Introduction to the PETSc library
Victor Eijkhout
eijkhout@tacc.utexas.edu
TACC Training 2022/02/04
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>SPMD parallelism</td>
</tr>
<tr>
<td>3</td>
<td>Getting started</td>
</tr>
<tr>
<td>4</td>
<td>Vec datatype: vectors</td>
</tr>
<tr>
<td>5</td>
<td>Mat Datatype: matrix</td>
</tr>
<tr>
<td>6</td>
<td>KSP &amp; PC: Iterative solvers</td>
</tr>
<tr>
<td>7</td>
<td>Grid manipulation</td>
</tr>
<tr>
<td>8</td>
<td>IS and VecScatter: irregular grids</td>
</tr>
<tr>
<td>9</td>
<td>SNES: Nonlinear solvers</td>
</tr>
<tr>
<td>10</td>
<td>TS: Time stepping</td>
</tr>
<tr>
<td>11</td>
<td>Profiling, debugging</td>
</tr>
</tbody>
</table>
2. To set the stage

Developing parallel, nontrivial PDE solvers that deliver high performance is still difficult and requires months (or even years) of concentrated effort. PETSc is a toolkit that can ease these difficulties and reduce the development time, but it is not black-box PDE solver, nor a silver bullet.

Barry Smith
3. More specifically...

Portable Extendable Toolkit for Scientific Computations

- Scientific Computations: parallel linear algebra, in particular linear and nonlinear solvers
- Toolkit: Contains high level solvers, but also the low level tools to roll your own.
- Portable: Available on many platforms, basically anything that has MPI

Why use it? It’s big, powerful, well supported.
4. What is in PETSc?

• Linear algebra data structures, all serial/parallel
• Linear system solvers (sparse/dense, iterative/direct)
• Nonlinear system solvers
• Optimization: TAO (used to be separate library)
• Tools for distributed matrices
• Support for profiling, debugging, graphical output
5. Structure of a PETSc application

- Matrices
- Vectors
- Index Sets

Level of Abstraction

Application Codes
- SNES (Nonlinear Equations Solvers)
- PC (Preconditioners)
- KSP (Krylov Subspace Methods)
- TS (Time Stepping)

BLAS
MPI
# 6. Hierarchy of tools

## Parallel Numerical Components of PETSc

### Nonlinear Solvers

<table>
<thead>
<tr>
<th>Newton–based Methods</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line Search</td>
<td>Trust Region</td>
</tr>
</tbody>
</table>

### Time Steppers

<table>
<thead>
<tr>
<th>Euler</th>
<th>Backward Euler</th>
<th>Pseudo–Time Stepping</th>
<th>Other</th>
</tr>
</thead>
</table>

### Krylov Subspace Methods

<table>
<thead>
<tr>
<th>GMRES</th>
<th>CG</th>
<th>CGS</th>
<th>Bi–CG–Stab</th>
<th>TFQMR</th>
<th>Richardson</th>
<th>Chebychev</th>
<th>Other</th>
</tr>
</thead>
</table>

### Preconditioners

<table>
<thead>
<tr>
<th>Additive Schwarz</th>
<th>Block Jacobi</th>
<th>Jacobi</th>
<th>ILU</th>
<th>ICC</th>
<th>LU (sequential only)</th>
<th>Other</th>
</tr>
</thead>
</table>

### Matrices

<table>
<thead>
<tr>
<th>Compressed Sparse Row (AIJ)</th>
<th>Block Compressed Sparse Row (BAIJ)</th>
<th>Block Diagonal (BDiag)</th>
<th>Dense</th>
<th>Other</th>
</tr>
</thead>
</table>

### Vectors

<table>
<thead>
<tr>
<th>Indices</th>
<th>Block Indices</th>
<th>Stride</th>
<th>Other</th>
</tr>
</thead>
</table>

### Index Sets

<table>
<thead>
<tr>
<th>Indices</th>
<th>Block Indices</th>
<th>Stride</th>
<th>Other</th>
</tr>
</thead>
</table>
7. Documentation and help

- **Web page:** [https://petsc.org/](https://petsc.org/)
- **Documentation (pdf/html):** [https://petsc.org/release/docs/](https://petsc.org/release/docs/)
- **Follow-up to this tutorial:** eijkhout@tacc.utexas.edu
- **PETSc on your local cluster:** ask your local support
- **General questions about PETSc:** petsc-maint@mcs.anl.gov
- **Example codes, found online, and in** `$PETSC_DIR/src/mat/examples` et cetera
- **Sometimes consult include files, for instance**
  `$PETSC_DIR/include/petscmat.h`
8. External packages

PETSc does not do everything, but it interfaces to other software:

- **Dense linear algebra:** Scalapack, Plapack, Elemental
- **Grid partitioning software:** ParMetis, Jostle, Chaco, Party
- **ODE solvers:** PVODE
- **Optimization:** TAO (now integrated)
- **Eigenvalue solvers (including SVD):** SLEPc (integrated)
9. PETSc and parallelism

PETSc is layered on top of MPI

- MPI has basic tools: send elementary datatypes between processors
- PETSc has intermediate tools:
  - insert matrix element in arbitrary location,
  - do parallel matrix-vector product
- Transparent: same code works sequential and parallel.
  (Some objects explicitly declared Seq/MPI)
- ⇒ you do not need to know much MPI when you use PETSc
- All objects in Petsc are defined on a communicator;
  can only interact if on the same communicator
- No OpenMP used in the library:
  user can use shared memory programming.
- Likewise, threading is kept outside of PETSc code.
- Limited Graphics Processing Unit (GPU) support; know what you’re doing!

  *TACC note.* Only available on the Frontera RTX nodes (single precision).
10. Object oriented design

Petsc uses objects: vector, matrix, linear solver, nonlinear solver

Overloading:

\[
\text{MATMult}(A, x, y); \quad // \quad y \leftarrow A \times
\]

same for sequential, parallel, dense, sparse, FFT
To support this uniform interface, the implementation is hidden:

\[
\text{MatSetValue}(A, i, j, v, \text{INSERT\_VALUES}); // A[i,j] <- v
\]

There are some direct access routines, but most of the time you don’t need them.

(And don’t worry about function call overhead.)
# Table of Contents

1. **Introduction**

2. **SPMD parallelism**

3. **Getting started**

4. **Vec datatype: vectors**

5. **Mat Datatype: matrix**

6. **KSP & PC: Iterative solvers**

7. **Grid manipulation**

8. **IS and VecScatter: irregular grids**

9. **SNES: Nonlinear solvers**

10. **TS: Time stepping**

11. **Profiling, debugging**
One processor and one process per node; all communication goes through the network.
A node has multiple sockets, each with multiple cores. Pure MPI puts a process on each core: pretend shared memory doesn’t exist.
Hybrid programming puts a process per node or per socket; further parallelism comes from threading. No use of threading in PETSc.
PETSc is largely aimed at MPI programming; however

- You can of course use OpenMP in between PETSc calls;
- there is support for GPUs
  
  \textit{TACC note. At the moment only on frontera:} module load petsc/3.16-rtx.

- OpenMP can be used in external packages.
16. Terminology

‘Processor’ is ambiguous: is that a chip or one independent instruction processing unit?

- Socket: the processor chip
- Processor: we don’t use that word
- Core: one instruction-stream processing unit
- Process: preferred terminology in talking about MPI.
The basic model of MPI is ‘Single Program Multiple Data’: each process is an instance of the same program.

Symmetry: There is no ‘master process’, all processes are equal, start and end at the same time.

Communication calls do not see the cluster structure: data sending/receiving is the same for all neighbours.
18. Compiling and running

MPI compilers are usually called mpicc, mpif90, mpicxx.

These are not separate compilers, but scripts around the regular C/Fortran compiler. You can use all the usual flags.

