> unknowns(N);
    sycl::range<1> with_boundary(N + 2);
    std::vector<float> old_values(with_boundary.size(), 0.),
                        new_values(with_boundary.size(), 0.);
    old_values.back() = 1.; new_values.back() = 1.;

    {
        sycl::buffer<float,1> new_buf(new_values.data(), with_boundary);
        sycl::buffer<float,1> old_buf(old_values.data(), with_boundary);

        sycl::event last_event = queue.submit( [&] (sycl::handler &h) {
            sycl::accessor old_array = old_buf.get_access<sycl::access::mode::write>(h);
            sycl::id<1> offset(1);
            h.parallel_for
                ( unknowns,offset,
                  [=] (sycl::id<1> idx) {
                      int i = idx[0];
                      old_array[i] = 0.;
                  } );
        });
        for (int iteration=0; iteration<N; iteration++) {
            last_event = queue.submit( [&] (sycl::handler &h) {
                h.depends_on(last_event);
                sycl::accessor new_array = new_buf.get_access<sycl::access::mode::write>(h);
                sycl::accessor old_array = old_buf.get_access<sycl::access::mode::read>(h);
                sycl::id<1> offset(1);
                h.parallel_for
                    ( unknowns,offset,
                      [=] (sycl::id<1> idx) {
                          int i = idx[0];
                          float self = old_array[i];
                          ...
```cpp
float left = old_array[i - 1];
float right = old_array[i + 1];
new_array[i] = (self + left + right) / 3.0f;
}
}
last_event = queue.submit([&](sycl::handler &h) {
    h.depends_on(last_event);
sycl::accessor new_array = new_buf.get_access<sycl::access::mode::read>(h);
sycl::accessor old_array = old_buf.get_access<sycl::access::mode::write>(h);
sycl::id<1> offset(1);
h.parallel_for(unknowns, offset,
    [=] (sycl::id<1> idx) {
        int i = idx[0];
        old_array[i] = new_array[i];
    });
});
queue.wait();

for (int i=0; i<with_boundary.size(); i++) {
    std::cout << std::scientific << std::setprecision(3) << new_values[i] << " ";
}
std::cout << std::endl;
return 0;
```
Chapter 43

Python multiprocessing

Python has a *multiprocessing* toolbox. This is a parallel processing library that relies on subprocesses, rather than threads.

43.1 Software and hardware

```python
## pool.py
nprocs = mp.cpu_count()
print(f"I detect {nprocs} cores")
```

43.2 Process

A process is an object that will execute a python function:

```python
## quicksort.py
import multiprocessing as mp
import random
import os

def quicksort(numbers):
    if len(numbers)==1:
        return numbers
    else:
        median = numbers[0]
        left = [ i for i in numbers if i<median ]
        right = [ i for i in numbers if i>=median ]
        with mp.Pool(2) as pool:
            [sortleft,sortright] = pool.map( quicksort,[left,right] )
        return sortleft.append( sortright )

if __name__ == '__main__':
    numbers = [ random.randint(1,50) for i in range(32) ]
    process = mp.Process(target=quicksort, args=[numbers])
    process.start()
    process.join()
```
Creating a process does not start it: for that use the `start` function. Execution of the process is not guaranteed until you call the `join` function on it:

```python
if __name__ == '__main__':
    for p in processes:
        p.start()
    for p in processes:
        p.join()
```

By making the start and join calls less regular than in a loop like this, arbitrarily complicated code can be produced.

### 43.2.1 Arguments

Arguments can be passed to the function of the process with the `args` keyword. This accepts a list (or tuple) of arguments, leading to a somewhat strange syntax for a single argument:

```python
proc = Process(target=print_func, args=(name,))
```

### 43.2.2 Process details

Note the test on `__main__`: the processes started read the current file in order to execute the function specified. Without this clause, the import would first execute more process start calls, before getting to the function execution.

Processes have a name that you can retrieve as `current_process().name`. The default is `Process-5` and such, but you can specify custom names:

```python
Process(name="Your name here")
```

The target function of a process can get hold of that process with the `current_process` function.

Of course you can also query `os.getpid()` but that does not offer any further possibilities.

```python
def say_name(iproc):
    print(f"Process {os.getpid()} has name: {mp.current_process().name}")
    if __name__ == '__main__':
        processes = [ mp.Process(target=say_name, name=f"proc{iproc}", args=[iproc])
                      for iproc in range(6) ]
```

### 43.3 Pools and mapping

Often you want a number of processes to do apply to a number of arguments, for instance in a `parameter sweep`. For this, create a `Pool` object, and apply the `map` method to it:
```
pool = mp.Pool(nprocs)
results = pool.map(print_value, range(1, 2*nprocs))
```

For the full source of this example, see section 43.5.3

Note that this is also the easiest way to get return values from a process, which is not directly possible with a `Process` object. Other approaches are using a shared object, or an object in a `Queue` or `Pipe` object; see below.

### 43.4 Shared data

The best way to deal with shared data is to create a `Value` or `Array` object, which come equipped with a lock for safe updating.

```
pi = mp.Value('d')
pi.value = 0
```

For instance, one could stochastically calculate $\pi$ by

1. generating random points in $[0,1)^2$, and
2. recording how many fall in the unit circle, after which
3. $\pi$ is $4 \times$ the ratio between points in the circle and the total number of points.

```
# pi.py
def calc_pi1(pi,n):
    for i in range(n):
        x = random.random()
        y = random.random()
        with pi.get_lock():
            if x*x+y*y<1:
                pi.value += 1.
```

**Exercise 43.1.** Do you see a way to improve the speed of this calculation?

#### 43.4.1 Pipes

A pipe, object type `Pipe`, corresponds to what used to be called a `channel` in older parallel programming systems: a First-In / First-Out (FIFO) object into which one process can place items, and from which another process can take them. However, a pipe is not associated with any particular pair: creating the pipe gives the entrance and exit from the pipe

```
q_entrance, q_exit = mp.Pipe()
```

And they can be passed to any process

```
producer1 = mp.Process(target=add_to_pipe, args=[1,q_entrance])
producer2 = mp.Process(target=add_to_pipe, args=[2,q_entrance])
printer = mp.Process(target=print_from_pipe, args=[q_exit])
```

which can then can put and get items, using the `send` and `recv` commands.
43. Python multiprocessing

```python
# pipemulti.py

def add_to_pipe(v, q):
    for i in range(10):
        print(f"put {v}")
        q.send(v)
        time.sleep(1)
    q.send("END")

def print_from_pipe(q):
    ends = 0
    while True:
        v = q.recv()
        print(f"Got: {v}")
        if v == "END":
            ends += 1
        if ends == 2:
            break
    print("pipe is empty")
```

43.4.2 Queues
43.5 Sources used in this chapter

43.5.1 Listing of code header

43.5.2 Listing of code code/multiprocessing/p/hello.py

```python
import multiprocessing as mp
import os

def say_hello(iproc):
    print(f"Process has input value: {iproc}\n")

if __name__ == '__main__':
    processes = [ mp.Process(target=say_hello,args=[iproc])
                  for iproc in range(6) ]

    for p in processes:
        p.start()

    for p in processes:
        p.join()

def say_name(iproc):
    print(f"Process {os.getpid()} has name: {mp.current_process().name}\")

if __name__ == '__main__':
    processes = [ mp.Process(target=say_name,name=f"proc{iproc}\",args=[iproc])
                  for iproc in range(6) ]

    for p in processes:
        p.start()

    for p in processes:
        p.join()
```

43.5.3 Listing of code code/multiprocessing/p/pool.py

```python
import multiprocessing as mp

def print_value(ivalue):
    mp.current_process()
    #print( f"Value: {ivalue}" )
    return 2*ivalue

if __name__ == '__main__':
    print("Test 1: what does cpu_count() return?\")
    nprocs = mp.cpu_count()
    print(f"I detect {nprocs} cores\")
    print("Test 2: create a pool and process an array on it\")
    pool = mp.Pool( nprocs )
    results = pool.map( print_value,range(1,2*nprocs) )
    print(results)
```
43. Python multiprocessing
PART V

THE REST
Chapter 44

Exploring computer architecture

There is much that can be said about computer architecture. However, in the context of parallel programming we are mostly concerned with the following:

- How many networked nodes are there, and does the network have a structure that we need to pay attention to?
- On a compute node, how many sockets (or other Non-Uniform Memory Access (NUMA) domains) are there?
- For each socket, how many cores and hyperthreads are there? Are caches shared?

44.1 Tools for discovery

An easy way for discovering the structure of your parallel machine is to use tools that are written especially for this purpose.

44.1.1 Intel cpuinfo

The Intel compiler suite comes with a tool cpuinfo that reports on the structure of the node you are running on. It reports on the number of packages, that is: sockets, cores, and threads.

44.1.2 hwloc

The open source package hwloc does similar reporting to cpuinfo, but it has been ported to many platforms. Additionally, it can generate ascii and pdf graphic renderings of the architecture.
44.2 Sources used in this chapter

44.2.1 Listing of code header
Chapter 45

Hybrid computing

So far, you have learned to use MPI for distributed memory and OpenMP for shared memory parallel programming. However, distribute memory architectures actually have a shared memory component, since each cluster node is typically of a multicore design. Accordingly, you could program your cluster using MPI for inter-node and OpenMP for intra-node parallelism.

You now have to find the right balance between processes and threads, since each can keep a core fully busy. Complicating this story, a node can have more than one *socket*, and corresponding **NUMA** domain. Figure 45.1 illustrates three modes: pure MPI with no threads used; one MPI process per node and full multi-threading; two MPI processes per node, one per socket, and multiple threads on each socket.

Figure 45.1: Three modes of MPI/OpenMP usage on a multi-core cluster
45.1 Affinity

In the preceding chapters we mostly considered all MPI nodes or OpenMP thread as being in one flat pool. However, for high performance you need to worry about affinity: the question of which process or thread is placed where, and how efficiently they can interact.

Here are some situations where you affinity becomes a concern.

- In pure MPI mode processes that are on the same node can typically communicate faster than processes on different nodes. Since processes are typically placed sequentially, this means that a scheme where process $p$ interacts mostly with $p + 1$ will be efficient, while communication with large jumps will be less so.
- If the cluster network has a structure (processor grid as opposed to fat-tree), placement of processes has an effect on program efficiency. MPI tries to address this with graph topology; section 11.2.
- Even on a single node there can be asymmetries. Figure 45.2 illustrates the structure of the four sockets of the Ranger supercomputer (no longer in production). Two cores have no direct connection. This asymmetry affects both MPI processes and threads on that node.
- Another problem with multi-socket designs is that each socket has memory attached to it. While every socket can address all the memory on the node, its local memory is faster to access. This asymmetry becomes quite visible in the first-touch phenomenon; section 25.2.
- If a node has fewer MPI processes than there are cores, you want to be in control of their placement. Also, the operating system can migrate processes, which is detrimental to performance since it negates data locality. For this reason, utilities such as numactl (and at TACC tacc_affinity) can be used to pin a thread or process to a specific core.
- Processors with hyperthreading or hardware threads introduce another level or worry about where threads go.

![Figure 45.2: The NUMA structure of a Ranger node](image)
45. Hybrid computing

45.2 What does the hardware look like?

If you want to optimize affinity, you should first know what the hardware looks like. The hwloc utility is valuable here [10] (https://www.open-mpi.org/projects/hwloc/).

Figure 45.3 depicts a Stampede compute node, which is a two-socket Intel Sandybridge design; figure 45.4 shows a Stampede largemem node, which is a four-socket design. Finally, figure 45.5 shows a Lonestar5 compute node, a two-socket design with 12-core Intel Haswell processors with two hardware threads each.

45.3 Affinity control

See chapter 25 for OpenMP affinity control.

45.4 Discussion

The performance implications of the pure MPI strategy versus hybrid are subtle.

- First of all, we note that there is no obvious speedup: in a well balanced MPI application all cores are busy all the time, so using threading can give no immediate improvement.
Figure 45.4: Structure of a Stampede largemem four-socket compute node
Both MPI and OpenMP are subject to Amdahl’s law that quantifies the influence of sequential code; in hybrid computing there is a new version of this law regarding the amount of code that is MPI-parallel, but not OpenMP-parallel.

MPI processes run unsynchronized, so small variations in load or in processor behavior can be tolerated. The frequent barriers in OpenMP constructs make a hybrid code more tightly synchronized, so load balancing becomes more critical.

On the other hand, in OpenMP codes it is easier to divide the work into more tasks than there are threads, so statistically a certain amount of load balancing happens automatically.

Each MPI process has its own buffers, so hybrid takes less buffer overhead.

Exercise 45.1. Review the scalability argument for 1D versus 2D matrix decomposition in HPSC-6.2. Would you get scalable performance from doing a 1D decomposition (for instance, of the rows) over MPI processes, and decomposing the other directions (the columns) over OpenMP threads?

Another performance argument we need to consider concerns message traffic. If let all threads make MPI calls (see section 13.1) there is going to be little difference. However, in one popular hybrid computing strategy we would keep MPI calls out of the OpenMP regions and have them in effect done by the master thread. In that case there are only MPI messages between nodes, instead of between cores. This leads to a decrease in message traffic, though this is hard to quantify. The number of messages goes down approximately by the number of cores per node, so this is an advantage if the average message size is small. On the other hand, the amount of data sent is only reduced if there is overlap in content between the messages.

Limiting MPI traffic to the master thread also means that no buffer space is needed for the on-node com-
Processes and cores and affinity

In OpenMP, threads are purely a software construct and you can create however many you want. The hardware limit of the available cores can be queried with `omp_get_num_procs` (section 17.5). How does that work in a hybrid context? Does the ‘proc’ count return the total number of cores, or does the MPI scheduler limit it to a number exclusive to each MPI process?

The following code fragment explore this:

```c
int ncores;
#pragma omp parallel
#pragma omp master
ncores = omp_get_num_procs();
int totalcores;
MPI_Reduce(&ncores,&totalcores,1,MPI_INT,MPI_SUM,0,comm);
if (procid==0) {
    printf("Omp procs on this process: %d\n",ncores);
    printf("Omp procs total: %d\n",totalcores);
}
```

For the full source of this example, see section 45.7.2

Running this with Intel MPI (version 19) gives the following:

```
---- nprocs: 14
Omp procs on this process: 4
Omp procs total: 56
---- nprocs: 15
Omp procs on this process: 3
Omp procs total: 45
---- nprocs: 16
Omp procs on this process: 3
Omp procs total: 48
```

We see that

- Each process get an equal number of cores, and
- Some cores will go unused.

While the OpenMP ‘proc’ count is such that the MPI processes will not oversubscribe cores, the actual placement of processes and threads is not expressed here. This assignment is known as affinity and it is determined by the MPI/OpenMP runtime system. Typically it can be controlled through environment variables, but one hopes the default assignment makes sense. Figure 45.6 illustrates this for the Intel Knights Landing:

- Placing four MPI processes on 68 cores gives 17 cores per process.
- Each process receives a contiguous set of cores.
Hybrid computing

However, cores are grouped in ‘tiles’ of two, so processes 1 and 3 start halfway a tile.
Therefore, thread zero of that process is bound to the second core.

45.6 Practical specification

Say you use 100 cluster nodes, each with 16 cores. You could then start 1600 MPI processes, one for each core, but you could also start 100 processes, and give each access to 16 OpenMP threads.

In your slurm scripts, the first scenario would be specified 
\[ \text{-N 100 -n 1600} \]
and the second as
\[ \text{#$ SBATCH -N 100} \]
\[ \text{#$ SBATCH -n 100} \]

- export OMP_NUM_THREADS=16

There is a third choice, in between these extremes, that makes sense. A cluster node often has more than one socket, so you could put one MPI process on each socket, and use a number of threads equal to the number of cores per socket.

The script for this would be:
\[ \text{#$ SBATCH -N 100} \]
\[ \text{#$ SBATCH -n 200} \]
export OMP_NUM_THREADS=8
ibrun tacc_affinity yourprogram

The `tacc_affinity` script unsets the following variables:

- `export MV2_USE_AFFINITY=0`
- `export MV2_ENABLE_AFFINITY=0`
- `export VIADEV_USE_AFFINITY=0`
- `export VIADEV_ENABLE_AFFINITY=0`

If you don’t use `tacc_affinity` you may want to do this by hand, otherwise `mvapich2` will use its own affinity rules.
45. Hybrid computing

45.7 Sources used in this chapter

45.7.1 Listing of code header

45.7.2 Listing of code examples/hybrid/c/procthread.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"
#include "omp.h"

int main(int argc,char **argv) {

    int procid,nprocs;
    int requested=MPI_THREAD_MULTIPLE,provided;

    MPI_Init_thread(&argc,&argv,requested,&provided);
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_rank(comm,&procid);
    MPI_Comm_size(comm,&nprocs);
    if (procid==0)
        printf("Threading level requested=%d, provided=%d\n",
               requested,provided);

    int ncores;
    #pragma omp parallel
    #pragma omp master
    ncores = omp_get_num_procs();

    int totalcores;
    MPI_Reduce(&ncores,&totalcores,1,MPI_INT,MPI_SUM,0,comm);
    if (procid==0) {
        printf("Omp procs on this process: %d\n",ncores);
        printf("Omp procs total: %d\n",totalcores);
    }

    MPI_Finalize();
    return 0;
}
```

Parallel Computing – r428
Chapter 46

Random number generation

Here is how you initialize the random number generator uniquely on each process:

C:

```c
// Initialize the random number generator
srand((int)(mytid*(double)RAND_MAX/ntids));
// compute a random float between [0,1]
randomfraction = (rand() / (double)RAND_MAX);
// compute random integer between [0,N-1]
randomfraction = rand() % N;
```

Fortran:

```fortran
integer :: randsize
integer, allocatable, dimension(:) :: randseed
real :: random_value

call random_seed(size=randsize)
allocate(randseed(randsize))
randseed(:) = 1023*mytid
call random_seed(put=randseed)
```
46. Random number generation

46.1 Sources used in this chapter

46.1.1 Listing of code header
Chapter 47

Parallel I/O

Parallel I/O is a tricky subject. You can try to let all processors jointly write one file, or to write a file per process and combine them later. With the standard mechanisms of your programming language there are the following considerations:

- On clusters where the processes have individual file systems, the only way to write a single file is to let it be generated by a single processor.
- Writing one file per process is easy to do, but
  - You need a post-processing script;
  - if the files are not on a shared file system (such as Lustre), it takes additional effort to bring them together;
  - if the files are on a shared file system, writing many files may be a burden on the metadata server.
- On a shared file system it is possible for all files to open the same file and set the file pointer individually. This can be difficult if the amount of data per process is not uniform.

Illustrating the last point:

```c
// pseek.c
FILE *pfile;
pfile = fopen("pseek.dat","w");
seek(pfile,procid*sizeof(int),SEEK_CUR);
// fseek(pfile,procid*sizeof(char),SEEK_CUR);
printf(pfile, "%d\n", procid);
fclose(pfile);
```

For the full source of this example, see section 47.1.2

MPI also has its own portable I/O: MPI I/O, for which see chapter 10.

Alternatively, one could use a library such as hdf5.
47. Parallel I/O

47.1 Sources used in this chapter

47.1.1 Listing of code header

47.1.2 Listing of code code/mpi/c/pseek.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc,char **argv) {

    int nprocs,procid;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&procid);

    FILE *pfile;
    pfile = fopen("pseek.dat","w");
    fseek(pfile,procid*sizeof(int),SEEK_CUR);
    // fseek(pfile,procid*sizeof(char),SEEK_CUR);
    fprintf(pfile,"%d\n",procid);
    fclose(pfile);

    MPI_Finalize();

    return 0;
}
```
Chapter 48

Support libraries

There are many libraries related to parallel programming to make life easier, or at least more interesting, for you.

48.1 SimGrid

SimGrid [15] is a simulator for distributed systems. It can for instance be used to explore the effects of architectural parameters. It has been used to simulate large scale operations such as High Performance Linpack (HPL) [4].

48.2 Other

ParaMesh
Global Arrays
Hdf5 and Silo
48. Support libraries

48.3 Sources used in this chapter

48.3.1 Listing of code header
PART VI

TUTORIALS
here are some tutorials specific to parallel programming.

1. Debugging, first sequential, then parallel. Chapter 49.
2. Tracing and profiling. Chapter 50.
5. Parallel I/O. Chapter 53.
Chapter 49

Debugging

When a program misbehaves, debugging is the process of finding out why. There are various strategies of finding errors in a program. The crudest one is debugging by print statements. If you have a notion of where in your code the error arises, you can edit your code to insert print statements, recompile, rerun, and see if the output gives you any suggestions. There are several problems with this:

- The edit/compile/run cycle is time consuming, especially since
- often the error will be caused by an earlier section of code, requiring you to edit, compile, and rerun repeatedly. Furthermore,
- the amount of data produced by your program can be too large to display and inspect effectively, and
- if your program is parallel, you probably need to print out data from all processors, making the inspection process very tedious.

For these reasons, the best way to debug is by the use of an interactive debugger, a program that allows you to monitor and control the behaviour of a running program. In this section you will familiarize yourself with gdb, which is the open source debugger of the GNU project. Other debuggers are proprietary, and typically come with a compiler suite. Another distinction is that gdb is a commandline debugger; there are graphical debuggers such as ddd (a frontend to gdb) or DDT and TotalView (debuggers for parallel codes). We limit ourselves to gdb, since it incorporates the basic concepts common to all debuggers.

In this tutorial you will debug a number of simple programs with gdb and valgrind. The files can be found in the repository in the directory tutorials/debug_tutorial_files.

49.1 Step 0: compiling for debug

You often need to recompile your code before you can debug it. A first reason for this is that the binary code typically knows nothing about what variable names corresponded to what memory locations, or what lines in the source to what instructions. In order to make the binary executable know this, you have to include the symbol table in it, which is done by adding the \texttt{-g} option to the compiler line.

Usually, you also need to lower the compiler optimization level: a production code will often be compiled with flags such as \texttt{-O2} or \texttt{-Xhost} that try to make the code as fast as possible, but for debugging you need
to replace this by -O0 (’oh-zero’). The reason is that higher levels will reorganize your code, making it hard to relate the execution to the source.

49.2 Invoking gdb

There are three ways of using gdb: using it to start a program, attaching it to an already running program, or using it to inspect a core dump. We will only consider the first possibility.

Here is an example of how to start gdb with a program that has no arguments (Fortran users, use hello.F):

```
tutorials/gdb/c/hello.c
```%

```
cc -g -o hello hello.c
# regular invocation:
%. ./hello
hello world
# invocation from gdb:
%. gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
Copyright 2004 Free Software Foundation, Inc. ..... copyright info ....
(gdb) run
Starting program: /home/eijkhout/tutorials/gdb/hello
Reading symbols for shared libraries +. done
hello world
Program exited normally.
(gdb) quit
```%

Important note: the program was compiled with the debug flag -g. This causes the symbol table (that is, the translation from machine address to program variables) and other debug information to be included in the binary. This will make your binary larger than strictly necessary, but it will also make it slower, for instance because the compiler will not perform certain optimizations.

To illustrate the presence of the symbol table do:

```
cc -g -o hello hello.c
gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

and compare it with leaving out the -g flag:

---

1. Typically, actual code motion is done by -O3, but at level -O2 the compiler will inline functions and make other simplifications.
2. Compiler optimizations are not supposed to change the semantics of a program, but sometimes do. This can lead to the nightmare scenario where a program crashes or gives incorrect results, but magically works correctly with compiled with debug and run in a debugger.
### 49.3 Finding errors

Let us now consider some programs with errors.

#### 49.3.1 C programs

```bash
tutorials/gdb/c/square.c
```
```bash
%% cc -g -o square square.c
%% ./square
5000
```

Segmentation fault

The *segmentation fault* (other messages are possible too) indicates that we are accessing memory that we are not allowed to, making the program stop. A debugger will quickly tell us where this happens:

```bash
%% gdb square
(gdb) run
50000
```

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x00000000000eb4a
0x000007fff824295ca in __svfscanf_l ()
```
49. Debugging

Apparently the error occurred in a function __svfscanf_l, which is not one of ours, but a system function. Using the backtrace (or bt, also where or w) command we quickly find out how this came to be called:

(gdb) backtrace
#0 0x00007fff824295ca in __svfscanf_l ()
#1 0x00007fff8244011b in fscanf ()
#2 0x00000001000000e89 in main (argc=1, argv=0x7fff5fbfc7c0) at square.c:7

We take a close look at line 7, and see that we need to change nmax to &nmax.

There is still an error in our program:

(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x000000010000f000
0x000000010000ebe in main (argc=2, argv=0x7fff5fbfc7a8) at square1.c:9
9 squares[i] = 1./(i*i); sum += squares[i];

We investigate further:

(gdb) print i
$1 = 11237
(gdb) print squares[i]
Cannot access memory at address 0x10000f000
and we quickly see that we forgot to allocate squares.

By the way, we were lucky here: this sort of memory errors is not always detected. Starting our program with a smaller input does not lead to an error:

(gdb) run
50
Sum: 1.625133e+00

Program exited normally.

49.3.2 Fortran programs

Compile and run the following program:

tutorials/gdb/f/square.F

It should end prematurely with a message such as 'Illegal instruction'. Running the program in gdb quickly tells you where the problem lies:

(gdb) run
Starting program: tutorials/gdb//fsquare
Reading symbols for shared libraries +++. done

Program received signal EXC_BAD_INSTRUCTION, Illegal instruction/operand.
49.4 Memory debugging with Valgrind

We take a close look at the code and see that we did not allocate squares properly.

49.4 Memory debugging with Valgrind

Insert the following allocation of squares in your program:

```c
squares = (float *) malloc( nmax*sizeof(float) );
```

Compile and run your program. The output will likely be correct, although the program is not. Can you see the problem?

To find such subtle memory errors you need a different tool: a memory debugging tool. A popular (because open source) one is valgrind; a common commercial tool is purify.

Compile this program with cc -o square1 square1.c and run it with valgrind square1 (you need to type the input value). You will lots of output, starting with:

```
%% valgrind square1
==53695== Memcheck, a memory error detector
==53695== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==53695== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==53695== Command: a.out
==53695==
==53695== Invalid write of size 4
==53695== at 0x100000EB0: main (square1.c:10)
==53695== Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695== at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695== by 0x100000E77: main (square1.c:8)
==53695==
==53695== Invalid read of size 4
==53695== at 0x100000EC1: main (square1.c:11)
==53695== Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695== at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695== by 0x100000E77: main (square1.c:8)
```

Valgrind is informative but cryptic, since it works on the bare memory, not on variables. Thus, these error messages take some exegesis. They state that a line 10 writes a 4-byte object immediately after a block of 40 bytes that was allocated. In other words: the code is writing outside the bounds of an allocated array. Do you see what the problem in the code is?

