Advanced Features of MPI-3 and MPI-4
Victor Eijkhout eijkhout@tacc.utexas.edu
ISS2022
Materials

Textbook and lecture slides:
https://tinyurl.com/vle335course

Source repository containing exercises:
https://bitbucket.org/VictorEijkhout/parallel-programming-public/
Version 3 of the MPI standard has added a number of features, some geared purely towards functionality, others with an eye towards efficiency at exascale.

Version 4 adds yet more features for exascale, and more flexible process management.

Note: MPI-3 as of 2012, 3.1 as of 2015. Fully supported everywhere. MPI-4 as of June 2021. Partial support in mpich version 4.0.1.
Part I

Fortran bindings
2. Overview

The Fortran interface to MPI had some defects. With Fortran2008 these have been largely repaired.

- The trailing error parameter is now optional;
- MPI data types are now actual Type objects, rather than Integer
- Strict type checking on arguments.
3. MPI headers

New module:

use mpi_f08 ! for Fortran2008
use mpi ! for Fortran90

True Fortran bindings as of the 2008 standard. Provided in

- Intel compiler version 18 or newer,
- gcc 9 and later (not with Intel MPI, use mvapich).
4. Optional error parameter

Old Fortran90 style:

```fortran
! your code
call MPI_Init(ierr)

! your code
call MPI_Finalize(ierr)
```

New Fortran2008 style:

```fortran
! your code
call MPI_Init()

call MPI_Finalize()
```
5. Communicators

!! Fortran 2008 interface
use mpi_f08
Type(MPI_Comm) :: comm = MPI_COMM_WORLD

!! Fortran legacy interface
#include <mpif.h>
! or: use mpi
Integer :: comm = MPI_COMM_WORLD
6. Requests

Requests are also derived types
note that ...NULL entities are now objects, not integers

```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
```

(Q for the alert student: do you see anything halfway remarkable about that index?)
7. More

\[
\begin{align*}
\text{Type(MPI\_Datatype)} & : \text{ newtype } ! \text{ F2008} \\
\text{Integer} & : \text{ newtype } ! \text{ F90}
\end{align*}
\]

Also: \text{MPI\_Comm}, \text{MPI\_Datatype}, \text{MPI\_Errhandler}, \text{MPI\_Group}, \text{MPI\_Info}, \text{MPI\_File}, \text{MPI\_Op}, \text{MPI\_Request}, \text{MPI\_Status}, \text{MPI\_Win}
8. Status

Fortran2008: status is a Type with fields:

```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
```

Fortran90: status is an array with named indexing

```fortran
!! anysource.F90
integer :: status(MPI_STATUS_SIZE)
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)
```
9. Type checking

Type checking catches potential problems:

```fortran
!! typecheckarg.F90
integer,parameter :: n=2
Integer,dimension(n) :: source
call MPI_Init()
call MPI_Send(source,MPI_INTEGER,n, &
              1,0,MPI_COMM_WORLD)
```

typecheck.F90(20): error #6285:
There is no matching specific subroutine for this generic subroutine call.  [MPI_SEND]
call MPI_Send(source,MPI_INTEGER,n, &
              1,0,MPI_COMM_WORLD)

Eijkhout: MPI course
Type checking does not catch all problems:

```
!! typecheckbuf.F90
integer,parameter :: n=1
Real,dimension(n) :: source
call MPI_Init()
call MPI_Send(source,n,MPI_INTEGER, &
1,0,MPI_COMM_WORLD)
```

Buffer/type mismatch is not caught.
Part II

Big data communication
11. Overview

This section discusses big messages.

Commands learned:

- `MPI_Send_c`, `MPI_Allreduce_c`, `MPI_Get_count_c` (MPI-4)
- `MPI_Get_elements_x`, `MPI_Type_get_extent_x`, `MPI_Type_get_true_extent_x` (MPI-3)
12. The problem with large messages

- There is no problem allocating large buffers:

  ```
  size_t bigsize = 1<<33;
  double *buffer =
      (double*) malloc(bigsize*sizeof(double));
  ```

- But you can not tell MPI how big the buffer is:

  ```
  MPI_Send(buffer,bigsize,MPI_DOUBLE,...) // WRONG
  ```

  because the size argument has to be int.
Count type since MPI 3

C:

\| MPI_Count count;

Fortran:

\| Integer(kind=MPI_COUNT_KIND) :: count

Big enough for

- int;
- MPI_Aint, used in one-sided;
- MPI_Offset, used in file I/O.