At TACC:
ibrun yourprog
the number of processes is determined by SLURM.
19. Do I need a supercomputer?

- With `mpiexec` and such, you start a bunch of processes that execute your PETSc program.
- Does that mean that you need a cluster or a big multicore?
- No! You can start a large number of processes, even on your laptop. The OS will use ‘time slicing’.
- Of course it will not be very efficient...
20. Cluster setup

Typical cluster:

- Login nodes, where you ssh into; usually shared with 100 (or so) other people. You don’t run your parallel program there!
- Compute nodes: where your job is run. They are often exclusive to you: no other users getting in the way of your program.

Hostfile: the description of where your job runs. Usually generated by a job scheduler.
21. In a picture

```c
int main() {
    .....  
    printf("Hello world\n");  
    .....  
}
```

```
int main() {
    .....  
    printf("Hello world\n");  
    .....  
}
```

```
int main() {
    .....  
    printf("Hello world\n");  
    .....  
}
```

```
int main() {
    .....  
    printf("Hello world\n");  
    .....  
}
```
22. Process identification

Every process has a number (with respect to a communicator)

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

For now, the communicator will be `MPI_COMM_WORLD`.

Note: mapping of ranks to actual processes and cores is not predictable!
# Table of Contents

1. Introduction
2. SPMD parallelism
3. Getting started
   - **Vec** datatype: vectors
   - **Mat** Datatype: matrix
   - **KSP & PC**: Iterative solvers
   - Grid manipulation
   - IS and VecScatter: irregular grids
   - **SNES**: Nonlinear solvers
   - **TS**: Time stepping
   - Profiling, debugging
23. Include files

C:

```c
#include "petsc.h"

int main(int argc, char **argv)
```
24. Variable declarations, C

```c
KSP solver;
Mat A;
Vec x, y;
PetscInt n = 20;
PetscScalar v;
PetscReal nrm;
```

Note: Scalar vs Real
// init.c
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");

Can replace MPI_Init

General: Every routine has an error return. Catch that value!
26. A word about datatypes

PETSc programs can not mix single and double precision, nor real/complex: **PetscScalar** is single/double/complex depending on the installation. **PetscReal** is always real, even in complex installations.

Similarly, **PetscInt** is 32/64 bit depending.

Other scalar data types: **PetscBool, PetscErrorCode**

*TACC note.*

module spider petsc
module avail petsc

module load petsc/3.16-i64 # et cetera
27. Debug and production

While you are developing your code:

```bash
module load petsc/3.16-debug
# or 3.16-complexdebug, i64debug, rtxdebug &c
```

This does bounds tests and other time-wasting error checking.

Production:

```bash
module load petsc/3.16
```

This will just bomb if your program is not correct.

Every petsc configuration is available as debug and non-debug.
Exercise 1 (hello)

Look up the function \texttt{PetscPrintf} and print a message ‘This program runs on 27 processors’ from process zero.

- Start with the template code \texttt{hello.c/hello.F}
- (or see slide 22)
- Compile with \texttt{make hello}
- Part two: use \texttt{PetscSynchronizedPrintf}
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
28. About routine prototypes: C/C++

Prototype:

```
PetscErrorCode VecCreate(MPI_Comm comm, Vec *v);
```

Use:

```
PetscErrorCode ierr;
MPI_Comm comm = MPI_COMM_WORLD;
Vec v;
ierr = VecCreate(comm,&vec); CHKERRQ(ierr).
```

(always good idea to catch that error code)
29. Note to self

```c
PetscInitialize
    (&argc,&args,0,"Usage: prog -o1 v1 -o2 v2\n");
```

run as

```
./program -help
```

This displays the usage note, plus all available petsc options.

Not available in Fortran
Debugging support:

```c
PetscFunctionBeginUser;
// all statements
PetscFunctionReturn(0);
```

leads to informative tracebacks.

(Only in C, not in Fortran)
31. Example: function with error

```c
// backtrace.c

PetscErrorCode this_function_bombs() {
    PetscFunctionBegin;
    SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this");
    PetscFunctionReturn(0);
}
```

Victor Eijkhout

Introduction to the PETSc library
32. Example: error traceback

[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
Exercise 2 (root)

Start with root.c. Write a function that computes a square root, or displays an error on negative input: Look up the definition of SETERRQ1.

```c
x = 1.5; ierr = square_root(x,&rootx); CHKERRQ(ierr);
PetscPrintf(PETSC_COMM_WORLD,"Root of %f is %f\n",x,rootx);
x = -2.6; ierr = square_root(x,&rootx); CHKERRQ(ierr);
PetscPrintf(PETSC_COMM_WORLD,"Root of %f is %f\n",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
(I’m leaving out the `CHKERRQ(ierr)` in the examples, but do use this in actual code)

```
ierr = PetscOptionsGetInt
      (PETSC_NULL,PETSC_NULL,"-n",&n,&flag); CHKERRQ(ierr);
ierr = PetscPrintf
      (comm,"Input parameter: %d\n",n); CHKERRQ(ierr);
```

Read commandline argument, print out from processor zero; flag can be `PETSC_NULL` if not wanted
Everything in PETSc is an object, with create and destroy calls:

```c
VecCreate(MPI_Comm comm, Vec *v);
VecDestroy(Vec *v);

/* C */
Vec V;
VecCreate(MPI_COMM_WORLD, &V);
VecDestroy(&V);
```
35. More about vectors

A vector is a vector of **PetscScalar**: there are no vectors of integers (see the **IS** datatype later)

The vector object is not completely created in one call:

```c
VecSetType(V, VECMPI) // or VECSEQ
VecSetSizes(Vec v, int m, int M);
```

Other ways of creating: make more vectors like this one:

```c
VecDuplicate(Vec v, Vec *w);
```
VecSetSizes(Vec v, int m, int M);

Local size can be specified as PETSC_DECIDE.

VecSetSizes(V,PETSC_DECIDE,8)
VecSetSizes(V,PETSC_DECIDE,8)
VecSetSizes(V,PETSC_DECIDE,8)
37. Parallel layout specified

Local or global size in

\[
\text{VecSetSizes}(\text{Vec } v, \text{ int } m, \text{ int } M);
\]

Global size can be specified as \text{PETSC_DECIDE}.

\[
\begin{align*}
\text{VecSetSizes}(V,2,5) \\
\text{VecSetSizes}(V,3,5) \\
\text{VecSetSizes}(V,2,\text{PETSC_DECIDE}) \\
\text{VecSetSizes}(V,3,\text{PETSC_DECIDE})
\end{align*}
\]
38. Query parallel layout

Query vector layout:

\[ \text{VecGetSize}(\text{Vec}, \text{PetscInt} \,* \text{globalsize}) \]
\[ \text{VecGetLocalSize}(\text{Vec}, \text{PetscInt} \,* \text{localsize}) \]
\[ \text{VecGetOwnershipRange}(\text{Vec} \, x, \text{PetscInt} \,* \text{low}, \text{PetscInt} \,* \text{high}) \]

On 2nd processor:

\[ \text{low} = 3 \]
\[ \text{high} = 6 \]
39. Layout, regardless object

Query general layout:

```
| PetscSplitOwnership(MPI_Comm comm, PetscInt *n, PetscInt *N);
```

(get local/global given the other)
40. Setting values

Set vector to constant value:

\[ \text{VecSet}(\text{Vec } x, \text{PetscScalar } \text{value}); \]

Set individual elements (global indexing!):

\[ \text{VecSetValue} \]
\[ (\text{Vec } x, \text{int } \text{row}, \text{PetscScalar } \text{value}, \]
\[ \text{InsertMode } \text{mode}); \]

\[ i = 1; \ v = 3.14; \]
\[ \text{VecSetValue}(x, i, v, \text{INSERT_VALUES}); \]

\[ \text{call VecSetValue}(x, i, v, \text{INSERT_VALUES}) \]

The other insertmode is ADD_VALUES.
41. Setting values by block

Set individual elements (global indexing!):

