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
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
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

Level of Abstraction

Application Codes

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

Matrices
Vectors
Index Sets

BLAS
MPI
### 6. Hierarchy of tools

#### Parallel Numerical Components of PETSc

<table>
<thead>
<tr>
<th><strong>Nonlinear Solvers</strong></th>
<th><strong>Time Steppers</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Newton–based Methods</td>
<td>- Euler</td>
</tr>
<tr>
<td>- Line Search</td>
<td>- Backward Euler</td>
</tr>
<tr>
<td>- Trust Region</td>
<td>- Pseudo–Time Stepping</td>
</tr>
<tr>
<td>- Other</td>
<td>- Other</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Krylov Subspace Methods</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- GMRES</td>
</tr>
<tr>
<td>- CG</td>
</tr>
<tr>
<td>- CGS</td>
</tr>
<tr>
<td>- Bi–CG–Stab</td>
</tr>
<tr>
<td>- TFQMR</td>
</tr>
<tr>
<td>- Richardson</td>
</tr>
<tr>
<td>- Chebychev</td>
</tr>
<tr>
<td>- Other</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Preconditioners</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Additive Schwarz</td>
</tr>
<tr>
<td>- Block Jacobi</td>
</tr>
<tr>
<td>- Jacobi</td>
</tr>
<tr>
<td>- ILU</td>
</tr>
<tr>
<td>- ICC</td>
</tr>
<tr>
<td>- LU (sequential only)</td>
</tr>
<tr>
<td>- Other</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Matrices</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Compressed Sparse Row (AIJ)</td>
</tr>
<tr>
<td>- Block Compressed Sparse Row (BAIJ)</td>
</tr>
<tr>
<td>- Block Diagonal (BDiag)</td>
</tr>
<tr>
<td>- Dense</td>
</tr>
<tr>
<td>- Other</td>
</tr>
</tbody>
</table>

| **Vectors** |

<table>
<thead>
<tr>
<th><strong>Index Sets</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Indices</td>
</tr>
<tr>
<td>- Block Indices</td>
</tr>
<tr>
<td>- Stride</td>
</tr>
<tr>
<td>- Other</td>
</tr>
</tbody>
</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
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:

```c
MATMult(A, x, y);  // y <- A x
```

same for sequential, parallel, dense, sparse, FFT
11. Data hiding

To support this uniform interface, the implementation is hidden:

```c
MatSetValue(A, i, j, v, 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.)
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>
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.
MPI compilers are usually called \texttt{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:
\begin{verbatim}
ibrun yourprog
\end{verbatim}
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"); 
    ..... 
}
```

proc1 proc2 proc3 proc4
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**
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**
Include file for preprocessor definitions, module for library definitions

```
program basic
#include <petsc/finclude/petsc.h>
use petsc
implicit none
```
24. Variable declarations, F

\begin{tabular}{ll}
KSP & :: solver \\
Mat & :: A \\
Vec & :: x, y \\
PetscInt & :: j(3) \\
PetscScalar & :: mv \\
PetscReal & :: nrm \\
\end{tabular}

Much like in C
call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
    CHKERRQ(ierr)
// all the petsc work
    call PetscFinalize(ierr); CHKERRQ(ierr)

Error code is now final parameter. This holds for every PETSc routine.
26. A word about datatypes

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

Similarly, \texttt{PetscInt} is 32/64 bit depending.

Other scalar data types: \texttt{PetscBool, PetscErrorCode}

\textit{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:

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

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

Production:

```
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 `PetscPrintf` and print a message ‘This program runs on 27 processors’ from process zero.

- Start with the template code `hello.c/hello.F`
- (or see slide 22)
- Compile with `make hello`
- Part two: use `PetscSynchronizedPrintf`
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
Prototype

```fortran
Subroutine VecCreate
  ( comm, v, ierr )

Type(MPI_Comm) :: comm
Vec :: v
PetscErrorCode :: ierr
```

Use:

```fortran
Type(MPI_Comm) :: &
  comm = MPI_COMM_WORLD
Vec :: v
PetscErrorCode :: ierr

call VecCreate(comm, v, ierr)
```

- Final parameter always error parameter. Do not forget!
- MPI types are of often `Type(MPI_Comm)` and such,
- PETSc datatypes are handled through the preprocessor.
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
30. 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);
}
```
31. Example: error traceback