Note that valgrind also reports at the end of the program run how much memory is still in use, meaning not properly freed.

If you fix the array bounds and recompile and rerun the program, valgrind still complains:

```
==53785== Conditional jump or move depends on uninitialised value(s)
==53785== at 0x10006FC68: __dtoa (in /usr/lib/libSystem.B.dylib)
```
Although no line number is given, the mention of printf gives an indication where the problem lies. The reference to an ‘uninitialized value’ is again cryptic: the only value being output is sum, and that is not uninitialized: it has been added to several times. Do you see why valgrind calls is uninitialized all the same?

49.5 Stepping through a program

Often the error in a program is sufficiently obscure that you need to investigate the program run in detail. Compile the following program

```c
#include <stdio.h>
#include <math.h>

float root(float a, float b)
{
    float x = 0;
    for (int i = 100; i > -100; i--)
        x = 0.5 * (x + a / root(b, a - x * x));
    return x;
}

int main()
{
    float a = 16;
    float b = 4;
    printf("sum: \%g\n", root(a, b));
    return 0;
}
```

and run it:

```bash
%% ./roots
sum: nan
```

Start it in gdb as follows:

```bash
%% gdb roots
GNU gdb 6.3.50-20050815 (Apple version gdb-1469) (Wed May  5 04:36:56 UTC 2010)
Copyright 2004 Free Software Foundation, Inc.
....
(gdb) break main
Breakpoint 1 at 0x100000ea6: file root.c, line 14.
(gdb) run
Starting program: tutorials/gdb/c/roots
Reading symbols for shared libraries +. done

Breakpoint 1, main () at roots.c:14
14 float x=0;
```

Here you have done the following:

- Before calling run you set a breakpoint at the main program, meaning that the execution will stop when it reaches the main program.
- You then call run and the program execution starts;
- The execution stops at the first instruction in main.

If execution is stopped at a breakpoint, you can do various things, such as issuing the step command:

```bash
(gdb) step
15 for (i=100; i>-100; i--)
```

---

Parallel Computing – r428
49.6 Inspecting values

16 \quad x += \text{root}(i);

(gdb)

(if you just hit return, the previously issued command is repeated). Do a number of steps in a row by
hitting return. What do you notice about the function and the loop?

Switch from doing step to doing next. Now what do you notice about the loop and the function?

Set another breakpoint: break 17 and do cont. What happens?

Rerun the program after you set a breakpoint on the line with the sqrt call. When the execution stops
there do where and list.

- If you set many breakpoints, you can find out what they are with info breakpoints.
- You can remove breakpoints with delete n where n is the number of the breakpoint.
- If you restart your program with run without leaving gdb, the breakpoints stay in effect.
- If you leave gdb, the breakpoints are cleared but you can save them: save breakpoints <file>. Use
  source <file> to read them in on the next gdb run.

49.6 Inspecting values

Run the previous program again in gdb: set a breakpoint at the line that does the sqrt call before you
actually call run. When the program gets to line 8 you can do print n. Do cont. Where does the program stop?

If you want to repair a variable, you can do set var=value. Change the variable n and confirm that the
square root of the new value is computed. Which commands do you do?

If a problem occurs in a loop, it can be tedious keep typing cont and inspecting the variable with print.
Instead you can add a condition to an existing breakpoint: the following:

condition 1 if (n<0)

or set the condition when you define the breakpoint:

break 8 if (n<0)

Another possibility is to use ignore 1 50, which will not stop at breakpoint 1 the next 50 times.

Remove the existing breakpoint, redefine it with the condition n<0 and rerun your program. When the
program breaks, find for what value of the loop variable it happened. What is the sequence of commands
you use?

49.7 Parallel debugging

Debugging parallel programs is harder than than sequential programs, because every sequential bug may
show up, plus a number of new types, caused by the interaction of the various processes.

Here are a few possible parallel bugs:
• Processes can **deadlock** because they are waiting for a message that never comes. This typically happens with blocking send/receive calls due to an error in program logic.
• If an incoming message is unexpectedly larger than anticipated, a memory error can occur.
• A collective call will hang if somehow one of the processes does not call the routine.

There are few low-budget solutions to parallel debugging. The main one is to create an xterm for each process. We will describe this next. There are also commercial packages such as **DDT** and **TotalView**, that offer a GUI. They are very convenient but also expensive. The **Eclipse** project has a parallel package, **Eclipse PTP**, that includes a graphic debugger.

**49.7.1 MPI debugging with gdb**

You can not run parallel programs in gdb, but you can start multiple gdb processes that behave just like MPI processes! The command

```
mpirun -np <NP> xterm -e gdb ./program
```

create a number of xterm windows, each of which execute the commandline `gdb ./program`. And because these xterms have been started with mpirun, they actually form a communicator.

**49.8 Further reading**


Chapter 50

Tracing, timing, and profiling

50.1 Timing

Many systems have their own timers:

- MPI see section 15.6.1;
- OpenMP see section 29.2;
- PETSc see section 39.4.

50.1.1 Parallel timing

Timing parallel operations is fraught with peril, as processes or threads can interact with each other. This means that you may be measuring the wait time induced by synchronization. Sometimes that is actually what you want, as in the case of a *ping-pong* operation; section 4.1.1.

Other times, this is not what you want. Consider the code

```c
if (procno==0)
do_big_setup();
t = timer();
mpi_some_collective();
duration = timer() - t;
```

Figure 50.1 illustrates this:

- in the naive scenario, processes other than zero start the collective immediately, but process zero first does the setup;
- all processes presumably finish more or less together.

On the non-zero processes we now get a time measurement, which we intended to be just the collective operation, that includes the setup time of process zero.

The solution is to put a barrier around the section that you want to time; see again figure 50.1.
50. Tracing, timing, and profiling

50.2 Tau

TAU [http://www.cs.uoregon.edu/Research/tau/home.php](http://www.cs.uoregon.edu/Research/tau/home.php) is a utility for profiling and tracing your parallel programs. Profiling is the gathering and displaying of bulk statistics, for instance showing you which routines take the most time, or whether communication takes a large portion of your runtime. When you get concerned about performance, a good profiling tool is indispensable.

Tracing is the construction and displaying of time-dependent information on your program run, for instance showing you if one process lags behind others. For understanding a program's behaviour, and the reasons behind profiling statistics, a tracing tool can be very insightful.

50.2.1 Instrumentation

Unlike such tools as VTune which profile your binary as-is, TAU works by adding instrumentation to your code: in effect it is a source-to-source translator that takes your code and turns it into one that generates run-time statistics.

This instrumentation is largely done for you; you mostly need to recompile your code with a script that does the source-to-source translation, and subsequently compiles that instrumented code. You could for instance have the following in your makefile:

```makefile
ifdef TACC_TAU_DIR
    CC = tau_cc.sh
else
    CC = mpicc
endif

% : %.c
  <TAB>${CC} -o $@ $^

If TAU is to be used (which we detect here by checking for the environment variable TACC_TAU_DIR), we define the CC variable as one of the TAU compilation scripts; otherwise we set it to a regular MPI compiler.
50.2. Tau

Fortran note 24: Cpp includes. If your source contains

```plaintext
#include "something.h"
```

directives, add the option

```
-0ptPreProcess
```

to the TAU compiler.

50.2.2 Running

You can now run your instrumented code; trace/profile output will be written to file if environment variables TAU_PROFILE and/or TAU_TRACE are set:

```plaintext
export TAU_PROFILE=1
export TAU_TRACE=1
```

A TAU run can generate many files: typically at least one per process. It is therefore advisable to create a directory for your tracing and profiling information. You declare them to TAU by setting the environment variables PROFILEDIR and TRACEDIR.

```plaintext
mkdir tau_trace
mkdir tau_profile
export PROFILEDIR=tau_profile
export TRACEDIR=tau_trace
```

The actual program invocation is then unchanged:

```plaintext
mpirun -np 26 myprogram
```

**TACC note.** At TACC, use `ibrun` without a processor count; the count is derived from the queue submission parameters.

While this example uses two separate directories, there is no harm in using the same for both.

50.2.3 Output

The tracing/profiling information is spread over many files, and hard to read as such. Therefore, you need some further programs to consolidate and display the information.

You view profiling information with `paraprof`

```plaintext
paraprof tau_profile
```

Viewing the traces takes a few steps:

```plaintext
cd tau_trace
rm -f tau.trc tau.edf align.trc align.edf
tau_treemerge.pl
tau_timecorrect tau.trc tau.edf align.trc align.edf
tau2slog2 align.trc align.edf -o yourprogram.slog2
```

If you skip the `tau_timecorrect` step, you can generate the `slog2` file by:
tau2slog2 tau.trc tau.edf -o yourprogram.slog2
The slog2 file can be viewed with jumpshot:
jumpshot yourprogram.slog2

50.2.4 Without instrumentation

Event-based sampling on uninstrumented code:
tau_exec -ebs yourprogram
The resulting .trc file can be viewed with paraprof.

50.2.5 Examples

50.2.5.1 Bucket brigade

Let’s consider a bucket brigade implementation of a broadcast: each process sends its data to the next higher rank.

```c
int sendto =
    ( procno<nprocs-1 ? procno+1 : MPI_PROC_NULL )
;
int recvfrom =
    ( procno>0 ? procno-1 : MPI_PROC_NULL )
;
MPI_Recv( leftdata,1,MPI_DOUBLE,recvfrom,0,comm,MPI_STATUS_IGNORE);
myvalue = leftdata
MPI_Send( myvalue,1,MPI_DOUBLE,sendto,0,comm);
```

We implement the bucket brigade with blocking sends and receives: each process waits to receive from its predecessor, before sending to its successor. missing snippet bucketblock The TAU trace of this is in figure 50.2, using 4 nodes of 4 ranks each. We see that the processes within each node are fairly well synchronized, but there is less synchronization between the nodes. However, the bucket brigade then imposes its own synchronization on the processes because each has to wait for its predecessor, no matter if it posted the receive operation early.

Next, we introduce pipelining into this operation: each send is broken up into parts, and these parts are sent and received with non-blocking calls.

```c
// bucketpipenonblock.c
MPI_Request requests[PARTS];
for (int ipart=0; ipart<PARTS; ipart++) {
    MPI_Irecv
    {
        leftdata+partition_starts[ipart],partition_sizes[ipart],
        MPI_DOUBLE,recvfrom,ipart,comm,rrequests+ipart);
    }
```

The TAU trace is in figure 50.3.
50.2.5.2 Butterfly exchange

The NAS Parallel Benchmark suite [22] contains a Conjugate Gradients (CG) implementation that spells out its all-reduce operations as a *butterfly exchange*.

```fortran
!! cgb.f

    do i = 1, l2npcols
      call mpi_recv( d,
                     1,
                     dp_type,
                     reduce_exch_proc(i),
                     i,
                     mpi_comm_world,
                     request,
                     ierr )
      call mpi_send( sum,
                     1,
                     dp_type,
                     reduce_exch_proc(i),
                     i,
                     mpi_comm_world,
                     ierr )
      call mpi_wait( request, status, ierr )

      sum = sum + d
    enddo
```

Figure 50.2: Trace of a bucket brigade broadcast
Figure 50.3: Trace of a pipelined bucket brigade broadcast

We recognize this structure in the TAU trace: figure 50.4. Upon closer examination, we see how this particular algorithm induces a lot of wait time. Figures 50.6 and 50.7 show a whole cascade of processes waiting for each other.
Figure 50.4: Trace of a butterfly exchange

Figure 50.5: Trace of a butterfly exchange
Figure 50.6: Four stages of processes waiting caused by a single lagging process
Figure 50.7: Four stages of processes waiting caused by a single lagging process
Chapter 51

SimGrid

Many readers of this book will have access to some sort of parallel machine so that they can run simulations, maybe even some realistic scaling studies. However, not many people will have access to more than one cluster type so that they can evaluate the influence of the interconnect. Even then, for didactic purposes one would often wish for interconnect types (fully connected, linear processor array) that are unlikely to be available.

In order to explore architectural issues pertaining to the network, we then resort to a simulation tool, SimGrid.

Installation

Compilation  You write plain MPI files, but compile them with the SimGrid compiler smpicc.

Running  SimGrid has its own version of mpirun: smpirun. You need to supply this with options:

• -np 123456 for the number of (virtual) processors;
• -hostfile simgridhostfile which lists the names of these processors. You can basically make these up, but are defined in:
• -platform arch.xml which defines the connectivity between the processors.

For instance, with a hostfile of 8 hosts, a linearly connected network would be defined as:

```xml
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid/simgrid.dtd">

<platform version="4">

<zone id="first zone" routing="Floyd">
   <!-- the resources -->
   <host id="host1" speed="1Mf"/>
   <host id="host2" speed="1Mf"/>
   <host id="host3" speed="1Mf"/>
</zone>
</platform>
```
<host id="host4" speed="1Mf"/>
<host id="host5" speed="1Mf"/>
<host id="host6" speed="1Mf"/>
<host id="host7" speed="1Mf"/>
<host id="host8" speed="1Mf"/>
<link id="link1" bandwidth="125MBps" latency="100us"/>
<!-- the routing: specify how the hosts are interconnected -->
<route src="host1" dst="host2"><link_ctn id="link1"></route>
<route src="host2" dst="host3"><link_ctn id="link1"></route>
<route src="host3" dst="host4"><link_ctn id="link1"></route>
<route src="host4" dst="host5"><link_ctn id="link1"></route>
<route src="host5" dst="host6"><link_ctn id="link1"></route>
<route src="host6" dst="host7"><link_ctn id="link1"></route>
<route src="host7" dst="host8"><link_ctn id="link1"></route>
</zone>
</platform>

(such files can be generated with a short shell script).

The Floyd designation of the routing means that any route using the transitive closure of the paths given can be used. It is also possible to use routing="Full" which requires full specification of all pairs that can communicate.
Chapter 52

Batch systems

Supercomputer clusters can have a large number of nodes, but not enough to let all their users run simultaneously, and at the scale that they want. Therefore, users are asked to submit jobs, which may start executing immediately, or may have to wait until resources are available.

The decision when to run a job, and what resources to give it, is not done by a human operator, but by software called a batch system. (The Stampede cluster at TACC ran close to 10 million jobs over its lifetime, which corresponds to starting a job every 20 seconds.)

This tutorial will cover the basics of such systems, and in particular Simple Linux Utility for Resource Management (SLURM).

52.1 Cluster structure

A supercomputer cluster usually has two types of nodes:

- login nodes, and
- compute nodes.

When you make an ssh connection to a cluster, you are connecting to a login node. The number of login nodes is small, typically less than half a dozen.

Exercise 52.1. Connect to your favourite cluster. How many people are on that login node? If you disconnect and reconnect, do you find yourself on the same login node?

Compute nodes are where your jobs are run. Different clusters have different structures here:

- Compute nodes can be shared between users, or they can be assigned exclusively.
  - Sharing makes sense if user jobs have less parallelism than the core count of a node.
  - ... on the other hand, it means that users sharing a node can interfere with each other’s jobs, with one job using up memory or bandwidth that the other job needs.
  - With exclusive nodes, a job has access to all the memory and all the bandwidth of that node.
- Clusters can homogeneous, having the same processor type on each compute node, or they can have more than one processor type. For instance, the TACC Stampede2 cluster has Intel Knightslanding and Intel Skylake nodes.
Often, clusters have a number of 'large memory' nodes, on the order of a Terabyte of memory or more. Because of the cost of such hardware, there is usually only a small number of these nodes.

52.2 Queues

Jobs often cannot start immediately, because not enough resources are available, or because other jobs may have higher priority (see section 52.7). It is thus typical for a job to be put on a queue, scheduled, and started, by a batch system such as SLURM.

Batch systems do not put all jobs in one big pool: jobs are submitted to any of a number of queues, that are all scheduled separately.

Queues can differ in the following ways:

- If a cluster has different processor types, those are typically in different queues. Also, there may be separate queues for the nodes that have a GPU attached. Having multiple queues means you have to decide what processor type you want your job to run on, even if your executable is binary compatible with all of them.
- There can be 'development' queues, which have restrictive limits on runtime and node count, but where jobs typically start faster.
- Some clusters have 'premium' queues, which have a higher charge rate, but offer higher priority.
- 'Large memory nodes' are typically also in a queue of their own.
- There can be further queues for jobs with large resource demands, such as large core counts, or longer-than-normal runtimes.

For slurm, the `sinfo` command can tell you much about the queues.

```sh
# what queues are there?
sinfo -o "%P"
# what queues are there, and what is their status?
sinfo -o "%20P %.5a"
```

**Exercise 52.2.** Enter these commands. How many queues are there? Are they all operational at the moment?

52.2.1 Queue limits

Queues have limits on

- the runtime of a job;
- the node count of a job; or
- how many jobs a user can have in that queue.
52.3 Job running

There are two main ways of starting a job on a cluster that is managed by slurm. You can start a program run synchronously with `srun`, but this may hang until resources are available. In this section, therefore, we focus on asynchronously executing your program by submitting a job with `sbatch`.

52.3.1 The job submission cycle

In order to run a batch job, you need to write a job script, or batch script. This script describes what program you will run, where its inputs and outputs are located, how many processes it can use, and how long it will run.

In its simplest form, you submit your script without further parameters:

```
sbatch yourscript
```

All options regarding the job run are contained in the script file, as we will now discuss.

As a result of your job submission you get a job id. After submission you can query your job with `squeue`:

```
squeue -j 123456
```

or query all your jobs:

```
squeue -u yourname
```

The `squeue` command reports various aspects of your job, such as its status (typically pending or running); and if it is running, the queue (or ‘partition’) where it runs, its elapsed time, and the actual nodes where it runs.

```
squeue -j 5807991
```

```
JOBID PARTITION NAME USER ST TIME NODES Nodelist(REASON)
5807991 development packingt eijkhout R 0:04 2 c456-[012,034]
```

If you discover errors in your script after submitting it, including when it has started running, you can cancel your job with `scancel`:

```
scancel 1234567
```

52.4 The script file

A job script looks like an executable shell script:

- It has an ‘interpreter’ line such as
  ```bash
  #!/bin/bash
  ```
  at the top, and
- it contains ordinary unix commands, including
- the (parallel) startup of your program:
  ```bash
  # sequential program:
  ./yourprogram youroptions
  # parallel program, general:
  mpiexec -n 123 parallelprogram options
  ```
52.4. The script file

# parallel program, TACC:
ibrun parallelprogram options
• ... and then it has many options specifying the parallel run.

52.4.1 sbatch options

In addition to the regular unix commands and the interpreter line, your script has a number of SLURM directives, each starting with #SBATCH. (This makes them comments to the shell interpreter, so a batch script is actually a legal shell script.)

Directives have the form

#SBATCH -option value

Common options are:

• -J: the jobname. This will be displayed when you call squeue.
• -o: name of the output file. This will contain all the stdout output of the script.
• -e: name of the error file. This will contain all the stderr output of the script, as well as slurm error messages.

It can be a good idea to make the output and error file unique per job. To this purpose, the macro %j is available, which at execution time expands to the job number. You will then get an output file with a name such as myjob.o2384737.
• -p: the partition or queue. See above.
• -t hh:mm:ss: the maximum running time. If your job exceeds this, it will get cancelled. Two considerations:
  1. You can not specify a duration here that is longer than the queue limit.
  2. The shorter your job, the more likely it is to get scheduled sooner rather than later.
• -w c452-[101-104,111-112,115] specific nodes to place the job.
• -A: the name of the account to which your job should be billed.
• --mail-user=you@where Slurm can notify you when a job starts or ends. You may for instance want to connect to a job when it starts (to run top), or inspect the results when it's done, but not sit and stare at your terminal all day. The action of which you want to be notified is specified with (among others)
  --mail-type=begin/end/fail/all
• --dependency=after:123467 indicates that this job is to start after jobs 1234567 finished. Use afterok to start only if that job successfully finished. (See https://cvw.cac.cornell.edu/slurm/submission_depend for more options.)
• --nodelist allows you to specify specific nodes. This can be good for getting reproducible timings, but it will probably increase your wait time in the queue.
• --array=0-30 is a specification for ‘array jobs’: a task that needs to be executed for a range of parameter values.

TACC note. Array jobs are not supported at TACC; use a launcher instead; section 52.5.3.
• --mem=10000 specifies the desired amount of memory per node. Default units are megabytes, but can be explicitly indicated with K/M/G/T.

TACC note. This option can not be used to request arbitrary memory: jobs always have access to all available physical memory, and use of shared memory is not allowed.
See https://slurm.schedmd.com/sbatch.html for a full list.

**Exercise 52.3.** Write a script that executes the *date* command twice, with a *sleep* in between. Submit the script and investigate the output.

### 52.4.2 Environment

Your job script acts like any other shell script when it is executed. In particular, it inherits the calling *environment* with all its environment variables. Additionally, slurm defines a number of environment variables, such as the job ID, the hostlist, and the node and process count.

### 52.5 Parallelism handling

We discuss parallelism options separately.

#### 52.5.1 MPI jobs

On most clusters there is a structure with compute nodes, that contain one or more multi-core processors. Thus, you want to specify the node and core count. For this, there are options `-N` and `-n` respectively.

```bash
#SBATCH -N 4  # Total number of nodes
#SBATCH -n 4  # Total number of mpi tasks
```

It would be possible to specify only the node count or the core count, but that takes away flexibility:

- If a node has 40 cores, but your program stops scaling at 10 MPI ranks, you would use:
  ```bash
  #SBATCH -N 1
  #SBATCH -n 10
  ```
- If your processes use a large amount of memory, you may want to leave some cores unused. On a 40-core node you would either use
  ```bash
  #SBATCH -N 2
  #SBATCH -n 40
  ```
  or
  ```bash
  #SBATCH -N 1
  #SBATCH -n 20
  ```

Rather than specifying a total core count, you can also specify the core count per node with `--ntasks-per-node`.

**Exercise 52.4.** Go through the above examples and replace the `-n` option by an equivalent `--ntasks-per-node` values.

**Python note 43: Python MPI programs.** Python programs using mpi4py should be treated like other MPI programs, except that instead of an executable name you specify the python executable and the script name:

```bash
ibrun python3 mympi4py.py
```
52.5.2 Threaded jobs

The above discussion was mostly of relevance to MPI programs. Some other cases:

- For pure-OpenMP programs you need only one node, so the -N value is 1. Maybe surprisingly, the -n value is also 1, since only one process needs to be created: OpenMP uses thread-level parallelism, which is specified through the OMP_NUM_THREADS environment variable.
- A similar story holds for the Matlab parallel computing toolbox (note: note the distributed computing toolbox), and the Python multiprocessing module.

Exercise 52.5. What happens if you specify an -n value greater than 1 for a pure-OpenMP program?

For hybrid computing MPI-OpenMP programs, you use a combination of slurm options and environment variables, such that, for instance, the product of the --tasks-per-node and OMP_NUM_THREADS is less than the core count of the node.

52.5.3 Parameter sweeps / ensembles / massively parallel

So far we have focused on jobs where a single parallel executable is scheduled. However, there are use cases where you want to run a sequential (or very modestly parallel) executable for a large number of inputs. This is called variously a parameter sweep or an ensemble.

Slurm can support this itself with array jobs, though there are more sophisticated launcher tools for such purposes.

TACC note. TACC clusters do not support array jobs. Instead, use the launcher or pylauncher modules.

52.6 Job running

When your job is running, its status is reported as R by squeue. That command also reports which nodes are allocated to it.

\[
\text{squeue -j 5807991}
\]

\[
\begin{array}{cccccccc}
\text{JOBID} & \text{PARTITION} & \text{NAME} & \text{USER} & \text{ST} & \text{TIME} & \text{NODES} & \text{NODELIST(REASON)} \\
5807991 & \text{development} & \text{packingt eijkhout} & \text{R} & 0:04 & 2 & \text{c456-[012,034]} \\
\end{array}
\]

You can then ssh into the compute nodes of your job; normally, compute nodes are off-limits. This is useful if you want to run top to see how your processes are doing.

52.7 Scheduling strategies

Such a system looks at resource availability and the user’s priority to determine when a job can be run.

Of course, if a user is requesting a large number of nodes, it may never happen that that many become available simultaneously, so the batch system will force the availability. It does so by determining a time when that job is set to run, and then let nodes go idle so that they are available at that time.
An interesting side effect of this is that, right before the really large job starts, a ‘fairly’ large job can be run, if it only has a short running time. This is known as backfill, and it may cause jobs to be run earlier than their priority would warrant.

### 52.8 File systems

File systems come in different types:

- They can be backed-up or not;
- they can be shared or not; and
- they can be permanent or purged.

On many clusters each node has as local disc, either spinning or a RAM disc. This is usually limited in size, and should only be used for temporary files during the job run.

Most of the file system lives on discs that are part of RAID arrays. These discs have a large amount of redundancy to make them fault-tolerant, and in aggregate they form a shared file system: one unified file system that is accessible from any node and where files can take on any size, or at least much larger than any individual disc in the system.

**TACC note.** The HOME file system is limited in size, but is both permanent and backed up. Here you put scripts and sources.

The WORK file system is permanent but not backed up. Here you can store output of your simulations. However, currently the work file system can not immediately sustain the output of a large parallel job.

The SCRATCH file system is purged, but it has the most bandwidth for accepting program output. This is where you would write your data. After post-processing, you can then store on the work file system, or write to tape.

**Exercise 52.6.** If you install software with *cmake*, you typically have

1. a script with all your cmake options;
2. the sources,
3. the installed header and binary files
4. temporary object files and such.

How would you organize these entities over your available file systems?

### 52.9 Examples

Very sketchy section.
52.9.1  Job dependencies

```
JOB=`sbatch my_batchfile.sh | egrep -o -e \"\[0-9]+\"`

#!/bin/sh

# Launch first job
JOB=`sbatch job.sh | egrep -o -e \"\[0-9]+\"`

# Launch a job that should run if the first is successful
sbatch --dependency=afterok:${JOB} after_success.sh

# Launch a job that should run if the first job is unsuccessful
sbatch --dependency=afternotok:${JOB} after_fail.sh
```

52.9.2  Multiple runs in one script

