



TEXAS ADVANCED COMPUTING CENTER

WWW.TACC.UTEXAS.EDU



TEXAS

The University of Texas at Austin

# Introduction to the PETSc library

## Victor Eijkhout

eijkhout@tacc.utexas.edu

TACC Training 2022/02/04

# 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

# 1. 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

## 2. More specifically...

Portable Extendable Toolkit for Scientific Computations

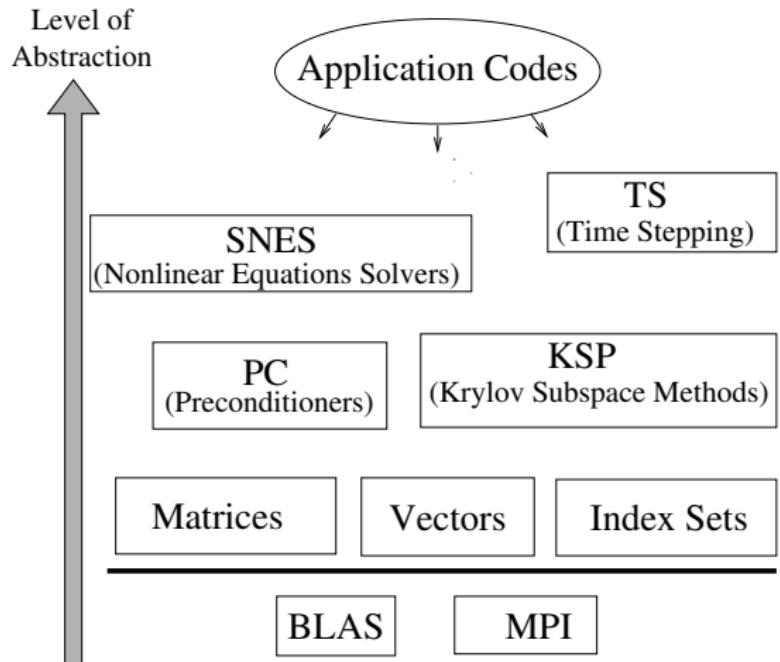
- Scientific Computations: parallel linear algebra, in particular linear and nonlinear solvers
- Toolkit: Contains high level solvers, but also the low level tools to roll your own.
- Portable: Available on many platforms, basically anything that has MPI

Why use it? It's big, powerful, well supported.

### 3. 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

# 4. Structure of a PETSc application



# 5. Hierarchy of tools

## Parallel Numerical Components of PETSc

Nonlinear Solvers	
Newton-based Methods	Other
Line Search	Trust Region

Time Steppers			
Euler	Backward Euler	Pseudo-Time Stepping	Other

Krylov Subspace Methods							
GMRES	CG	CGS	Bi-CG-Stab	TFQMR	Richardson	Chebychev	Other

Preconditioners						
Additive Schwarz	Block Jacobi	Jacobi	ILU	ICC	LU (sequential only)	Other

Matrices				
Compressed Sparse Row (AIJ)	Block Compressed Sparse Row (BAIJ)	Block Diagonal (BDiag)	Dense	Other

Vectors
---------

Index Sets			
Indices	Block Indices	Stride	Other

## 6. Documentation and help

- Web page: <https://petsc.org/>
- Documentation (pdf/html): <https://petsc.org/release/docs/>
- Follow-up to this tutorial: [eijkhout@tacc.utexas.edu](mailto:eijkhout@tacc.utexas.edu)
- PETSc on your local cluster: ask your local support
- General questions about PETSc: [petsc-maint@mcs.anl.gov](mailto: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

# 7. 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)

# 8. 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).*

# 9. Object oriented design

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

Overloading:

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

same for sequential, parallel, dense, sparse, FFT

# 10. Data hiding

To support this uniform interface, the implementation is hidden:

```
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.)

# 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

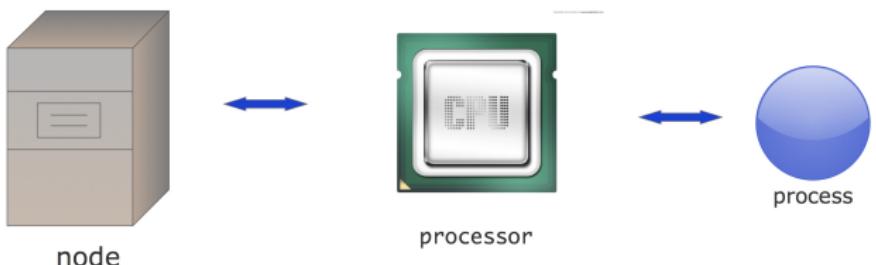
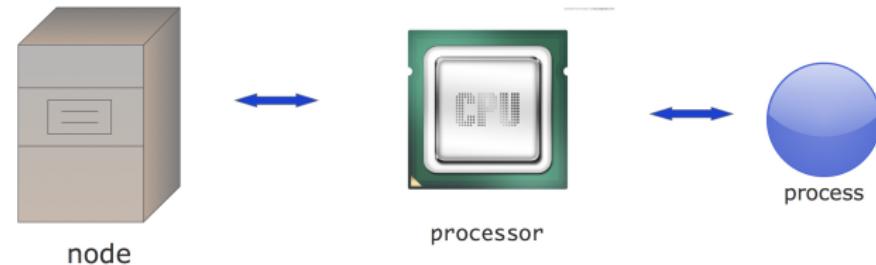
8 IS and VecScatter: irregular grids

9 SNES: Nonlinear solvers

10 TS: Time stepping

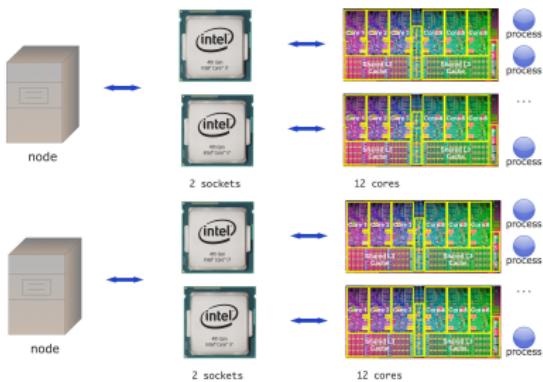
11 Profiling, debugging

# 11. Computers when MPI was designed



One processor and one process per node;  
all communication goes through the network.

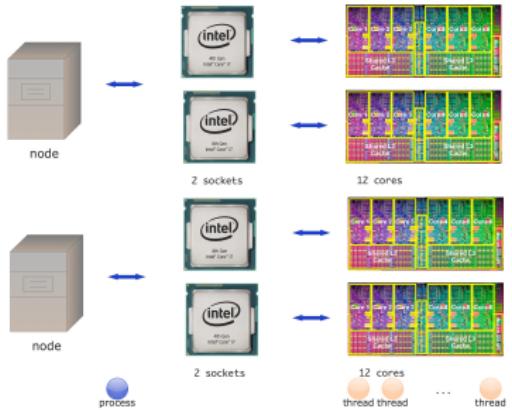
# 12. Pure MPI



A node has multiple sockets, each with multiple cores.

Pure MPI puts a process on each core: pretend shared memory doesn't exist.

# 13. Hybrid programming



Hybrid programming puts a process per node or per socket;  
further parallelism comes from threading.  
No use of threading in PETSc

# 14. Multi-mode programming 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

*TACC note. At the moment only on frontera: module load petsc/3.16-rtx.*

- OpenMP can be used in external packages.

# 15. 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.

# 16. SPMD

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.

# 17. 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:

`srun yourprog`

the number of processes is determined by SLURM.

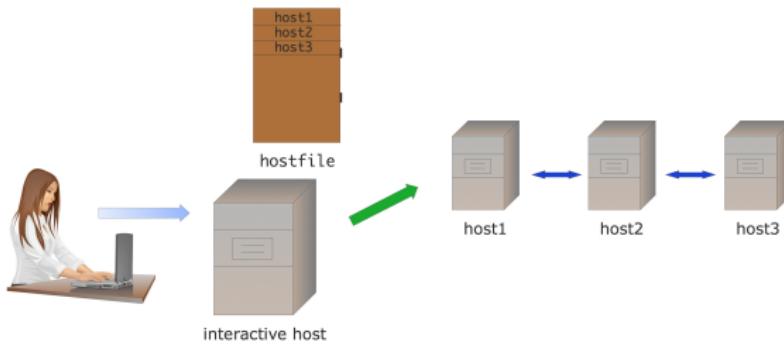
## 18. Do I need a supercomputer?

- 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...

# 19. Cluster setup

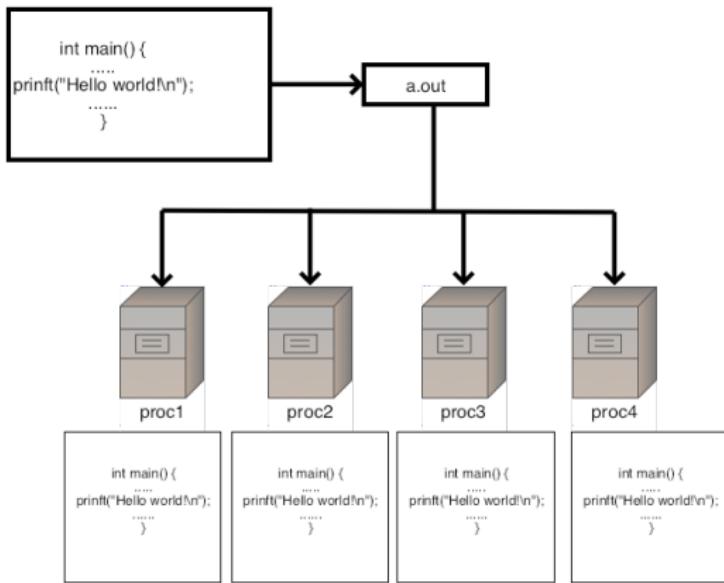
Typical cluster:

- 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.

# 20. In a picture



# 21. Process identification

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

```
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

## 22. Include files, C

```
1 #include "petsc.h"
2 int main(int argc,char **argv)
```

# 23. Include files, Fortran

Include file for preprocessor definitions,  
module for library definitions

```
1 program basic
2 #include <petsc/finclude/petsc.h>
3 use petsc
4 implicit none
```