```
VecSetValues(Vec x, int n, int *rows, PetscScalar *values, InsertMode mode); // INSERT_VALUES or ADD_VALUES

ii[0] = 1; ii[1] = 2; vv[0] = 2.7; vv[1] = 3.1;
VecSetValues(x, 2, ii, vv, INSERT_VALUES);

ii(1) = 1; ii(2) = 2; vv(1) = 2.7; vv(2) = 3.1
call VecSetValues(x, 2, ii, vv, INSERT_VALUES, ierr, e)
```
42. Setting values

No restrictions on parallelism; after setting, move values to appropriate processor:

```c
VecAssemblyBegin(Vec x);
VecAssemblyEnd(Vec x);
```

‘Latency hiding’: some of the implementation is visible here to the user
43. Basic operations

```c
VecAXPY(Vec y, PetscScalar a, Vec x); /* y <- y + a x */
VecAYPX(Vec y, PetscScalar a, Vec x); /* y <- a y + x */
VecScale(Vec x, PetscScalar a);
VecDot(Vec x, Vec y, PetscScalar *r); /* several variants */
VecMDot(Vec x, int n, Vec y[], PetscScalar *r);
VecNorm(Vec x, NormType type, PetscReal *r);
VecSum(Vec x, PetscScalar *r);
VecCopy(Vec x, Vec y);
VecSwap(Vec x, Vec y);
VecPointwiseMult(Vec w, Vec x, Vec y);
VecPointwiseDivide(Vec w, Vec x, Vec y);
VecMAXPY(Vec y, int n, PetscScalar *a, Vec x[]);
VecMax(Vec x, int *idx, double *r);
VecMin(Vec x, int *idx, double *r);
VecAbs(Vec x);
VecReciprocal(Vec x);
VecShift(Vec x, PetscScalar s);
```
Exercise 3 (vec)

Create a vector where the values are a single sine wave. using `VecGetSize`, `VecGetLocalSize`, `VecGetOwnershipRange`. Quick visual inspection:

```
ibrun vec -n 12 -vec_view
```
Exercise 4 (vec)

Use the routines VecDot, VecScale and VecNorm to compute the inner product of vectors $x, y$, scale the vector $x$, and check its norm:

\[
p \leftarrow x^t y \\
x \leftarrow x/p \\
n \leftarrow \|x\|_2
\]
44. Split dot products and norms

MPI is capable (in principle) of ‘overlapping computation and communication’.

- Start inner product / norm with `VecDotBegin` / `VecNormBegin`;
- Conclude inner product / norm with `VecDotEnd` / `VecNormEnd`;

Also: start/end multiple norm/dotproduct operations.
45. Direct access to vector values (C)

Setting values is done without user access to the stored data. Getting values is often not necessary: many operations provided. What if you do want access to the data?

Solution 1. Create vector from user provided array:

```c
VecCreateSeqWithArray(MPI_Comm comm, PetscInt n, const PetscScalar array[], Vec *V)
VecCreateMPIWithArray(MPI_Comm comm, PetscInt n, PetscInt N, const PetscScalar array[], Vec *vv)
```
46. Direct access’

Solution 2. Retrieve the internal array:

```c
VecGetArray(Vec x, PetscScalar *a[])
/* do something with the array */
VecRestoreArray(Vec x, PetscScalar *a[])
```

Note: local only; see VecScatter for more general mechanism)
47. Getting values example

```c
int localsize, first, i;
PetscScalar *a;
VecGetLocalSize(x,&localsize);
VecGetOwnershipRange(x,&first,PETSC_NULL);
VecGetArray(x,&a);
for (i=0; i<localsize; i++)
    printf("Vector element %d : %e\n", first+i,a[i]);
VecRestoreArray(x,&a);
```

Fortran: `PETSC_NULL_INTEGER`
48. More array juggling

- **VecPlaceArray**: replace the internal array; the original can be restored with **VecRestoreArray**
- **VecReplaceArray**: replace and free the internal array.
### 49. Matrix creation

The usual create/destroy calls:

- `MatCreate(MPI_Comm comm, Mat *A)`
- `MatDestroy(Mat *A)`

Several more aspects to creation:

- `MatSetType(A, MATSEQAIJ) /* or MATMPIAIJ or MATAIJ */`
- `MatSetSizes(Mat A, int m, int n, int M, int N)`
- `MatSeqAIJSetPreallocation /* more about this later*/`
  - `(Mat B, PetscInt nz, const PetscInt nnz[])`

Local or global size can be `PETSC_DECIDE` (as in the vector case)
50. If you already have a CRS matrix

```c
PetscErrorCode  MatCreateSeqAIJWithArrays
    (MPI_Comm  comm, PetscInt  m, PetscInt  n,
     PetscInt*  i, PetscInt*  j, PetscScalar*  a, Mat*  mat)
```

(also from triplets)

Do not use this unless you interface to a legacy code. And even then...
51. Matrix Preallocation

- PETSc matrix creation is very flexible:
  - No preset sparsity pattern
  - any processor can set any element
    ⇒ potential for lots of malloc calls
- tell PETSc the matrix’ sparsity structure
  (do construction loop twice: once counting, once making)
- Re-allocating is expensive:

  \[ \text{MatSetOption}(A, \text{MAT_NEW_NONZERO_LOCATIONS}, \text{PETSC_FALSE}); \]

(is default) Otherwise:

  [1]PETSC ERROR: Argument out of range
  [1]PETSC ERROR: New nonzero at (0,1) caused a malloc
MatSeqAIJSetPreallocation

(Mat B,PetscInt nz,const PetscInt nnz[])

- $nz$ number of nonzeros per row (or slight overestimate)
- $nnz$ array of row lengths (or overestimate)
- considerable savings over dynamic allocation!

In Fortran use PETSC_NULL_INTEGER if not specifying $nnz$ array
53. Parallel matrix structure

Off-diagonal block has off-processor connections

Diagonal block has on-processor connections

A

B
54. (why does it do this?)

- \( y \leftarrow A x_A + B x_b \)
- \( x_B \) needs to be communicated; \( A x_A \) can be computed in the meantime

**Algorithm**
- Initiate asynchronous sends/receives for \( x_b \)
- compute \( A x_A \)
- make sure \( x_b \) is in
- compute \( B x_B \)

- so by splitting matrix storage into \( A, B \) part, code for the sequential case can be reused.

- This is one of the few places where PETSc’s design is visible to the user.
55. Parallel matrix structure description

- $m, n$ local size; $M, N$ global. Note: If the matrix is square, specify $m, n$ equal, even though distribution by block rows
- $d_{nz}$: number of nonzeros per row in diagonal part
- $o_{nz}$: number of nonzeros per row in off-diagonal part
- $d_{nnz}$: array of numbers of nonzeros per row in diagonal part
- $o_{nnz}$: array of numbers of nonzeros per row in off-diagonal part

```
MatMPIAIJSetPreallocation
(Mat B,
 PetscInt d_nz, const PetscInt d_nnz[],
 PetscInt o_nz, const PetscInt o_nnz[])
```

In Fortran use `PETSC_NULL_INTEGER` if not specifying arrays
56. Matrix creation all in one

\begin{verbatim}
MatCreateSeqAIJ(MPI_Comm comm, PetscInt m, PetscInt n,
    PetscInt nz, const PetscInt nnz[], Mat *A)
MatCreateMPIAIJ(MPI_Comm comm,
    PetscInt m, PetscInt n, PetscInt M, PetscInt N,
    PetscInt d_nz, const PetscInt d_nnz[],
    PetscInt o_nz, const PetscInt o_nnz[],
    Mat *A)
\end{verbatim}
Matrix partitioned by block rows:

```c
MatGetSize(Mat mat, PetscInt *M, PetscInt* N);
MatGetLocalSize(Mat mat, PetscInt *m, PetscInt* n);
MatGetOwnershipRange(Mat A, int *first row, int *last row);
```

In query functions, unneeded components can be specified as `PETSC_NULL`. Fortran: `PETSC_NULL_INTEGER`
58. Setting values

Set one value:

\[
\text{MatSetValue}(\text{Mat } A, \text{PetscInt } i, \text{PetscInt } j, \text{PetscScalar } va, \text{InsertMode mode})
\]

where insert mode is \text{INSERT_VALUES, ADD_VALUES}

Set block of values:

\[
\text{MatSetValues}(\text{Mat } A, \text{int } m, \text{const int idxm}[], \text{int } n, \text{const int idxn}[], \text{const PetscScalar } values[], \text{InsertMode mode})
\]

\((v \text{ is row-oriented})\)
59. Set only one element

\[ \textbf{MatSetValue}(A, i, j, &v, \text{INSERT\_VALUES}); \]

Special case of the general case:

\[ \textbf{MatSetValues}(A, 1, &i, 1, &j, &v, \text{INSERT\_VALUES}); \]
60. Assembling the matrix

Setting elements is independent of parallelism; move elements to proper processor:

```c
MatAssemblyBegin(Mat A, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(Mat A, MAT_FINAL_ASSEMBLY);
```

Cannot mix inserting/adding values: need to do assembly in between with `MAT_FLUSH_ASSEMBLY`
Exercise 5 (matvec)

Pretend that you do not know how the matrix is created. Use `MatGetOwnershipRange` or `MatGetLocalSize` to create a vector with the same distribution, and then compute $y \leftarrow Ax$.

(Part of the code has been disabled with `#if 0`. We will get to that next.)
61. Getting values (C)

- Values are often not needed: many matrix operations supported
- Matrix elements can only be obtained locally.

```c
PetscErrorCode MatGetRow(Mat mat,
  PetscInt row, PetscInt *ncols, const PetscInt *cols[],
  const PetscScalar *vals[])

PetscErrorCode MatRestoreRow(/* same parameters */
```

Note: for inspection only; possibly expensive.
Exercise 6 (matvec)

Advanced exercise: create a sequential (uni-processor) vector. Question: how does the code achieve this? Give it the data of the distributed vector. Use that to compute the vector norm on each process separately.

(Start by removing the \#if 0 and \#endif.)
62. Other matrix types

**MATBAIJ**: blocked matrices (dof per node)

(see `PETSC_DIR/include/petscmat.h`)

**Dense:**

```c
MatCreateSeqDense(PETSC_COMM_SELF, int m, int n,
  PetscScalar *data, Mat *A);
MatCreateDense(MPI_Comm comm,
  PetscInt m, PetscInt n, PetscInt M, PetscInt N, 
  PetscScalar *data, Mat *A)
```

Data argument optional: `PETSC_NULL` or `PETSC_NULL_SCALAR` causes allocation
63. GPU support

- Create as GPU matrix,
- Otherwise transparent through overloading

```c
// cudainit.c
ierr = PetscCUDAInitialize(comm, PETSC_DECIDE); CHKERRQ(ierr);
ierr = PetscCUDAInitializeCheck(); CHKERRQ(ierr);
```

VECCUDA, MatCreateDenseCUDA, MATAIJCUSPARSE
```c
ierr = MatCreate(comm, &A); CHKERRQ(ierr);
#ifdef PETSC_HAVE_CUDA
ierr = MatSetType(A, MATMPIAIJCUSPARSE); CHKERRQ(ierr);
#else
ierr = MatSetType(A, MATMPIAIJ); CHKERRQ(ierr);
#endif
```
65. Matrix operations

Main operations are matrix-vector:

- `MatMult(Mat A, Vec in, Vec out);`
- `MatMultAdd`
- `MatMultTranspose`
- `MatMultTransposeAdd`

Simple operations on matrices:

- `MatNorm`
- `MatScale`
- `MatDiagonalScale`
66. Some matrix-matrix operations

\begin{verbatim}
MatMatMult(Mat, Mat, MatReuse, PetscReal, Mat*);
MatPtAP(Mat, Mat, MatReuse, PetscReal, Mat*);
MatMatMultTranspose(Mat, Mat, MatReuse, PetscReal, Mat*);
MatAXPY(Mat, PetscScalar, Mat, MatStructure);
\end{verbatim}
67. Matrix viewers

\begin{verbatim}
MatView(A, PETSC_VIEWER_STDOUT_WORLD);

row 0: (0, 1) (2, 0.333333) (3, 0.25) (4, 0.2)
row 1: (0, 0.5) (1, 0.333333) (2, 0.25) (3, 0.2)
...
\end{verbatim}

(Fortran: \texttt{PETSC\_NULL\_INTEGER})

- also invoked by \texttt{-mat_view}
- Sparse: only allocated positions listed
- other viewers: for instance \texttt{-mat_view_draw (X terminal)}
68. General viewers

Any PETSc object can be ‘viewed’

- Terminal output: useful for vectors and matrices but also for solver objects.
- Binary output: great for vectors and matrices.
- Viewing can go both ways: load a matrix from file or URL into an object.
- Viewing through a socket, to Matlab or Mathematica, HDF5, VTK.

```c
PetscViewer fd;
PetscViewerCreate( comm, &fd );
PetscViewerSetType( fd,PETSCVIEWERVTX );
MatView( A,fd );
PetscViewerDestroy(fd);
```
69. Shell matrices

What if the matrix is a user-supplied operator, and not stored?

```c
MatSetType(A, MATSHELL); /* or */
MatCreateShell(MPI Comm comm,
    int m, int n, int M, int N, void *ctx, Mat *mat);

PetscErrorCode UserMult(Mat mat, Vec x, Vec y);

MatShellSetOperation(Mat mat, MatOperation MATOP_MULT,
    (void (*)(void)) PetscErrorCode (*UserMult)(Mat, Vec, Vec));
```

Inside iterative solvers, PETSc calls `MatMult(A, x, y)`:
no difference between stored matrices and shell matrices
Shell matrices need custom data

```c
MatShellSetContext(Mat mat, void *ctx);
MatShellGetContext(Mat mat, void **ctx);
```

(This does not work in Fortran: use Common or Module or write interface block)

User program sets context, matmult routine accesses it
Shell matrix example

```c
... 
MatSetType(A, MATSHELL);
MatShellSetOperation(A, MATOP_MULT, (void*)&mymatmult);
MatShellSetContext(A, (void*)&mystruct);
...

PetscErrorCode mymatmult(Mat mat, Vec in, Vec out)
{
    PetscFunctionBegin;
    MatShellGetContext(mat, (void**)&mystruct);
    /* compute out from in, using mystruct */
    PetscFunctionReturn(0);
}
```
72. Submatrices

Extract one parallel submatrix:

```
MatGetSubMatrix(Mat mat, IS isrow, IS iscol, PetscInt csizes, MatReuse cll, Mat *newmat)
```

Extract multiple single-processor matrices:

```
MatGetSubMatrices(Mat mat, PetscInt n, const IS irow[], const IS icol[], MatReuse scall, Mat *submat[])```

Collective call, but different index sets per processor
73. Load balancing

```
MatPartitioningCreate
    (MPI_Comm comm, MatPartitioning *part);
```

Various packages for creating better partitioning: Chaco, Parmetis
# Table of Contents

1. Introduction
2. SPMD parallelism
3. Getting started
4. Vec datatype: vectors
5. Mat Datatype: matrix
6. KSP & PC: Iterative solvers
7. Grid manipulation
8. IS and VecScatter: irregular grids
9. SNES: Nonlinear solvers
10. TS: Time stepping
11. Profiling, debugging
74. What are iterative solvers?