```
ibrun stuff &
sleep 10
for h in hostlist ; do
  ssh $h "top"
done
wait
```

52.10  Review questions

For all true/false questions, if you answer False, what is the right answer and why?

**Exercise 52.7.**  T/F? When you submit a job, it starts running immediately once sufficient resources are available.

**Exercise 52.8.**  T/F? If you submit the following script:

```
#!/bin/bash
#SBATCH -N 10
#SBATCH -n 10
echo "hello world"
```

you get 10 lines of ‘hello world’ in your output.

**Exercise 52.9.**  T/F? If you submit the following script:

```
#!/bin/bash
#SBATCH -N 10
#SBATCH -n 10
hostname
```

you get the hostname of the login node from which your job was submitted.

**Exercise 52.10.**  Which of these are shared with other users when your job is running:

- Memory;
52. Batch systems

- CPU;
- Disc space?

**Exercise 52.11.** What is the command for querying the status of your job?

- sinfo
- squeue
- sacct

**Exercise 52.12.** On 4 nodes with 40 cores each, what’s the largest program run, measured in

- MPI ranks;
- OpenMP threads?
Chapter 53
Parallel I/O

For a great discussion see [20], from which figures here are taken.

53.1 Use sequential I/O

MPI processes can do anything a regular process can, including opening a file. This is the simplest form of parallel I/O: every MPI process opens its own file. To prevent write collisions,

• you use MPI_Comm_rank to generate a unique file name, or
• you use a local file system, typically /tmp, that is unique per process, or at least per the group of processes on a node.

For reading it is actually possible for all processes to open the same file, but for reading this is not really feasible. Hence the unique files.

53.2 MPI I/O

In chapter 10 we discussed MPI I/O. This is a way for all processes on a communicator to open a single file, and write to it in a coordinated fashion. This has the big advantage that the end result is an ordinary Unix file.

53.3 Higher level libraries

Libraries such as NetCDF or HDF5 offer advantages over MPI I/O:

• Files can be OS-independent, removing worries such as about little-endian storage.
• Files are self-documenting: they contain the metadata describing their contents.
53. Parallel I/O
PART VII

CLASS PROJECTS
Chapter 54

A Style Guide to Project Submissions

Here are some guidelines for how to submit assignments and projects. As a general rule, consider programming as an experimental science, and your writeup as a report on some tests you have done: explain the problem you’re addressing, your strategy, your results.

Structure of your writeup Most of the projects in this book use a scientific question to allow you to prove your coding skills. That does not mean that turning in the code is sufficient, nor code plus sample output. Turn in a writeup in pdf form that was generated from a text processing program such (preferably) \LaTeX{} (for a tutorial, see HPSC-40).

Your writeup should have:

- Foremost, a short description of the purpose of your project and your results;
- An explanation of your algorithms or solution strategy;
- Relevant fragments of your code;
- A scientific discussion of what you observed,
- Any code-related observations.
- If applicable: graphs, both of application quantities and performance issues. (For parallel runs possibly TAU plots; see 50.2).

Observe, measure, hypothesize, deduce Your project may be a scientific investigation of some phenomenon. Formulate hypotheses as to what you expect to observe, report on your observations, and draw conclusions.

Quite often your program will display unexpected behaviour. It is important to observe this, and hypothesize what the reason might be for your observed behaviour.

In most applications of computing machinery we care about the efficiency with which we find the solution. Thus, make sure that you do measurements. In general, make observations that allow you to judge whether your program behaves the way you would expect it to.
Including code If you include code samples in your writeup, make sure they look good. For starters, use a mono-spaced font. In \LaTeX, you can use the verbatim environment or the verbatiminput command. In that section option the source is included automatically, rather than cut and pasted. This is to be preferred, since your writeup will stay current after you edit the source file.

Including whole source files makes for a long and boring writeup. The code samples in this book were generated as follows. In the source files, the relevant snippet was marked as

```plaintext
... boring stuff
//snippet samplex
  .. interesting! ..
//snippet end
... more boring stuff
```

The files were then processed with the following command line (actually, included in a makefile, which requires doubling the dollar signs):

```bash
for f in *.{c,cxx,h} ; do
cat $x | awk 'BEGIN {f=0}
    /snippet end/ {f=0}
    f==1 {print $0 > file}
    /snippet/ && !/end/ {f=1; file=$2 }'
done
```

which gives (in this example) a file `samplex`. Other solutions are of course possible.

### Code formatting

Included code snippets should be readable. At a minimum you could indent the code correctly in an editor before you include it in a verbatim environment. (Screenshots of your terminal window are a decidedly suboptimal solution.) But it’s better to use the listing package which formats your code, include syntax coloring. For instance,

```latex
\lstset{language=C++} % or Fortran or so
\begin{lstlisting}
for (int i=0; i<N; i++)
  s += 1;
\end{lstlisting}
```

### Running your code

A single run doesn’t prove anything. For a good report, you need to run your code for more than one input dataset (if available) and in more than one processor configuration. When you choose problem sizes, be aware that an average processor can do a billion operations per second: you need to make your problem large enough for the timings to rise above the level of random variations and startup phenomena.
When you run a code in parallel, beware that on clusters the behaviour of a parallel code will always be different between one node and multiple nodes. On a single node the MPI implementation is likely optimized to use the shared memory. This means that results obtained from a single node run will be unrepresentative. In fact, in timing and scaling tests you will often see a drop in (relative) performance going from one node to two. Therefore you need to run your code in a variety of scenarios, using more than one node.

*Reporting scaling*  If you do a scaling analysis, a graph reporting runtimes should not have a linear time axis: a logarithmic graph is much easier to read. A speedup graph can also be informative.

Some algorithms are mathematically equivalent in their sequential and parallel versions. Others, such as iterative processes, can take more operations in parallel than sequentially, for instance because the number of iterations goes up. In this case, report both the speedup of a single iteration, and the total improvement of running the full algorithm in parallel.

*Repository organization*  If you submit your work through a repository, make sure you organize your submissions in subdirectories, and that you give a clear name to all files. Object files and binaries should not be in a repository since they are dependent on hardware and things like compilers.
Chapter 55

Warmup Exercises

We start with some simple exercises.

55.1 Hello world

For background, see section 2.3.

First of all we need to make sure that you have a working setup for parallel jobs. The example program helloworld.c does the following:

```c
// helloworld.c
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&ntids);
MPI_Comm_rank(MPI_COMM_WORLD,&mytid);
printf("Hello, this is processor \%d out of \%d\n",mytid,ntids);
MPI_Finalize();
```

Compile this program and run it in parallel. Make sure that the processors do not all say that they are processor 0 out of 1!

55.2 Collectives

It is a good idea to be able to collect statistics, so before we do anything interesting, we will look at MPI collectives; section 3.1.

Take a look at time_max.cxx. This program sleeps for a random number of seconds: and measures how long the sleep actually was: In the code, this quantity is called 'jitter', which is a term for random deviations in a system.

Exercise 55.1. Change this program to compute the average jitter by changing the reduction operator.
Exercise 55.2. Now compute the standard deviation

$$\sigma = \sqrt{\frac{\sum (x_i - m)^2}{n}}$$

where $m$ is the average value you computed in the previous exercise.
- Solve this exercise twice: once by following the reduce by a broadcast operation and once by using an Allreduce.
- Run your code both on a single cluster node and on multiple nodes, and inspect the TAU trace. Some MPI implementations are optimized for shared memory, so the trace on a single node may not look as expected.
- Can you see from the trace how the allreduce is implemented?

Exercise 55.3. Finally, use a gather call to collect all the values on processor zero, and print them out. Is there any process that behaves very differently from the others?

55.3 Linear arrays of processors

In this section you are going to write a number of variations on a very simple operation: all processors pass a data item to the processor with the next higher number.

- In the file linear-serial.c you will find an implementation using blocking send and receive calls.
- You will change this code to use non-blocking sends and receives; they require an MPI_Wait call to finalize them.
- Next, you will use MPI_Sendrecv to arrive at a synchronous, but deadlock-free implementation.
- Finally, you will use two different one-sided scenarios.

In the reference code linear-serial.c, each process defines two buffers:

// linear-serial.c
int my_number = mytid, other_number=-1.;

where other_number is the location where the data from the left neighbour is going to be stored.

To check the correctness of the program, there is a gather operation on processor zero:

```c
int *gather_buffer=NULL;
if (mytid==0) {
    gather_buffer = (int*) malloc(ntids*sizeof(int));
    if (!gather_buffer) MPI_Abort(comm,1);
}
MPI_Gather(&other_number,1,MPI_INT,
           gather_buffer,1,MPI_INT, 0,comm);
if (mytid==0) {
    int i,error=0;
    for (i=0; i<ntids; i++)
        if (gather_buffer[i]!=i-1) {
```
55.3. Linear arrays of processors

```c
printf("Processor \%d was incorrect: \%d should be \%d\n", 
i,gather_buffer[i],i-1);
error =1;
}
if (!error) printf("Success!\n");
free(gather_buffer);
}
```

55.3.1 Coding with blocking calls

Passing data to a neighbouring processor should be a very parallel operation. However, if we code this naively, with MPI_Send and MPI_Recv, we get an unexpected serial behaviour, as was explained in section 4.1.4.

```c
if (mytid<ntids-1)
    MPI_Ssend( /* data: */ &my_number,1,MPI_INT,
        /* to: */ mytid+1, /* tag: */ 0, comm);
if (mytid>0)
    MPI_Recv( /* data: */ &other_number,1,MPI_INT,
        /* from: */ mytid-1, 0, comm, &status);
```

(Note that this uses an Ssend; see section 15.8 for the explanation why.)

**Exercise 55.4.** Compile and run this code, and generate a TAU trace file. Confirm that the execution is serial. Does replacing the Ssend by Send change this?

Let’s clean up the code a little.

**Exercise 55.5.** First write this code more elegantly by using MPI_PROC_NULL.

55.3.2 A better blocking solution

The easiest way to prevent the serialization problem of the previous exercises is to use the MPI_Sendrecv call. This routine acknowledges that often a processor will have a receive call whenever there is a send. For border cases where a send or receive is unmatched you can use MPI_PROC_NULL.

**Exercise 55.6.** Rewrite the code using MPI_Sendrecv. Confirm with a TAU trace that execution is no longer serial.

Note that the Sendrecv call itself is still blocking, but at least the ordering of its constituent send and recv are no longer ordered in time.

55.3.3 Non-blocking calls

The other way around the blocking behaviour is to use Isend and Irecv calls, which do not block. Of course, now you need a guarantee that these send and receive actions are concluded; in this case, use MPI_Waitall.

**Exercise 55.7.** Implement a fully parallel version by using MPI_Isend and MPI_Irecv.
55.3.4 One-sided communication

Another way to have non-blocking behaviour is to use one-sided communication. During a Put or Get operation, execution will only block while the data is being transferred out of or into the origin process, but it is not blocked by the target. Again, you need a guarantee that the transfer is concluded; here use `MPI_Win_fence`.

**Exercise 55.8.** Write two versions of the code: one using `MPI_Put` and one with `MPI_Get`. Make TAU traces.

Investigate blocking behaviour through TAU visualizations.

**Exercise 55.9.** If you transfer a large amount of data, and the target processor is occupied, can you see any effect on the origin? Are the fences synchronized?
Chapter 56

Mandelbrot set

If you’ve never heard the name Mandelbrot set, you probably recognize the picture; figure 56.1. Its formal definition is as follows:

A point $c$ in the complex plane is part of the Mandelbrot set if the series $x_n$ defined by

\[
\begin{cases}
  x_0 = 0 \\
  x_{n+1} = x_n^2 + c
\end{cases}
\]

satisfies

\[
\forall n : |x_n| \leq 2.
\]
56. Mandelbrot set

It is easy to see that only points \( c \) in the bounding circle \( |c| < 2 \) qualify, but apart from that it’s hard to say much without a lot more thinking. Or computing; and that’s what we’re going to do.

In this set of exercises you are going to take an example program \texttt{mandel\_main.cxx} and extend it to use a variety of MPI programming constructs. This program has been set up as a \textit{manager-worker} model: there is one manager processor (for a change this is the last processor, rather than zero) which gives out work to, and accepts results from, the worker processors. It then takes the results and constructs an image file from them.

56.1 Invocation

The \texttt{mandel\_main} program is called as

\begin{verbatim}
mpirun -np 123 mandel\_main steps 456 iters 789
\end{verbatim}

where the \texttt{steps} parameter indicates how many steps in \( x, y \) direction there are in the image, and \texttt{iters} gives the maximum number of iterations in the \texttt{belongs} test.

If you forget the parameter, you can call the program with

\begin{verbatim}
mandel\_serial -h
\end{verbatim}

and it will print out the usage information.

56.2 Tools

The driver part of the Mandelbrot program is simple. There is a circle object that can generate coordinates \textbf{missing snippet \texttt{circledef}} and a global routine that tests whether a coordinate is in the set, at least up to an iteration bound. It returns zero if the series from the given starting point has not diverged, or the iteration number in which it diverged if it did so.

```c
int belongs(struct coordinate xy, int itbound) {
    double x=xy.x, y=xy.y; int it;
    for (it=0; it<itbound; it++) {
        double xx,yy;
        xx = x*x - y*y + xy.x;
        yy = 2*x*y + xy.y;
        x = xx; y = yy;
        if (x*x+y*y>4.) {
            return it;
        }
    }
    return 0;
}
```

In the former case, the point could be in the Mandelbrot set, and we colour it black, in the latter case we give it a colour depending on the iteration number.
if (iteration==0)
    memset(colour,0,3*sizeof(float));
else {
    float rfloat = ((float) iteration) / workcircle->infty;
    colour[0] = rfloat;
    colour[1] = MAX((float)0,(float)(1-2*rfloat));
    colour[2] = MAX((float)0,(float)(2*(rfloat-.5)));
}

We use a fairly simple code for the worker processes: they execute a loop in which they wait for input, process it, return the result.

A very simple solution using blocking sends on the manager is given:

```cpp
// mandel_serial.cxx
class serialqueue : public queue {
private :
    int free_processor;
public :
    serialqueue(MPI_Comm queue_comm, circle *workcircle) :
        queue(queue_comm,workcircle) {
            free_processor=0;
    }
    /**
     * The 'addtask' routine adds a task to the queue. In this simple case it immediately sends the task to a worker and waits for the result, which is added to the image.
     * This routine is only called with valid coordinates; the calling environment will stop the process once an invalid coordinate is encountered.
     */
    int addtask(struct coordinate xy) {
        MPI_Status status; int contribution, err;
```
56. Mandelbrot set

```c
err = MPI_Send(&xy, 1, coordinate_type, free_processor, 0, comm); CHK(err);
err = MPI_Recv(&contribution, 1, MPI_INT, free_processor, 0, comm, &status); CHK(err);

coordinate_to_image(xy, contribution);
total_tasks++;
free_processor = (free_processor+1)%(ntids-1);

return 0;
};
```

**Exercise 56.1.** Explain why this solution is very inefficient. Make a trace of its execution that bears this out.

**Figure 56.2: Trace of a serial Mandelbrot calculation**

56.3 Bulk task scheduling

The previous section showed a very inefficient solution, but that was mostly intended to set up the code base. If all tasks take about the same amount of time, you can give each process a task, and then wait on them all to finish. A first way to do this is with non-blocking sends.

**Exercise 56.2.** Code a solution where you give a task to all worker processes using non-blocking sends and receives, and then wait for these tasks with MPI_Waitall
to finish before you give a new round of data to all workers. Make a trace of the execution of this and report on the total time.
You can do this by writing a new class that inherits from `queue`, and that provides its own `addtask` method:

```cpp
// mandel_bulk.cxx
class bulkqueue : public queue {
public:
    bulkqueue(MPI_Comm queue_comm, circle *workcircle)
        : queue(queue_comm, workcircle) {
```

You will also have to override the `complete` method: when the `circle` object indicates that all coordinates have been generated, not all workers will be busy, so you need to supply the proper `MPI_Waitall` call.

![Figure 56.3: Trace of a bulk Mandelbrot calculation](image)

56.4 Collective task scheduling

Another implementation of the bulk scheduling of the previous section would be through using collectives.

**Exercise 56.3.** Code a solution which uses scatter to distribute data to the worker tasks, and gather to collect the results. Is this solution more or less efficient than the previous?
56.5 **Asynchronous task scheduling**

At the start of section 56.3 we said that bulk scheduling mostly makes sense if all tasks take similar time to complete. In the Mandelbrot case this is clearly not the case.

**Exercise 56.4.** Code a fully dynamic solution that uses `MPI_Probe` or `MPI_Waitany`. Make an execution trace and report on the total running time.

![Figure 56.4: Trace of an asynchronous Mandelbrot calculation](image)

56.6 **One-sided solution**

Let us reason about whether it is possible (or advisable) to code a one-sided solution to computing the Mandelbrot set. With active target synchronization you could have an exposure window on the host to which the worker tasks would write. To prevent conflicts you would allocate an array and have each worker write to a separate location in it. The problem here is that the workers may not be sufficiently synchronized because of the differing time for computation.

Consider then passive target synchronization. Now the worker tasks could write to the window on the manager whenever they have something to report; by locking the window they prevent other tasks from interfering. After a worker writes a result, it can get new data from an array of all coordinates on the manager.

It is hard to get results into the image as they become available. For this, the manager would continuously have to scan the results array. Therefore, constructing the image is easiest done when all tasks are concluded.
Chapter 57

Data parallel grids

In this section we will gradually build a semi-realistic example program. To get you started some pieces have already been written: as a starting point look at code/mpi/c/grid.cxx.

57.1 Description of the problem

With this example you will investigate several strategies for implementing a simple iterative method. Let’s say you have a two-dimensional grid of datapoints \( G = \{ g_{ij} : 0 \leq i < n_i, 0 \leq j < n_j \} \) and you want to compute \( G' \) where

\[
g'_{ij} = \frac{1}{4} \cdot (g_{i+1,j} + g_{i-1,j} + g_{i,j+1} + g_{i,j-1}).
\]

(57.1)

This is easy enough to implement sequentially, but in parallel this requires some care.

Let’s divide the grid \( G \) and divide it over a two-dimensional grid of \( p_i \times p_j \) processors. (Other strategies exist, but this one scales best; see section HPSC-6.5.) Formally, we define two sequences of points

\[
0 = i_0 < \cdots < i_{p_i-1} < i_{p_i} = n_i, \quad 0 < j_0 < \cdots < j_{p_j-1} < j_{p_j} = n_j
\]

and we say that processor \((p,q)\) computes \( g_{ij} \) for

\[
i_p \leq i < i_{p+1}, \quad j_q \leq j < j_{q+1}.
\]

From formula (57.1) you see that the processor then needs one row of points on each side surrounding its part of the grid. A picture makes this clear; see figure 57.1. These elements surrounding the processor’s own part are called the *halo* or *ghost region* of that processor.

The problem is now that the elements in the halo are stored on a different processor, so communication is needed to gather them. In the upcoming exercises you will have to use different strategies for doing so.
57. Data parallel grids

Figure 57.1: A grid divided over processors, with the 'ghost' region indicated

57.2 Code basics

The program needs to read the values of the grid size and the processor grid size from the commandline, as well as the number of iterations. This routine does some error checking: if the number of processors does not add up to the size of MPI_COMM_WORLD, a nonzero error code is returned.

```c
ierr = parameters_from_commandline
    (argc,argv,comm,&ni,&nj,&pi,&pj,&nit);
if (ierr) return MPI_Abort(comm,1);
```

From the processor parameters we make a processor grid object:

```c
processor_grid *pgrid = new processor_grid(comm,pi,pj);
```

And from the numerical parameters we make a number grid:

```c
number_grid *grid = new number_grid(pgrid,ni,nj);
```

Number grids have a number of methods defined. To set the value of all the elements belonging to a processor to that processor's number:

```c
grid->set_test_values();
```

To set random values:

```c
grid->set_random_values();
```

If you want to visualize the whole grid, the following call gathers all values on processor zero and prints them:

```c
grid->gather_and_print();
```

Next we need to look at some data structure details.

The definition of the number_grid object starts as follows:
class number_grid {
public:
  processor_grid *pgrid;
  double *values,*shadow;

where values contains the elements owned by the processor, and shadow is intended to contain the
values plus the ghost region. So how does shadow receive those values? Well, the call looks like
grid->build_shadow();
and you will need to supply the implementation of that. Once you’ve done so, there is a routine that prints
out the shadow array of each processor
grid->print_shadow();
In the file code/mpi/c/grid_impl.cxx you can see several uses of the macro INDEX. This translates
from a two-dimensional coordinate system to one-dimensional. Its main use is letting you use (i,j) coor-
dinates for indexing the processor grid and the number grid: for processors you need the translation to
the linear rank, and for the grid you need the translation to the linear array that holds the values.

A good example of the use of INDEX is in the number_grid::relax routine: this takes points from the
shadow array and averages them into a point of the values array. (To understand the reason for this
particular averaging, see HPSC-4.2.3 and HPSC-5.5.3.) Note how the INDEX macro is used to index in a
ilength × jlength target array values, while reading from a (ilength + 2) × (jlength + 2) source
array shadow.

for (i=0; i<ilength; i++) {
  for (j=0; j<jlength; j++) {
    int c=0;
    double new_value=0.;
    for (c=0; c<5; c++) {
      int ioff=i+1+ioffsets[c],joff=j+1+joffsets[c];
      new_value += coefficients[c] * 
                    shadow[ INDEX(ioff,joff,ilength+2,jlength+2) ];
    }
    values[ INDEX(i,j,ilength,jlength) ] = new_value/8.;
  }
}
Chapter 58

N-body problems

N-body problems describe the motion of particles under the influence of forces such as gravity. There are many approaches to this problem, some exact, some approximate. Here we will explore a number of them. For background reading see HPSC-10.

58.1 Solution methods

It is not in the scope of this course to give a systematic treatment of all methods for solving the N-body problem, whether exactly or approximately, so we will just consider a representative selection.

1. Full $N^2$ methods. These compute all interactions, which is the most accurate strategy, but also the most computationally demanding.
2. Cutoff-based methods. These use the basic idea of the $N^2$ interactions, but reduce the complexity by imposing a cutoff on the interaction distance.
3. Tree-based methods. These apply a coarsening scheme to distant interactions to lower the computational complexity.

58.2 Shared memory approaches

58.3 Distributed memory approaches
PART VIII

DIDACTICS
Chapter 59

Teaching guide

Based on two lectures per week, here is an outline of how MPI can be taught in a college course. Links to the relevant exercises.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Exercises</th>
<th>Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1: SPMD and collectives</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intro: cluster structure</td>
<td>hello: 2.1, 2.2</td>
<td></td>
</tr>
<tr>
<td>Functional parallelism</td>
<td>commrank: 2.4, 2.5, prime: 2.6</td>
<td></td>
</tr>
<tr>
<td>Allreduce, broadcast</td>
<td>3.1, randommax: 3.2</td>
<td>week 2</td>
</tr>
<tr>
<td></td>
<td>jordan: 3.8</td>
<td></td>
</tr>
<tr>
<td>Scan, Gather</td>
<td>3.13, scangather: 3.11, 3.15</td>
<td>week 3</td>
</tr>
<tr>
<td>Block 2: Two-sided point-to-point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send and receive</td>
<td>pingpong: 4.1, rightsend: 4.4</td>
<td></td>
</tr>
<tr>
<td>Sendrecv</td>
<td>bucketblock: 4.6, sendrecv: 4.8, 4.9</td>
<td>week 4</td>
</tr>
<tr>
<td>Nonblocking</td>
<td>isendirecv: 4.13, isendirecvarray: 4.14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bucketpipenonblock: 4.11</td>
<td></td>
</tr>
<tr>
<td>Block 3: Derived datatypes</td>
<td></td>
<td>week 5</td>
</tr>
<tr>
<td>Contiguous, Vector, Indexed</td>
<td>stridesend: 6.4, cubegather: 6.6</td>
<td></td>
</tr>
<tr>
<td>Extent and resizing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block 4: Communicators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Duplication, split</td>
<td>procgrid: 7.1, 7.2</td>
<td>week 6</td>
</tr>
<tr>
<td>Groups</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block 5: I/O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>File open, write, views</td>
<td>blockwrite: 10.1, viewwrite 10.4</td>
<td>week 7</td>
</tr>
<tr>
<td>Block 6: Neighborhood collectives</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neighbor allgather</td>
<td>rightgraph: 11.1</td>
<td></td>
</tr>
</tbody>
</table>
59.1 Sources used in this chapter

59.1.1 Listing of code header
Chapter 60

Teaching from mental models

Distributed memory programming, typically through the MPI library, is the *de facto* standard for programming large scale parallelism, with up to millions of individual processes. Its dominant paradigm of Single Program Multiple Data (SPMD) programming is different from threaded and multicore parallelism, to an extent that students have a hard time switching models. In contrast to threaded programming, which allows for a view of the execution with central control and a central repository of data, SPMD programming has a symmetric model where all processes are active all the time, with none privileged, and where data is distributed.

This model is counterintuitive to the novice parallel programmer, so care needs to be taken how to instill the proper ‘mental model’. Adoption of an incorrect mental model leads to broken or inefficient code.

We identify problems with the currently common way of teaching MPI, and propose a structuring of MPI courses that is geared to explicit reinforcing the symmetric model. Additionally, we advocate starting from realistic scenarios, rather than writing artificial code just to exercise newly-learned routines.

60.1 Introduction

The MPI library \([25, 21]\) is the *de facto* tool for large scale parallelism as it is used in engineering sciences. In this paper we want to discuss the manner it is usually taught, and propose a rethinking.

We argue that the topics are typically taught in a sequence that is essentially dictated by level of complexity in the implementation, rather than by conceptual considerations. Our argument will be for a sequencing of topics, and use of examples, that is motivated by typical applications of the MPI library, and that explicitly targets the required mental model of the parallelism model underlying MPI.

We have written an open-source textbook \([8]\) with exercise sets that follows the proposed sequencing of topics and the motivating applications.

60.1.1 Short background on MPI

The MPI library dates back to the early days of cluster computing, the first half of the 1990s. It was an academic/industrial collaboration to unify earlier, often vendor-specific, message passing libraries. MPI is
typically used to code large-scale Finite Element Method (FEM) and other physical simulation applications, which share characteristics of a relatively static distribution of large amounts of data – hence the use of clusters to increase size of the target problem – and the need for very efficient exchange of small amounts of data.

The main motivation for MPI is the fact that it can be scaled to more or less arbitrary scales, currently up to millions of cores \cite{1}. Contrast this with threaded programming, which is limited more or less by the core count on a single node, currently about 70.

Considering this background, the target audience for MPI teaching consists of upper level undergraduate students, graduate students, and even post-doctoral researchers who are engaging for the first time in large scale simulations. The typical participant in an MPI course is likely to understand more than the basics of linear algebra and some amount of numerics of Partial Diffential Equation (PDE).

**60.1.2 Distributed memory parallelism**

Corresponding to its origins in cluster computing, MPI targets distributed memory parallelism\(^1\). Here, network-connected cluster nodes run codes that share no data, but synchronize through explicit messages over the network. Its main model for parallelism is described as Single Program Multiple Data (SPMD): multiple instances of a single program run on the processing elements, each operating on their own data. The MPI library then implements the communication calls that allow processes to combine and exchange data.

While MPI programs can solve many or all of the same problems that can be solved with a multicore approach, the programming approach is different, and requires an adjustment in the programmer’s 'mental model' \cite{6, 29} of the parallel execution. This paper addresses the question of how to teach MPI to best effect this shift in mindset.

**Outline of this paper.** We use section 60.2 to address explicitly the mental models that govern parallel thinking and parallel programming, pointing out why MPI is different, and difficult initially. In section 60.3 we consider the way MPI is usually taught, while in section 60.4 we offer an alternative that is less likely to lead to an incorrect mental model.

Some details of our proposed manner of teaching are explored in sections 60.5, 60.6, 60.7. We conclude with discussion in sections 60.8 and 60.9.

**60.2 Implied mental models**

Denning \cite{7} argued how computational thinking consists in finding an abstract machine (a ‘computational model’) that solves the problem in a simple algorithmic way. In our case of teaching parallel programming, the complication to this story is that the problem to be solved is already a computational system. That doesn’t lessen the need to formulate an abstract model, since the full explanation of MPI’s workings are unmanageable for a beginning programmer, and often not needed for practical purposes.

\footnote{Recent additions to the MPI standard target shared memory too.}

\victor
In this section we consider in more detail the mental models that students may implicitly be working under, and the problems with them; targeting the right mental model will then be the subject of later sections. The two (interrelated) aspects of a correct mental model for distributed memory programming are control and synchronization. We here discuss how these can be misunderstood by students.

60.2.1  The traditional view of parallelism

The problem with mastering the MPI library is that beginning programmers take a while to overcome a certain mental model for parallelism. In this model, which we can call ‘sequential semantics’ (or more whimsically the ‘big index finger’ model), there is only a single strand of execution, which we may think of as a big index finger going down the source code.

This mental model corresponds closely to the way algorithms are described in the mathematical literature of parallelism, and it is actually correct to an extent in the context of threaded libraries such as OpenMP, where there is indeed initially a single thread of execution, which in some places spawns a team of threads to execute certain sections of code in parallel. However, in MPI this model is factually incorrect, since there are always multiple processes active, with none essentially privileged over others, and no shared or central data store.

60.2.2  The misconceptions of centralized control

The sequential semantics mental model that, as described above, underlies much of the theoretical discussion of parallelism, invites the student to adopt certain programming techniques, such as the master-worker approach to parallel programming. While this is often the right approach with thread-based coding, where we indeed have a master thread and spawned threads, it is usually incorrect for MPI. The strands of execution in an MPI run are all long-living processes (as opposed to dynamically spawned threads), and are symmetric in their capabilities and execution.

Lack of recognition of this process symmetry also induces students to solve problems by having a form of ‘central data store’ on one process, rather than adopting a symmetric, distributed, storage model. For instance, we have seen a student solve a data transposition problem by collecting all data on process 0, and subsequently distributing it again in transposed form. While this may be reasonable in shared memory with OpenMP, with MPI it is unrealistic in that no process is likely to have enough storage for the full problem. Also, this introduces a sequential bottleneck in the execution.

In conclusion, we posit that beginning MPI programmers may suffer from a mental model that makes them insufficiently realize the symmetry of MPI processes, and thereby arrive at inefficient and nonscalable solutions.

60.2.3  The reality of distributed control

An MPI program run consists of multiple independent threads of control. One problem in recognizing this is that there is only a single source code, so there is an inclination to envision the program execution as

---

2. We carefully avoid the word ‘thread’ which carries many connotations in the context of parallel programming.
3. To first order; second order effects such as affinity complicate this story.
60.3. Teaching MPI, the usual way

A single thread of control: the above-mentioned ‘index finger’ going down the statements of the source. A second factor contributing to this view is that a parallel code incorporates statements with values (int x = 1.5;) that are replicated over all processes. It is easy to view these as centrally executed.

Interestingly, work by Ben-David Kolikant [2] shows that students with no prior knowledge of concurrency, when invited to consider parallel activities, will still think in terms of centralized solutions. This shows that distributed control, such as it appears in MPI, is counterintuitive and needs explicit enforcement in its mental model. In particular, we explicitly target process symmetry and process differentiation.

The centralized model can still be maintained in MPI to an extent, since the scalar operations that would be executed by a single thread become replicated operations in the MPI processes. The distinction between sequential execution and replicated execution escapes many students at first, and in fact, since nothing it gained by explaining this, we do not do so.

60.2.4 The misconception of synchronization

Even with multiple threads of control and distributed data, there is still a temptation to see execution as ‘bulk synchronous processing’ (BSP [28]). Here, the execution proceeds by supersteps, implying that processes are largely synchronized. (The BSP model has several components more, which are usually ignored, notably one-sided communication and processor oversubscription.)

Supersteps as a computational model allow for small differences in control flow, for instance conditional inside a big parallelizable loop, but otherwise imply a form of centralized control (as above) on the level of major algorithm steps. However, codes using the pipeline model of parallelism, such idioms as