# 24. Include files, Python

```
1 from petsc4py import PETSc
```

# 25. Variable declarations, C

```
1   KSP           solver;
2   Mat           A;
3   Vec           x,y;
4   PetscInt      n = 20;
5   PetscScalar    v;
6   PetscReal      nrm;
```

Note Scalar vs Real

# 26. Variable declarations, F

```
1   KSP          :: solver  
2   Mat          :: A  
3   Vec          :: x,y  
4   PetscInt    :: j(3)  
5   PetscScalar  :: mv  
6   PetscReal    :: nrm
```

Much like in C

# 27. Library setup, C

```
1 // init.c
2 PetscCall( PetscInitialize
3     (&argc,&argv,(char*)0,help) );
4 int flag;
5 MPI_Initialized(&flag);
6 if (flag)
7     printf("MPI was initialized by PETSc\n");
8 else
9     printf("MPI not yet initialized\n");
```

Can replace `MPI_Init`

General: Every routine has an error return. Catch that value!

# 28. Library setup, F

```
1  call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
2      CHKERRQ(ierr)
3 // all the petsc work
4  call PetscFinalize(ierr); CHKERRQ(ierr)
```

Error code is now final parameter. This holds for every PETSc routine.

## 29. 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
```

# 30. 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 24)
- Compile with `make hello`
- Part two: use `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
```

# 31. PetscPrintf in Fortran

Can only print character buffer:

```
1      character*80      msg
2      write(msg,10) n
3      10 format("Input parameter:",i5)
4      call PetscPrintf(PETSC_COMM_WORLD,msg,ierr)
```

Less elegant than `PetscPrintf` in C

## 32. About routine prototypes: C/C++

Prototype:

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

Use:

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

(always good idea to catch that error code)

# 33. About routine prototypes: Fortran

Prototype

```
1 Subroutine VecCreate
2     ( comm,v,ierr )
3 Type(MPI_Comm) :: comm
4 Vec          :: v
5 PetscErrorCode :: ierr
```

- Final parameter always error parameter. Do not forget!
- MPI types are often `Type(MPI_Comm)` and such,
- PETSc datatypes are handled through the preprocessor.

Use:

```
1 Type(MPI_Comm) :: &
2     comm = MPI_COMM_WORLD
3 Vec          :: v
4 PetscErrorCode :: ierr
5
6 call VecCreate(comm,v,ierr)
```

# 34. About routine prototypes: Python

Object methods:

```
1 # definition
2 PETSc.Mat.setSizes(self, size, bsize=None)
3
4 # use
5 A = PETSc.Mat().create(comm=comm)
6 A.setSizes( ( (None,matrix_size), (None,matrix_size) ) )
```

Class methods:

```
1 # definition
2 PETSc.Sys.Print(type cls, *args, **kwargs)
3
4 # use
5 PETSc.Sys.Print("detecting n option")
```

## 35. Note to self

```
1 PetscInitialize  
2 (&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

# 36. Routine start/end, C

Debugging support:

```
1 PetscFunctionBeginUser;  
2 // all statements  
3 PetscFunctionReturn(0);
```

leads to informative tracebacks.

(Only in C, not in Fortran)

## 37. Example: function with error

```
1 // backtrace.c
2 PetscErrorCode this_function_bombs() {
3     PetscFunctionBegin;
4     SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this");
5     PetscFunctionReturn(0);
6 }
```

## 38. Example: error traceback

```
[0]PETSC ERROR: We cannot go on like this
[0]PETSC ERROR: See https://www.mcs.anl.gov/petsc/documentation/faq.html fo
[0]PETSC ERROR: Petsc Release Version 3.12.2, Nov, 22, 2019
[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`.

```
1     x = 1.5; ierr = square_root(x,&rootx); CHKERRQ(ierr);
2     PetscPrintf(PETSC_COMM_WORLD,"Root of %f is %f\n",x,rootx);
3     x = -2.6; ierr = square_root(x,&rootx); CHKERRQ(ierr);
4     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
```

# 39. Commandline arguments, C

(I'm leaving out the `CHKERRQ(ierr)` in the examples,  
but do use this in actual code)

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

Read commandline argument, print out from processor zero;  
flag can be `PETSC_NULL` if not wanted

# 40. Commandline argument, F

```
1  call PetscOptionsGetInt(  
2      PETSC_NULL_OPTIONS, PETSC_NULL_CHARACTER, &  
3      "-n",n,PETSC_NULL_BOOL,ierr)
```

Note the *PETSC\_NULL\_XXX*: Fortran has strict type checking.