Solving a linear system $Ax = b$ with Gaussian elimination can take lots of time/memory.

Alternative: iterative solvers use successive approximations of the solution:

- Convergence not always guaranteed
- Possibly much faster / less memory
- Basic operation: $y \leftarrow Ax$ executed once per iteration
- Also needed: preconditioner $B \approx A^{-1}$
75. Topics

- All linear solvers in PETSc are iterative, even the direct ones
- Preconditioners
- Fargoing control through commandline options
- Tolerances, convergence and divergence reason
- Custom monitors and convergence tests
76. Iterative solver basics

- **KSP** object: solver
- set linear system operator
- solve with rhs/sol vector
- this is a default setup

```c
KSPCreate(comm, &solver); KSPDestroy(solver);
// set Amat and Pmat
KSPSetOperators(solver, A, B); // usually: A, A
// solve
KSPSolve(solver, rhs, sol);
```

Optional: **KSPSetUp**(solver)
77. Solver settings

Change default settings by program calls
example: solver type

```c
KSPSetType(solver, KSPGMRES);
```

Settings can be controlled from the commandline:

```c
KSPSetFromOptions(solver);
/* right before KSPSolve or KSPSetUp */
```

then options `-ksp....` are parsed.

- **type:** `-ksp_type gmres -ksp_gmres_restart 20`
- `-ksp_view` for seeing all settings
Iterative solvers can fail

- Solve call itself gives no feedback: solution may be completely wrong
- \texttt{KSPGetConvergedReason}(\texttt{solver}, &\texttt{reason})
  - positive is convergence, negative divergence
- \texttt{KSPConvergedReasons}[\texttt{reason}] is string
- \texttt{KSPGetIterationNumber}(\texttt{solver}, &\texttt{nits}) : after how many iterations did the method stop?
79. Reason for convergence

Query the solver object:

```c
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);
 }
```
System $Ax = b$ is transformed:

$$M^{-1}A = M^{-1}b$$

- $M$ is constructed once, applied in every iteration
- If $M = A$: convergence in one iteration
- Tradeoff: $M$ expensive to construct $\Rightarrow$ low number of iterations; construction can sometimes be amortized.
- Other tradeoff: $M$ more expensive to apply and only modest decrease in number of iterations
- Symmetry: $A, M$ symmetric $\not\Rightarrow M^{-1}A$ symmetric, however can be symmetrized by change of inner product
- Can be tricky to make both parallel and efficient
81. PC basics

- PC usually created as part of KSP: separate create and destroy calls exist, but are (almost) never needed.

```c
// kspcg.c
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);
}
```

- Many choices, some with options: `PCJACOBI`, `PCILU` (only sequential), `PCASM`, `PCBJACOBI`, `PCMG`, et cetera

- Controllable through commandline options:
  `-pc_type ilu -pc_factor_levels 3`
82. Preconditioner reuse

In context of nonlinear solvers, the preconditioner can sometimes be reused:

- If the jacobian doesn’t change much, reuse the preconditioner completely
- If the preconditioner is recomputed, the sparsity pattern probably stays the same

\texttt{KSPSetOperators(solver,A,B)}

- \( B \) is basis for preconditioner, need not be \( A \)
- if \( A \) or \( B \) is to be reused, use \texttt{NULL}
83. Types of preconditioners

- Simple preconditioners: Jacobi, SOR, ILU
- Compose simple preconditioners:
  - composing in space: Block Jacobi, Schwarz
  - composing in physics: Fieldsplit
- Global parallel preconditioners: multigrid, approximate inverses
84. Simple preconditioners

\[ A = D_A + L_A + U_A, \quad M = \ldots \]

- None: \( M = I \)
- Jacobi: \( M = D_A \)
  - very simple, better than nothing
  - Watch out for zero diagonal elements
- Gauss-Seidel: \( M = D_A + L_A \)
  - Non-symmetric
  - popular as multigrid smoother
- SOR: \( M = \omega^{-1}D_A + L_A \)
  - estimating \( \omega \) often infeasible
- SSOR: \( M = (I + (\omega^{-1}D_A)^{-1} + L_A)(\omega^{-1}D_A + U_A) \)

Mostly of textbook value.
See next for more state-of-the-art.
85. Factorization preconditioners

Exact factorization: $A = LU$

Inexact factorization: $A \approx M = LU$ where $L, U$ obtained by throwing away ‘fill-in’ during the factorization process.

Exact:

$$\forall i,j : a_{ij} \leftarrow a_{ij} - a_{ik} a^{-1}_{kk} a_{kj}$$

Inexact:

$$\forall i,j : \text{if } a_{ij} \neq 0 \ a_{ij} \leftarrow a_{ij} - a_{ik} a^{-1}_{kk} a_{kj}$$

Application of the preconditioner (that is, solve $Mx = y$) approx same cost as matrix-vector product $y \leftarrow Ax$
PCICC: symmetric, PCILU: nonsymmetric
many options:

\[
\text{PCFactorSetLevels}(\text{PC } pc, \text{int } levels);
\]
\[-pc\text{-factor}\_\text{levels} <\text{levels}>
\]

Prevent indefinite preconditioners:

\[
\text{PCFactorSetShiftType}(\text{PC } pc, \text{MatFactorShiftType } type);
\]

value \text{MAT\_SHIFT\_POSITIVE\_DEFINITE} et cetera

Factorization preconditioners are sequential
but still useful; see later
87. Block Jacobi and Additive Schwarz, theory

- Both methods parallel
- Jacobi fully parallel
  - Schwarz local communication between neighbours
- Both require sequential local solver: composition with simple preconditioners
- Jacobi limited reduction in iterations
  - Schwarz can be optimal
88. Block Jacobi and Additive Schwarz, coding

```c
KSP *ksp; int nlocal, firstlocal; PC pc;
PCBJacobiGetSubKSP(pc,&nlocal,&firstlocal,&ksp);
for (i=0; i<nlocal; i++) {
    KSPSetType( ksp[i], KSPGMRES );
    KSPGetPC( ksp[i], &pc );
    PCSetType( pc, PCILU );
}
```

Much shorter: commandline options `-sub_ksp_type` and `-sub_pc_type`
(subksp is PREONLY by default)

```c
PCASMSSetOverlap(PC pc, int overlap);
```
Exercise 7 (**ksp**)

File **ksp.c / ksp.F90** contains the solution of a (possibly nonsymmetric) linear system.

Compile the code and run it. Now experiment with commandline options. Make notes on your choices and their outcomes.

- The code has two custom commandline switch:
  - `-n 123` set the domain size to 123 and therefore the matrix size to $123^2$.
  - `-unsymmetry 456` adds a convection-like term to the matrix, making it unsymmetric. The numerical value is the actual element size that is set in the matrix.
- What is the default solver in the code? Run with `-ksp_view`.
- Print out the matrix for a small size with `-mat_view`.
- Now out different solvers for different matrix sizes and amounts of unsymmetry. See the instructions in the code.
Exercise 8 (shell)

After the main program, a routine `mymatmult` is declared, which is attached by
`MatShellSetOperation` to the matrix $A$ as the means of computing the
product $\text{MatMult}(A, \text{in, out})$, for instance inside an iterative method.

In addition to the shell matrix $A$, the code also creates a traditional matrix $AA$. Your assignment is to make it so that `mymatmult` computes the product
$y \leftarrow A^tAx$.

In C, use `MatShellSetContext` to attach $AA$ to $A$ and `MatShellGetContext` to
retrieve it again for use; in Fortran use a common block (or a module) to store
$AA$.