```c
MPI_Recv( /* from: */ my_process-1)
// do some major work
MPI_Send( /* to: */ my_process+1)
```

fall completely outside either the sequential semantics or BSP model and require an understanding of one process’ control being dependent on another’s. Gaining an mental model for this sort of unsynchronized execution is nontrivial to achieve. We target this explicitly in section 60.5.1.

60.3 Teaching MPI, the usual way

The MPI library is typically taught as follows. After an introduction about parallelism, covering concepts such as speedup and shared versus distributed memory parallelism, students learn about the initialization and finalization routines, and the MPI_Comm_size and MPI_Comm_rank calls for querying the number of processes and the rank of the current process.

After that, the typical sequence is

1. two-sided communication, with first blocking and later nonblocking variants;
2. collectives; and
3. any number of advanced topics such as derived data types, one-sided communication, subcommunicators, MPI I/O et cetera, in no particular order.
This sequence is defensible from a point of the underlying implementation: the two-sided communication calls are a close map to hardware behavior, and collectives are both conceptually equivalent to, and can be implemented as, a sequence of point-to-point communication calls. However, this is not a sufficient justification for teaching this sequence of topics.

60.3.1 Criticism

We offer three points of criticism against this traditional approach to teaching MPI.

First of all, there is no real reason for teaching collectives after two-sided routines. They are not harder, nor require the latter as prerequisite. In fact, their interface is simpler for a beginner, requiring one line for a collective, as opposed to at least two for a send/receive pair, probably surrounded by conditionals testing the process rank. More importantly, they reinforce the symmetric process view, certainly in the case of the MPI_All... routines.

Our second point of criticism is regarding the blocking and nonblocking two-sided communication routines. The blocking routines are typically taught first, with a discussion of how blocking behavior can lead to load unbalance and therefore inefficiency. The nonblocking routines are then motivated from a point of latency hiding and solving the problems inherent in blocking. In our view such performance considerations should be secondary. Nonblocking routines should instead be taught as the natural solution to a conceptual problem, as explained below.

Thirdly, starting with point-to-point routines stems from a Communicating Sequential Processes (CSP) view of a program: each process stands on its own, and any global behavior is an emergent property of the run. This may make sense for the teacher who know how concepts are realized ‘under the hood’, but it does not lead to additional insight with the students. We believe that a more fruitful approach to MPI programming starts from the global behavior, and then derives the MPI process in a top-down manner.

60.3.2 Teaching MPI and OpenMP

In scientific computing, another commonly used parallel programming system is OpenMP. OpenMP and MPI are often taught together, with OpenMP taught earlier because it is supposedly easier, or because its parallelism would be easier to grasp. Regardless our opinion on the first estimate, we argue that OpenMP should be taught after MPI because of its ‘central control’ parallelism model. If students come to associate parallelism with a model that has a ‘master thread’ and ‘parallel regions’ they will find it much harder to make idiomatic use of the symmetric model of MPI.

60.4 Teaching MPI, our proposal

As alternative to the above sequence of introducing MPI concepts, we propose a sequence that focuses on practical scenarios, and that actively reinforces the mental model of SPMD execution.

Such reinforcement is often an immediate consequence of our strategy of illustrating MPI constructs in the context of an application: most MPI applications (as we shall briefly discuss next) operate on large ‘distributed objects’. This immediately leads to a mental model of the workings of each process being the
60.4. Teaching MPI, our proposal

‘projection’ onto that process of the global calculation. The opposing view, where the overall computation is emergent from the individual processes, is the CSP model mentioned above.

60.4.1 Motivation from applications

The typical application for MPI comes from Computational Science and Engineering, such as N-body problems, aerodynamics, shallow water equations, Lattice Boltzman methods, weather modeling with Fast Fourier Transform. Of these, the PDE based applications can readily be explained to need a number of MPI mechanisms.

Nonnumeric applications exist:

- Graph algorithms such as shortest-path or PageRank are straightforward to explain sequentially. However, the distributed memory algorithms need to be approached fundamentally different from the more naive shared memory variants. Thus they require a good amount of background knowledge. Additionally, they do not feature the regular communications that one-dimensional PDE applications have. Scalability arguments make this story even more complicated. Thus, these algorithms are in fact a logical next topic after discussion of parallel PDE algorithm.

- N-body problems, in their naive implementation, are easy to explain to any student who knows inverse-square laws such as gravity. It is a good illustration of some collectives, but nothing beyond that.

- Sorting. Sorting algorithms based on a sorting network (this includes bubblesort, but not quicksort) can be used as illustration. In fact, we use odd-even transposition sort as a ‘midterm’ exam assignment, which can be solved with MPI_Sendrecv. Algorithms such as bitonic sort can be used to illustrate some advanced concepts, but quicksort, which is relatively easy to explain as a serial algorithm, or even in shared memory, is quite hard in MPI.

- Point-to-point operations can also be illustrated by graphics operations such as a ‘blur’, since these correspond to a ‘stencil’ applied to a cluster of pixels. Unfortunately, this example suffers from the fact that neither collectives, nor irregular communications have a use in this application. Also, using graphics to illustrate simple MPI point-to-point communication is unrealistic in two ways: first, to start out simple we have to posit a one-dimensional pixel array; secondly, graphics is hardly ever of the scale that necessitates distributed memory, so this example is far from ‘real world’. (Ray tracing is naturally done distributed, but that has a completely different computational structure.)

Based on this discussion of possible applications, and in view of the likely background of course attendants, we consider Finite Difference solution of PDEs as a prototypical application that exercises both the simplest and more sophisticated mechanisms. During a typical MPI training, even a one-day short course, we insert a lecture on sparse matrices and their computational structure to motivate the need for various MPI constructs.

60.4.2 Process symmetry

Paradoxically, the first way to get students to appreciate the notion of process symmetry in MPI is to run a non-MPI program. Thus, students are asked to write a ‘hello world’ program, and execute this with

Victor Eijkhout
mpileaks, as if it were an MPI program. Every process executes the print statement identically, bearing out the total symmetry between the processes.

Next, students are asked to insert the initialize and finalize statements, with three different ‘hello world’ statements before, between, and after them. This will prevent any notion of the code between initialization and finalization being considered as an OpenMP style ‘parallel region’.

A simple test to show that while processes are symmetric they are not identical is offered by the exercise of using the MPI_Get_processor_name function, which will have different output for some or all of the processes, depending on how the hostfile was arranged.

### 60.4.3 Functional parallelism

The MPI_Comm_rank function is introduced as a way of distinguishing between the MPI processes. Students are asked to write a program where only one process prints the output of MPI_Comm_size.

Having different execution without necessarily different data is a case of ‘functional parallelism’. At this point there are few examples that we can assign. For instance, in order to code the evaluation of an integral by Riemann sums ($\pi/4 = \int_0^1 \sqrt{1 - x^2}dx$ is a popular one) would need a final sum collective, which has not been taught at this point.

A possible example would be primality testing, where each process tries to find a factor of some large integer $N$ by traversing a subrange of $[2, \sqrt{N}]$, and printing a message if a factor is found. Boolean satisfiability problems form another example, where again a search space is partitioned without involving any data space; a process finding a satisfying input can simply print this fact. However, this example requires background that students typically don’t have.

### 60.4.4 Introducing collectives

At this point we can introduce collectives, for instance to find the maximum of a random value that is computed locally on each process. This requires teaching the code for random number generation and, importantly, setting a process-dependent random number seed. Generating random 2D or 3D coordinates and finding the center of mass is an example that requires a send and receive buffer of length greater than 1, and illustrates that reductions are then done pointwise.

These examples evince both process symmetry and a first form of local data. However, a thorough treatment of distributed parallel data will come in the discussion of point-to-point routines.

It is an interesting question whether we should dispense with ‘rooted’ collectives such as MPI_Reduce at first, and start with MPI_Allreduce. The latter is more symmetric in nature, and has a buffer treatment that is easier to explain; it certainly reinforces the symmetric mindset. There is also essentially no difference in efficiency.

---

4. The MPI_Reduce call performs a reduction on data found on all processes, leaving the result on a ‘root’ process. With MPI_Allreduce the result is left on all processes.
Certainly, in most applications the ‘allreduce’ is the more common mechanism, for instance where the algorithm requires computations such as
\[
\hat{y} \leftarrow \hat{x} / \|\hat{x}\|
\]
where \(x, y\) are distributed vectors. The quantity \(\|\hat{x}\|\) is then needed on all processes, making the Allreduce the natural choice. The rooted reduction is typically only used for final results. Therefore we advocate introducing both rooted and nonrooted collectives, but letting the students initially do exercises with the nonrooted variants.

This has the added advantage of not bothering the students initially with the asymmetric treatment of the receive buffer between the root and all other processes.

60.4.5 Distributed data

As motivation for the following discussion of point-to-point routines, we now introduce the notion of distributed data. In its simplest form, a parallel program operates on a linear array the dimensions of which exceed the memory of any single process.

```c
int n;
double data[n];
```

![Figure 60.2: A distributed array versus multiple local arrays](image)

The lecturer stresses that the global structure of the distributed array is only ‘in the programmer’s mind’: each MPI process sees an array with indexing starting at zero. The following snippet of code is given for the students to use in subsequent exercises:

```c
int myfirst = ......;
for (int ilocal=0; ilocal<nlocal; ilocal++) {
    int iglobal = myfirst+ilocal;
    array[ilocal] = f(iglobal);
}
```

At this point, the students can code a second variant of the primality testing exercise above, but with an array allocated to store the integer range. Since collectives are now known, it becomes possible to have a single summary statement from one process, rather than a partial result statement from each.

The inner product of two distributed vectors is a second illustration of working with distributed data. In this case, the reduction for collecting the global result is slightly more useful than the collective in the previous examples. For this example no translation from local to global numbering is needed.
60.4.6 Point-to-point motivated from operations on distributed data

We now state the importance of local combining operations such as

$$y_i = \frac{x_{i-1} + x_i + x_{i+1}}{3} : i = 1, \ldots, N - 1$$

applied to an array. Students who know about PDEs will recognize that with different coefficients this is the heat equation; for others a graphics ‘blur’ operation can be used as illustration, if they accept that a one-dimensional pixel array is a stand-in for a true graphic.

Under the ‘owner computes’ regime, where the process that stores location $y_i$ performs the full calculation of that quantity, we see the need for communication in order to compute the first and last element of the local part of $y$:

We then state that this data transfer is realized in MPI by two-sided send/receive pairs.

60.4.7 Detour: deadlock and serialization

The concept of ‘blocking’ is now introduced, and we discuss how this can lead to deadlock. A more subtle behavior is ‘unexpected serialization’: processes interacting to give serial behavior on a code that conceptually should be parallel. (The classroom protocol is discussed in detail in section 60.5.1.) For completeness, the ‘eager limit’ can be discussed.

This introduces students to an interesting phenomenon in the concept of parallel correctness: a program may give the right result, but not with the proper parallel efficiency. Asking a class to come up with a solution that does not have a running time proportional the number of processes, will usually lead to at least one student suggesting splitting processes in odd and even subsets. The limits to this approach, code complexity and the reliance on regular process connectivity, are explained to the students as a preliminary to the motivation for nonblocking sends; section 60.4.10.

60.4.8 Detour: ping-pong

At this point we briefly abandon the process symmetry, and consider the ping-pong operation between two processes A and B. We ask students to consider what the ping-pong code looks like for A and, for B. Since we are working with SPMD code, we arrive at a program where the A code and B code are two branches of a conditional.

We ask the students to implement this, and do timing with MPI_Wtime. The implementation of the ping-pong is itself a good exercise in SPMD thinking; finding the right sender/receiver values usually takes the students a nontrivial amount of time. Many of them will initially write a code that deadlocks.

---

5. In this operations, process A sends to B, and B subsequently sends to A. Thus the time for a message is half the time of a ping-pong. It is not possible to measure a single message directly, since processes can not be synchronized that finely.
The concepts of latency and bandwidth can be introduced, as the students test the ping-pong code on messages of increasing size. The concept of halfbandwidth can be introduced by letting half of all processes execute a ping-pong with a partner process in the other half.

### 60.4.9 Back to data exchange

The foregoing detours into the behavior of two-sided send and receive calls were necessary, but they introduced asymmetric behavior in the processes. We return to the averaging operation given above, and with it to a code that treats all processes symmetrically. In particular, we argue that, except for the first and last, each process exchanges information with its left and right neighbor.

This could be implemented with blocking sends and receive calls, but students recognize how this could be somewhere between tedious and error-prone. Instead, to prevent deadlock and serialization as described above, we now offer the `MPI_Sendrecv` routine. Students are asked to implement the classroom exercise above with the `sendrecv` routine. Ideally, they use timing or tracing to gather evidence that no serialization is happening.

As a nontrivial example (in fact, this takes enough programming that one might assign it as an exam question, rather than an exercise during a workshop) students can now implement an odd-even transposition sort algorithm using `MPI_Sendrecv` as the main tool. For simplicity they can use a single array element per process. (If each process has a subarray one has to make sure their solution has the right parallel complexity. It is easy to make errors here and implement a correct algorithm that, however, performs too slowly.)

Note that students have at this point not done any serious exercises with the blocking communication calls, other than the ping-pong. No such exercises will in fact be done.

### 60.4.10 Nonblocking sends

Nonblocking sends are now introduced as the solution to a specific problem: the above schemes required paired-up processes, or careful orchestration of send and receive sequences. In the case of irregular communications this is no longer possible or feasible. Life would be easy if we could declare ‘this data needs to be sent’ or ‘these messages are expected’, and then wait for these messages collectively. Given this motivation, it is immediately clear that multiple send or receive buffers are needed, and that requests need to be collected.

Implementing the three-point averaging with nonblocking calls is at this point an excellent exercise.

Note that we have here motivated the nonblocking routines to solve a symmetric problem. Doing this should teach the students the essential point that each nonblocking call needs its own buffer and generates its own request. Viewing nonblocking routines as a performance alternative to blocking routines is likely to lead to students re-using buffers or failing to save the request objects. Doing so is a correctness bug that is very hard to find, and at large scale it induces a memory leak since many requests objects are lost.

---

6. The `MPI_Sendrecv` call combines a send a receive operation, specifying for each process both a sending and receiving communication. The execution guarantees that no deadlock or serialization will occur.
60.4.11 Taking it from here

At this point various advanced topics can be discussed. For instance, Cartesian topologies can be introduced, extending the linear averaging operation to a higher dimensional one. Subcommunicators can be introduced to apply collectives to rows and columns of a matrix. The recursive matrix transposition algorithm is also an excellent application of subcommunicators.

However, didactically these topics do not require the careful attention that the introduction of the basic concepts needs, so we will not go into further detail here.

60.5 ‘Parallel computer games’

Part of the problem in developing an accurate mental model of parallel computation is that there is no easy way to visualize the execution. While sequential execution can be imagined with the ‘big index finger’ model (see section 60.2.1), the possibly unsynchronized execution of an MPI program makes this a gross simplification. Running a program in a parallel graphical environment (such as the DDT debugger or the Eclipse PTP IDE) would solve this, but they introduce much learning overhead. Ironically, the low tech solution of

    mpiexec -n 4 xterm -e gdb program

is fairly insightful, but having to learn gdb is again a big hurdle.

We have arrived at the somewhat unusual solution of having students act out the program in front of the class. With each student acting out the program, any interaction is clearly visible to an extent that is hard to achieve any other way.

60.5.1 Sequentialization

Our prime example is to illustrate the blocking behavior of MPI_Send and MPI_Recv\(^7\). Deadlock is easy enough to understand as a consequence of blocking – in the simplest case of deadlock to processes are both blocked expecting a receive from the other – but there are more subtle effects that will come as a surprise to students. (This was alluded to in section 60.4.7.)

Consider the following basic program:

- Pass a data item to the next higher numbered process.

Note that this is conceptually a fully parallel program, so it should execute in time \(O(1)\) in terms of the number of processes.

In terms of send and receive calls, the program becomes

- Send data to the next higher process;
- Receive data from the next lower process.

The final detail concerns the boundary conditions: the first process has nothing to receive and the last one has nothing to send. This makes the final version of the program:

\[^7\text{Blocking is defined as the process executing a send or receive call halting until the corresponding operation is executing.}\]
• If you are not the last process, send data to the next higher process; then
• If you are not the first process, receive data from the next lower process.

To have students act this out, we tell them to hold a pen in their right hand, and put the left hand in a pocket or behind their back. Thus, they have only one 'communication channel'. The ‘send data’ instruction becomes ‘turn to your right and give your pen’, and ‘receive data’ becomes ‘turn to your left and receive a pen’.

Executing this program, the students first all turn to the right, and they see that giving data to a neighbor is not possible because no one is executing the receive instruction. The last process is not sending, so moves on to the receive instruction, after which the penultimate process can receive, et cetera.

This exercise makes the students see, better than any explanation or diagram, how a parallel program can compute the right result, but with unexpectedly low performance because of the interaction of the processes. (In fact, we have had explicit feedback that this game was the biggest lightbulb moment of the class.)

60.5.2 Ping-pong

While in general we emphasize the symmetry of MPI processes, during the discussion of send and receive calls we act out the ping-pong operation (one process sending data to another, followed by the other sending data back), precisely to demonstrate how asymmetric actions are handled. For this, two students throw a pen back and forth between them, calling out ‘send’ and ‘receive’ when they do so.

The teacher then asks each student what program they executed, which is ‘send-receive’ for the one, and ‘receive-send’ for the other student. Incorporating this in the SPMD model then leads to a code with conditionals to determine the right action for the right process.

60.5.3 Collectives and other games

Other operations can be acted out by the class. For instance, the teacher can ask one student to add the grades of all students, as a proxy for a reduction operation. The class quickly sees that this will take a long time, and strategies such as taking by-row sums in the classroom quickly suggest themselves.

We have at one point tried to have a pair of student act out a ‘race condition’ in shared memory programming, but modeling this quickly became too complicated to be convincing.

60.5.4 Remaining questions

Even with our current approach, however, we still see students writing idioms that are contrary to the symmetric model. For instance, they will write

```c
for (p=0; p<nprocs; p++)
    if (p==myrank)
        // do some function of p
```
This code computes the correct result, and with the correct performance behavior, but it still shows a conceptual misunderstanding. As one of the ‘parallel computer games’ (section 60.5) we have put a student stand in front of the class with a sign ‘I am process 5’, and go through the above loop out loud (‘Am I process zero? No. Am I process one? No.’) which quickly drives home the point about the futility of this construct.

60.6 Further course summary

We have taught MPI based on the above ideas in two ways. First, we teach an academic class, that covers MPI, OpenMP, and general theory of parallelism in one semester. The typical enrollment is around 30 students, who do lab exercises and a programming project of their own choosing. We also teach a two-day intensive workshop (attendance 10–40 students depending on circumstances) of 6–8 hours per day. Students of the academic class are typically graduate or upper level undergraduate students; the workshops get attendance from post-docs, academics, and industry too. The typical background is applied math, engineering, physical sciences.

We cover the following topics, with division over two days in the workshop format:

- Day 1: familiarity with SPMD, collectives, blocking and nonblocking two-sided communication.
- Day 2: exposure to: sub-communicators, derived datatypes. Two of the following: MPI-I/O, one-sided communication, process management, the profiling and tools interfaces, neighborhood collectives.

60.6.1 Exercises

On day 1 the students do approximately 10 programming exercises, mostly finishing a skeleton code given by the instructor. For the day 2 material students do two exercises per topic, again starting with a given skeleton. (Skeleton codes are available as part of the repository [8].)

The design of these skeleton codes is an interesting problem in view of our concern with mental models. The skeletons are intended to take the grunt work away from the students, to both indicate a basic code structure and relieve them from making elementary coding errors that have no bearing on learning MPI. On the other hand, the skeletons should leave enough unspecified that multiple solutions are possible, including wrong ones: we want students to be confronted with conceptual errors in their thinking, and a too-far-finished skeleton would prevent them from doing that.

Example: the prime finding exercise mentioned above (which teaches the notion of functional parallelism) has the following skeleton:

```c
int myfactor;
// Specify the loop header:
// for ( ... myfactor ... )
for (
    /**** your code here ****/
) {
    if (bignum%myfactor==0)
        printf("Process %d found factor %d\n", 
```
This leaves open the possibility of both a blockwise and a cyclic distribution of the search space, as well as incorrect solutions where each process runs through the whole search space.

60.6.2 Projects

Students in our academic course do a programming project in place of a final exam. Students can choose between one of a set of standard projects, or doing a project of their own choosing. In the latter case, some students will do a project in context of their graduate research, which means that they have an existing codebase; others will write code from scratch. It is this last category, that will most clearly demonstrate their correct understanding of the mental model underlying SPMD programs. However, we note that this is only a fraction of the students in our course, a fraction made even smaller by the fact that we also give a choice of doing a project in OpenMP rather than MPI. Since OpenMP is, at least to the beginning programmer, simpler to use, there is an in fact a clear preference for it among the students who pick their own project.

60.7 Prospect for an online course

Currently the present author teaches MPI in the form of an academic course or short workshop, as outlined in section 60.6. In both cases, lecture time is far less than lab time, making the setup very intensive in teacher time. It also means that this setup is not scalable to a larger number of students. Indeed, while the workshops are usually webcast, we have not sufficiently solved the problem of supporting remote students. (The Pittsburgh Supercomputing Center offers courses that have remotely located teaching assistants, which seems a promising approach.) Such problems of support would be even more severe with an online course, where in-person support is completely absent.

One obvious solution to online teaching is automated grading: a student submits an exercise, which is then run through a checker program that tests the correct output. Especially if the programming assignment takes input, a checker script can uncover programming errors, notably in boundary cases.

However, the whole target of this paper is to uncover conceptual misunderstandings, for instance such as can lead to correct results with sub-optimal performance. In a classroom situation such misunderstandings are quickly caught and cleared up, but to achieve this in a context of automated grading we need to go further.

We have started experiments with actually parsing code submitted by the students. This effort started in a beginning programming class taught by the present author, but is now being extended to the MPI courses.

It is possible to uncover misconceptions in students' understanding by detecting the typical manifestations of such misconceptions. For instance, the code in section 60.5.4 can be uncovered by detecting a loop where the upper bound involves a variable that was set by MPI_Comm_size. Many MPI codes have no need for such a loop over all processes, so detecting one leads to an alert for the student.

Note that no tools exist for such automated evaluation. The source code analysis needed falls far short of full parsing. On the other hand, the sort of constructs is it supposed to detect, are normally not of interest.
to the writers of compilers and source translators. This means that by writing fairly modest parsers (say, less than 200 lines of python) we can perform a sophisticated analysis of the students’ codes. We hope to report on this in more detail in a follow-up paper.

60.8 Evaluation and discussion

At the moment, no rigorous evaluation of the efficacy of the above ideas has been done. We intend to perform a comparison between outcomes of the proposed way of teaching and the traditional way by comparing courses at two (or more) different institutions and from different syllabi. The evaluation will then be based on evaluating the independent programming project.

However, anecdotal evidence suggests that students are less likely to develop ‘centralized’ solutions as described in section 60.2.2. This was especially the case in our semester-long course, where the students have to design and implement a parallel programming project of their own choosing. After teaching the ‘symmetric’ approach, no students wrote code based on a manager-worker model, or using centralized storage. In earlier semesters, we had seen students do this, even though this model was never taught as such.

60.9 Summary

In this paper we have introduced a nonstandard sequence for presenting the basic mechanisms in MPI. Rather than starting with sends and receives and building up from there, we start with mechanisms that emphasize the inherent symmetry between processes in the SPMD programming model. This symmetry requires a substantial shift in mindset of the programmer, and therefore we target it explicitly.

In general, it is the opinion of this author that it pays off to teach from the basis of instilling a mental model, rather than of presenting topics in some order of (perceived) complexity or sophistication.

Comparing our presentation as outlined above to the standard presentation, we recognize the downplaying of the blocking send and receive calls. While students learn these, and in fact learn them before other send and receive mechanisms, they will recognize the dangers and difficulties in using them, and will have the combined sendrecv call as well as nonblocking routines as standard tools in their arsenal.
60.10 Sources used in this chapter

60.10.1 Listing of code header
Teaching from mental models
PART IX

BIBLIOGRAPHY, INDEX, AND LIST OF ACRONYMS
Chapter 61

Bibliography


Chapter 62

List of acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programmer Interface</td>
</tr>
<tr>
<td>AMG</td>
<td>Algebraic MultiGrid</td>
</tr>
<tr>
<td>AVX</td>
<td>Advanced Vector Extensions</td>
</tr>
<tr>
<td>BLAS</td>
<td>Basic Linear Algebra Subprograms</td>
</tr>
<tr>
<td>BSP</td>
<td>Bulk Synchronous Parallel</td>
</tr>
<tr>
<td>CAF</td>
<td>Co-array Fortran</td>
</tr>
<tr>
<td>CPP</td>
<td>C PreProcessor</td>
</tr>
<tr>
<td>CRS</td>
<td>Compressed Row Storage</td>
</tr>
<tr>
<td>CSP</td>
<td>Communicating Sequential Processes</td>
</tr>
<tr>
<td>CG</td>
<td>Conjugate Gradients</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute-Unified Device Architecture</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
</tr>
<tr>
<td>DPCPP</td>
<td>Data Parallel C++</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>FEM</td>
<td>Finite Element Method</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In / First-Out</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating Point Unit</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>FSA</td>
<td>Finite State Automaton</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HPC</td>
<td>High-Performance Computing</td>
</tr>
<tr>
<td>HPF</td>
<td>High Performance Fortran</td>
</tr>
<tr>
<td>HPL</td>
<td>High Performance Linpack</td>
</tr>
<tr>
<td>ICV</td>
<td>Internal Control Variable</td>
</tr>
<tr>
<td>LAPACK</td>
<td>Linear Algebra Package</td>
</tr>
<tr>
<td>MG</td>
<td>Multi-Grid</td>
</tr>
<tr>
<td>MIC</td>
<td>Many Integrated Cores</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instruction Multiple Data</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>MPL</td>
<td>Message Passing Layer</td>
</tr>
<tr>
<td>MPMD</td>
<td>Multiple Program Multiple Data</td>
</tr>
<tr>
<td>MTA</td>
<td>Multi-Threaded Architecture</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Interface Card</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-Uniform Memory Access</td>
</tr>
<tr>
<td>OO</td>
<td>Object-Oriented</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PGAS</td>
<td>Partitioned Global Address Space</td>
</tr>
<tr>
<td>PDE</td>
<td>Partial Differential Equation</td>
</tr>
<tr>
<td>PRAM</td>
<td>Parallel Random Access Machine</td>
</tr>
<tr>
<td>RDMA</td>
<td>Remote Direct Memory Access</td>
</tr>
<tr>
<td>RMA</td>
<td>Remote Memory Access</td>
</tr>
<tr>
<td>SAN</td>
<td>Storage Area Network</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as-a Service</td>
</tr>
<tr>
<td>SFC</td>
<td>Space-Filling Curve</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single Instruction Multiple Thread</td>
</tr>
<tr>
<td>SLURM</td>
<td>Simple Linux Utility for Resource Management</td>
</tr>
<tr>
<td>SM</td>
<td>Streaming Multiprocessor</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multi Processing</td>
</tr>
<tr>
<td>SOR</td>
<td>Successive Over-Relaxation</td>
</tr>
<tr>
<td>SP</td>
<td>Streaming Processor</td>
</tr>
<tr>
<td>SPMD</td>
<td>Single Program Multiple Data</td>
</tr>
<tr>
<td>SPD</td>
<td>symmetric positive definite</td>
</tr>
<tr>
<td>SSE</td>
<td>SIMD Streaming Extensions</td>
</tr>
<tr>
<td>STL</td>
<td>Standard Template Library</td>
</tr>
<tr>
<td>TACC</td>
<td>Texas Advanced Computing Center</td>
</tr>
<tr>
<td>TLB</td>
<td>Translation Look-aside Buffer</td>
</tr>
<tr>
<td>UMA</td>
<td>Uniform Memory Access</td>
</tr>
<tr>
<td>UPC</td>
<td>Unified Parallel C</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide Area Network</td>
</tr>
</tbody>
</table>
Chapter 63

General Index
Index

-malloc_debug, 659
-malloc_test, 659
.petscrc, 665

active target synchronization, 311, 316
adaptive integration, see quadrature, adaptive address
    physical, 388, 526
    virtual, 388, 526
adjacency
    graph, 608
affinity, 711, 715
    process and thread, 711–712
    thread
        on multi-socket nodes, 522
alignment, 225
all-to-all, 43
allocate
    and private/shared data, 493
allreduce, 43
AMD
    Milan, 470
argc, 29, 32
argv, 29, 32
array
    static, 121
asynchronous, 417
atomic operation, 499, 531
    file, 364
    MPI, 325–330
    OpenMP, 501–502
backfill, 752
bandwidth, 87

    bisection, 92
barrier
    for timing, 420
    implicit, 531
    nonblocking, 85
Basic Linear Algebra Subprograms (BLAS), 583
batch
    job, 18, 748
    scheduler, 18
    script, 748
    system, 746
Beowulf cluster, 17
block row, 595
Boolean satisfiability, 37
boost, 20
breakpoint, 732
broadcast, 41
btl_openib_eager_limit, 127
btl_openib_rndv_eager_limit, 127
bucket brigade, 65, 91, 129, 738
buffer
    MPI, in C, 47
    MPI, in Fortran, 48
    MPI, in MPL, 48
    MPI, in Python, 48
    receive, 122
butterfly exchange, 739

C
    C99, 195
    MPI bindings, see MPI, C bindings
C++
    bindings, see MPI, C++ bindings
    first-touch, see first-touch, in C++
standard library, 217
  vector, 217
C++ iterators
  in OMP reduction, 482
c_sizeof, 202
cacheline, 478
callback, 644
cast, 47
CC, 736
channel, 703
Charmpp, 18
chunk, 466
chunk, 467
Clang, 535
clang, 19
client, 293
CLINKER, 574
cluster, 746
cmake, 752
Codimension, 676
coherent memory, see memory, coherence
  collective
    split, 360
collectives, 41
    neighborhood, 373, 422
    nonblocking, 81
canceling, 428
column-major storage, 209
combiner, 231
communication
  asynchronous, 187
  blocking, 124–129
    vs nonblocking, 421
  buffered, 188, 188, 422
  local, 188
  nonblocking, 135–147
  nonlocal, 188
  one-sided, 311–338
  one-sided, implementation of, 338
  partitioned, see partitioned, communication
  persistent, 422
  synchronous, 187
  two-sided, 159
communicator, 33, 271–281
  info object, 408
    inter, 277, 278, 289
    from socket, 296
    intra, 278, 279
    object, 33
    peer, 278
    variable, 33
compare-and-swap, 132
compiler, 220
  optimization level, 727
completion, 332
  local, 332
  remote, 332
compute
  node, 746
configure.log, 578
construct, 445
contention, 92
contiguous
  data type, 202
control variable
  access, 401
  handle, 400
core, 26, 440
core dump, 728
cosubscript, 677
cpp, 443
cpuinfo, 708
Cray
  MPI, 19
  T3E, 428
critical section, 477, 500, 544
    flush at, 533
curly braces, 444
Dalcin
  Lisandro, 21, 574
data dependency, 514
Data Parallel C++ (DPCPP), 681
data race, see race condition, see race condition
  datatype, 194–234
    big, 221–224
derived, 194, 202–356
different on sender and receiver, 207
  elementary, 194–202
    in C, 195
    in Fortran, 195
    in Python, 198
  signature, 218
  date, 750
  ddd, 727
  DDT, 727, 734
  deadlock, 85, 121, 125, 135, 421, 423, 734
  debug flag, 728
  debug_mt, 418
  debugger, 727
  debugging, 727–734
    parallel, 733
  Dekker’s algorithm, 532
  dense linear algebra, 273
  Depth First Search (DFS), 565
  destructor, 270
  directive
    end-of, 444
  directives, 443, 443–444
    cpp, 452
  displacement unit, 335
  distributed array, 37
  distributed shared memory, 311
  doubling
    recursive, see recursive doubling
  dynamic mode, 441
  eager limit, 125–127, 421
  eager send
    and non-blocking, 127
  Eclipse, 734
    PTP, 734
  edge
    cuts, 608
    weight, 608
  ensemble, 751
  envelope, see message, envelope
  environment
    of batch job, 750
  environment variables, 396
  epoch, 316
  access, 316, 318, 331
  communication, 316
  completion, 334
  exposure, 316, 318
  passive target, 330
  error return, 20
  ethernet, 20
  false sharing, 461, 478
  Fast Fourier Transform (FFT), 59, 605, 606
  fat-tree, 711
  fence, 316
  fftw, 578, 605
  Fibonacci sequence, 505–507
  file
    pointer
      advance by write, 360
      individual, 360
      system
        shared, 752
  file system
    shared, 356
  first-touch, 526, 711
    in C++, 527
  five-point stencil, 85
  FLINKER, 574
  fork/join model, 440, 447, 514
  Fortran
    1-based indexing, 140
    2008, 33
    array syntax, 488
    assumed-shape arrays in MPI, 416
    fixed-form source, 444
    forall loops, 488
    Fortran2003, 151, 480
    Fortran2008, 203, 208, 321
      MPI bindings, see MPI, Fortran2008
        bindings
    Fortran2018, 417
    Fortran77, 77
      PETSc interface, 573
    Fortran90, 23, 29, 151, 197, 208
      bindings, 21
    Fortran9090
PETSc interface, 573
line length, 657
MPI bindings, see MPI, Fortran bindings
MPI equivalences of scalar types, 197
MPI issues, 416–417
gather, 41
Gauss-Jordan algorithm, 55
Gaussian elimination, 636
GCC, 535
gcc
   thread affinity, 528
gdb, 727–734
gemv, 601
getrusage, 665
ghost region, 771
GNU, 727
gdb, see gdb
GPUDirect, 654
Gram-Schmidt, 46
graph
   partitioning
      packages, 608
topology, 373, 422, 711
   unweighted, 375
grid
   Cartesian, 367, 617
   periodic, 367
   processor, 711
group, 317
group of
   processors, 319
halo, 771
   update, 322
halo region, 606, 620
handshake, 423
HDF5, 755
hdf5, 356, 721
heap, 446
heat equation, 523
Hessian, 649
hipsycl, 688
histogram, 502
hostname, 411
hwloc, 708, 712
hybrid
   computing, 751
   hydra, 295
   hyperthread, 462
   hyperthreading, 711
Hypre, 578, 642, 643
   pilut, 643
I/O
   in OpenMP, 487
I_MPI_ASYNC_PROGRESS, 418
I_MPI_ASYNC_PROGRESS_THREADS, 418
I_MPI_EAGER_THRESHOLD, 127
I_MPI_SHM_EAGER_THRESHOLD, 127
IBM
   Power9, 471
ibrun, 292, 576, 737
ICC, 535
idle, 751
image, 676
image_index, 677
immediate operation, 135
implicit barrier, 500
incomplete operation, see operation, incomplete
   indexed
   data type, 202
inner product, 46
input redirection
   shell, 424
instrumentation, 736
Intel, 423
   Cascade Lake, 461, 470, 526, 561
   compiler
      optimization report, 462
      thread affinity, 528
   compiler suite, 708
Haswell, 712
Knight’s Landing, 440, 536
   thread placement, 525
Knights Landing, 462, 470, 715
Knightslanding, 746
MPI, 19, 92, 127, 292, 417, 425, 715
INDEX

Paragon, 417
Sandybridge, 440, 712
Skylake, 470, 746
interconnect, 744
Internal Control Variable (ICV), 543–545
Java, 17
job, 746
array, 751
cancel, 749
job script, 748
jumpshot, 738
KIND, 200
KMP_AFFINITY, 528
Laplace equation, 598
latency, 87, 432
hiding, 144, 421, 589, 591, 600
launcher, 751
lcobound, 677
league(OpenMP), 541
lexical scope, 490
Linear Algebra Package (LAPACK), 583
linked list, 511
linker
weak symbol, 420
listing, 759
little-endian, 755
load
balancing, 464
imbalance, 464
local
operation, 188, 189
local operation, 146
local refinement, 85
lock, 502, 502–505
flush at, 533
nested, 504
login
node, 746
Lonestar5, 712
LU factorization, 466, 636
Lustre, 721
malloc, 446, 526
make.log, 578
macports, 411
matrix
sparse, 81, 600
transposition, 274
matrix-vector product, 636
dense, 64
sparse, 67
Mellanox, 422
memory
coherent, 334
high-bandwidth, 496
model, see window, memory, model
non-volatile, 496
page, 526
shared, MPI, 314
memory leak, 147
message
collision, 422
count, 152
envelope, 125, 155
source, 122
status, 123, 149–155
tag, 152
error, 152
source, 150
tag, 152
synchronous, 125
tag, 121, 396
Message Passing Layer (MPL), 20, 35
message
target, 121
MKL, 583
mkll, 579
ML, 643
Monte Carlo codes, 37
motherboard, 439
move_pages, 528

MPI
  C bindings, 20
  C++ bindings, 20
  constants, 426–427
    compile-time, 426
    link-time, 426
  datatype
    extent, 224
    size, 201
    subarray, 225
    vector, 201
  Fortran bindings, 20–21
  Fortran issues, see Fortran, MPI issues
  Fortran2008 bindings, 20–21
  I/O, 414, 721
  initialization, 29
  MPI-1, 367, 379
  MPI-2, 289, 408, 409, 413
  MPI-3, 45, 81, 212, 217, 222, 325, 384, 416
    C++ bindings removed, 20
    Fortran2008 interface, 21
  MPI-3.1, 426
  MPI-3.2, 428
  MPI-4, 20, 45, 202, 221, 384, 414
  MPI-4.1, 196
  Python bindings, 21
  semantics, 423
  tools interface, 399–404
  version, 411
mpi.h, 21, 29
MPI/O, 356–364
mp14py, 21, 35, 578, 750
mpi_f08, 21, 29, 150, 222, 321
mpicc, 19
mpich, 19
mpich, 425
mpicxx, 19
mpiexec
  and environment variables, 396
  options, 19
stdout/err of, 424
mpiexec, 18–20, 24, 28, 32, 266, 289, 292, 396, 424, 425, 784
mpif.h, 29
mpif90, 19
MPIR, 423
mpirun, 18, 32
mpirun, 425
MPL, 20
  compiling and linking, 20
mulpd, 536
mulsd, 536
multicore, 441
Multiple Program Multiple Data (MPMD), 28, 425
multiprocessing, 701
mumps, 578
MV2_IBA_EAGER_THRESHOLD, 127
mvapich, 396
mvapich2, 127, 717

N-body problem, 560
name server, 295
nested parallelism, 452–454
NetCDF, 755
netcdf, 356
network
  card, 422
  contention, 422
  port
  oversubscription, 422
new, 446
Newton’s method, 649
node, 26, 746
  cluster, 439
  non-blocking communication, 133
Non-Uniform Memory Access (NUMA), 710
norm
  one, 80
np.frombuffer, 79
NULL, 399
null terminator, 406
num_images, 677
numactl, 528, 711
INDEX

numerical integration, 460
Numpy, 198
  1.20, 198
numpy, 21, 48, 200, 314
od, 356
offloading
  vs onloading, 422
omp
  for
    barrier behavior, 500
    reduction, 476–483
    user-defined, 480–483
OMP_NUM_THREADS, 396, 751
OneAPI, 681
onloading, see offloading, vs onloading
opaque handle, 33, 47
OpenMP, 384
  accelerator support in, 547
  co-processor support in, 547
  compiling, 442–443
  environment variables, 443, 447, 543–545
  library routines, 447
  library routines, 543–545
macro, 443
OpenMP-3, 460
OpenMP-3.1, 479
OpenMP-4, 568
OpenMP-4.0, 517, 547
OpenMP-4.5, 479, 548
OpenMP-5, 496
OpenMP-5.0, 463, 516, 547, 548
OpenMP-5.1, 548
OpenMP-5.2, 548
places, 522
running, 443
tasks, 511–518
  data, 512–513
  dependencies, 514–516
  synchronization, 513–514
OpenMPI, 19, 127, 385
operating system, 547
operation
  non-local, 124
operator, 76–80
  predefined, 76
  user-defined, 78
option
  prefix, 664
origin, 311, 318
overlapping computation and communication,
  see latency, hiding
owner computes, 118
package, 708
packing, 232
page
  small, 387
  table, 388, 526
page, memory, see memory, page
parallel
  data, 528
  embarrassingly, 528
  parallel region, 441, 450–454, 486
    dynamic scope, 453, 491
  flush at, 533
parallel regions
  nested, 544
parameter sweep, 702, 751
paraprof, 737, 738
parasails, 642