# 41. Program parameters, Python

```
1 nlocal = PETSc.Options().getInt("n",10)
```

# 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

## 42. Create calls

Everything in PETSc is an object, with create and destroy calls:

```
1 VecCreate(MPI_Comm comm,Vec *v);
2 VecDestroy(Vec *v);
3
4 Vec V;
5 VecCreate(MPI_COMM_WORLD,&V);
6 VecDestroy(&V);
```

# 43. Create calls, Fortran

```
1 Vec :: V
2 call VecCreate(MPI_COMM_WORLD,V,e)
3 call VecDestroy(V,e)
```

Note: in Fortran there are no 'star' arguments

## 44. More about vectors

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

The vector object is not completely created in one call:

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

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

```
1 VecDuplicate(Vec v, Vec *w);
```

# 45. Python

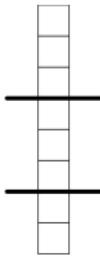
Create is a class method:

```
1 ## setvalues.py
2 comm = PETSc.COMM_WORLD
3 x = PETSc.Vec().create(comm=comm)
4 x.setType(PETSc.Vec.Type.MPI)
```

# 46. Parallel layout up to PETSc

```
1 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)

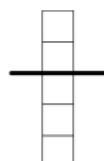
VecSetSizes(V,PETSC\_DECIDE,8)

# 47. Parallel layout specified

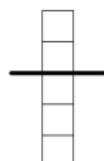
Local or global size in

```
1 VecSetSizes(Vec v, int m, int M);
```

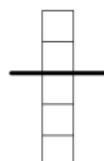
Global size can be specified as PETSC\_DECIDE.



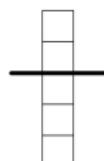
VecSetSizes(V,2,5)



VecSetSizes(V,3,5)



VecSetSizes(V,2,PETSC\_DECIDE)



VecSetSizes(V,3,PETSC\_DECIDE)

# 48. Vector layout in python

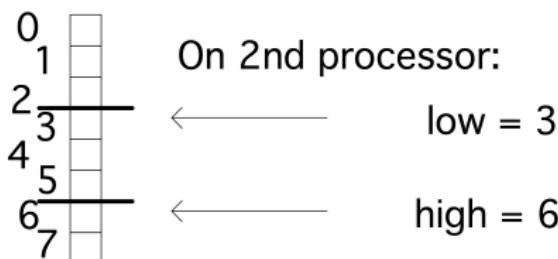
Local and global sizes in a tuple,  
`PETSc.DECIDE` for parameter not specified.

```
1 x.setSizes([2,PETSc.DECIDE])
```

# 49. Query parallel layout

Query vector layout:

```
1 VecGetSize(Vec,PetscInt *globalsize)
2 VecGetLocalSize(Vec,PetscInt *localsize)
3 VecGetOwnershipRange(Vec x,PetscInt *low,PetscInt *high)
```



On 2nd processor:

low = 3

high = 6

# 50. Layout, regardless object

Query general layout:

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

(get local/global given the other)

# 51. Setting values

Set vector to constant value:

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

Set individual elements (global indexing!):

```
1 VecSetValue
2     (Vec x,int row,PetscScalar value,
3      InsertMode mode);
4
5 i = 1; v = 3.14;
6 VecSetValue(x,i,v,INSERT_VALUES);
7
8 call VecSetValue(x,i,v,INSERT_VALUES,e)
```

The other insertmode is ADD\_VALUES.

## 52. Setting values by block

Set individual elements (global indexing!):

```
1 VecSetValues(Vec x,int n,int *rows,PetscScalar *values,
2      InsertMode mode); // INSERT_VALUES or ADD_VALUES
3
4 ii[0] = 1; ii[1] = 2; vv[0] = 2.7; vv[1] = 3.1;
5 VecSetValues(x,2,ii,vv,INSERT_VALUES);
6
7 ii(1) = 1; ii(2) = 2; vv(1) = 2.7; vv(2) = 3.1
8 call VecSetValues(x,2,ii,vv,INSERT_VALUES,ierr,e)
```

# 53. Setting values: Python

```
1 x.setValue(0,1.)  
  
1 x.setValues( [2*procno,2*procno+1], [2.,3.] )
```

## 54. Setting values

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

```
1 VecAssemblyBegin(Vec x);  
2 VecAssemblyEnd(Vec x);
```

'Latency hiding':  
some of the implementation is visible here to the user

# 55. Basic operations

```
1 VecAXPY(Vec y,PetscScalar a,Vec x); /* y <- y + a x */
2 VecAYPX(Vec y,PetscScalar a,Vec x); /* y <- a y + x */
3 VecScale(Vec x, PetscScalar a);
4 VecDot(Vec x, Vec y, PetscScalar *r); /* several variants */
5 VecMDot(Vec x,int n,Vec y[],PetscScalar *r);
6 VecNorm(Vec x,NormType type, PetscReal *r);
7 VecSum(Vec x, PetscScalar *r);
8 VecCopy(Vec x, Vec y);
9 VecSwap(Vec x, Vec y);
10 VecPointwiseMult(Vec w,Vec x,Vec y);
11 VecPointwiseDivide(Vec w,Vec x,Vec y);
12 VecMAXPY(Vec y,int n, PetscScalar *a, Vec x[]);
13 VecMax(Vec x, int *idx, double *r);
14 VecMin(Vec x, int *idx, double *r);
15 VecAbs(Vec x);
16 VecReciprocal(Vec x);
17 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:

$$\begin{aligned} p &\leftarrow x^t y \\ x &\leftarrow x/p \\ n &\leftarrow \|x\|_2 \end{aligned}$$

## 56. 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.

## 57. 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:

```
1 VecCreateSeqWithArray(MPI_Comm comm,  
2   PetscInt n,const PetscScalar array[],Vec *V)  
3 VecCreateMPIWithArray(MPI_Comm comm,  
4   PetscInt n,PetscInt N,const PetscScalar array[],Vec *vv)
```

# 58. Direct access'

Solution 2. Retrieve the internal array:

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

Note: local only; see [VecScatter](#) for more general mechanism)

# 59. Getting values example

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

Fortran: PETSC\_NULL\_INTEGER

# 60. More array juggling

- `VecPlaceArray`: replace the internal array; the original can be restored with `VecRestoreArray`
- `VecReplaceArray`: replace and free the internal array.