However, this type could not be used in MPI-3 to describe send buffers.
14. MPI 4 large count routines

C: routines with _c suffix

```c
MPI_Count count;
MPI_Send_c( buff, count, MPI_INT, ... );
```

also `MPI_Reduce_c`, `MPI_Get_c`, ... (some 190 routines in all)

Fortran: polymorphism rules

```fortran
Integer(kind=MPI_COUNT_KIND) :: count
call MPI_Send( buff, count, MPI_INTEGER, ... )
```
15. Big count example

```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_Ssend_c(buffer,length,MPI_DOUBLE,
                 processB,0,comm);
    MPI_Recv_c(buffer,length,MPI_DOUBLE,
                processB,0,comm,MPI_STATUS_IGNORE);
}
```
### MPI_Send

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</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>
</tr>
<tr>
<td>MPI_Send_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>buf</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>int</td>
<td>INTEGER</td>
<td>INTEGER(KIND=MPI_COUNT_KIND)</td>
<td>IN</td>
</tr>
<tr>
<td>datatype</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>dest</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>tag</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
<td></td>
</tr>
</tbody>
</table>

- **buf**: initial address of send buffer
- **count**: number of elements in send buffer
- **datatype**: datatype of each send buffer element
- **dest**: rank of destination
- **tag**: message tag
- **comm**: communicator
16. MPI 4 large count querying

C:

```c
MPI_Count count;
MPI_Get_count_c( &status,MPI_INT, &count );
MPI_Get_elements_c( &status,MPI_INT, &count );
```

Fortran:

```fortran
Integer(kind=MPI_COUNT_KIND) :: count
call MPI_Get_count( status,MPI_INTEGER,count )
call MPI_Get_elements( status,MPI_INTEGER,count )
```
17. MPI 3 kludge: use semi-large types

Make a derived datatype, and send a couple of those:

```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);
}
else if (procno==receiver) {
    MPI_Status recv_status;
    MPI_Recv(target,nblocks,blocktype,sender,0,comm, &recv_status);
}
```

You can even receive them:
18. Large int counting

MPI-3 mechanism, deprecated (probably) in MPI-4.1:

By composing types you can make a ‘big type’. Use
MPI_Type_get_extent_x, MPI_Type_get_true_extent_x, MPI_Get_elements_x to query.

```c
MPI_Count recv_count;
MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);
```
Part III

Atomic operations
19. Justification

MPI-1/2 lacked tools for race condition-free one-sided communication. These have been added in MPI-3.
20. Emulating shared memory with one-sided communication

- One process stores a table of work descriptors, and a ‘stack pointer’ stating how many there are.
- Each process reads the pointer, reads the corresponding descriptor, and decrements the pointer; and
- A process that has read a descriptor then executes the corresponding task.
- Non-collective behavior: processes only take a descriptor when they are available.
21. Simplified model

- One process has a counter, which models the shared memory;
- Each process, if available, reads the counter; and
- . . . decrements the counter.
- No actual work: random decision if process is available.
22. Shared memory problems: what is a race condition?

Race condition: outward behavior depends on timing/synchronization of low-level events. In shared memory associated with shared data.