ParMetis, 422, 608
partition, 749
partitioned communication, 185–187
passive target synchronization, 312, 327, 330
pbing, 528
persistent
  collectives, 184–185
  communication, 185
  point-to-point, 182–183
persistent communication, 133, see
  communication, persistent
PETSc, 422
  interoperability with BLAS, 583
  interoperability with MPI, 583
  log files, 578
PETSC_OPTIONS, 665
pin a thread, 711
ping-pong, 119, 420, 678, 735
pipe, 703
PMI_RANK, 425
PMI_SIZE, 425
point-to-point, 118
pointer
 null, 63
polling, 142, 417
posting
 of send/receive, 136
pragma, 443
preconditioner, 637, 641
 block jacobi, 664
 field-split, 605
prefix operation, 57
private variables, 446
proc_bind, 451
process, 26
 set, 298
processes status of, 425
producer-consumer, 530
PROFFILEDIR, 737
progress
 asynchronous, 144, 417
protocol, 125
 rendezvous, 125
pthreads, 417
purify, 731
PVM, 18, 289
pylauncher, 751
Python
 MPI bindigs, see MPI, Python bindings
 multiproccessing, 751
PETSc interface, 574
 quadrature
 adaptive, 465
queue, 747
 in-order, 683
 SYCL, 682
 race condition, 325, 395, 445, 476, 501, 501, 531, 545
 in co-array Fortran, 677
 in MPI/OpenMP, 395
 in OpenMP, 531–533
 in SYCL, 687
 radix sort, 67
RAID
 array, 752
RAM
 disc, 752
random number generator, 496, 546
Ranger, 711
 rar, 336
 raw, 336
ray tracing, 388
recursive doubling, 92
redirection, see shell, input redirection
reduction, 41
region of code, 445
register
 SSE2, 536
release_mt, 418
residual, 637
Riemann
 sum, 476
Riemann sums, 461
RMA
 active, 311
 passive, 312
root, 49
 root process, 41
sbatch, 748
scalable
 in space, 91
 in time, 90
scalapack, 646
 scan, 43
 exclusive, 58
 inclusive, 57
scancel, 748
scanf, 661
scatter, 41
sched_setaffinity, 528
scope
 lexical, 441, 445
of variables, 441, 445
SEEK_SET, 364
segfault, 659
segmentation fault, 729
segmented scan, 60
send
  buffer, 121
  ready mode, 188
  synchronous, 188
sentinel, 444, 452
sequential
  semantics, 573
sequential consistency, 532
serialization, 128
server, 293
session, 297
  performance experiment, 402
session model, 297, 297
sessions model, 30
SetThreadAffinityMask, 528
shared data, 441
shared memory, see memory, shared
shared variables, 446
shell
  matrix, 644
shmem, 428
siilo, 356
SimGrid, 90, 744–745
  compiler, 744
sinfo, 747
Single Program Multiple Data (SPMD), 444, 450
sizeof, 200, 314, 416
sleep, 750
SLURM, 365
smpicc, 744
smpirun, 744
socket, 26, 296, 386, 439, 710, 716
  dual, 526
sort
  odd-even transposition, 132
  radix, 67
  swap, 132
sparse approximate inverse, 641, 642
sparse matrix vector product, 60
spin loop, 417
spin-lock, 544
ssqueue, 748, 749, 751
ssrun, 748
ssh, 18
  connection, 746
ssh, 751
stack, 446, 544
  overflow, 491
  per thread, 491
Stampede, 746
  compute node, 712
  largemem node, 712
  node, 440
Stampede2, 746
standard deviation, 42
start/affinity
status
  of received message, 149
stderr, 424
stdout, 424
stencil, 617
storage association, 491, 494
storage_size, 202
stride, 209
stringstream, 451
struct
  data type, 202
structured block, 445
Sun
  compiler, 528
SUNW_MP_PROCBIND, 528
SYCL, 681
symbol table, 727, 728
sync, 677
synchronization
  in OpenMP, 499–507
T3PIO, 362
TACC, 746
  Frontera, 461, 526, 561
  portal, 574
  Stampede2, 462
tacc_affinity, 528, 711, 717
TACC_TAU_DIR, 736
tag, see message, tag
    bound on value, 410
target, 311, 318
    active synchronization, see active target synchronization
    passive synchronization, see passive target synchronization
task
    generating, 540
    initial, 540
    scheduler, 511
    scheduling point, 516
target, 540
taskset, 528
TAU, 736–740
TAU_PROFILE, 737
tau_timecorrect, 737
TAU_TRACE, 737
team(OpenMP), 541
this_image, 677
thread
    affinity, 522–525
    initial, 451
    master, 451
    migrating a, 525
    primary, 451
    private data, 494
thread-safe, 545–546
threads, 440
    hardware, 441, 711
    master, 441
team of, 440, 451
time slicing, 26, 441
time-slicing, 289
timing
    MPI, 418–420
top, 749, 751
topology
    virtual, 367
TotalView, 727, 734
TRACEDIR, 737
transpose, 84
    and all-to-all, 66–67
    data, 606
    recursive, 274
    through derived types, 230–231
tree
    traversal
        post-order, 564
tunnel
    ssh, 424
ucobound, 677
ulimit, 491
Unix
    process, 491
valgrind, 731–732
vector
    data type, 202
    instructions, 535
verbatim, 759
virtual shared memory, 311
VTune, 736
wall clock, 418
war, 336
waw, 336
weak symbol, see linker, weak symbol
while loop, 511
while loops, 469
window, 311–315
    consistency, 332
    displacement, 319
    displacement unit, 336
    info object, 408
memory, see also memory model
    model, 333
    separate, 333
    memory allocation, 313–315
    private, 334
    public, 334
work sharing, 441
work sharing construct, 447, 486
workshare
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>flush after</td>
<td>533</td>
</tr>
<tr>
<td>worksharing constructs</td>
<td></td>
</tr>
<tr>
<td>implied barriers at</td>
<td>500</td>
</tr>
<tr>
<td>world model</td>
<td>296, 297</td>
</tr>
<tr>
<td>wormhole routing</td>
<td>92</td>
</tr>
<tr>
<td>wraparound connections</td>
<td>367</td>
</tr>
<tr>
<td>X11, 601, 662</td>
<td></td>
</tr>
<tr>
<td>XSEDE portal</td>
<td>574</td>
</tr>
<tr>
<td>Zoltan, 422, 608</td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 64

**Lists of notes**

#### 64.1 MPI-4 notes

List of MPI-4 notes

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Non-blocking/persistent sendrecv</td>
</tr>
<tr>
<td>2</td>
<td>Persistent collectives</td>
</tr>
<tr>
<td>3</td>
<td>Neighborhood collectives, init</td>
</tr>
<tr>
<td>4</td>
<td>Partitioned communication</td>
</tr>
<tr>
<td>5</td>
<td>Count type</td>
</tr>
<tr>
<td>6</td>
<td>MPI Count type</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>8</td>
<td>Session model</td>
</tr>
<tr>
<td>9</td>
<td>Info key for alignment</td>
</tr>
<tr>
<td>10</td>
<td>Split by guided types</td>
</tr>
<tr>
<td>11</td>
<td>Window memory alignment</td>
</tr>
<tr>
<td>12</td>
<td>Info with null terminator</td>
</tr>
<tr>
<td>13</td>
<td>Memory alignment</td>
</tr>
<tr>
<td>14</td>
<td>Error code for aborted process</td>
</tr>
<tr>
<td>15</td>
<td>Error handler for session</td>
</tr>
<tr>
<td>16</td>
<td>Abort on communicator</td>
</tr>
</tbody>
</table>
64.2 Fortran notes

List of Fortran notes

MPI ................................................................. 16
1 Formatting of Fortran notes ................................ 20
2 New developments only in f08 module .................. 29
3 Communicator type ............................................ 33
4 MPI send/recv buffers ........................................ 48
5 Min/maxloc types ............................................. 77
6 Index of requests ............................................. 140
7 Status object in f08 ........................................... 150
8 Derived types for handles ................................... 194
9 Subarrays ....................................................... 211
10 Displacement unit ............................................ 320
11 Offset literals ................................................ 363
OpenMP .............................................................. 438
12 OpenMP version ............................................. 443
13 OMP do pragma ............................................... 458
14 Reductions on derived types .............................. 482
15 Private variables in parallel region ..................... 491
16 Private common blocks .................................... 495
17 Array sizes in map clause ................................. 541
PETSc ............................................................... 572
18 Petsc Initialization ......................................... 577
19 F90 array access through pointer ....................... 594
20 Error code handling ........................................ 657
21 Backtrace on error ........................................... 658
22 Print string construction .................................. 660
23 Printing and newlines ..................................... 660
Other ................................................................. 676
Tutorials ............................................................ 726
24 Cpp includes .................................................. 737
64.3 C++ notes

List of C++ notes

MPI .......................................................... 16
  1 Buffer treatment .................................................. 47
OpenMP .......................................................... 438
  2 Range syntax ...................................................... 463
  3 Reduction over iterators ........................................... 482
  4 Templated reductions ................................................ 482
  5 Reduction on class objects ......................................... 482
  6 Lock inside overloaded operator ................................... 504
  7 Uninitialized containers ............................................ 527
PETSc .......................................................... 572

64.4 The MPL C++ interface

List of Mpl notes

  1 Notes format ..................................................... 20
  2 Header file ....................................................... 29
  3 Init, finalize ...................................................... 30
  4 Processor name .................................................... 33
  5 World communicator ................................................ 33
  6 Communicator copying ................................................ 34
  7 Communicator passing ............................................... 34
  8 Rank and size ...................................................... 36
  9 Reduction operator ................................................ 46
 10 Scalar buffers ..................................................... 48
 11 Vector buffers ..................................................... 48
 12 Iterator buffers ................................................... 49
 13 Reduce in place .................................................... 53
 14 Reduce on non-root ................................................ 53
<table>
<thead>
<tr>
<th></th>
<th>LIST OF MPL NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Broadcast</td>
</tr>
<tr>
<td>16</td>
<td>Scan operations</td>
</tr>
<tr>
<td>17</td>
<td>Gather/scatter</td>
</tr>
<tr>
<td>18</td>
<td>Gather on nonroot</td>
</tr>
<tr>
<td>19</td>
<td>Operators</td>
</tr>
<tr>
<td>20</td>
<td>User-defined operators</td>
</tr>
<tr>
<td>21</td>
<td>Lambda operator</td>
</tr>
<tr>
<td>22</td>
<td>Nonblocking collectives</td>
</tr>
<tr>
<td>23</td>
<td>Blocking send and receive</td>
</tr>
<tr>
<td>24</td>
<td>Sending arrays</td>
</tr>
<tr>
<td>25</td>
<td>Iterator buffers</td>
</tr>
<tr>
<td>26</td>
<td>Iterator layout</td>
</tr>
<tr>
<td>27</td>
<td>Any source</td>
</tr>
<tr>
<td>28</td>
<td>Send-recv call</td>
</tr>
<tr>
<td>29</td>
<td>Requests from nonblocking calls</td>
</tr>
<tr>
<td>30</td>
<td>Request pools</td>
</tr>
<tr>
<td>31</td>
<td>Wait any</td>
</tr>
<tr>
<td>32</td>
<td>Status object</td>
</tr>
<tr>
<td>33</td>
<td>Status source querying</td>
</tr>
<tr>
<td>34</td>
<td>Message tag</td>
</tr>
<tr>
<td>35</td>
<td>Receive count</td>
</tr>
<tr>
<td>36</td>
<td>Persistent requests</td>
</tr>
<tr>
<td>37</td>
<td>Buffered send</td>
</tr>
<tr>
<td>38</td>
<td>Buffer attach and detach</td>
</tr>
<tr>
<td>39</td>
<td>Other types</td>
</tr>
<tr>
<td>40</td>
<td>Data types</td>
</tr>
<tr>
<td>41</td>
<td>Derived type handling</td>
</tr>
<tr>
<td>42</td>
<td>Contiguous type</td>
</tr>
<tr>
<td>43</td>
<td>Contiguous composing</td>
</tr>
<tr>
<td>44</td>
<td>Vector type</td>
</tr>
<tr>
<td>45</td>
<td>Subarray layout</td>
</tr>
<tr>
<td>46</td>
<td>Indexed type</td>
</tr>
<tr>
<td>47</td>
<td>Layouts for gatherv</td>
</tr>
<tr>
<td>48</td>
<td>Indexed block type</td>
</tr>
<tr>
<td>49</td>
<td>Struct type scalar</td>
</tr>
<tr>
<td>50</td>
<td>Struct type general</td>
</tr>
<tr>
<td>51</td>
<td>Extent resizing</td>
</tr>
<tr>
<td>52</td>
<td>Predefined communicators</td>
</tr>
<tr>
<td>53</td>
<td>Raw communicators</td>
</tr>
<tr>
<td>54</td>
<td>Communicator duplication</td>
</tr>
<tr>
<td>55</td>
<td>Communicator splitting</td>
</tr>
<tr>
<td>56</td>
<td>Distributed graph creation</td>
</tr>
<tr>
<td>57</td>
<td>Graph communicators</td>
</tr>
<tr>
<td>814</td>
<td></td>
</tr>
</tbody>
</table>
### 64.5 Python notes

**List of Python notes**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>16</td>
</tr>
<tr>
<td>1 Running mpi4py programs</td>
<td>20</td>
</tr>
<tr>
<td>2 Python notes</td>
<td>21</td>
</tr>
<tr>
<td>3 Import mpi module</td>
<td>29</td>
</tr>
<tr>
<td>4 Initialize/finalize</td>
<td>30</td>
</tr>
<tr>
<td>5 Communicator objects</td>
<td>33</td>
</tr>
<tr>
<td>6 Communicator rank and size</td>
<td>35</td>
</tr>
<tr>
<td>7 Buffers from numpy</td>
<td>48</td>
</tr>
<tr>
<td>8 Buffers from subarrays</td>
<td>48</td>
</tr>
<tr>
<td>9 In-place collectives</td>
<td>52</td>
</tr>
<tr>
<td>10 Sending objects</td>
<td>54</td>
</tr>
<tr>
<td>11 Define reduction operator</td>
<td>78</td>
</tr>
<tr>
<td>12 Reduction function</td>
<td>79</td>
</tr>
<tr>
<td>13 Handling a single request</td>
<td>139</td>
</tr>
<tr>
<td>14 Request arrays</td>
<td>140</td>
</tr>
<tr>
<td>15 Status object</td>
<td>150</td>
</tr>
<tr>
<td>16 Data types</td>
<td>195</td>
</tr>
<tr>
<td>17 Derived type handling</td>
<td>203</td>
</tr>
<tr>
<td>18 Vector type</td>
<td>208</td>
</tr>
<tr>
<td>19 Sending from the middle of a matrix</td>
<td>210</td>
</tr>
<tr>
<td>20 Big data</td>
<td>224</td>
</tr>
<tr>
<td>21 Communicator types</td>
<td>267</td>
</tr>
<tr>
<td>22 Communicator duplication</td>
<td>267</td>
</tr>
<tr>
<td>23 Comm split key is optional</td>
<td>274</td>
</tr>
<tr>
<td>24 Displacement byte computations</td>
<td>314</td>
</tr>
<tr>
<td>25 Window buffers</td>
<td>315</td>
</tr>
<tr>
<td>26 MPI one-sided transfer routines</td>
<td>321</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>27</td>
<td>File open is class method</td>
</tr>
<tr>
<td>28</td>
<td>Graph communicators</td>
</tr>
<tr>
<td>29</td>
<td>Thread level</td>
</tr>
<tr>
<td>30</td>
<td>Utility functions</td>
</tr>
<tr>
<td>31</td>
<td>Error policy</td>
</tr>
<tr>
<td></td>
<td>OpenMP</td>
</tr>
<tr>
<td></td>
<td>PETSc</td>
</tr>
<tr>
<td>32</td>
<td>Init, and with commandline options</td>
</tr>
<tr>
<td>33</td>
<td>Communicator object</td>
</tr>
<tr>
<td>34</td>
<td>petsc4py interface</td>
</tr>
<tr>
<td>35</td>
<td>Vector creation</td>
</tr>
<tr>
<td>36</td>
<td>Vector size</td>
</tr>
<tr>
<td>37</td>
<td>Vector operations</td>
</tr>
<tr>
<td>38</td>
<td>Setting vector values</td>
</tr>
<tr>
<td>39</td>
<td>Vector access</td>
</tr>
<tr>
<td>40</td>
<td>Petsc print and python print</td>
</tr>
<tr>
<td>41</td>
<td>HDF5 file generation</td>
</tr>
<tr>
<td>42</td>
<td>Petsc options</td>
</tr>
<tr>
<td>Other</td>
<td></td>
</tr>
<tr>
<td>Tutorials</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>Python MPI programs</td>
</tr>
</tbody>
</table>
Chapter 65

Index of MPI commands and keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0_MPI_OFFSET_KIND</td>
<td>363</td>
</tr>
<tr>
<td>access_style</td>
<td>408</td>
</tr>
<tr>
<td>accumulate_ops</td>
<td>336</td>
</tr>
<tr>
<td>accumulate_ordering</td>
<td>336</td>
</tr>
<tr>
<td>alloc_shared_noncontig</td>
<td>387</td>
</tr>
<tr>
<td>cb_block_size</td>
<td>409</td>
</tr>
<tr>
<td>cb_buffer_size</td>
<td>409</td>
</tr>
<tr>
<td>cb_nodes</td>
<td>409</td>
</tr>
<tr>
<td>chunked</td>
<td>409</td>
</tr>
<tr>
<td>chunked_item</td>
<td>409</td>
</tr>
<tr>
<td>chunked_size</td>
<td>409</td>
</tr>
<tr>
<td>collective_buffering</td>
<td>409</td>
</tr>
<tr>
<td>control_variable</td>
<td>399–401</td>
</tr>
<tr>
<td>cvvar, see control_variable</td>
<td></td>
</tr>
<tr>
<td>file_perm</td>
<td>409</td>
</tr>
<tr>
<td>io_node_list</td>
<td>409</td>
</tr>
<tr>
<td>irequest_pool</td>
<td>142</td>
</tr>
<tr>
<td>KSPSolve</td>
<td>638</td>
</tr>
<tr>
<td>mpi://SELF</td>
<td>298</td>
</tr>
<tr>
<td>mpi://WORLD</td>
<td>298</td>
</tr>
<tr>
<td>MPI_2DOUBLE_PRECISION</td>
<td>77</td>
</tr>
<tr>
<td>MPI_2INT</td>
<td>77</td>
</tr>
<tr>
<td>MPI_2INTEGER</td>
<td>77</td>
</tr>
<tr>
<td>MPI_2REAL</td>
<td>77</td>
</tr>
<tr>
<td>MPI_Abort</td>
<td>30, 80, 415, 425</td>
</tr>
<tr>
<td>MPI_Accumulate</td>
<td>316, 323, 329, 336</td>
</tr>
<tr>
<td>MPI_Add_error_class</td>
<td>415</td>
</tr>
<tr>
<td>MPI_Add_error_code</td>
<td>415</td>
</tr>
<tr>
<td>MPI_Add_error_string</td>
<td>415</td>
</tr>
<tr>
<td>MPI_Address (deprecated)</td>
<td>217</td>
</tr>
<tr>
<td>MPI_ADDRESS_KIND</td>
<td>196, 200, 320, 426</td>
</tr>
<tr>
<td>MPI_AINT</td>
<td>199</td>
</tr>
<tr>
<td>MPI_Aint</td>
<td>195, 196, 199, 199, 222, 315, 319</td>
</tr>
<tr>
<td>in Fortran</td>
<td>200</td>
</tr>
<tr>
<td>MPI_Aint_add</td>
<td>199, 315</td>
</tr>
<tr>
<td>MPI_Aint_diff</td>
<td>199, 315</td>
</tr>
<tr>
<td>MPI_Allgather</td>
<td>64, 94</td>
</tr>
<tr>
<td>MPI_Allgather_init</td>
<td>185</td>
</tr>
<tr>
<td>MPI_Allgatherv</td>
<td>72</td>
</tr>
<tr>
<td>MPI_Allgatherv_init</td>
<td>185</td>
</tr>
<tr>
<td>MPI_Alloc_mem</td>
<td>313, 315, 330, 413</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>44, 45, 51, 67, 94</td>
</tr>
<tr>
<td>MPI_Allreduce_init</td>
<td>184, 185</td>
</tr>
<tr>
<td>MPI_Alltoall</td>
<td>65, 67, 607</td>
</tr>
<tr>
<td>MPI_Alltoall_init</td>
<td>185</td>
</tr>
<tr>
<td>MPI_Alltoallv</td>
<td>67, 67, 72, 76, 216</td>
</tr>
<tr>
<td>MPI_Alltoallv_init</td>
<td>185</td>
</tr>
<tr>
<td>MPI_AppNum</td>
<td>292, 410, 425</td>
</tr>
<tr>
<td>MPI_ANY_SOURCE</td>
<td>71, 90, 122, 122, 146, 149, 150, 154, 311, 413, 423, 427</td>
</tr>
<tr>
<td>MPI_ANY_TAG</td>
<td>123, 131, 149, 152, 427</td>
</tr>
<tr>
<td>MPI_APPNUM</td>
<td></td>
</tr>
</tbody>
</table>

817
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Argv_null</td>
<td>427</td>
</tr>
<tr>
<td>MPI_Argvs_null</td>
<td>427</td>
</tr>
<tr>
<td>MPI_async_protects_nonblocking</td>
<td>417</td>
</tr>
<tr>
<td>MPI_attr_get</td>
<td>409</td>
</tr>
<tr>
<td>MPI_band</td>
<td>77</td>
</tr>
<tr>
<td>MPI_barrier</td>
<td>71, 364, 420</td>
</tr>
<tr>
<td>MPI_barrier_init</td>
<td>185</td>
</tr>
<tr>
<td>MPI_bcast</td>
<td>53, 94</td>
</tr>
<tr>
<td>MPI_bcast_init</td>
<td>185</td>
</tr>
<tr>
<td>MPI_bottom</td>
<td>199, 336, 426, 427</td>
</tr>
<tr>
<td>MPI_bsend</td>
<td>188, 189, 190</td>
</tr>
<tr>
<td>MPI_bsend_init</td>
<td>183, 188, 191, 191</td>
</tr>
<tr>
<td>MPI_bsend_overhead</td>
<td>190, 191, 427</td>
</tr>
<tr>
<td>MPI_buffer_attach</td>
<td>190, 190</td>
</tr>
<tr>
<td>MPI_buffer_detach</td>
<td>190, 190</td>
</tr>
<tr>
<td>MPI_bxor</td>
<td>77</td>
</tr>
<tr>
<td>MPI_byte</td>
<td>196, 197, 202, 224</td>
</tr>
<tr>
<td>MPI_c_bool</td>
<td>196</td>
</tr>
<tr>
<td>MPI_c_complex</td>
<td>196</td>
</tr>
<tr>
<td>MPI_c_double_complex</td>
<td>196</td>
</tr>
<tr>
<td>MPI_c_float_complex</td>
<td>196</td>
</tr>
<tr>
<td>MPI_c_long_double_complex</td>
<td>196</td>
</tr>
<tr>
<td>MPI_c_long_double_complex</td>
<td>196</td>
</tr>
<tr>
<td>MPI_cancel</td>
<td>427, 428</td>
</tr>
<tr>
<td>MPI.cart</td>
<td>367</td>
</tr>
<tr>
<td>MPI_cart_coords</td>
<td>369</td>
</tr>
<tr>
<td>MPI_cart_create</td>
<td>368, 370</td>
</tr>
<tr>
<td>MPI_cart_get</td>
<td>369</td>
</tr>
<tr>
<td>MPI_cart_map</td>
<td>372</td>
</tr>
<tr>
<td>MPI_cart_rank</td>
<td>369</td>
</tr>
<tr>
<td>MPI_cart_sub</td>
<td>371</td>
</tr>
<tr>
<td>MPI_cartdim_get</td>
<td>369</td>
</tr>
<tr>
<td>MPI_char</td>
<td>196</td>
</tr>
<tr>
<td>MPI_character</td>
<td>197</td>
</tr>
<tr>
<td>MPI_close_port</td>