# 61. Array handling in F90

```
1     PetscScalar, pointer :: xx_v(:)
2     ....
3     call VecGetArrayF90(x,xx_v,ierr)
4     a = xx_v(3)
5     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

# 62. Matrix creation

The usual create/destroy calls:

```
1 MatCreate(MPI_Comm comm,Mat *A)  
2 MatDestroy(Mat *A)
```

Several more aspects to creation:

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

Local or global size can be PETSC\_DECIDE (as in the vector case)

## 63. If you already have a CRS matrix

```
1 PetscErrorCode MatCreateSeqAIJWithArrays  
2   (MPI_Comm comm, PetscInt m, PetscInt n,  
3    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...

# 64. 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:

```
1 MatSetOption(A,MAT_NEW_NONZERO_LOCATIONS,PETSC_FALSE);
```

(is default) Otherwise:

[1]PETSC ERROR: Argument out of range

[1]PETSC ERROR: New nonzero at (0,1) caused a malloc

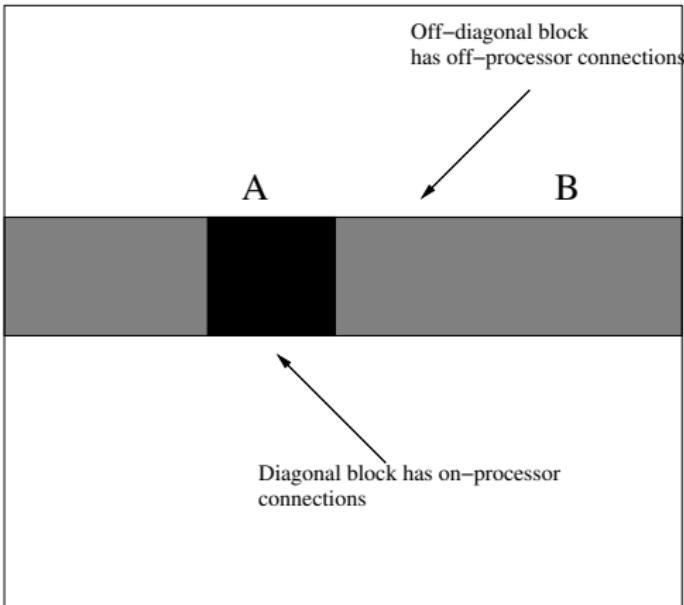
# 65. Sequential matrix structure

```
1 MatSeqAIJSetPreallocation  
2     (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

# 66. Parallel matrix structure



## 67. (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.

# 68. 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

```
1 MatMPIAIJSetPreallocation
2   (Mat B,
3    PetscInt d_nz,const PetscInt d_nnz[],
4    PetscInt o_nz,const PetscInt o_nnz[])
```

In Fortran use `PETSC_NULL_INTEGER` if not specifying arrays

# 69. Matrix creation all in one

```
1 MatCreateSeqAIJ(MPI_Comm comm,PetscInt m,PetscInt n,
2     PetscInt nz,const PetscInt nnz[],Mat *A)
3 MatCreateMPIAIJ(MPI_Comm comm,
4     PetscInt m,PetscInt n,PetscInt M,PetscInt N,
5     PetscInt d_nz,const PetscInt d_nnz[],
6     PetscInt o_nz,const PetscInt o_nnz[],
7     Mat *A)
```

# 70. Querying parallel structure

Matrix partitioned by block rows:

```
1 MatGetSize(Mat mat,PetscInt *M,PetscInt* N);  
2 MatGetLocalSize(Mat mat,PetscInt *m,PetscInt* n);  
3 MatGetOwnershipRange(Mat A,int *first_row,int *last_row);
```

In query functions, unneeded components can be specified as `PETSC_NULL`.

Fortran: `PETSC_NULL_INTEGER`

# 71. Setting values

Set one value:

```
1 MatSetValue(Mat A,  
2   PetscInt i,PetscInt j,PetscScalar va,InsertMode mode)
```

where insert mode is `INSERT_VALUES`, `ADD_VALUES`

Set block of values:

```
1 MatSetValues(Mat A,int m,const int idxm[],  
2   int n,const int idxn[],const PetscScalar values[],  
3   InsertMode mode)
```

( $v$  is row-oriented)

## 72. Set only one element

```
1 MatSetValue(A,i,j,&v,INSERT_VALUES);
```

Special case of the general case:

```
1 MatSetValues(A,1,&i,1,&j,&v,INSERT_VALUES);
```

## 73. Assembling the matrix

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

```
1 MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY);  
2 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.)