PETSC ERROR: We cannot go on like this
PETSC ERROR: See https://www.mcs.anl.gov/petsc/documentation/faq.html for
PETSC ERROR: backtrace on a [computer name]
PETSC ERROR: Configure options [all options]
PETSC ERROR: #1 this_function_bombs() line 20 in backtrace.c
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);
```

```c
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
32. Commandline argument, F

```fortran
character*80    msg
call PetscOptionsGetInt(
    PETSC_NULL_OPTIONS, PETSC_NULL_CHARACTER, &
    "-n", n,PETSC_NULL_BOOL, ierr)
write(msg,10) n
10 format("Input parameter:",i5)
call PetscPrintf(PETSC_COMM_WORLD,msg,ierr)
```

Less elegant than `PetscPrintf` in C

Note the `PETSC_NULL_XXX`: Fortran has strict type checking.
<table>
<thead>
<tr>
<th></th>
<th>Section</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 <code>datatype</code>: vectors</td>
</tr>
<tr>
<td>5</td>
<td>Mat <code>Datatype</code>: 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>
33. Create calls

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);
```
34. Create calls, Fortran

! Fortran

Vec V

call VecCreate(MPI_COMM_WORLD, V, e)
call VecDestroy(V, e)

Note: in Fortran there are no ‘star’ arguments
35. More about vectors

A vector is a vector of \texttt{PetscScalar}s: there are no vectors of integers (see the \texttt{IS} datatype later)

The vector object is not completely created in one call:

\begin{verbatim}
VecSetType(V, VECMPI) // or VECSEQ
VecSetSizes(Vec v, int m, int M);
\end{verbatim}

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

\begin{verbatim}
VecDuplicate(Vec v, Vec *w);
\end{verbatim}
36. Parallel layout up to PETSc

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

Local size can be specified as `PETSC_DECIDE`.

- \( \text{VecSetSizes}(V,\text{PETSC_DECIDE},8) \)
- \( \text{VecSetSizes}(V,\text{PETSC_DECIDE},8) \)
- \( \text{VecSetSizes}(V,\text{PETSC_DECIDE},8) \)
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 PETSC_DECIDE.

- VecSetSizes(V,2,5)
- VecSetSizes(V,3,5)
- VecSetSizes(V,2,PETSC_DECIDE)
- VecSetSizes(V,3,PETSC_DECIDE)
38. Query parallel layout

Query vector layout:

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

On 2nd processor:

\[
\begin{align*}
\text{low} & = 3 \\
\text{high} & = 6
\end{align*}
\]
Query general layout:

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

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

Set vector to constant value:

```
VecSet(Vec x, PetscScalar value);
```

Set individual elements (global indexing!):

```
VecSetValue
    (Vec x, int row, PetscScalar value,
     InsertMode mode);
```

```
i = 1; v = 3.14;
VecSetValue(x, i, v, INSERT_VALUES);
```

```
call VecSetValue(x, i, v, INSERT_VALUES)
```

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

Set individual elements (global indexing!):

```c
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:

```
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);
```
Create a vector where the values are a single sine wave using \texttt{VecGetSize}, \texttt{VecGetLocalSize}, \texttt{VecGetOwnershipRange}. Quick visual inspection:

\texttt{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:

\[
\begin{align*}
  p & \leftarrow x^t y \\
  x & \leftarrow x/p \\
  n & \leftarrow \|x\|_2
\end{align*}
\]
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)
```
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.
PetscScalar, pointer :: xx_v(:)

....
call VecGetArrayF90(x, xx_v, ierr)
a = xx_v(3)
call VecRestoreArrayF90(x, xx_v, ierr)

More separate F90 versions for ‘Get’ routines
there are some ugly hacks for F77
# 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**
50. Matrix creation

The usual create/destroy calls:

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

Several more aspects to creation:

```c
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)
51. 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...
52. 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
53. Sequential matrix structure

```
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
54. Parallel matrix structure

Off-diagonal block has off-processor connections

Diagonal block has on-processor connections
55. (why does it do this?)

- $y \leftarrow Ax_A + Bx_b$
- $x_B$ needs to be communicated; $Ax_A$ can be computed in the meantime
- Algorithm
  - Initiate asynchronous sends/receives for $x_b$
  - compute $Ax_A$
  - make sure $x_b$ is in
  - compute $Bx_B$
- 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.
56. 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

```c
(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
MatCreateSeqAIJ(MPI_Comm comm, PetscInt m, PetscInt n, PetscInt nz, const PetscInt nz[], 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)
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`
59. 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 INSERT_VALUES, ADD_VALUES