Example:

process 1: \( I = I + 2 \)
process 2: \( I = I + 3 \)

<table>
<thead>
<tr>
<th>scenario 1.</th>
<th>scenario 2.</th>
<th>scenario 3.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>I = 0</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read ( I = 0 )</td>
<td>read ( I = 0 )</td>
<td>read ( I = 0 )</td>
</tr>
<tr>
<td>local ( I = 2 )</td>
<td>local ( I = 3 )</td>
<td>local ( I = 2 )</td>
</tr>
<tr>
<td>write ( I = 2 )</td>
<td>write ( I = 3 )</td>
<td>write ( I = 2 )</td>
</tr>
<tr>
<td>write ( I = 3 )</td>
<td>write ( I = 2 )</td>
<td></td>
</tr>
<tr>
<td><strong>I = 3</strong></td>
<td><strong>I = 2</strong></td>
<td><strong>I = 5</strong></td>
</tr>
</tbody>
</table>

(In MPI, the read/write would be `MPI_Get` / `MPI_Put` calls)
23. Case study in shared memory: 1, wrong

```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;
    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);
```
24. Discussion

- The multiple `MPI_Put` calls conflict.
- Code is correct if in each iteration there is only one writer.
- Question: In that case, can we take out the middle fence?
- Question: what is wrong with

```c
MPI_Win_fence(0, the_window);
if (i_am_available) {
    MPI_Get( &counter_value, ... )
    MPI_Win_fence(0, the_window);
    MPI_Put( ... )
}
MPI_Win_fence(0, the_window);
```

?
25. Case study in shared memory: 2, hm

```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);
```
26. Discussion: need for atomics

- MPI_Accumulate is atomic, so no conflicting writes.
- What is the problem?
- Processes are not reading unique counter_value values.
- Read and update need to come together: read unique value and immediately update.


MPI_Fetch_and_op / MPI_Get_accumulate.
Former is syntactic sugar around the latter.
# MPI\_Fetch\_and\_op

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Fetch_and_op (</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>origin_addr</td>
<td>const void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>initial address of buffer</td>
<td></td>
</tr>
<tr>
<td>result_addr</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>initial address of result buffer</td>
<td></td>
</tr>
<tr>
<td>datatype</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>datatype of the entry in origin, result, and target buffers</td>
<td></td>
</tr>
<tr>
<td>target_rank</td>
<td>int</td>
<td>INTEGER</td>
<td>rank of target</td>
<td></td>
</tr>
<tr>
<td>target_disp</td>
<td>MPI_Aint</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td>displacement from start of window to beginning of target buffer</td>
<td></td>
</tr>
<tr>
<td>op</td>
<td>MPI_Op</td>
<td>TYPE(MPI_Op)</td>
<td>reduce operation</td>
<td></td>
</tr>
<tr>
<td>win</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td>window object</td>
<td></td>
</tr>
</tbody>
</table>
27. Case study in shared memory: 3, good

```c
MPI_Win_fence(0, the_window);
int
counter_value;
if (i_am_available) {
    int
decrement = -1;
total_decrement++;
MPI_Fetch_and_op
    ( /* operate with data from origin: */ &decrement,
     /* retrieve data from target: */ &counter_value,
        MPI_INT, counter_process, 0, MPI_SUM,
            the_window);
}
MPI_Win_fence(0, the_window);
if (i_am_available) {
    my_counter_values[n_my_counter_values++] = counter_value;
}
```
### Allowable Operators (Hint!)

<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</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_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>

No user-defined operators.
We are using fences, which are collective. What if a process is still operating on its local work?

Better (but more tricky) solution: use passive target synchronization and locks.
Exercise 1 (lockfetch)

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 processes 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?
Exercise 2 (lockfetchshared)

As exercise 1, 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 \texttt{MPI\_Win\_flush\_local} to force coherence of a window (on another process) and the local variable from \texttt{MPI\_Fetch\_and\_op}. 
Part IV

Advanced collectives
30. Non-blocking collectives

- Collectives are blocking.
- Compare blocking/non-blocking sends:
  - `MPI_Send` → `MPI_Isend`
    - immediate return of control, produce request object.
- Non-blocking collectives:
  - `MPI_Bcast` → `MPI_Ibcast`

  Same:
  ```c
  || MPI_Isomething( <usual arguments>, MPI_Request *req);
  ```

- Considerations:
  - Calls return immediately;
  - the usual story about buffer reuse
  - Requires `MPI_Wait`... for completion.
  - Multiple collectives can complete in any order

- Why?
  - Use for overlap communication/computation
  - Imbalance resilience
  - Allows pipelining
### MPI_Ibcast