## 74. Getting values (C)

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

```
1 PetscErrorCode MatGetRow(Mat mat,
2   PetscInt row,PetscInt *ncols,const PetscInt *cols[],
3   const PetscScalar *vals[])
4 PetscErrorCode MatRestoreRow(/* same parameters */)
```

Note: for inspection only; possibly expensive.

# 75. Getting values (F)

```
1   MatGetRow(A, row, ncols, cols, vals, ierr)
2   MatRestoreRow(A, row, ncols, cols, vals, ierr)
```

where `cols(maxcols)`, `vals(maxcols)` are long enough arrays (allocated by the user)

## 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`.)

# 76. Other matrix types

*MATBAIJ* : blocked matrices (dof per node)

(see PETSC\_DIR/include/petscmat.h)

Dense:

```
1 MatCreateSeqDense(PETSC_COMM_SELF,int m,int n,  
2 PetscScalar *data,Mat *A);  
3 MatCreateDense(MPI_Comm comm,  
4 PetscInt m,PetscInt n,PetscInt M,PetscInt N,  
5 PetscScalar *data,Mat *A)  
6 fg
```

Data argument optional: `PETSC_NULL` or `PETSC_NULL_SCALAR` causes allocation

# 77. GPU support

- Create as GPU matrix,
- Otherwise transparent through overloading

```
1 // cudainit.c
2 PetscDeviceType cuda = PETSC_DEVICE_CUDA;
3 ierr = PetscDeviceInitialize(cuda);
4 PetscBool has_cuda;
5 has_cuda = PetscDeviceInitialized(cuda);
```

VECCUDA, MatCreateDenseCUDA, MATAIJCUSPARSE

```
1 ierr = MatCreate(comm,&A);
2 #ifdef PETSC_HAVE_CUDA
3 ierr = MatSetType(A,MATMPIAIJCUSPARSE);
4 #else
5 ierr = MatSetType(A,MATMPIAIJ);
6#endif
```

# 79. Matrix operations

Main operations are matrix-vector:

- 1 `MatMult(Mat A, Vec in, Vec out);`
- 2 `MatMultAdd`
- 3 `MatMultTranspose`
- 4 `MatMultTransposeAdd`

Simple operations on matrices:

- 1 `MatNorm`
- 2
- 3 `MatScale`
- 4 `MatDiagonalScale`

# 80. Some matrix-matrix operations

```
1 MatMatMult(Mat,Mat,MatReuse,PetscReal,Mat*);  
2  
3 MatPtAP(Mat,Mat,MatReuse,PetscReal,Mat*);  
4  
5 MatMatMultTranspose(Mat,Mat,MatReuse,PetscReal,Mat*);  
6  
7 MatAXPY(Mat,PetscScalar,Mat,MatStructure);
```

# 81. Matrix viewers

```
1 MatView(A,PETSC_VIEWER_STDOUT_WORLD);  
2  
3 row 0: (0, 1)  (2, 0.333333)  (3, 0.25)  (4, 0.2)  
4 row 1: (0, 0.5)  (1, 0.333333)  (2, 0.25)  (3, 0.2)  
5 ....
```

(Fortran: PETSC\_NULL\_INTEGER)

- also invoked by `-mat_view`
- Sparse: only allocated positions listed
- other viewers: for instance `-mat_view_draw` (X terminal)

## 82. 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.

```
1 PetscViewer fd;
2 PetscViewerCreate( comm, &fd );
3 PetscViewerSetType( fd,PETSCVIEWERVTK );
4 MatView( A,fd );
5 PetscViewerDestroy(fd);
```

# 83. Shell matrices

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

```
1 MatSetType(A,MATSHELL); /* or */
2 MatCreateShell(MPI_Comm comm,
3     int m,int n,int M,int N,void *ctx,Mat *mat);
4
5 PetscErrorCode UserMult(Mat mat,Vec x,Vec y);
6
7 MatShellSetOperation(Mat mat,MatOperation MATOP_MULT,
8     (void*)(void)) PetscErrorCode (*UserMult)(Mat,Vec,Vec));
```

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

# 84. Shell matrix context

Shell matrices need custom data

```
1 MatShellSetContext(Mat mat,void *ctx);  
2 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

# 85. Shell matrix example

```
1  ...
2  MatSetType(A,MATSHELL);
3  MatShellSetOperation(A,MATOP_MULT,(void*)&mymatmult);
4  MatShellSetContext(A,(void*)&mystruct);
5  ...
6
7  PetscErrorCode mymatmult(Mat mat,Vec in,Vec out)
8  {
9      PetscFunctionBegin;
10     MatShellGetContext(mat,(void**)&mystruct);
11     /* compute out from in, using mystruct */
12     PetscFunctionReturn(0);
13 }
```

# 86. Submatrices

Extract one parallel submatrix:

```
1 MatGetSubMatrix(Mat mat,
2   IS isrow, IS iscol, PetscInt csize, MatReuse cll,
3   Mat *newmat)
```

Extract multiple single-processor matrices:

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

Collective call, but different index sets per processor

# 87. Load balancing

```
1 MatPartitioningCreate  
2     (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