<td>294</td>
</tr>
<tr>
<td>MPI_combiner_vector</td>
<td>231</td>
</tr>
<tr>
<td>MPI_comm</td>
<td>21–23, 33, 266, 267, 426</td>
</tr>
<tr>
<td>MPI_comm_accept</td>
<td>293, 293</td>
</tr>
<tr>
<td>MPI_comm_attr_function</td>
<td>410</td>
</tr>
<tr>
<td>MPI_comm_compare</td>
<td>268, 280, 298</td>
</tr>
<tr>
<td>MPI_comm_connect</td>
<td>293, 294, 294, 413</td>
</tr>
<tr>
<td>MPI_comm_create</td>
<td>271, 275</td>
</tr>
<tr>
<td>MPI_comm_create_errhandler</td>
<td>415, 415</td>
</tr>
<tr>
<td>MPI_Comm_create_group</td>
<td>276</td>
</tr>
<tr>
<td>MPI_Comm_create_keyval</td>
<td>410</td>
</tr>
<tr>
<td>MPI_Comm_delete_attr</td>
<td>410</td>
</tr>
<tr>
<td>MPI_Comm_disconnect</td>
<td>272, 294</td>
</tr>
<tr>
<td>MPI_Comm_dup</td>
<td>34, 267, 268, 270, 271, 408, 577</td>
</tr>
<tr>
<td>MPI_Comm_dup_with_info</td>
<td>267, 408</td>
</tr>
<tr>
<td>MPI_Comm_free</td>
<td>270, 272</td>
</tr>
<tr>
<td>MPI_Comm_free_keyval</td>
<td>410</td>
</tr>
<tr>
<td>MPI_Comm_get_attr</td>
<td>289, 409</td>
</tr>
<tr>
<td>MPI_Comm_get_errhandler</td>
<td>413, 415</td>
</tr>
<tr>
<td>MPI_Comm_get_info</td>
<td>408</td>
</tr>
<tr>
<td>MPI_Comm_get_parent</td>
<td>280, 291, 292, 297</td>
</tr>
<tr>
<td>MPI_Comm_group</td>
<td>275, 279</td>
</tr>
<tr>
<td>MPI_Comm_idup</td>
<td>267</td>
</tr>
<tr>
<td>MPI_Comm_idup_with_info</td>
<td>267</td>
</tr>
<tr>
<td>MPI_Comm_join</td>
<td>296, 297</td>
</tr>
<tr>
<td>MPI_Comm_null</td>
<td>266, 267, 269, 272, 280, 291, 369, 384</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>34, 35, 273, 279, 375, 755</td>
</tr>
<tr>
<td>MPI_Comm_remote_group</td>
<td>280</td>
</tr>
<tr>
<td>MPI_Comm_remote_size</td>
<td>280, 292</td>
</tr>
<tr>
<td>MPI_COMM_SELF</td>
<td>266, 267, 584</td>
</tr>
<tr>
<td>MPI_Comm_set_attr</td>
<td>409, 410</td>
</tr>
<tr>
<td>MPI_Comm_set_errhandler</td>
<td>413, 413, 415</td>
</tr>
<tr>
<td>MPI_Comm_set_info</td>
<td>408</td>
</tr>
<tr>
<td>MPI_Comm_set_name</td>
<td>267</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>34, 35, 121, 279, 375</td>
</tr>
<tr>
<td>MPI_Comm_spawn</td>
<td>271, 289, 418</td>
</tr>
<tr>
<td>MPI_Comm_spawn_multiple</td>
<td>292, 410</td>
</tr>
<tr>
<td>MPI_Comm_split</td>
<td>71, 271, 272, 281, 371</td>
</tr>
<tr>
<td>MPI_Comm_split_type</td>
<td>274, 384, 385, 388</td>
</tr>
<tr>
<td>MPI_Comm_test_inter</td>
<td>280</td>
</tr>
<tr>
<td>MPI_COMM_TYPE_HW_GUIDED</td>
<td>384</td>
</tr>
<tr>
<td>MPI_COMM_TYPE_HW_UNGUIDED</td>
<td>384</td>
</tr>
<tr>
<td>MPI_COMM_TYPE_SHARED</td>
<td>384</td>
</tr>
<tr>
<td>MPI_COMM_WORLD</td>
<td>33, 266, 267, 271, 272, 277, 278, 281, 289, 292, 298, 389, 410, 415, 418, 426, 573, 577, 585</td>
</tr>
<tr>
<td>MPI.Compare_and_swap</td>
<td>327</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>197</td>
</tr>
<tr>
<td>MPI_CONGRUENT</td>
<td>269</td>
</tr>
<tr>
<td>MPI_Count</td>
<td>45, 196, 221, 222, 222, 223, 226</td>
</tr>
<tr>
<td>MPI_COUNT_KIND</td>
<td>196, 426</td>
</tr>
</tbody>
</table>
MPI_Datatype, 62, 194, 203
MPI_DATATYPE_NULL, 195, 204
MPI_Dims_create, 367
MPI_DISPLACEMENT_CURRENT, 361
MPI_DIST_GRAPH, 367, 367
MPI_Dist_graph_create, 373, 373, 374, 377, 378
MPI_Dist_graph_create_adjacent, 373, 376
MPI_Dist_graph_neighbors, 375–377, 378
MPI_Dist_graph_neighbors_count, 375, 376, 378
MPI_DOUBLE, 195, 196
MPI_DOUBLE_COMPLEX, 197
MPI_DOUBLE_INT, 77, 77
MPI_DOUBLE_PRECISION, 195, 197
MPI_ERR_ARG, 412
MPI_ERR_BUFFER, 191, 412
MPI_ERR_COMM, 155, 412, 418
MPI_ERR_COUNT, 155, 412
MPI_ERR_IN_PLACE, 143, 152, 413
MPI_ERR_INFO, 413
MPI_ERR_UNHANDLED, 191, 412
MPI_ERR_LASTCODE, 412, 415
MPI_ERR_NODEFAULT, 315, 413
MPI_ERR_OTHER, 413
MPI_ERR_PORT, 294, 413
MPI_ERR_PROC_ABORTED, 413
MPI_ERR_RANK, 155, 413
MPI_ERR_SERVICE, 296, 413
MPI_ERR_TAG, 155
MPI_ERR_TYPE, 155
MPI_ERRCODES_IGNORE, 289, 427
MPI_Errhandler, 413, 414
MPI_Errhandler_c2f, 300
MPI_Errhandler_create, 413
MPI_Errhandler_f2c, 300
MPI_Errhandler_free, 300, 413
MPI_ERROR, 150, 152, 413, 414
MPI_Error_class, 300
MPI_Error_string, 300, 413, 414
MPI_ERRORS_ABORT, 413
MPI_ERRORS_ARE_FATAL, 364, 413, 414
MPI_ERRORS_RETURN, 364, 413, 414, 414, 415
MPI_Exscan, 57, 58, 60
MPI_Exscan_init, 185
MPI_F08_status, 150
MPI_F_sync_reg, 417
MPI_Fetch_and_op, 325, 327, 329, 333
MPI_File, 276, 357
MPI_File_call_errhandler, 413
MPI_File_close, 357
MPI_File_delete, 358
MPI_File_get_errhandler, 364
MPI_File_get_group, 276
MPI_File_get_info, 408
MPI_File_get_size, 363
MPI_File_get_view, 363
MPI_File_iwrite, 360
MPI_File_iwrite_all, 360
MPI_File_iwrite_at, 360
MPI_File_iwrite_at_all, 360
MPI_File_iwrite_shared, 360, 363
MPI_File_iwrite_all, 360
MPI_File_iwrite_at, 360
MPI_File_iwrite_at_all, 360
MPI_File_iwrite_shared, 360, 363
MPI_FILE_NULL, 364
MPI_File_open, 43, 357, 408
MPI_File_preallocate, 363
MPI/File_read, 359, 365
MPI/File_read_all, 359
MPI/File_read_all_begin, 360
MPI/File_read_all_end, 360
MPI/File_read_at, 359
MPI/File_read_at_all, 359
MPI/File_read_shared, 360, 363
MPI/File NULL, 364
MPI/File_set_atomicity, 364
MPI/File_set_errhandler, 364, 413
MPI/File_set_info, 408
MPI/File_set_size, 363
MPI/File_set_view, 361, 363, 365, 408
MPI/File_sync, 358
MPI/File_write, 358, 360, 363
MPI_File_write_all, 358
MPI_File_write_all_begin, 360
MPI_File_write_all_end, 360
MPI_File_write_at, 359, 360, 360
MPI_File_write_at_all, 359
MPI_File_write_ordered, 363
MPI_File_write_shared, 363
MPI_Finalize, 29, 30, 31, 296, 297, 577
MPI_Finalized, 30, 31, 299
MPI_Fint, 150
MPI_FLOAT, 195, 196
MPI_FLOAT_INT, 77
MPI_Gather, 62, 72, 94, 211, 356
MPI_Gather_init, 185
MPI_Gatherv, 64, 72, 72, 216
MPI_Gatherv_init, 185
MPI_Get, 316, 321, 328, 331, 333, 335
MPI_Get_accumulate, 324, 325, 325, 336
MPI_Get_address, 199, 217, 219, 335
MPI_Get_count, 147, 150, 152, 222
MPI_Get_elements, 153, 222
MPI_Get_elements_x, 153, 223, 224
MPI_Get_hw_resource_types, 384
MPI_Get_library_version, 299
MPI_Get_processor_name, 31, 32, 33, 411
MPI_Get_version, 299, 411
MPI_GRAPH, 367
MPI_Graph_create, 379
MPI_Graph_get, 379
MPI_Graph_map, 379
MPI_Graph_neighbors, 379
MPI_Graph_neighbors_count, 379
MPI_Graphdims_get, 379
MPI_Group, 275, 275
MPI_Group_difference, 276
MPI_Group_excl, 276
MPI_Group_incl, 276
MPI HOST (deprecated), 410
MPI_Iallgather, 82
MPI_Iallgatherv, 82
MPI_Iallreduce, 82
MPI_Ialltoall, 82
MPI_Ialltoallw, 82
MPI_Ialltoallw, 82
MPI_Ibarrier, 81, 82, 85
MPI_Ibsend, 188, 189, 191
MPI_IDENT, 268
MPI_Iexscan, 82
MPI_Igather, 82, 84
MPI_Igatherv, 82
MPI/info, 184, 185, 297, 315, 336, 362, 387, 406, 408
MPI_I_info_c2f, 299
MPI_I_info_create, 299, 406
MPI_I_info_create_env, 299
MPI_I_info_delete, 299, 406
MPI_I_info_dup, 299, 406
MPI_INFO_ENV, 31, 32, 406
MPI_I_info_f2c, 299
MPI_I_info_free, 299, 406
MPI_I_info_get (deprecated), 406
MPI_I_info_get, 299, 406
MPI_I_info_get_nkeys, 299, 406
MPI_I_info_get_nthkey, 299, 406
MPI_I_info_get_string, 406
MPI_I_info_get_valueLen (deprecated), 406
MPI_I_info_get_valueLen, 299
MPI_INFO_NULL, 362
MPI_I_info_set, 299, 406
MPI_Init, 29, 30–32, 296, 297, 395, 399, 426, 576, 672
in Fortran, 416
MPI_Init_thread, 30, 296, 297, 395, 395, 396, 399
MPI_Initialized, 31, 299
MPI_INT, 194, 196
MPI_INT16_T, 197
MPI_INT32_T, 197
MPI_INT64_T, 197
MPI_INT8_T, 197
MPI_INTEGER, 195, 197
MPI_INTEGER1, 197
MPI_INTEGER16, 195, 197
MPI_INTEGER2, 197
MPI_INTEGER4, 197
MPI_INTEGER8, 197
MPI_INTEGER_KIND, 426
MPI_Intercomm_create, 271, 277
MPI_Intercomm_merge, 280
MPI_IO, 410
MPI_Iprobe, 86, 147, 417
MPI_Irecv, 82, 92, 135, 136, 143, 145, 146, 150, 159, 182
MPI_Ireduce, 82
MPI_Ireduce_scatter, 82
MPI_Ireduce_scatter_block, 82
MPI_Irsend, 188
MPI_Is_thread_main, 395
MPI_Isend, 92, 135, 136, 159, 182, 190, 324
MPI_Isendrecv, 133
MPI_Isendrecv_replace, 133
MPI_Issend, 187, 188
MPI_KEYVAL_INVALID, 427
MPI_LAND, 77
MPI_LASTUSEDCODE, 415
MPI_LOCK_EXCLUSIVE, 331, 427
MPI_LOCK_SHARED, 331, 427
MPI_LOGICAL, 197
MPI_LONG, 196
MPI_LONG_DOUBLE, 196
MPI_LONG_DOUBLE_INT, 77
MPI_LONG_INT, 77, 195
MPI_LONG_LONG_INT, 196
MPI_LOR, 77
MPI_LXOR, 77
MPI_MAX, 49, 77
MPI_MAX_DATAREP_STRING, 426
MPI_MAX_ERROR_STRING, 414, 426
MPI_MAX_INFO_KEY, 406, 426
MPI_MAX_INFO_VAL, 426
MPI_MAX_LIBRARY_VERSION_STRING, 426
MPI_MAX_OBJECT_NAME, 426
MPI_MAX_PORT_NAME, 293, 411, 426
MPI_MAXLOC, 77, 77
MPI_MESSAGE, 148
MPI_MIN, 77
mpi_minimum_memory_alignment, 315, 387, 409
MPI_MINLOC, 77, 77
MPI_MODE_APPEND, 357
MPI_MODE_CREATE, 357
MPI_MODE_DELETE_ON_CLOSE, 357
MPI_MODE_Excl, 357
MPI_MODE_NOPRECEDE, 317, 337
MPI_MODE_NOPUT, 317, 337
MPI_MODE_NOSTORE, 317, 337
MPI_MODE_NOSUCCEED, 317, 337
MPI_MODE_RDONLY, 357
MPI_MODE_RDWR, 357
MPI_MODE_SEQUENTIAL, 357, 358, 359
MPI_MODE_UNIQUE_OPEN, 357
MPI_MODE_WRONLY, 357
MPI_Mprobe, 148
MPI_Mrecv, 148
MPI_Neighbor_allgather, 376, 377
MPI_Neighbor_allgather_init, 185
MPI_Neighbor_allgatherv, 377
MPI_Neighbor_allgatherv_init, 185
MPI_Neighbor_allreduce, 377
MPI_Neighbor_alltoall, 377
MPI_Neighbor_alltoall_init, 185
MPI_Neighbor_alltoallv, 377
MPI_Neighbor_alltoallw, 377
MPI_Neighbor_alltoallw_init, 185
MPI_NO_OP, 77, 324, 327, 336
MPI_Offset, 419, 222, 360
MPI_OFFSET_KIND, 197, 363, 426
MPI_Op, 47, 58, 76, 78, 80, 319, 327, 415
MPI_Op_commutative, 80
MPI_Op_create, 60, 78, 80

Victor Eijkhout

821
<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Op_free</td>
<td>80</td>
</tr>
<tr>
<td>MPI_OP_NULL</td>
<td>80</td>
</tr>
<tr>
<td>MPI_Open_port</td>
<td>293, 293</td>
</tr>
<tr>
<td>MPI_ORDER_C</td>
<td>213</td>
</tr>
<tr>
<td>MPI_ORDER_Fortran</td>
<td>213</td>
</tr>
<tr>
<td>MPI_Pack</td>
<td>232</td>
</tr>
<tr>
<td>MPI_Pack_size</td>
<td>190, 233</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>196, 197, 224, 232</td>
</tr>
<tr>
<td>MPI_Parrived</td>
<td>187</td>
</tr>
<tr>
<td>MPI_Pready</td>
<td>185, 185</td>
</tr>
<tr>
<td>MPI_Pready_list</td>
<td>186</td>
</tr>
<tr>
<td>MPI_Pready_range</td>
<td>185</td>
</tr>
<tr>
<td>MPI_Precv_init</td>
<td>187</td>
</tr>
<tr>
<td>MPI_Probe</td>
<td>122, 147, 148, 417</td>
</tr>
<tr>
<td>MPI_PROC_NULL</td>
<td>122, 129, 131, 132, 143, 279, 319, 371, 410, 412, 427, 763</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>49, 58, 77</td>
</tr>
<tr>
<td>MPI_Psend_init</td>
<td>185, 186</td>
</tr>
<tr>
<td>MPI_Publish_name</td>
<td>295</td>
</tr>
<tr>
<td>MPI_Put</td>
<td>316, 319, 321, 322, 328, 329, 331, 333</td>
</tr>
<tr>
<td>MPI_Query_thread</td>
<td>395</td>
</tr>
<tr>
<td>MPI_Raccumulate</td>
<td>324</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>195, 197</td>
</tr>
<tr>
<td>MPI_REAL2</td>
<td>197</td>
</tr>
<tr>
<td>MPI_REAL4</td>
<td>197</td>
</tr>
<tr>
<td>MPI_REAL8</td>
<td>197</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>122, 122–124, 128, 129, 134, 145, 149, 150, 153, 417, 427</td>
</tr>
<tr>
<td>MPI_Recv_init</td>
<td>182</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>49, 51, 72, 94, 323</td>
</tr>
<tr>
<td>MPI_Reduce_init</td>
<td>185</td>
</tr>
<tr>
<td>MPI_Reduce_local</td>
<td>80</td>
</tr>
<tr>
<td>MPI_Reduce_scatter</td>
<td>67, 70, 71, 94, 338</td>
</tr>
<tr>
<td>MPI_Reduce_scatter_block</td>
<td>67, 69</td>
</tr>
<tr>
<td>MPI_Reduce_scatter_block_init</td>
<td>185</td>
</tr>
<tr>
<td>MPI_Reduce_scatter_init</td>
<td>185</td>
</tr>
<tr>
<td>MPI_REPLACE</td>
<td>77, 323, 324, 327</td>
</tr>
<tr>
<td>MPI_Request</td>
<td>23, 81, 136, 146, 181, 184, 186, 187, 360</td>
</tr>
<tr>
<td>MPI_Request_free</td>
<td>147, 147, 181, 184, 428</td>
</tr>
<tr>
<td>MPI_Request_get_status</td>
<td>147, 418</td>
</tr>
<tr>
<td>MPI_REQUEST_NULL</td>
<td>138, 146, 147</td>
</tr>
<tr>
<td>MPI_Rget</td>
<td>324</td>
</tr>
<tr>
<td>MPI_Rget_accumulate</td>
<td>324</td>
</tr>
<tr>
<td>MPI_ROOT</td>
<td>279, 427</td>
</tr>
<tr>
<td>MPI_Rput</td>
<td>324</td>
</tr>
<tr>
<td>MPI_Rsend</td>
<td>188, 423</td>
</tr>
<tr>
<td>MPI_Rsend_init</td>
<td>183</td>
</tr>
<tr>
<td>MPI_Rsend_init</td>
<td>183</td>
</tr>
<tr>
<td>MPI.Scan</td>
<td>57, 57, 58, 60</td>
</tr>
<tr>
<td>MPI_Scan_init</td>
<td>185</td>
</tr>
<tr>
<td>MPI_Scatter</td>
<td>61, 62, 94</td>
</tr>
<tr>
<td>MPI_Scatter_init</td>
<td>185</td>
</tr>
<tr>
<td>MPI_Scatterv</td>
<td>72</td>
</tr>
<tr>
<td>MPI_Scatterv_init</td>
<td>185</td>
</tr>
<tr>
<td>MPI_SEEK_CUR</td>
<td>358, 360, 361</td>
</tr>
<tr>
<td>MPI_SEEK_END</td>
<td>358</td>
</tr>
<tr>
<td>MPI_SEEK_SET</td>
<td>358, 364</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>82, 91, 120, 122, 124–126, 128, 129, 134, 145, 149, 221, 417, 423</td>
</tr>
<tr>
<td>MPI_Send_c</td>
<td>221</td>
</tr>
<tr>
<td>MPI_Send_init</td>
<td>182, 185</td>
</tr>
<tr>
<td>MPI_Sendrecv</td>
<td>129, 131, 132, 134, 145, 370, 763</td>
</tr>
<tr>
<td>MPI_Sendrecv_init</td>
<td>133</td>
</tr>
<tr>
<td>MPI_Sendrecv_replace</td>
<td>132</td>
</tr>
<tr>
<td>MPI_Sendrecv_replace_init</td>
<td>133</td>
</tr>
<tr>
<td>MPI_Session_call_errhandler</td>
<td>300, 413</td>
</tr>
<tr>
<td>MPI_Session_create_errhandler</td>
<td>298, 300</td>
</tr>
<tr>
<td>MPI_Session_finalize</td>
<td>297</td>
</tr>
<tr>
<td>MPI_Session_get_info</td>
<td>297</td>
</tr>
<tr>
<td>MPI_Session_get_nth_pset</td>
<td>298</td>
</tr>
<tr>
<td>MPI_Session_get_num_psets</td>
<td>298</td>
</tr>
<tr>
<td>MPI_Session_get_pset_info</td>
<td>298</td>
</tr>
<tr>
<td>MPI_Session_init</td>
<td>187</td>
</tr>
<tr>
<td>MPI_Session_set_errhandler</td>
<td>413</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>196</td>
</tr>
<tr>
<td>MPI_SHORT_INT</td>
<td>77</td>
</tr>
<tr>
<td>MPI_SIGNED_CHAR</td>
<td>196</td>
</tr>
<tr>
<td>MPI_SIMILAR</td>
<td>269</td>
</tr>
<tr>
<td>mpi_size</td>
<td>298</td>
</tr>
<tr>
<td>MPI_Sizeof</td>
<td>199, 200, 202, 416</td>
</tr>
<tr>
<td>MPI_SOURCE</td>
<td>143, 146, 150, 150, 154, 155</td>
</tr>
<tr>
<td>MPI_Ssend</td>
<td>126, 187, 188, 423</td>
</tr>
<tr>
<td>MPI_Ssend_init</td>
<td>183, 188</td>
</tr>
<tr>
<td>MPI_Start</td>
<td>181, 182, 184, 185</td>
</tr>
<tr>
<td>MPI_Startall</td>
<td>181, 182, 184</td>
</tr>
</tbody>
</table>
| MPI_Status                                  | 123, 131, 136, 139, 143, 146, 149, 149,
MPI_Status_f082f, 150
MPI_Status_f2f08, 150
MPI_STATUS_IGNORE, 123, 123, 136, 143, 149, 426, 427
MPI_STATUS_SIZE, 426
MPI_STATUSES_IGNORE, 139, 143, 427
MPI_SUBARRAYS_SUPPORTED, 212, 426
MPI_SUBVERSION, 411, 426
MPI_SUCCESS, 22, 155, 191, 412, 413, 414
MPI_SUM, 49, 58, 72, 77
MPI_T_BIND_NO_OBJECT, 399
MPI_T_category_changed, 403
MPI_T_category_get_categories, 403
MPI_T_category_get_cvars, 403
MPI_T_category_get_index, 403
MPI_T_category_get_info, 403
MPI_T_category_get_num, 403
MPI_T_category_get_pvars, 403
MPI_T_cvar_get_index, 400
MPI_T_cvar_get_info, 399, 403
MPI_T_cvar_handle_alloc, 402
MPI_T_cvar_handle_free, 402
MPI_T_ERR_INVALID_HANDLE, 402
MPI_T_ERR_INVALID_INDEX, 399
MPI_T_ERR_INVALID_NAME, 400
MPI_T_ERR_PVAR_NO_STARTSTOP, 402
MPI_T_ERR_PVAR_NO_WRITE, 403
MPI_T_finalize, 402
MPI_T_init_thread, 399
MPI_T_PVAR_ALL_HANDLES, 402, 403
MPI_T_PVAR_CLASS_AGGREGATE, 401
MPI_T_PVAR_CLASS_COUNTER, 401
MPI_T_PVAR_CLASS GENERIC, 401
MPI_T_PVAR_CLASS_HIGHWATERMARK, 401
MPI_T_PVAR_CLASS_LEVEL, 401
MPI_T_PVAR_CLASS_LOWWATERMARK, 401
MPI_T_PVAR_CLASS_PERCENTAGE, 401
MPI_T_PVAR_CLASS_SIZE, 401
MPI_T_PVAR_CLASS_STATE, 401
MPI_T_PVAR_CLASS_TIMER, 401
MPI_T_pvar_get_index, 402
MPI_T_pvar_get_info, 401, 402, 403
MPI_T_pvar_get_num, 401
MPI_T_pvar_handle_alloc, 402
MPI_T_pvar_handle_free, 402
MPI_T_PVAR_HANDLE_NULL, 402
MPI_T_pvar_read, 402
MPI_T_pvar_readreset, 403
MPI_T_pvar_session_create, 402
MPI_T_pvar_session_free, 402
MPI_T_PVAR_SESSION_NULL, 402
MPI_T_pvar_start, 402, 402
MPI_T_pvar_stop, 402, 402
MPI_T_pvar_write, 402
MPI_Txxx, 300
MPI_Type_commit, 204
MPI_Type_contiguous, 203, 204
MPI_Type_create_f90_complex, 197
MPI_Type_create_f90_integer, 197
MPI_Type_create_f90_real, 197
MPI_Type_create_hindexed, 199, 216
MPI_Type_create_hindexed_block, 216
MPI_Type_create_keyval, 410
MPI_Type_create_resized, 227, 229
MPI_Type_create_subarray, 204, 217
MPI_Type_delete_attr, 410
MPI_Type_extent (deprecated), 225
MPI_Type_free, 204
MPI_Type_get_attr, 409
MPI_Type_get_contents, 231

Victor Eijkhout 823
MPI_Type_get_envelope, 231
MPI_Type_get_extent, 224, 225
MPI_Type_get_extent_c, 226
MPI_Type_get_extent_x, 223, 226
MPI_Type_get_true_extent, 225
MPI_Type_get_true_extent_c, 226
MPI_Type_get_true_extent_x, 223, 226
MPI_Type_hindexed, 204
MPI_Type_indexed, 204, 213, 216
MPI_Type_lb (deprecated), 225
MPI_Type_match_size, 201
MPI_Type_size, 201
MPI_Type_size_x, 196
MPI_Type_struct, 217
MPI_Type_ub (deprecated), 225
MPI_Type_vector, 204, 206
MPI_TYPECLASS_COMPLEX, 201
MPI_TYPECLASS_INTEGER, 201
MPI_TYPECLASS_REAL, 201
MPI_UB, 220, 230
MPI_UINT16_T, 197
MPI_UINT32_T, 197
MPI_UINT64_T, 197
MPI_UINT8_T, 197
MPI_UNDEFINED, 367, 427
MPI_UNEQUAL, 269
MPI_UNIVERSE_SIZE, 289, 410
MPI_Unpack, 322
MPI_Unpublish_name, 296, 413
MPI UNSIGNED, 196
MPI_UNSIGNED_CHAR, 196
MPI_UNSIGNED_LONG, 196
MPI_UNSIGNED_SHORT, 196
MPI_UNWEIGHTED, 375, 427
MPI VERSION, 411, 426
MPI_Wait, 81, 136, 138, 143, 146, 185, 187, 360, 428
in Python, 139
MPI_Wait..., 149, 150
MPI_Waitall, 139, 143, 146, 152, 182, 312
in Python, 140
MPI_Waitany, 140, 142, 143, 146
MPI_Waitsome, 143, 146
MPI_WEIGHTS_EMPTY, 375, 427
MPI_Win, 199, 276, 312, 384
MPI_Win_allocate, 314, 315, 333, 336
MPI_Win_allocate_shared, 314, 334, 336, 386, 386
MPI_Win_attach, 334
MPI_WIN_BASE, 336
MPI_Win_call_errhandler, 413
MPI_Win_complete, 318, 318
MPI_Win_create, 200, 313, 314, 315, 330, 333, 334, 336
MPI_Win_create_dynamic, 314, 334, 336
MPI_WIN_CREATE_FLAVOR, 336
MPI_Win_delete_attr, 410
MPI_Win_detach, 335
MPI_WIN_DISP_UNIT, 336
MPI_Win_fence, 316, 316, 331, 331, 332, 337, 764
MPI_WIN_FLAVOR_ALLOCATE, 336
MPI_WIN_FLAVOR_CREATE, 336
MPI_WIN_FLAVOR_DYNAMIC, 336
MPI_WIN_FLAVOR_SHARED, 336
MPI_Win_flash, 333
MPI_Win_flush..., 325
MPI_Win_flush_all, 333
MPI_Win_flush_local, 333, 333
MPI_Win_flush_local_all, 333
MPI_Win_free, 312
MPI_Win_get_attr, 334, 409
MPI_Win_get_group, 276
MPI_Win_get_info, 408
MPI_Win_lock, 327, 330, 331, 337
MPI_Win_lock_all, 331, 332, 333, 337
MPI_Win_lockall, 337
MPI_WIN_MODEL, 334, 336
MPI_Win_post, 318, 337
MPI_WIN_SEPARATE, 334
MPI_Win_set_attr, 410
MPI_Win_set_errhandler, 413
MPI_Win_set_info, 408
MPI_Win_set_cursor, 386, 388
MPI_Win_shared_query, 386, 388
MPI_WIN_SIZE, 336
MPI_Win_start, 318, 337
MPI_Win_sync, 333
MPI_Win_test, 318, 418
MPI_WIN_UNIFIED, 334
MPI_Win_unlock, 331, 332
MPI_Win_unlock_all, 331, 332
MPI_Win_wait, 318, 318
MPI_Wtick, 419, 420
MPI_Wtime, 123, 418, 665
MPI_WTIME_IS_GLOBAL, 410, 419

nb_proc, 409
no_locks, 336
num_io_nodes, 409

OMPI_COMM_TYPE_SOCKET, 385

performance variable, 401–403
PMPI_..., 420
pvar, see performance variable

same_op, 336
same_op_no_op, 336
striping_factor, 409
striping_unit, 409

testany, 142
thread_support, 297

vector_layout, 206

wtime, 419
65.1 From the standard document

This is an automatically generated list of every function, type, and constant in the MPI standard document. Where these appear in this book, a page reference is given.

65.1.1 List of all functions

- MPI_Abort 30
- MPI_Accumulate 323
- MPI_Address ??
- MPI_Add_error_class 415
- MPI_Add_error_string 415
- MPI_Aint_add 199
- MPI_Aint_diff 199
- MPI_Allgather 64
- MPI_Allgatherv 72
- MPI_Allgatherv_init 185
- MPI_Allgather_init 185
- MPI_Algmem 315
- MPI_Algmem_cptr ??
- MPI_Allreduce 44
- MPI_Allreduce_init 184
- MPI_Alltoall 65
- MPI_Alltoallv 67
- MPI_Alltoallv_init 185
- MPI_Alltoallv ??
- MPI_Alltoallw_init 185
- MPI_Alltoallw_init 185
- MPI_Attr_delete ??
- MPI_Attr_get 409
- MPI_Attr_put ??
- MPI_Accumulate 323
- MPI_Barrier 71
- MPI_Barrier_init 185
- MPI_Bcast 53
- MPI_Bcast_init 185
- MPI_Bsend 189
- MPI_Bsend_init 191
- MPI_Buffer_attach 190
- MPI_Buffer_detach 190
- MPI_Cancel 427
- MPI_Cartdim_get 369
- MPI_Cart_coords 369
- MPI_Cart_create 368
- MPI_Cart_get 369
- MPI_Cart_map 372
- MPI_Cart_rank 369
- MPI_Cart_shift ??
- MPI_Cart_sub 371
- MPI_Close_port 294