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Ibcast</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Ibcast_c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>buffer</td>
<td>void*</td>
<td>TYPE(*), DIMENSION(..)</td>
<td>INOUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>starting address of buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER(KIND=MPI_COUNT_KIND)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>number of entries in buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>datatype</td>
<td>MPI_Datatype</td>
<td>TYPE(MPI_Datatype)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>datatype of buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>root</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rank of broadcast root</td>
<td></td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>communicator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>request</td>
<td>MPI_Request*</td>
<td>TYPE(MPI_Request)</td>
<td>OUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>communication request</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
31. Overlapping collectives

Independent collective and local operations:

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

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

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_Request reqs[2];
MPI_Iallreduce
    ( &local_xy, &global_xy, 1, MPI_DOUBLE, MPI_SUM, comm,
      &(reqs[0]) );
MPI_Iallreduce
    ( &local_xinf, &global_xin, 1, MPI_DOUBLE, MPI_MAX, comm,
      &(reqs[1]) );
MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
```
Revisit exercise ???. 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.
33. Matching collectives

Blocking and non-blocking don’t match: either all processes call the non-blocking or all call the blocking one. Thus the following code is incorrect:

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

This is unlike the point-to-point behavior of non-blocking calls: you can catch a message with `MPI_Irecv` that was sent with `MPI_Send`.
34. Transpose as gather/scatter

Every process needs to do a scatter or gather.
35. Simultaneous collectives

Transpose matrix by scattering all rows simultaneously. Each scatter involves all processes, but with a different spanning tree.

```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);
```
Persistent collectives
36. Persistent collectives (MPI-4)

Similar to persistent send/recv:

```c
MPI_Allreduce_init( ...., &request );
for ( ... ) {
    MPI_Start( request );
    MPI_Wait( request );
}
MPI_Request_free( &request );
```

Available for all collectives and neighborhood collectives.
// 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 );

Note also the MPI_Info parameter.
38. Persistent vs non-blocking

Both request-based.

- Non-blocking is ‘ad hoc’: buffer info not known before the collective call.
- Persistent allows ‘planning ahead’: management of internal buffers and such.
Non-blocking barrier
39. Just what is a barrier?

- Barrier is not *time* synchronization but *state* synchronization.
- Test on non-blocking barrier: ‘has everyone reached some state’
Some processes decide locally to alter their structure

... need to communicate that to neighbors

Problem: neighbors don’t know whether to expect update calls, if at all.

Solution:

- send update msgs, if any;
- then post barrier.
- Everyone probe for updates, test for barrier.
41. Use case: distributed termination detection

- Distributed termination detection (Matocha and Kamp, 1998): draw a global conclusion with local operations
- Everyone posts the barrier when done;
- keeps doing local computation while testing for the barrier to complete
<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Ibarrier (</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>comm</td>
<td></td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td></td>
</tr>
<tr>
<td>communicato</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>request</td>
<td></td>
<td>MPI_Request*</td>
<td>TYPE(MPI_Request)</td>
<td></td>
</tr>
<tr>
<td>communication request</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Do sends, post 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);
```
Poll for barrier and messages

```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!
        }
    }
}
```
Exercise 4 (ibARRIERUPDATE)

- Let each process send to a random number of randomly chosen neighbors. Use MPI_Isend.
- Write the main loop with the MPI_Test call.
- Insert an MPI_Iprobe call and process incoming messages.
- Can you make sure that all sends are indeed processed?
Part V

Shared memory
Myth:

*MPI processes use network calls, whereas OpenMP threads access memory directly, therefore OpenMP is more efficient for shared memory.*

Truth:

*MPI implementations use copy operations when possible, whereas OpenMP has thread overhead, and affinity/coherence problems.*

Main problem with MPI on shared memory: data duplication.
Shared memory access: two processes can access each other’s memory through `double*` (and such) pointers, if they are on the same shared memory.

Limitation: only window memory.

Non-use case: remote update. This has all the problems of traditional shared memory (race conditions, consistency).