The code uses a preconditioner `PCNONE`. What happens if you run it with option
`-pc_type jacobi`?
89. Monitors and convergence tests

| KSPSetTolerances(solver, rtol, atol, dtol, maxit); |

Monitors can be set in code, but simple cases:

- -ksp_monitor
- -ksp_monitor_true_residual
90. Custom monitors and convergence tests

```c
KSPMonitorSet(KSP ksp,
    PetscErrorCode (*monitor)(KSP,PetscInt,PetscReal,void*),
    void *mctx,
    PetscErrorCode (*monitordestroy)(void*));

KSPSetConvergenceTest(KSP ksp,
    PetscErrorCode (*converge)(KSP,PetscInt,PetscReal,KSPConvergedReason*,void*),
    void *cctx,
    PetscErrorCode (*destroy)(void*))
```
PetscErrorCode resconverge
(KSP solver, PetscInt it, PetscReal res,
KSPConvergedReason *reason, void *ctx)
{
    MPI_Comm comm; Mat A; Vec X,R; PetscErrorCode ierr;
    PetscFunctionBegin;
    KSPGetOperators(solver,&A,PETSC_NULL,PETSC_NULL);
    PetscObjectGetComm((PetscObject)A,&comm);
    KSPBuildResidual(solver,PETSC_NULL,PETSC_NULL,&R);
    KSPBuildSolution(solver,PETSC_NULL,&X);
    /* stuff */
    if (sometest) *reason = 15;
    else *reason = KSP_CONVERGED_ITERATING;
    PetscFunctionReturn(0);
}
92. Advanced options

Many options for the (mathematically) sophisticated user
some specific to one method

- KSPSetInitialGuessNonzero
- KSPGMRESSetRestart
- KSPSetPreconditionerSide
- KSPSetNormType

Many options easier through commandline.
93. Null spaces

Iterating orthogonal to the null space of the operator:

```c
MatNullSpace sp;
MatNullSpaceCreate /* constant vector */
    (PETSC_COMM_WORLD,PETSC_TRUE,0,PETSC_NULL,&sp);
MatNullSpaceCreate /* general vectors */
    (PETSC_COMM_WORLD,PETSC_FALSE,5,vecs,&sp);
MatSetNullSpace(mat,sp);
```

The solver will now properly remove the null space at each iteration.
Shell matrix requires shell preconditioner in `KSPSetOperators`:

```
PCSetType(pc, PCSHELL);
PCShellSetContext(PC pc, void *ctx);
PCShellGetContext(PC pc, void **ctx);
PCShellSetApply(PC pc,
                PetscErrorCode (*apply)(void*, Vec, Vec));
PCShellSetSetUp(PC pc,
                PetscErrorCode (*setup)(void*))
```

similar idea to shell matrices

Alternative: use different operator for preconditioner
95. Fieldsplit preconditioners

If a problem contains multiple physics, separate preconditioning can make sense.

Matrix block storage: \texttt{MatCreateNest}

$$
\begin{pmatrix}
A_{00} & A_{01} & A_{02} \\
A_{10} & A_{11} & A_{12} \\
A_{20} & A_{21} & A_{22}
\end{pmatrix}
$$

However, it makes more sense to interleave these fields.
96. Fieldsplit use

Easy case: all fields are the same size

\[
\begin{align*}
\text{PCSetType}(prec, \text{PCFIELDSPLIT}); \\
\text{PCFieldSplitSetBlockSize}(prec, 3); \\
\text{PCFieldSplitSetType}(prec, \text{PC\_COMPOSITE\_ADDITIVE});
\end{align*}
\]

Subpreconditioners can be specified in code, but easier with options:

\[
\begin{align*}
\text{PetscOptionsSetValue} \\
\quad ("-fieldsplit_0_pc_type","lu"); \\
\text{PetscOptionsSetValue} \\
\quad ("-fieldsplit_0_pc_factor_mat_solver_package","mumps");
\end{align*}
\]

Fields can be named instead of numbered.
Non-strided, arbitrary fields: \textbf{PCFieldSplitSetIS()}

Stokes equation can be detected: \texttt{-pc_fieldsplit_detect_saddle_point}

Combining fields multiplicatively: solve

\[
\begin{pmatrix}
  I & A_{10}A_{00}^{-1} \\
  A_{10} & I
\end{pmatrix}
\begin{pmatrix}
  A_{00} & A_{01} \\
  A_{01} & A_{11}
\end{pmatrix}
\]

If there are just two fields, they can be combined by Schur complement

\[
\begin{pmatrix}
  I & A_{10}A_{00}^{-1} \\
  A_{10} & I
\end{pmatrix}
\begin{pmatrix}
  A_{00} & A_{01} \\
  A_{11} - A_{10}A_{00}^{-1}A_{01}
\end{pmatrix}
\]
Fieldsplit example

```c
KSPGetPC(solver,&prec);
PCSetType(prec,PCFIELDSPLIT);
PCFieldSplitSetBlockSize(prec,2);
PCFieldSplitSetType(prec,PC_COMPOSITE_ADDITIVE);
PetscOptionsSetValue(  
    "-fieldsplit_0_pc_type","lu");  
PetscOptionsSetValue(  
    "-fieldsplit_0_pc_factor_mat_solver_package","mumps");  
PetscOptionsSetValue(  
    "-fieldsplit_1_pc_type","lu");  
PetscOptionsSetValue(  
    "-fieldsplit_1_pc_factor_mat_solver_package","mumps");
```
Global preconditioners: MG

```c
PCSetType(PC pc, PCMG);
PCMGSetLevels(pc, int levels, MPI Comm *comms);
PCMGSetType(PC pc, PCMGType mode);
PCMGSetCycleType(PC pc, PCMGCycleType ctype);
PCMGSetNumberSmoothUp(PC pc, int m);
PCMGSetNumberSmoothDown(PC pc, int n);
PCMGGetCoarseSolve(PC pc, KSP *ksp);
PCMGSetInterpolation(PC pc, int level, Mat P); and
PCMGSetRestriction(PC pc, int level, Mat R);
PCMGSetResidual(PC pc, int level, PetscErrorCode (*residual)(Mat, Vec, Vec, Vec), Mat mat);
```
Hypre is a package like PETSc
- selling point: fancy preconditioners
- **Install with** `--with-hypre=yes --download-hypre=yes`
- **then use** `-pc_type hypre -pc_hypre_type parasails/boomeramg/euclid/pilut`
101. Direct methods

- Iterative method with direct solver as preconditioner would converge in one step
- Direct methods in PETSc implemented as special iterative method: `KSPPREONLY` only apply preconditioner
- All direct methods are preconditioner type `PCLU`:

```
myprog -pc_type lu -ksp_type preonly -pc_factor_mat_solver_package mumps
```
Table of Contents

1 Introduction

2 SPMD parallelism

3 Getting started

4 Vec datatype: vectors

5 Mat Datatype: matrix

6 KSP & PC: Iterative solvers

7 Grid manipulation

8 IS and VecScatter: irregular grids

9 SNES: Nonlinear solvers

10 TS: Time stepping

11 Profiling, debugging
102. Regular grid: DMDA

DMDAs are for storing vector field, not matrix.