- MPI_Comm_accept 293
- MPI_Comm_call_errhandler ??
- MPI_Comm_compare 268
- MPI_Comm_connect 294
- MPI_Comm_create 275
- MPI_Comm_create_errhandler 415
- MPI_Comm_create_from_group ??
- MPI_Comm_create_group 276
- MPI_Comm_create_keyval 410
- MPI_Comm_delete_attr 410
- MPI_Comm_disconnect 272
- MPI_Comm_dup 267
- MPI_Comm_dup_fn ??
- MPI_Comm_dup_with_info 267
- MPI_Comm_free 272
- MPI_Comm_free_keyval 410
- MPI_Comm_get_attr 409
- MPI_Comm_get_errhandler 413
- MPI_Comm_get_info 408
- MPI_Comm_get_name ??
- MPI_Comm_get_parent 280
- MPI_Comm_group 275
- MPI_Comm_idup 267
- MPI_Comm_idup_with_info 267
- MPI_Comm_join 296
- MPI_Comm_null_copy_fn ??
- MPI_Comm_null_delete_fn ??
- MPI_Comm_rank 34
- MPI_Comm_remote_size 280
- MPI_Comm_set_attr 409
- MPI_Comm_set_errhandler 413
- MPI_Comm_set_info 408
- MPI_Comm_set_name 267
- MPI_Comm_size 34
- MPI_Comm_spawn 289
- MPI_Comm_spawn_multiple 292
- MPI_Comm_split 272
- MPI_Comm_split_type 384
- MPI_Comm_test_inter 280
- MPI_Compare_and_swap 327
- MPI_Conversion_fn_null ??
- MPI_Conversion_fn_null_c ??
- MPI_Compare_and_swap 327
- MPI_Dims_create 367
- MPI_Dist_graph_create 373
- MPI_Dist_graph_create_adjacent ??
- MPI_Dist_graph_neighbours 378
- MPI_Dist_graph_neighbours_count 378
- MPI_Dup_fn ??
- MPI_Errhandler_create 413
- MPI_Errhandler_free 413
- MPI_Errhandler_get ??
- MPI_Errhandler_set ??
- MPI_Error_class ??
- MPI_Error_string 414
- MPI_Exscan 58
- MPI_Exscan_init 185
- MPI_Fetch_and_op 325
- MPI_File_call_errhandler 413
- MPI_File_close 357
- MPI_File_create_errhandler ??
- MPI_File_delete 358
- MPI_File_get_amode ??
- MPI_File_get_atomicity ??
- MPI_File_get_byte_offset ??
- MPI_File_get_errhandler ??
- MPI_File_get_group ??
- MPI_File_get_info ??
- MPI_File_get_position ??
- MPI_File_get_position_shared ??
- MPI_File_get_size 363
- MPI_File_get_type_extent ??
- MPI_File_get_view 363
- MPI_File_iread 360
- MPI_File_ireadall 360
- MPI_File_ireadat 360
- MPI_File_ireadallat 360
- MPI_File_ireadshared 363
- MPI_File_ivrite 360
- MPI_File_ivriteall 360
- MPI_File_ivriteat 360
- MPI_File_ivriteatall 360
- MPI_File_ivriteatshared 363
- MPI_File_open 357
- MPI_File_preadallocate 363
- MPI_File_read 359
- MPI_File_readall 359
- MPI_File_readat 359
- MPI_File_readatall 359
- MPI_File_readatallbegin 360
- MPI_File_readatallend 360
• MPI_File_read_at_all_end ??
• MPI_File_read_ordered 363
• MPI_File_read_ordered_begin ??
• MPI_File_read_ordered_end ??
• MPI_File_read_shared 363
• MPI_File_seek 358
• MPI_File_seek_shared 363
• MPI_File_set_atomicity 364
• MPI_File_set_errhandler 413
• MPI_File_set_info ??
• MPI_File_set_view ??
• MPI_Finalize 29
• MPI_Finalized 31
• MPI_Free_mem ??
• MPI_F_sync_reg ??
• MPI_Gather 62
• MPI_Gatherv 72
• MPI_Gatherv_init 185
• MPI_Gather_init 185
• MPI_Get 321
• MPI_Get_accumulate 325
• MPI_Get_address 217
• MPI_Get_count 152
• MPI_Get_count_c ??
• MPI_Get_elements 153
• MPI_Get_elements_c ??
• MPI_Get_elements_x 223
• MPI_Get_library_version 299
• MPI_Get_processor_name 32
• MPI_Get_version 411
• MPI_Graphdims_get 379
• MPI_Graph_create 379
• MPI_Graph_get 379
• MPI_Graph_map 379
• MPI_Graph_neighbors 379
• MPI_Graph_neighbors_count 379
• MPI_Grequest_complete ??
• MPI_Grequest_start ??
• MPI_Group_compare ??
• MPI_Group_excl 276
• MPI_Group_free ??
• MPI_Group_from_session_pset ??
• MPI_Group_incl 276
• MPI_Group_range_excl ??
• MPI_Group_range_incl ??
• MPI_Group_rank ??
• MPI_Group_size ??
• MPI_Group_translate_ranks ??
• MPI_Iallgather 82
• MPI_Iallgatherv 82
• MPI_Iallreduce 82
• MPI_Ialltoall 82
• MPI_Ialltoallw 82
• MPI_Ibsend 82
• MPI_Ibcast 82
• MPI_IBsend ??
• MPI_Iexscan 82
• MPI_Igather 82
• MPI_Igatherv 82
• MPI_Iimprove ??
• MPI_Iimrecv ??
• MPI_Ineighbor_allgather ??
• MPI_Ineighbor_allgatherv ??
• MPI_Ineighbor_alltoall ??
• MPI_Ineighbor_alltoallv ??
• MPI_Ineighbor_alltoallw ??
• MPI_Info_create 406
• MPI_Info_create_env ??
• MPI_Info_delete 406
• MPI_Info_dup 406
• MPI_Info_free 406
• MPI_Info_get 406
• MPI_Info_get_nkeys 406
• MPI_Info_get_string 406
• MPI_Info_get_valuelsen ??
• MPI_Info_set 406
• MPI_Init 29
• MPI_Initialized 31
• MPI_Init_thread 395
• MPI_Intercomm_create 277
• MPI_Intercomm_create_from_group ??
• MPI_Intercomm_create_from_group2 ??
• MPI_Intercomm_merge 280
• MPI_Iprobe 147
• MPI_Irecv 135
• MPI_Ireduce 82
• MPI_Ireduce_scatter 82
• MPI_Ireduce_scatter_block 82
• MPI_Iscatter 82
• MPI_Iscatterv 82
• MPI_Isend 135
• MPI_Isendrecv 133
• MPI_Isendrecv_replace 133
• MPI_Is_thread_main ??
• MPI_Keyval_create ??
• MPI_Keyval_free ??
• MPI_Lookup_name ??
• MPI_Mprobe 148
• MPI_Mrecv 148
• MPI_Neighbor_allgatherv ??
• MPI_Neighbor_allgatherv_init 185
• MPI_Neighbor_alltoall ??
• MPI_Neighbor_alltoallv ??
• MPI_Neighbor_alltoallw ??
• MPI_Neighbor_alltoallw_init 185
• MPI_Null_copy_fn ??
• MPI_Null_delete_fn ??
• MPI_Open_port 293
• MPI_Op_commulative 80
• MPI_Op_create 78
• MPI_P_control 80
• MPI_Pready 185
• MPI_Pready_list 186
• MPI_Pready_range 185
• MPI_Precv_init 147
• MPI_Psend_init 147
• MPI_Publish_name ??
• MPI_Pready 185
• MPI_Pready_list 186
• MPI_Pready_range 185
• MPI_Precv_init 147
• MPI_Psend_init 147
• MPI_Register_datarep ??
• MPI_Request_free ??
• MPI_Request_get_status 147
• MPI_Rget ??
• MPI_Rget_accumulate ??
### 65.1 From the standard document

- MPI_Win_wait 318
- MPI_Wtick 419
- MPI_Wtime 418
- PMPI_??
- PMPI_Aint_add ??
- PMPI_Aint_diff ??
- PMPI_Isend ??
- PMPI_Wtick ??
- PMPI_Wtime ??

#### 65.1.2 List of all dtypes

- Datatype ??
- Group ??
- MPI_Aint 199
- MPI_Comm_copy_attr_function ??
- MPI_Comm_delete_attr_function ??
- MPI_Copy_function ??
- MPI_Count 222
- MPI_Datarep_conversion_function ??
- MPI_Datarep_conversion_function_c ??
- MPI_Delete_function ??
- MPI_Offset 360
- MPI_Type_copy_attr_function ??
- MPI_Type_delete_attr_function ??
- MPI_Win_copy_attr_function ??
- MPI_Win_delete_attr_function ??
- Status ??
- _Bool ??
- bool ??
- char ??
- class ??
- double ??
- enum ??
- float ??
- int ??
- long ??
- short ??
- wchar_t ??

#### 65.1.3 List of all ctypes

- ALLOCATABLE ??
- ASYNCHRONOUS ??
- BLOCK ??
- CHARACTER ??
- COMMON ??
- COMM_COPY_ATTR_FUNCTION ??
- COMM_DELETE_ATTR_FUNCTION ??
- COMPLEX ??
- CONTAINS ??
- CONTIGUOUS ??
- COPY_FUNCTION ??
- C_F_POINTER ??
- C_PTR ??
- DATAREP_CONVERSION_FUNCTION ??
- DELETE_FUNCTION ??
- EXTERNAL ??
- FUNCTION ??
- IN ??
- INCLUDE ??
- INOUT ??
- INTEGER ??
- INTENT ??
- INTERFACE ??
- ISO_C_BINDING ??
- ISO_FORTRAN_ENV ??
- KIND ??
- LOGICAL ??
- MODULE ??
- MPI_Send 120
- MPI_Status 149
- MPI_User_function ??
- MPI_Waitall 139
- OPTIONAL ??
- OUT ??
- POINTER ??
- PROCEDURE ??
- REAL ??
- SEQUENCE ??
- TARGET ??
- TYPE ??
- TYPE_COPY_ATTR_FUNCTION ??
- TYPE_DELETE_ATTR_FUNCTION ??
- USER_FUNCTION ??
- VOLATILE ??
- WIN_COPY_ATTR_FUNCTION ??
- WIN_DELETE_ATTR_FUNCTION ??
- base ??
- foo ??
- int ??
- separated_sections ??

#### 65.1.4 List of all constants

- MPI_ADDRESS_KIND 200
- MPI_ANY_SOURCE 122
- MPI_ANY_TAG 123
- MPI_APHNUM 292
- MPI_Argvs_null ??
- MPI_Argv_null ??
- MPI_ASYNC_PROTECTS_NONBLOCKING ??
- MPI_Aint 199
- MPI_BAND ??
- MPI_BOTTOM ??
- MPI_BSEND_OVERHEAD 190
- MPI_BXOR ??
- MPI_CART 367
- MPI_COMBINER_CONTIGUOUS ??
- MPI_COMBINER_DARRAY ??
- MPI_COMBINER_DUP ??
- MPI_COMBINER_INDEXED ??
- MPI_COMBINER_INDEXED_BLOCK ??
- MPI_COMBINER_HINDEXED ??
- MPI_COMBINER_HINDEXED_BLOCK ??
- MPI_COMBINER_HVECTOR ??
- MPI_COMBINER_HVECTOR_INTEGER ??
- MPI_COMBINER_INDEXED ??
- MPI_COMBINER_INDEXED_BLOCK ??
- MPI_COMBINER_NAMED ??
- MPI_COMBINER_RESIZED ??
- MPI_COMBINER_STRUCT ??

Victor Eijkhout 829
65.2. MPI for Python

65.2.1 Buffer specifications

65.2.2 Listing of python routines

65.1.6 List of all callbacks

• COMM_COPY_ATTR_FUNCTION
• COMM_DELETE_ATTR_FUNCTION
• COPY_FUNCTION
• DELETE_FUNCTION
• MPI_Comm_copy_attr_function
• MPI_Comm_delete_attr_function
• MPI_Datarep_conversion_function
• MPI_Type_delete_attr_function
• MPI_User_function
• MPI_User_function_c
• MPI_Comm_errhandler_function
• MPI_Type_errhandler_function
• MPI_File_errhandler_function
• MPI_Win_errhandler_function
Chapter 66

Index of OpenMP commands

_OPENMP, 443, 548
aligned, 535
atomic, 501, 531
barrier
    cancelled by nowait, 469
barrier, 499, 499
cancel, 454, 517, 568, 569
chunk, 464
collapse, 467, 467
copyin, 495
copyprivate, 488, 495
critical, 461, 501, 546
declare, 480
declare simd, 535
default
    firstprivate, 493
    none, 493
    private, 492
    shared, 492
default, 492
depend, 514, 518
dist_schedule, 541
do, 454, 458
dynamic, 546
final, 517
firstprivate, 494, 512, 513, 540, 557
flush, 503, 533
for, 454, 458, 461
if, 517
implicit barrier
    after single directive, 488
in_reduction, 516
lastprivate, 469, 494
league, 541
linear, 535
master, 395, 487, 488, 523
nowait, 469, 500, 546, 546
omp
    barrier
        implicit, 500
omp for, 492
omp alloc, 496
OMP_CANCELLATION, 454, 543
omp_cgroup_mem_alloc, 496
omp_const_mem_alloc, 496
omp_const_mem_space, 496
OMP_DEFAULT_DEVICE, 543
omp_default_mem_alloc, 496
omp_default_mem_space, 496
omp_destroy_nest_lock, 504
OMP_DISPLAY_ENV, 522, 543
OMP_DYNAMIC, 496, 543, 544
omp_get_active_level, 543
omp_get_ancestor_thread_num, 543
omp_get_cancellation, 454
omp_get_dynamic, 543, 544
omp_get_level, 543
omp_get_max_active_levels, 452, 543
omp_get_max_threads, 447, 543, 544
omp_get_num_procs, 445, 447, 450, 452, 543, 544
omp_get_schedule, 466, 543, 544
omp_get_team_size, 543
omp_get_thread_limit, 445, 447, 450, 452, 543, 544
omp_get_wtick, 543, 545
omp_get_wtime, 543, 545
omp_high_bw_mem_alloc, 496
omp_high_bw_mem_space, 496
omp_in, 480, 481
omp_in_parallel, 454, 543, 544
omp_init_nest_lock, 504
omp_is_initial_device, 540
omp_large_cap_mem_alloc, 496
omp_large_cap_mem_space, 496
omp_low_lat_mem_alloc, 496
omp_low_lat_mem_space, 496
OMP_MAX_ACTIVE_LEVELS, 452, 543
OMP_MAX_TASK_PRIORITY, 517, 544
OMP_NESTED (deprecated), 453
OMP_NESTED, 448, 544, 544
OMP_NUM_THREADS, 443, 444, 447, 544, 544
omp_out, 480, 481
OMP_PLACES, 522, 524, 544
omp_priv, 481
OMP_PROC_BIND, 522, 524, 544, 545
omp_pthread_mem_alloc, 496
omp_sched_affinity, 466
omp_sched_auto, 466
omp_sched_dynamic, 466
omp_sched_guided, 466, 467
omp_sched_runtime, 467
omp_sched_static, 466
omp_sched_t, 466
OMP_SCHEDULE, 465–467, 544, 544
omp_set_dynamic, 543
omp_set_nested, 452, 543
omp_set_schedule, 466, 543, 544
OMP_STACKSIZE, 491, 544, 544
omp_test_nest_lock, 504
OMP_THREAD_LIMIT, 544
omp_thread_mem_alloc, 496
omp_unset_nest_lock, 504
OMP_WAIT_POLICY, 452, 544, 544
openmp_version, 443
ordered, 468, 468
parallel, 444, 444, 450, 451, 454, 458, 461, 525, 557
parallel region
  barrier at the end of, 500
pragma, see see under pragma name
priority, 517
private, 491, 557
proc_bind, 523, 525
reduction, 461, 465, 476, 479, 480, 481, 516
safelen(𝑛), 535
schedule
  auto, 465
  chunk, 464
  guided, 465
  runtime, 465
schedule, 464–466, 546
section, 486
sections, 453, 454, 476, 486, 494
simd, 535, 535
single, 487
target
enter data, 541
exit data, 541
map, 541
update from, 541
update to, 541
target, 540, 541
task, 511, 513, 514
task_reduction, 516
taskgroup, 454, 514, 516, 567, 568
taskwait, 513, 514, 516, 540, 567
taskyield, 516
team, 541
teams, 541
threadprivate, 494, 528, 546
tofrom, 540
untied, 516

wait-policy-var, 544
workshare, 488
Chapter 67

Index of PETSc commands

--sub_ksp_monitor, 664
-da_grid_x, 617
-da_refine, 624
-da_refine_x, 624
-download blas lapack, 583
-download mpich, 578
-ksp_atol, 638
-ksp_converged_reason, 639
-ksp_divtol, 638
-ksp_gmres_restart, 640
-ksp_mat_view, 601
-ksp_max_it, 638
-ksp_monitor, 645, 664
-ksp_monitor_true_residual, 645
-ksp_rtol, 638
-ksp_type, 640
-ksp_view, 638, 662, 664
-log_summary, 663
-log_view, 666
-malloc_dump, 667
-mat_view, 601, 662
-pc_factor_levels, 643
-snes_fd, 650
-snes_fd_color, 650

-vec_view, 662
-with-precision, 578
-with-scalar-type, 578

ADD_VALUES, 592, 600
AO, 608
AOViewFromOptions, 663

CHKERRA, 657
CHKERRABORT, 657
CHKERRMPI, 657
CHKERRQ, 657
CHKMEMA, 659
CHKMEMQ, 657, 659

DM, 617, 618, 624, 662
DM_BOUNDARY_GHOSTED, 617
DM_BOUNDARY_NONE, 617
DM_BOUNDARY_PERIODIC, 617
DMBoundaryType, 617
DMCreateGlobalVector, 620, 625
DMCreateLocalVector, 620, 625
DMDA, 617, 620, 624
DMDA_STENCIL_BOX, 617
DMDA_STENCIL_STAR, 617
DMDACreate1d, 617
DMDACreate2d, 617, 617
DMDACorners, 618, 625
DMDACreateLocalInfo, 618
DMDALocalInfo, 618, 619, 624
DMDASetRefinementFactor, 624
DMDAVecGetArray, 624
DMGetGlobalVector, 620
DMGetLocalVector, 620
DMGlobalToLocal, 620, 625
DMGlobalToLocalBegin, 625
DMGlobalToLocalEnd, 625
DMLocalToGlobal, 620, 625
DMLocalToGlobalBegin, 625
DMLocalToGlobalEnd, 625
DMLEX, 622
DMRestoreGlobalVector, 620
DMRestoreLocalVector, 620
DMStencilType, 617
DMViewFromOptions, 663
INSERT_VALUES, 592, 600
IS, 608
ISCreate, 606
ISCreateBlock, 606
ISCreateGeneral, 606
ISCreateStride, 606
ISGetIndices, 607
ISLocalToGlobalMappingViewFromOptions, 663
ISRestoreIndices, 607
ISViewFromOptions, 663
KSP, 601, 636
KSPBuildResidual, 646
KSPBuildSolution, 646
KSPConvergedDefault, 645
KSPConvergedReason, 638
KSPConvergedReasonView, 639
KSPConvergedReasonViewFromOptions, 663
KSPCreate, 637
KSPGetConvergedReason, 638
KSPGetIterationNumber, 639
KSPGetOperators, 638
KSPGetRhs, 646
KSPGetSolution, 646
KSPGMRESSetRestart, 640
KSPMatSolve, 640
KSPMonitorDefault, 645
KSPMonitorSet, 645
KSPMonitorTrueResidualNorm, 645
KSPReasonView (deprecated), 639
KSPSetConvergenceTest, 645
KSPSetFromOptions, 638, 640, 646
KSPSetOperators, 638
KSPSetOptionsPrefix, 664
KSPSetTolerances, 638
KSPSetType, 640
KSPView, 638, 661
KSPViewFromOptions, 663
MAT_FLUSH_ASSEMBLY, 600
MATAIJCUSPARSE, 655
MatAssemblyBegin, 600, 600
MatAssemblyEnd, 600, 600
MatCoarsenViewFromOptions, 663
MatCreate, 595
MatCreateDenseCUDA, 655
MatCreateFFT, 605
MatCreateSeqDenseCUDA, 655
MatCreateShell, 602
MatCreateSubMatrices, 602
MatCreateSubMatrix, 602, 608
MatCreateVecs, 587, 596
MatCreateVecsFFTW, 605
MATDENSECUDA, 655
MatDenseCUDAGetArray, 655
MatDenseGetArray, 600
MatDenseRestoreArray, 600
MatGetArray (deprecated), 600
MatGetRow, 600
MatImaginaryPart, 583
MatMatMult, 601
MATMPIAIJ, 595
MATMPIAIJCUSPARSE, 655
MatMPIAIJSetPreallocation, 598
MATMPIBIJ, 605
MATMPIDENSE, 595
MATMPIDENSECUDA, 655
MatMult, 601, 603
MatMultAdd, 601
MatMultHermitianTranspose, 601
MatMultTranspose, 601
MatPartitioning, 608
MatPartitioningApply, 608
MatPartitioningCreate, 608
MatPartitioningDestroy, 608
MatPartitioningSetType, 608
MatPartitioningViewFromOptions, 663
MatRealPart, 583
MatRestoreArray (deprecated), 600
MatRestoreRow, 600
MATSEQAIJ, 595
MATSEQAIJCUSPARSE, 655
MatSeqAIJGetArray, 600
MatSeqAIJRestoreArray, 600
MatSeqAIJSetPreallocation, 598
MATSEQDENSE, 595
MATSEQDENSECUDA, 655
MatSeqAIJGetArray, 600
MatSeqAIJRestoreArray, 600
MatSeqAIJSetPreallocation, 598
MATSEQDENSE, 595
MATSEQDENSECUDA, 655
MatSetOptionsPrefix, 665
MatSetSizes, 596
MatSetType, 595
MatSetValue, 599
MatSetValue, 599
MatSetValueStencil, 621
MatShellGetContext, 603
MatShellSetContext, 603
MatShellSetOperation, 603
MatSizes, 596
MATSOLVERMUMS, 646
MatSolverType, 646
MatStencil, 621
MatView, 601, 661, 662
MatViewFromOptions, 663
MPIU_COMPLEX, 583
MPIU_REAL, 583
MPIU_SCALAR, 583
PCCOMPOSITE, 645
PCFactorSetLevels, 643
PCGAMG, 643
PCHYPRESetType, 642
PCMG, 643
PCSetOptionsPrefix, 665
PCHE, 644
PCShellGetContext, 644
PCShellSetApply, 644
PCShellSetContext, 644
PCShellSetSetUp, 644
PCViewFromOptions, 663
PETSC_ARCH, 574
PETSC_CC_INCLUDES, 575
PETSC_COMM_SELF, 584, 625, 658
PETSC_COMM_WORLD, 577, 585, 659
PETSC_DECIDE, 582, 586
PETSC_DEFAULT, 638
PETSC_DIR, 574
PETSC_ERR_ARG_OUTOFRANGE, 583
PETSC_FALSE, 584
PETSC_FC_INCLUDES, 575
PETSC_HAVE_CUDA, 654
PETSC_i, 583
PETSC_MEMALIGN, 667
PETSC_NULL, 618
PETSC_NULL_CHARACTER, 577
PETSC_NULL_INTEGER, 574
PETSC_NULL_IS, 607
PETSC_NULL_OBJECT, 574
PETSC_NULL_VIEWER, 661
PETSC_STDOUT, 660
PETSC_TRUE, 584
PETSC_USE_DEBUG, 657
PETSC_VIEWER_STDOUT_WORLD, 661
PetscBLASInt, 583, 583
PetscBLASIntCast, 583
PetscBool, 584
PetscAlloc1, 666
PetscComm, 672
PetscComplex, 583, 583
PetscCUDAInitialize, 654
PetscDataType, 661
PetscDrawViewFromOptions, 663
PetscDSViewFromOptions, 663
PetscDualSpaceViewFromOptions, 663
PetscErrorCode, 583, 657
PetscFEViewFromOptions, 663
PetscFinalize, 577, 667
PetscFree, 666
PetscFunctionBegin, 657
PetscFunctionBeginUser, 657
PetscFunctionReturn, 657
PetscFinalize, 577, 667
PetscFree, 666
PetscFunctionBegin, 657
PetscFunctionBeginUser, 657
PetscFunctionReturn, 657
PetscSDLViewer, 669
PetscSDLViewer, 669
PetscGetCPUTime, 665
PetscImaginaryPart, 583
PetscInitialize, 576, 577, 663, 665, 672
PetscInitializeFortran, 577
PetscInt, 583, 583, 606
petscksp.h, 640
PetscLimiterViewFromOptions, 663
PetscLogDouble, 665
PetscLogView, 666
PetscLogViewFromOptions, 663
PetscMalloc, 593, 655, 666
PetscMalloc1, 655, 666, 667
PetscMallocDump, 667
PetscMallocResetCUDAHost, 655, 655
PetscMallocSetCUDAHost, 655, 655
PetscMPIInt, 583, 583
PetscMPIIntCast, 583
PetscNew, 666
PetscObjectSetOptionsPrefix, 665
PetscObjectViewFromOptions, 663
PetscOptionsBegin, 664
PetscOptionsEnd, 664
PetscOptionsGetInt, 663
PetscOptionsHasName, 664
PetscOptionsSetValue, 663
PetscPartitionerViewFromOptions, 663
PetscPrintf, 660, 660
PetscRandomViewFromOptions, 663
PetscReal, 200, 583, 583, 665
PetscRealPart, 583
PetscScalar, 200, 582, 583
PetscSectionViewFromOptions, 663
PetscSFViewFromOptions, 663
PetscSpaceViewFromOptions, 663
PetscSplitOwnership, 582
PetscSynchronizedFlush, 660
PetscSynchronizedPrintf, 660
PetscTime, 665
PetscViewer, 601, 661
PETSCVIEWERASCII, 662
PETSCVIEWERBINARY, 662
PetscViewerCreate, 662
PETSCVIEWERDRAW, 662
PETSCVIEWERHDF5, 662
PetscViewerPopFormat, 662
PetscViewerPushFormat, 662
PetscViewerRead, 661
PetscViewerSetOptionsPrefix, 665
PetscViewerSetType, 662
PETSCVIEWERSOCKET, 662
PETSCVIEWERSTRING, 662
PetscViewerViewFromOptions, 663
PETSCVIEWERVTK, 662
PFViewFromOptions, 663

SETERRA, 657
SETERRQ, 657
SETERRQ1, 657, 659
SETERRQ2, 657
SNES, 649
SNESconvergedReasonViewFromOptions, 663
SNESJacobianFunction, 650
SNESSetJacobian, 650
SNESSetOptionsPrefix, 665
SNESsolve, 649
SNESViewFromOptions, 663

TaoLineSearchViewFromOptions, 663
TaoViewFromOptions, 663
TSSetIFunction, 651
TSSetOptionsPrefix, 665
TSSetRHSFunction, 651
TSTrajectoryViewFromOptions, 663
TSViewFromOptions, 663

VecAssemblyBegin, 591, 592
VecAssemblyEnd, 591, 592
VecAXPY, 587
VecCreate, 584, 624
VecCreateMPICUDAWithArray, 655
VecCreateMPIWithArray, 587
VecCreateSeqCUDA, 655
VecCreateSeqWithArray, 586
VECCUDA, 655
VecCUDAGetArray, 655
VecDestroy, 584
VecDestroyVecs, 585
VecDot, 588
VecDotBegin, 589
VecDotEnd, 590
VecDuplicate, 585
VecDuplicateVecs, 585
VecGetArray, 592
VecGetArrayF90, 594
VecGetArrayRead, 592
VecGetLocalSize, 586, 588, 593
VecGetOwnershipRange, 586, 588
VecGetSize, 586, 588
VecImaginaryPart, 583
VecLoad, 595
VECMPI, 585
VECMPICUDA, 655
VecNorm, 588
VecNormBegin, 589
VecNormEnd, 590
VecPlaceArray, 594, 594
VecRealPart, 583
VecReplaceArray, 593
VecResetArray, 594
VecRestoreArray, 592
VecRestoreArrayF90, 594
VecRestoreArrayRead, 592
VecScale, 588
VecScatter, 607
VecScatterCreate, 607
VecScatterViewFromOptions, 663
VECSEQ, 584, 585
VECSEQCUDA, 655
VecSet, 590
VecSetOptionsPrefix, 665
VecSetSizes, 585, 605, 624
VecSetType, 584, 655
VecSetValue, 590, 591, 592
VecSetValues, 590, 591, 592, 624
VECSTANDARD, 585
VecView, 587, 595
VecViewFromOptions, 662, 663
Chapter 68

Index of SYCL commands

accessor, 688, 690
buffer, 688
combine, 688
cout, 691
cpu_selector, 682
deq, 691
free, 689
get_access, 690
get_range, 690
host_selector, 683
id<1>, 685
id<nd>, 685
is_cpu, 682
is_gpu, 682
is_host, 682
malloc, 689
malloc_device, 688, 689
malloc_host, 688, 689
malloc_shared, 688
nd_item, 686
offset, 692
parallel_for, 685, 688, 692
queue::memcpy, 688
range, 685, 685
read, 690
reduction, 688
runtime_error, 683
submit, 684
wait, 691
write, 690