Good use case: every process needs access to large read-only dataset. Example: ray tracing.
46. Shared memory threatsments in MPI

- MPI uses optimizations for shared memory: copy instead of socket call
- One-sided offers ‘fake shared memory’: yes, can access another process’ data, but only through function calls.
- MPI-3 shared memory gives you a pointer to another process’ memory, if that process is on the same shared memory.
47. Shared memory per cluster node

- Cluster node has shared memory
- Memory is attached to specific socket
- beware Non-Uniform Memory Access (NUMA) effects
48. Shared memory interface

Here is the high level overview; details next.

- Use `MPI_Comm_split_type` to find processes on the same shared memory
- Use `MPI_Win_allocate_shared` to create a window between processes on the same shared memory
- Use `MPI_Win_shared_query` to get pointer to another process’ window data.
- You can now use `memcpy` instead of `MPI_Put`.
49. Discover shared memory

- **MPI_Comm_split_type** splits into communicators of same type.
- **Use type:** **MPI_COMM_TYPE_SHARED** splitting by shared memory.
  (MPI-4: split by other hardware features through **MPI_COMM_TYPE_HW_GUIDED** and **MPI_Get_hw_resource_types**)

```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);
```

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 on c206-030.frontera.tacc.utexas.edu
[4] is processor 0 in a shared group of 4, running on c206-031.frontera.tacc.utexas.edu
TACC: Shutdown complete. Exiting.
Exercise 5

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.
```
50. Allocate shared window

Use `MPI_Win_allocate_shared` to create a window that can be shared;

- Has to be on a communicator on shared memory

```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);
```
51. Get pointer to other windows

Use `MPI_Win_shared_query`:

```c
MPI_Aint window_size0; int window_unit; double *win0_addr;
MPI_Win_shared_query
  ( node_window,0,
    &window_size0,&window_unit, &win0_addr );
```
MPI_Win_shared_query

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_shared_query</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>win</td>
<td>MPI_Win</td>
<td>TYPE(MPI_Win)</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>rank</td>
<td>int</td>
<td>INTEGER</td>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>MPI_Aint*</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td>OUT</td>
<td></td>
</tr>
<tr>
<td>disp_unit</td>
<td>int*</td>
<td>INTEGER(KIND=MPI_ADDRESS_KIND)</td>
<td>OUT</td>
<td></td>
</tr>
<tr>
<td>baseptr</td>
<td>void*</td>
<td>TYPE(C_PTR)</td>
<td>OUT</td>
<td></td>
</tr>
</tbody>
</table>

shared memory window object

rank in the group of window win or MPI_PROC_NULL

size of the window segment

local unit size for displacements, in bytes

address for load/store access to window segment
52. Allocated memory

Memory will be allocated contiguously convenient for address arithmetic, not for NUMA: set alloc_shared_noncontig true in MPI_Info object.

Example: each window stores one double. Measure distance in bytes:

Strategy: default behavior of shared window allocation

Distance 1 to zero: 8
Distance 2 to zero: 16

Strategy: allow non-contiguous shared window allocation

Distance 1 to zero: 4096
Distance 2 to zero: 8192
53. Exciting example: bulk data

- Application: ray tracing:
  large read-only data structure describing the scene
- traditional MPI would duplicate:
  excessive memory demands
- Better: allocate shared data on process 0 of the shared communicator
- every else points to this object.
Let the ‘shared’ data originate on process zero in \texttt{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.
Part VI

Process topologies
54. Overview

This section discusses topologies:

- Cartesian topology
- MPI-1 Graph topology
- MPI-3 Graph topology

Commands learned:

- `MPI_Dist_graph_create`, `MPI_DIST_GRAPH`, `MPI_Dist_graph_neighbors_count`
- `MPI_Neighbor_allgather` and such
55. Process topologies

- Processes don’t communicate at random
- Example: Cartesian grid, each process 4 (or so) neighbors
- Express operations in terms of topology
- Elegance of expression
- MPI can optimize
56. Process reordering

- Consecutive process numbering often the best:
  divide array by chunks
- Not optimal for grids or general graphs:
- MPI is allowed to renumbering ranks
- Graph topology gives information from which to deduce renumbering
57. MPI-1 topology

- Cartesian topology
- Graph topology, globally specified.
  Not scalable, do not use!
58. MPI-3 topology

- Graph topologies locally specified: scalable!
- Neighborhood collectives: expression close to the algorithm.
Graph topologies
59. Example: 5-point stencil

Neighbor exchange, spelled out:

- Each process communicates down/right/up/left
- Send and receive at the same time.
- Can optimally be done in four steps
60. Step 1
61. Step 2

The middle node is blocked because all its targets are already receiving or a channel is occupied:
one missed turn
62. Neighborhood collective

This is really a ‘local gather’: each node does a gather from its neighbors in whatever order.

\texttt{MPI\_Neighbor\_allgather}
63. Why neighborhood collectives?

- Using `MPI_Isend / MPI_Irecv` is like spelling out a collective;
- Collectives can use pipelining as opposed to sending a whole buffer;
- Collectives can use spanning trees as opposed to direct connections.
64. Create graph topology