Support for different stencil types:

- Star stencil
- Box stencil
103. Ghost regions around processors

A DMDA defines a global vector, which contains the elements of the grid, and a local vector for each processor which has space for "ghost points".
104. DMDA construction

\[
\text{DMDACreate2d}(\text{comm, } bndx, bndy, \text{ type, } M, N, m, n, \\
dof, s, lm[], \text{ ln[]}, \text{ DMDA *da})
\]

*bndx, bndy* boundary behaviour: none/ghost/periodic

type: Specifies stencil

DMDA_STENCIL_BOX or DMDA_STENCIL_STAR

M/N: Number of grid points in x/y-direction

m/n: Number of processes in x/y-direction

dof: Degrees of freedom per node

s: The stencil width (for instance, 1 for 2D five-point stencil)

lm/n: array of local sizes (optional; Use PETSC_NULL for the default)
// dmrhs.c
DM grid;
 ierr = DMDACreate2d
  ( comm,
    DM_BOUNDARY_NONE,DM_BOUNDARY_NONE,
    DMDA_STENCIL_STAR,
    100,100,
    PETSC_DECIDE,PETSC_DECIDE,
    NULL,NULL,
    &grid
  ); CHKERRQ(ierr);
 ierr =
    DMSetFromOptions(grid);
 CHKERRQ(ierr);
 ierr =
    DMSetUp(grid);
 CHKERRQ(ierr);
 ierr =
    DMViewFromOptions(grid,
    CHKERRQ(ierr);
106. Associated vectors

- Global vector: based on grid partitioning.
- Local vector: including halo regions

```c
Vec  ghostvector;
 ierr = DMGetLocalVector(grid,&ghostvector); CHKERRQ(ierr);
 ierr = DMGlobalToLocal(grid,xy,INSERT_VALUES,ghostvector);
    CHKERRQ(ierr);
PetscReal **xyarray,**gh;
 ierr = DMDAVecGetArray(grid,xy,&xyarray); CHKERRQ(ierr);
 ierr = DMDAVecGetArray(grid,ghostvector,&gh); CHKERRQ(ierr);
// computation on the arrays
 ierr = DMDAVecRestoreArray(grid,xy,&xyarray); CHKERRQ(ierr);
 ierr = DMDAVecRestoreArray(grid,ghostvector,&gh);
    CHKERRQ(ierr);
 ierr = DMLocalToGlobal(grid,ghostvector,INSERT_VALUES,xy);
    CHKERRQ(ierr);
 ierr = DMRestoreLocalVector(grid,&ghostvector); CHKERRQ(ierr);
```
typedef struct {
  PetscInt  dim, dof, sw;
  PetscInt  mx, my, mz;  /* grid points in x,y,z */
  PetscInt  xs, ys, zs;  /* starting point, excluding ghosts */
  PetscInt  xm, ym, zm;  /* grid points, excluding ghosts */
  PetscInt  gxs, gys, gzs;  /* starting point, including ghosts */
  PetscInt  gxm, gym, gzm;  /* grid points, including ghosts */
  DMBoundaryType  bx, by, bz;  /* type of ghost nodes */
  DMDAStencilType  st;
  DM  da;
} DMDALocalInfo;
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.;
    }
}
109. Associated matrix

Matrix that has knowledge of the grid:

```c
DMSetUp(DM grid);
DMCreateMatrix(DM grid, Mat *J)
```

Set matrix values based on stencil:

```c
MatSetValuesStencil(Mat mat,
  PetscInt m, const MatStencil idxm[],
  PetscInt n, const MatStencil idxn[],
  const PetscScalar v[], InsertMode addv)
```

(ordering of row/col variables too complicated for `MatSetValues`)
110. Set values by stencil

// 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;
        row.j = j; row.i = i;
        /*** 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);
        CHKERRQ(ierr);
    }
}
111. DMPLex

Support for unstructured grids and node/edge/cell relations.

This is complicated and under-documented.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>SPMD parallelism</td>
</tr>
<tr>
<td>3</td>
<td>Getting started</td>
</tr>
<tr>
<td>4</td>
<td>Vec datatype: vectors</td>
</tr>
<tr>
<td>5</td>
<td>Mat Datatype: matrix</td>
</tr>
<tr>
<td>6</td>
<td>KSP &amp; PC: Iterative solvers</td>
</tr>
<tr>
<td>7</td>
<td>Grid manipulation</td>
</tr>
<tr>
<td>8</td>
<td>IS and VecScatter: irregular grids</td>
</tr>
<tr>
<td>9</td>
<td>SNES: Nonlinear solvers</td>
</tr>
<tr>
<td>10</td>
<td>TS: Time stepping</td>
</tr>
<tr>
<td>11</td>
<td>Profiling, debugging</td>
</tr>
</tbody>
</table>
112. Irregular data movement

Example: collect distributed boundary onto a single processor (this happens in the matrix-vector product)

Problem: figuring out communication is hard, actual communication is cheap
113. VecScatter

Preprocessing: determine mapping between input vector and output:

```c
VecScatterCreate(Vec, IS, Vec, IS, VecScatter*)
// also Destroy
```

Application to specific vectors:

```c
VecScatterBegin(VecScatter, 
    Vec, Vec, InsertMode mode, ScatterMode direction)
VecScatterEnd (VecScatter, 
    Vec, Vec, InsertMode mode, ScatterMode direction)
```
114. IS: index set

Index Set is a set of indices

```c
ISCreateGeneral(comm, n, indices, PETSC_COPY_VALUES, &is);
    /* indices can now be freed */
ISCreateStride (comm, n, first, step, &is);
ISCreateBlock   (comm, bs, n, indices, &is);

ISDestroy(is);
```

Use `MPI_COMM_SELF` most of the time.

Various manipulations: `ISSum`, `ISDifference`, `ISInvertPermutations` et cetera.
Also `ISGetIndices` / `ISRestoreIndices` / `ISGetSize`
115. Example: split odd and even

Input:

Process [0]
0.
1.
2.
3.
4.
5.

Process [1]
6.
7.
8.
9.
10.
11.

Output:

Process [0]
0.
2.
4.
6.
8.
10.

Process [1]
1.
3.
5.
7.
9.
11.
116. index sets for this example

```c
// oddeven.c
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);
}
```
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);
Exercise 9 (odd\textit{even})

Now alter the \texttt{is} objects so that the output becomes:

<table>
<thead>
<tr>
<th>Process [0]</th>
<th>Process [1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.</td>
<td>11.</td>
</tr>
<tr>
<td>8.</td>
<td>9.</td>
</tr>
<tr>
<td>6.</td>
<td>7.</td>
</tr>
<tr>
<td>4.</td>
<td>5.</td>
</tr>
<tr>
<td>2.</td>
<td>3.</td>
</tr>
<tr>
<td>0.</td>
<td>1.</td>
</tr>
</tbody>
</table>
/ * create the distributed vector with one element per processor */
ierr = VecCreate(MPI_COMM_WORLD,&global);
ierr = VecSetType(global,VECMPI);
ierr = VecSetSizes(global,1,PETSC_DECIDE);

/* create the local copy */
ierr = VecCreate(MPI_COMM_SELF,&local);
ierr = VecSetType(local,VECSEQ);
ierr = VecSetSizes(local,ntids,ntids);
IS indices;
ierr = ISCreateStride(MPI_COMM_SELF, ntids, 0, 1, &indices);
/* create a scatter from the source indices to target */
ierr = VecScatterCreate(global, indices, local, indices, &scatter);
/* index set is no longer needed */
ierr = ISDestroy(&indices);
// oddeven.c
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);
}
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);
Table of Contents

1 Introduction
2 SPMD parallelism
3 Getting started
4 Vec datatype: vectors
5 Mat Datatype: matrix
6 KSP & PC: Iterative solvers
7 Grid manipulation
8 IS and VecScatter: irregular grids
9 SNES: Nonlinear solvers
10 TS: Time stepping
11 Profiling, debugging
122. Nonlinear problems

Basic equation

\[ f(u) = 0 \]

where \( u \) can be big, for instance nonlinear PDE.

Typical solution method:

\[ u_{n+1} = u_n - J(u_n)^{-1}f(u_n) \]

Newton iteration.