## 88. 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  $B \approx A^{-1}$

# 89. 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

# 90. Iterative solver basics

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

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

Optional: `KSPSetUp(solver)`

# 91. Solver settings

Change default settings by program calls

example: solver type

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

Settings can be controlled from the commandline:

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

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

- type: `-ksp_type gmres -ksp_gmres_restart 20`
- `-ksp_view` for seeing all settings

## 92. Convergence

Iterative solvers can fail

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

# 93. Reason for convergence

Query the solver object:

```
1 PetscInt its; KSPConvergedReason reason;
2 PetscCall( KSPGetConvergedReason(solver,&reason) );
3 PetscCall( KSPGetIterationNumber(solver,&its) );
4 if (reason<0) {
5     PetscPrintf
6         (comm,"Failure to converge after %d iterations; reason %s\n",
7          its,KSPConvergedReasons[reason]);
8 } else {
9     PetscPrintf
10        (comm,"Number of iterations to convergence: %d\n",
11         its);
12 }
```

# 94. Preconditioners

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

# 95. PC basics

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

```
1 // kspcg.c
2 PetscCall( KSPCreate(comm,&solver);
3             PetscCall( KSPSetOperators(solver,A,A) );
4             PetscCall( KSPSetType(solver,KSPCG) ) );
5 {
6     PC prec;
7     PetscCall( KSPGetPC(solver,&prec) );
8     PetscCall( PCSetType(prec,PCNONE) );
9 }
```

- 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`

# 96. 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

`KSPSetOperators(solver, A, B)`

- $B$  is basis for preconditioner, need not be  $A$
- if  $A$  or  $B$  is to be reused, use `NULL`

# 97. 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

# 98. Simple preconditioners

$$A = D_A + L_A + U_A, \quad M = \dots$$

- 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.

# 99. 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:

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

Application of the preconditioner (that is, solve  $Mx = y$ ) approx same cost as matrix-vector product  $y \leftarrow Ax$

# 100. ILU

PCICC: symmetric, PCILU: nonsymmetric  
many options:

```
1 PCFactorSetLevels(PC pc,int levels);  
2 -pc_factor_levels <levels>
```

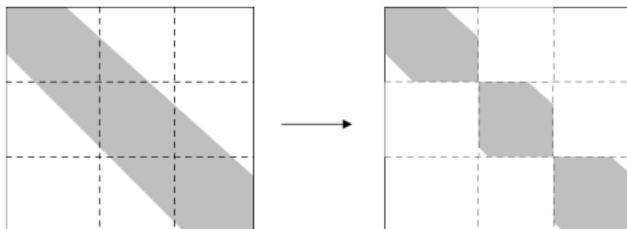
Prevent indefinite preconditioners:

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

value MAT\_SHIFT\_POSITIVE\_DEFINITE et cetera

Factorization preconditioners are sequential  
but still useful; see later

# 101. 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

# 102. Block Jacobi and Additive Schwarz, coding

```
1 KSP *ksp; int nlocal,firstlocal; PC pc;
2 PCBJacobiGetSubKSP(pc,&nlocal,&firstlocal,&ksp);
3 for (i=0; i<nlocal; i++) {
4     KSPSetType( ksp[i], KSPGMRES );
5     KSPGetPC( ksp[i], &pc );
6     PCSetType( pc, PCILU );
7 }
```

Much shorter: commandline options -sub\_ksp\_type and -sub\_pc\_type  
(subksp is PREONLY by default)

```
1 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 `mymatmult` is declared, which is attached by `MatShellSetOperation` to the matrix  $A$  as the means of computing the product `MatMult(A,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  $y \leftarrow A^t Ax$ .

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`?

# 103. Monitors and convergence tests

```
1 KSPSetTolerances(solver,rtol,atol,dtol,maxit);
```

Monitors can be set in code, but simple cases:

- `-ksp_monitor`
- `-ksp_monitor_true_residual`

# 104. Custom monitors and convergence tests

```
1 KSPMonitorSet(KSP ksp,
2     PetscErrorCode (*monitor)
3         (KSP,PetscInt,PetscReal,void*),
4     void *mctx,
5     PetscErrorCode (*monitordestroy)(void*));
6 KSPSetConvergenceTest(KSP ksp,
7     PetscErrorCode (*converge)
8         (KSP,PetscInt,PetscReal,KSPConvergedReason*,void*),
9     void *cctx,
10    PetscErrorCode (*destroy)(void*))
```

# 105. Example of convergence tests

```
1 PetscErrorCode resconverge
2   (KSP solver,PetscInt it,PetscReal res,
3    KSPConvergedReason *reason,void *ctx)
4 {
5   MPI_Comm comm; Mat A; Vec X,R; PetscErrorCode ierr;
6   PetscFunctionBegin;
7   KSPGetOperators(solver,&A,PETSC_NULL,PETSC_NULL);
8   PetscObjectGetComm((PetscObject)A,&comm);
9   KSPBuildResidual(solver,PETSC_NULL,PETSC_NULL,&R);
10  KSPBuildSolution(solver,PETSC_NULL,&X);
11  /* stuff */
12  if (sometest) *reason = 15;
13  else *reason = KSP_CONVERGED_ITERATING;
14  PetscFunctionReturn(0);
```

# 106. Advanced options

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

- 1 `KSPSetInitialGuessNonzero`
- 2 `KSPGMRESSetRestart`
- 3 `KSPSetPreconditionerSide`
- 4 `KSPSetNormType`

Many options easier through commandline.

# 107. Null spaces

Iterating orthogonal to the null space of the operator:

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

The solver will now properly remove the null space at each iteration.

# 108. Matrix-free solvers

Shell matrix requires shell preconditioner in `KSPSetOperators`):