```c
int MPI_Dist_graph_create
(MPI_Comm comm_old, int nsources, const int sources[],
 const int degrees[], const int destinations[],
 const int weights[], MPI_Info info, int reorder,
 MPI_Comm *comm_dist_graph)
```

- `nsources` how many source nodes described? (Usually 1)
- `sources` the processes being described (Usually `MPI_Comm_rank` value)
- `degrees` how many processes to send to
- `destinations` their ranks
- `weights`: usually set to `MPI_UNWEIGHTED`.
- `info`: `MPI_INFO_NULL` will do
- `reorder`: 1 if dynamically reorder processes
int MPI_Neighbor_allgather
(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype,
MPI_Comm comm)

Like an ordinary MPI_Allgather, but
the receive buffer has a length \textit{degree}
(instead of comm size).
After `MPI_Neighbor_allgather` data in the buffer is *not* in normal rank order.

- `MPI_Dist_graph_neighbors_count` gives actual number of neighbors. (Why do you need this?)
- `MPI_Dist_graph_neighbors` lists neighbor numbers.
MPI_Dist_graph_neighbors

<table>
<thead>
<tr>
<th>Name</th>
<th>Param name</th>
<th>C type</th>
<th>F type</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm</td>
<td>MPI_Comm</td>
<td>TYPE(MPI_Comm)</td>
<td>communicator with distributed graph topology</td>
<td></td>
</tr>
<tr>
<td>maxindegree</td>
<td>int</td>
<td>INTEGER</td>
<td>size of sources and sourceweights arrays</td>
<td></td>
</tr>
<tr>
<td>sources</td>
<td>int[]</td>
<td>INTEGER(maxindegree)</td>
<td>processes for which the calling process is a destination</td>
<td></td>
</tr>
<tr>
<td>sourceweights</td>
<td>int[]</td>
<td>INTEGER(*)</td>
<td>weights of the edges into the calling process</td>
<td></td>
</tr>
<tr>
<td>maxoutdegree</td>
<td>int</td>
<td>INTEGER</td>
<td>size of destinations and destweights arrays</td>
<td></td>
</tr>
<tr>
<td>destinations</td>
<td>int[]</td>
<td>INTEGER(maxoutdegree)</td>
<td>processes for which the calling process is a source</td>
<td></td>
</tr>
<tr>
<td>destweights</td>
<td>int[]</td>
<td>INTEGER(*)</td>
<td>weights of the edges out of the calling process</td>
<td></td>
</tr>
</tbody>
</table>
Revisit exercise ?? and solve it using \texttt{MPI\_Dist\_graph\_create}. Use figure 67 for inspiration.

Use a degree value of 1.
Solving the right-send exercise with neighborhood collectives
68. Hints for the previous exercise

Two approaches:

1. Declare just one source: the previous process. Do this! Or:
2. Declare two sources: the previous and yourself. In that case bear in mind slide 66.
69. More graph collectives

- Heterogeneous: `MPI_Neighbor_alltoallw`.
- Non-blocking: `MPI_Ineighbor_allgather` and such
- Persistent: `MPI_Neighbor_allgather_init`, `MPI_Neighbor_allgatherv_init`.
Part VII

Partitioned communication
Hybrid scenario:
multiple threads contribute to one large message

Partitioned send/recv:
the contributions can be declared/tested
71. Create partitions