Set block of values:

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

(\(v\) is row-oriented)
60. Set only one element

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

Special case of the general case:

\[ \text{MatSetValues}(A, 1, &i, 1, &j, &v, \text{INSERT\_VALUES}); \]
61. 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.)
• 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.
63. Getting values (F)

\[ \text{MatGetRow}(A, \text{row}, \text{ncols}, \text{cols}, \text{vals}, \text{ierr}) \]
\[ \text{MatRestoreRow}(A, \text{row}, \text{ncols}, \text{cols}, \text{vals}, \text{ierr}) \]

where \( \text{cols}(\text{maxcols}), \text{vals}(\text{maxcols}) \) are long enough arrays (allocated by the user)
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 \texttt{#if 0 and #endif}.)

64. 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
65. 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
ierr = MatCreate(comm, &A); CHKERRQ(ierr);
#ifdef PETSC_HAVE_CUDA
ierr = MatSetType(A, MATMPIAIJCUSPARSE); CHKERRQ(ierr);
#else
ierr = MatSetType(A, MATMPIAIJ); CHKERRQ(ierr);
#endif
67. Matrix operations

Main operations are matrix-vector:

\[ \text{MatMult}(\text{Mat } A, \text{Vec } \text{in}, \text{Vec } \text{out}); \]
\[ \text{MatMultAdd} \]
\[ \text{MatMultTranspose} \]
\[ \text{MatMultTransposeAdd} \]

Simple operations on matrices:

\[ \text{MatNorm} \]
\[ \text{MatScale} \]
\[ \text{MatDiagonalScale} \]
Some matrix-matrix operations

\begin{itemize}
\item \texttt{MatMatMult(Mat, Mat, MatReuse, PetscReal, Mat*)};
\item \texttt{MatPtAP(Mat, Mat, MatReuse, PetscReal, Mat*)};
\item \texttt{MatMatMultTranspose(Mat, Mat, MatReuse, PetscReal, Mat*)};
\item \texttt{MatAXPY(Mat, PetscScalar, Mat, MatStructure)};
\end{itemize}
69. Matrix viewers

```c
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)
....

(Fortran: `PETSC_NULL_INTEGER`)

- also invoked by `mat_view`
- Sparse: only allocated positions listed
- other viewers: for instance `mat_view_draw` (X terminal)
70. 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, PETSCVIEWERVTk );
MatView( A, fd );
PetscViewerDestroy(fd);
```
71. 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
72. Shell matrix context

Shell matrices need custom data

\[
\text{MatShellSetContext} (\text{Mat } \text{mat}, \text{void } *\text{ctx}); \\
\text{MatShellGetContext} (\text{Mat } \text{mat}, \text{void } **\text{ctx});
\]

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

User program sets context, matmult routine accesses it
...  

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

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

Extract one parallel submatrix:

```c
MatGetSubMatrix(mat, isrow, iscol, csize, MatReuse cll, newmat)
```

Extract multiple single-processor matrices:

```c
MatGetSubMatrices(mat, n, irow[], icol[], scall, submat[])
```

Collective call, but different index sets per processor
75. 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
76. 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 $B \approx A^{-1}$
77. 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
78. 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)
79. 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(solver, &reason)}:
  positive is convergence, negative divergence
- \texttt{KSPConvergedReasons[reason]} is string
- \texttt{KSPGetIterationNumber(solver, &nits)}: after how many iterations did the method stop?
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
83. 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`
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

**KSPSetOperators** (*solver*, *A*, *B*)

- *B* is basis for preconditioner, need not be *A*
- if *A* or *B* is to be reused, use *NULL*
85. 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
86. 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.
87. 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:

\[ \forall i,j : a_{ij} \leftarrow a_{ij} - a_{ik} a_{kk}^{-1} a_{kj} \]

Inexact:

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

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:

```cpp
PCFactorSetLevels(PC pc, int levels);
```