Needed: function and Jacobian.
123. Basic SNES usage

User supplies function and Jacobian:

```c
SNES snes;

SNESCreate(PETSC_COMM_WORLD,&snes)
SNESSetType(snes,type);
SNESSetFromOptions(snes);
SNESDestroy(SNES snes);
```

where type:

- `SNESLS` Newton with line search
- `SNESTR` Newton with trust region
- several specialized ones
PetscErrorCode (*FunctionEvaluation)(SNES, Vec, Vec, void*);
VecCreate(PETSC_COMM_WORLD, &r);
SNESSetFunction(snes, r, FunctionEvaluation, *ctx);
PetscErrorCode (*FormJacobian)(SNES, Vec, Mat, Mat, void*);
MatCreate(PETSC_COMM_WORLD, &J);
SNESSetJacobian(snes, J, J, FormJacobian, *ctx);
126. SNES solution

```c
SNESolve(snes, /* rhs= */ PETSC_NULL, x)
SNESGetConvergedReason(snes, &reason)
SNESGetIterationNumber(snes, &its)
```
127. Example: two-variable problem

Define a context

typedef struct {
    Vec xloc,rloc; VecScatter scatter; } AppCtx;

/* User context */
AppCtx user;

/* Work vectors in the user context */
VecCreateSeq(PETSC_COMM_SELF,2,&user.xloc);
VecDuplicate(user.xloc,&user.rloc);

/* Create the scatter between the global and local x */
ISCreateStride(MPI_COMM_SELF,2,0,1,&idx);
VecScatterCreate(x,idx,user.xloc,idx,&user.scatter);
PetscErrorCode FormFunction

(SNES snes, Vec x, Vec f, void *ctx)

{
    VecScatterBegin(user->scatter,
        x, user->xloc, INSERT_VALUES, SCATTER_FORWARD); // & End
    VecGetArray(xloc, &xx); CHKERRQ(ierr);
    VecSetValue
        (f, 0, /* something with xx[0] */ & xx[1] */,
            INSERT_VALUES);
    VecRestoreArray(x, &xx);
    PetscFunctionReturn(0);
}
129. Jacobian calculation through finite differences

Jacobian calculation is difficult. It can be approximated through finite differences:

\[ J(u)v \approx \frac{f(u + hv) - f(u)}{h} \]

MatCreateSNESMF(snes,&J);
SNESSetJacobian(snes,J,J,MatMFFDComputeJacobian,(void*)&user);
130. Further possibilities

```c
SNESSetTolerances
(SNES snes, double atol, double rtol, double stol,
int its, int fcts);
```

convergence test and monitoring, specific options for line search and trust region

adaptive convergence: `-snes_ksp_ew_conv` *(Eisenstat Walker)*
131. Solve customization

```c
SNESSetType(snes,SNESTR); /* newton with trust region */
SNESGetKSP(snes,&ksp)
KSPGetPC(ksp,&pc)
PCSetType(pc,PCNONE)
KSPSetTolerances(ksp,1.e-4,PETSC_DEFAULT,PETSC_DEFAULT,20)
```
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>SPMD parallelism</td>
</tr>
<tr>
<td>3</td>
<td>Getting started</td>
</tr>
<tr>
<td>4</td>
<td>Vec datatype: vectors</td>
</tr>
<tr>
<td>5</td>
<td>Mat Datatype: matrix</td>
</tr>
<tr>
<td>6</td>
<td>KSP &amp; PC: Iterative solvers</td>
</tr>
<tr>
<td>7</td>
<td>Grid manipulation</td>
</tr>
<tr>
<td>8</td>
<td>IS and VecScatter: irregular grids</td>
</tr>
<tr>
<td>9</td>
<td>SNES: Nonlinear solvers</td>
</tr>
<tr>
<td>10</td>
<td>TS: Time stepping</td>
</tr>
<tr>
<td>11</td>
<td>Profiling, debugging</td>
</tr>
</tbody>
</table>
132. Basic profiling

- **-log_summary** flop counts and timings of all PETSc events
- **-info** all sorts of information, in particular
  
  ```bash
  mpirun -info | grep malloc
  ```
  
  ```c
  [0] MatAssemblyEnd_SeqAIJ():
      Number of mallocs during MatSetValues() is 0
  ```
- **-log_trace** start and end of all events: good for hanging code
<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec):</td>
<td>5.493e-01</td>
<td>1.00006</td>
<td>5.493e-01</td>
<td></td>
</tr>
<tr>
<td>Objects:</td>
<td>2.900e+01</td>
<td>1.00000</td>
<td>2.900e+01</td>
<td></td>
</tr>
<tr>
<td>Flops:</td>
<td>1.373e+07</td>
<td>1.00000</td>
<td>1.373e+07</td>
<td>2.746e+07</td>
</tr>
<tr>
<td>Flops/sec:</td>
<td>2.499e+07</td>
<td>1.00006</td>
<td>2.499e+07</td>
<td>4.998e+07</td>
</tr>
<tr>
<td>Memory:</td>
<td>1.936e+06</td>
<td>1.00000</td>
<td></td>
<td>3.871e+06</td>
</tr>
<tr>
<td>MPI Messages:</td>
<td>1.040e+02</td>
<td>1.00000</td>
<td>1.040e+02</td>
<td>2.080e+02</td>
</tr>
<tr>
<td>MPI Msg Lengths:</td>
<td>4.772e+05</td>
<td>1.00000</td>
<td>4.588e+03</td>
<td>9.544e+05</td>
</tr>
<tr>
<td>MPI Reductions:</td>
<td>1.450e+02</td>
<td>1.00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Max Ratio</td>
<td>Max Ratio</td>
<td>Avg len</td>
<td>%T</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------</td>
<td>-----------</td>
<td>---------</td>
<td>----</td>
</tr>
<tr>
<td>MatMult</td>
<td>100</td>
<td>1.0</td>
<td>3.4934e-02</td>
<td>1.0</td>
</tr>
<tr>
<td>MatSolve</td>
<td>101</td>
<td>1.0</td>
<td>2.9381e-02</td>
<td>1.0</td>
</tr>
<tr>
<td>MatLUFactorNum</td>
<td>1</td>
<td>1.0</td>
<td>2.0621e-03</td>
<td>1.0</td>
</tr>
<tr>
<td>MatAssemblyBegin</td>
<td>1</td>
<td>1.0</td>
<td>2.8350e-03</td>
<td>1.1</td>
</tr>
<tr>
<td>MatAssemblyEnd</td>
<td>1</td>
<td>1.0</td>
<td>8.8258e-03</td>
<td>1.0</td>
</tr>
<tr>
<td>VecDot</td>
<td>101</td>
<td>1.0</td>
<td>8.3244e-03</td>
<td>1.2</td>
</tr>
<tr>
<td>KSPSetup</td>
<td>2</td>
<td>1.0</td>
<td>1.9123e-02</td>
<td>1.0</td>
</tr>
<tr>
<td>KSPSolve</td>
<td>1</td>
<td>1.0</td>
<td>1.4158e-01</td>
<td>1.0</td>
</tr>
</tbody>
</table>
135. User events

#include "petsclog.h"

int USER EVENT;

PetscLogEventRegister(&USER EVENT,"User event name",0);

PetscLogEventBegin(USER EVENT,0,0,0,0);

/* application code segment to monitor */

PetscLogFlops(number of flops for this code segment);

PetscLogEventEnd(USER EVENT,0,0,0,0);
136. Program stages

```c
PetscLogStagePush(int stage); /* 0 <= stage <= 9 */
PetscLogStagePop();
PetscLogStageRegister(int stage, char *name)
```
137. Debugging

- Use of `{CHKERRQ, SETERRQ}` for catching and generating error
- Use of `{PetscMalloc, PetscFree}` to catch memory problems; `{CHKMEMQ}` for instantaneous memory test (debug mode only)
- Better than `{PetscMalloc}: PetscMalloc1` aligned to `{PETSC_MEMALIGN}`