```c
// partition.c
int bufsize = nparts*SIZE;
int *partitions = (int*)malloc((nparts+1)*sizeof(int));
for (int ip=0; ip<=nparts; ip++)
    partitions[ip] = ip*SIZE;
if (procno==src) {
    double *sendbuffer = (double*)malloc(bufsize*sizeof(double));
```
72. Init calls

Similar to init calls for persistent sends, but specify the number of partitions.

```
MPI_Psend_init
  (sendbuffer, nparts, SIZE, MPI_DOUBLE, tgt, 0,
   comm, MPI_INFO_NULL, &send_request);

MPI_Precv_init
  (recvbuffer, nparts, SIZE, MPI_DOUBLE, src, 0,
   comm, MPI_INFO_NULL, &recv_request);
```
73. Partitioned send

```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);
```
74. Partitioned receive

define 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);

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 *= chk_buffer(recvbuffer, partitions[ip], partitions[ip+1], ip);
}

MPI_Request_free(&recv_request);
75. Partitioned receive tests

Use

```c
MPI_Parrived(recv_request, ipart, &flag);
```

to test for arrived partitions.
Part VIII

Sessions model
76. Problems with the ‘world model’

MPI is started exactly once:

- MPI can not close down and restart.
- Libraries using MPI need to agree on threading and such.
77. Sketch of a solution

Unique handles to the underlying MPI library
78. World and session model

- World model: what you have been doing so far; Start with `MPI_COMM_WORLD` and make subcommunicators, or spawn new world communicators and bridge them.
- Session model: have multiple sessions active, each starting/ending MPI separately.
79. Session model

- Create a session;
- a session has multiple ‘process sets’
- from a process set you make a communicator;
- Potentially create multiple sessions in one program run
- Can not mix objects from multiple simultaneous sessions
80. Session creating

```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");

Info object can also be MPI_INFO_NULL, then

MPI_Session the_session;
MPI_Session_init
    ( session_request_info,MPI_ERRORS_ARE_FATAL,
      &the_session );
MPI_Session_finalize( &the_session );
```
Process sets, identified by name (not a data type):

```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 sets mpi://SELF and mpi://WORLD are always defined.
82. Session: create communicator

Process set → group → communicator

```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);
```
83. Multiple sessions

```c
// sessionmulti.c
MPI_Info info1 = MPI_INFO_NULL, info2 = MPI_INFO_NULL;
char thread_key[] = "mpi_thread_support_level";
MPI_Info_create(&info1); MPI_Info_create(&info2);
MPI_Info_set(info1, thread_key, "MPI_THREAD_SINGLE");
MPI_Info_set(info2, thread_key, "MPI_THREAD_MULTIPLE");
MPI_Session session1, session2;
MPI_Session_init( info1, MPI_ERRORS_ARE_FATAL, &session1 );
MPI_Session_init( info2, MPI_ERRORS_ARE_FATAL, &session2 );
```
// 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_finalize(&session); }

    int compute(int x) {

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

    Library lib1, lib2;
    MPI_Init(0,0);
    MPI_Comm world = MPI_COMM_WORLD;
    int procno, nprocs;
    MPI_Comm_rank(world, &procno);
    MPI_Comm_size(world, &nprocs);
    auto sum1 = lib1.compute(procno);
    auto sum2 = lib2.compute(procno+1);
```

Part IX

Summary
86. Summary

- Fortran 2008 bindings (MPI-3)
- `MPI_Count` arguments for large buffers (MPI-4)
- Atomic one-sided communication (MPI-3)
- Non-blocking collectives (MPI-3) and persistent collectives (MPI-4)
- Shared memory (MPI-3)
- Graph topologies (MPI-3)
- Partitioned sends (MPI-4)
- Sessions model (MPI-4)