- `pc_factor_levels <levels>`

Prevent indefinite preconditioners:

```cpp
PCFactorSetShiftType(PC pc, MatFactorShiftType type);
```

value `MAT_SHIFT_POSITIVE_DEFINITE` et cetera

Factorization preconditioners are sequential but still useful; see later
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
90. 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)

```
PCASMSetOverlap(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 \texttt{mymatmult} is declared, which is attached by \texttt{MatShellSetOperation} to the matrix $A$ as the means of computing the product $\texttt{MatMult}(A, \texttt{in}, \texttt{out})$, 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 \texttt{mymatmult} computes the product $y \leftarrow A^tAx$.

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

The code uses a preconditioner \texttt{PCNONE}. What happens if you run it with option \texttt{-pc_type jacobi}?
91. Monitors and convergence tests

\[ \text{KSPSetTolerances}(\text{solver}, \text{rtol}, \text{atol}, \text{dtol}, \text{maxit}); \]

Monitors can be set in code, but simple cases:

- \text{-ksp\_monitor}
- \text{-ksp\_monitor\_true\_residual}
92. 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*))
```
93. Example of convergence tests

```c
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);
}
```
94. Advanced options

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

```
KSPSetInitialGuessNonzero
KSPGMRESSetRestart
KSPSetPreconditionerSide
KSPSetNormType
```

Many options easier through commandline.
Iterating orthogonal to the null space of the operator:

\[
\text{MatNullSpace } sp; \\
\text{MatNullSpaceCreate} /* constant vector */ \\
(PETSC_COMM_WORLD, PETSC_TRUE, 0, PETSC_NULL, &sp); \\
\text{MatNullSpaceCreate} /* general vectors */ \\
(PETSC_COMM_WORLD, PETSC_FALSE, 5, vecs, &sp); \\
\text{MatSetNullSpace}(mat, sp);
\]

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

\begin{verbatim}
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*))
\end{verbatim}

similar idea to shell matrices

Alternative: use different operator for preconditioner
97. 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
98. Fieldsplit use

Easy case: all fields are the same size

```c
PCSetType(prec, PCFIELDSPLIT);
PCFieldSplitSetBlockSize(prec, 3);
PCFieldSplitSetType(prec, PC_COMPOSITE_ADDITIVE);
```

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

```c
PetscOptionsSetValue
   ("-fieldsplit_0_pc_type", "lu");
PetscOptionsSetValue
   ("-fieldsplit_0_pc_factor_mat_solver_package", "mumps");
```

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

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

Combining fields multiplicatively: solve

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

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

\[
\begin{pmatrix}
I & A_{10}A^{-1}_{00} \\
A_{10}A^{-1}_{00} & I
\end{pmatrix}
\begin{pmatrix}
A_{00} & A_{01} \\
A_{11} - A_{10}A^{-1}_{00}A_{01}
\end{pmatrix}
\]
```c
KSPGetPC(solver,&prec);
PCTYPE(prec,PCFIELDSPLIT);
PCFieldSplitSetBlockSize(prec,2);
PCTYPE(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");
```
101. 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);
```
102. Global preconditioners: Hypre

- 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`
• 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
104. Regular grid: DMDA

DMDAs are for storing vector field, not matrix.

Support for different stencil types:

Star stencil

Box stencil
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".
106. DMDA construction

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

**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)
107. Grid info

```c
// dmrhs.c
DM grid;
ierr = DMDACreate2d (comm,
  DM_BOUNDARY_NONE, DM_BOUNDARY_NONE,
  DMDA_STENCIL_STAR, 100, 100,
  PETSC_DECIDE, PETSC_DECIDE,
  1, 1,
  NULL, NULL, &grid);
CHKERRQ(ierr);
ierr = DMSetFromOptions(grid);
CHKERRQ(ierr);
ierr = DMSetUp(grid);
CHKERRQ(ierr);
ierr = DMViewFromOptions(grid, NULL, "-dm_view");
CHKERRQ(ierr);
```
108. Associated vectors

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

```c
Vec  ghostvector;
int ierr = DMGetLocalVector(grid,&ghostvector); CHKERRQ(ierr);
ierr = DMGlobalToLocal(grid,xy,INSERT_VALUES,ghostvector);
  CHKERRQ(ierr);
PetscReal **xyarray,**gh;
int 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.xls && info.gys<info.xls)
            if (i-1>=info.xs && i+1<=info.xs+info.xm &&
                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.;
    }
}
111. Associated matrix

Matrix that has knowledge of the grid:

\[ \text{DMSSetUp}(\text{DM } \text{grid}); \]
\[ \text{DMCreateMatrix}(\text{DM grid,Mat } *J) \]

Set matrix values based on stencil:

\[ \text{MatSetValuesStencil}(\text{Mat mat,} \]
\[ \quad \text{PetscInt } m,\text{const MatStencil } idxm[], \]
\[ \quad \text{PetscInt } n,\text{const MatStencil } idxn[], \]
\[ \quad \text{const PetscScalar } v[],\text{InsertMode addv}) \]

(ordering of row/col variables too complicated for \text{MatSetValues})
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 =
        \[ \textbf{MatSetValuesStencil}(A, 1, &row, ncols, col, v, \texttt{INSERT_VALUES}); \texttt{CHKERRQ(ierr)} \]
    }
}
113. DMPIlex

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

This is complicated and under-documented.
### 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
114. 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
115. VecScatter

Preprocessing: determine mapping between input vector and output:

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

Application to specific vectors:

VecScatterBegin(VecScatter, Vec, Vec, InsertMode mode, ScatterMode direction)
VecScatterEnd  (VecScatter, Vec, Vec, InsertMode mode, ScatterMode direction)
116. 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`
117. 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.
118. index sets for this example

// oddeven.c
IS oddeven;
if (procid==0) {
  ierr = ISCCreateStride(comm,Nglobal/2,0,2,&oddeven);
  CHKERRQ(ierr);
} else {
  ierr = ISCCreateStride(comm,Nglobal/2,1,2,&oddeven);
  CHKERRQ(ierr);
}
119. scatter for this example

```c
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 (oddeven)

Now alter the `is` objects so that the output becomes:

Process [0]  Process [1]
10. 11.
 8. 9.
 6. 7.
 4. 5.
 2. 3.
 0. 1.
Example: simulate allgather

/* 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);
122. Example: even and odd indices

```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);
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.
125. 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
```c
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);
SNES Solve

SNES Solve: SNESSolve(snes, /* rhs = */ PETSC_NULL, x)
SNES Get Converged Reason: SNESGetConvergedReason(snes, &reason)
SNES Get Iteration Number: SNESGetIterationNumber(snes, &its)
129. 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);
n the user function:

```c
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);
}
```
131. 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);
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)
133. Solve customization

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)
134. Basic profiling

- **-log_summary** flop counts and timings of all PETSc events
- **-info** all sorts of information, in particular

```bash
%% mpiexec yourprogram -info | grep malloc
[0] MatAssemblyEnd_SeqAIJ():
    Number of mallocs during MatSetValues() is 0
```

- **-log_trace** start and end of all events: good for hanging code
## 135. Log summary: overall

<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>1.00000</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>
</tbody>
</table>
### Log summary: details

<table>
<thead>
<tr>
<th>Function</th>
<th>Max Ratio</th>
<th>Ratio Avg</th>
<th>Max Ratio Avg</th>
<th>len</th>
<th>%T</th>
<th>%F</th>
<th>%M</th>
<th>%L</th>
<th>%R</th>
<th>%T</th>
<th>%F</th>
<th>%M</th>
<th>%L</th>
<th>%R</th>
<th>Mflop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatMult</td>
<td>100</td>
<td>1.0</td>
<td>3.4934e-02</td>
<td>1.0</td>
<td>8.0e+02</td>
<td>6 32 96 17 0</td>
<td>6 32 96 17 0</td>
<td>255</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MatSolve</td>
<td>101</td>
<td>1.0</td>
<td>2.9381e-02</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>5 33 0 0 0</td>
<td>5 33 0 0 0</td>
<td>305</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MatLUFactorNum</td>
<td>1</td>
<td>1.0</td>
<td>2.0621e-03</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>43</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MatAssemblyBegin</td>
<td>1</td>
<td>1.0</td>
<td>2.8350e-03</td>
<td>1.1</td>
<td>0.0e+00</td>
<td>0 0 3 83 1</td>
<td>0 0 3 83 1</td>
<td>43</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MatAssemblyEnd</td>
<td>1</td>
<td>1.0</td>
<td>8.8258e-03</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>2 0 1 0 3</td>
<td>2 0 1 0 3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VecDot</td>
<td>101</td>
<td>1.0</td>
<td>8.3244e-03</td>
<td>1.2</td>
<td>0.0e+00</td>
<td>1 7 0 0 35</td>
<td>1 7 0 0 35</td>
<td>243</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KSPSetup</td>
<td>2</td>
<td>1.0</td>
<td>1.9123e-02</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>3 0 0 0 2</td>
<td>3 0 0 0 2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KSPSolve</td>
<td>1</td>
<td>1.0</td>
<td>1.4158e-01</td>
<td>1.0</td>
<td>8.0e+02</td>
<td>26100 96 17 92</td>
<td>26100 96 17 92</td>
<td>194</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
#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);
138. Program stages

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

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