```
1  PCSetType(pc,PCSHELL);  
2  PCShellSetContext(PC pc,void *ctx);  
3  PCShellGetContext(PC pc,void **ctx);  
4  PCShellSetApply(PC pc,  
5      PetscErrorCode (*apply)(void*,Vec,Vec));  
6  PCShellSetSetUp(PC pc,  
7      PetscErrorCode (*setup)(void*))
```

similar idea to shell matrices

Alternative: use different operator for preconditioner

# 109. Fieldsplit preconditioners

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

Matrix block storage: `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

# 110. Fieldsplit use

Easy case: all fields are the same size

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

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

```
1 PetscOptionsSetValue
2 (" -fieldsplit_0_pc_type ", " lu ");
3 PetscOptionsSetValue
4 (" -fieldsplit_0_pc_factor_mat_solver_package ", " mumps ");
```

Fields can be named instead of numbered.

# 111. more

Non-strided, arbitrary fields: `PCFieldSplitSetIS()`

Stokes equation can be detected:

`-pc_fieldsplit_detect_saddle_point`

Combining fields multiplicatively: solve

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

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

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

# 112. Fieldsplit example

```
1 KSPGetPC(solver,&prec);
2 PCSetType(prec,PCFIELDSPLIT);
3 PCFieldSplitSetBlockSize(prec,2);
4 PCFieldSplitSetType(prec,PC_COMPOSITE_ADDITIVE);
5 PetscOptionsSetValue
6     ("-fieldsplit_0_pc_type","lu");
7 PetscOptionsSetValue
8     ("-fieldsplit_0_pc_factor_mat_solver_package","mumps");
9 PetscOptionsSetValue
10    ("-fieldsplit_1_pc_type","lu");
11 PetscOptionsSetValue
12    ("-fieldsplit_1_pc_factor_mat_solver_package","mumps");
```

# 113. Global preconditioners: MG

```
1  PCSetType(PC pc,PCMG);
2  PCMGSetsLevels(pc,int levels,MPI Comm *comms);
3  PCMGSetsType(PC pc,PCMGTyoe mode);
4  PCMGSetsCycleTyoe(PC pc,PCMGCycleTyoe ctype);
5  PCMGSetsNumberSmoothUp(PC pc,int m);
6  PCMGSetsNumberSmoothDown(PC pc,int n);
7  PCMGGetCoarseSolve(PC pc,KSP *ksp);
8  PCMGSetsInterpolation(PC pc,int level,Mat P); and
9  PCMGSetsRestriction(PC pc,int level,Mat R);
10 PCMGSetsResidual(PC pc,int level,PetscErrorCode
11      (*residual)(Mat,Vec,Vec,Vec),Mat mat);
```

# 114. 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`

# 115. 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

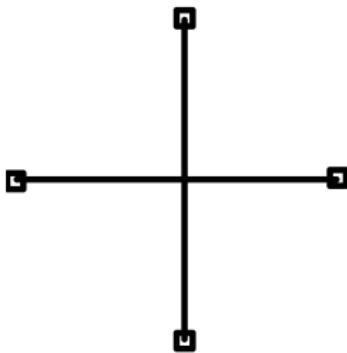
11 Profiling, debugging

# 116. Regular grid: DMDA

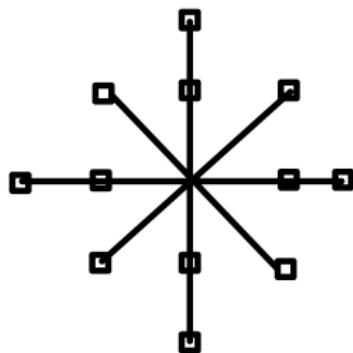
DMDAs are for storing vector field, not matrix.

Support for different stencil types:

Star stencil

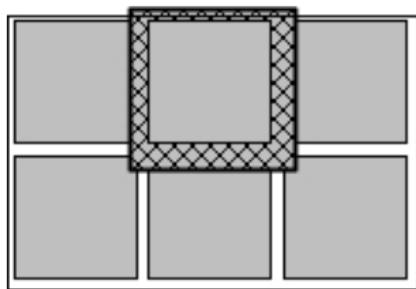


Box stencil



## 117. Ghost regions around processors

A DM<sub>A</sub> defines a global vector, which contains the elements of the grid, and a local vector for each processor which has space for "ghost points".



# 118. DMDA construction

```
1 DMDACreate2d(comm, bndx,bndy, type, M, N, m, n,  
2 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)

# 119. Grid info

Divide  $100 \times 100$  grid over 4 processes,  
stencil width= 1:

## Code:

```
1 // dmrhs.c
2 DM grid;
3 PetscCall( DMDACreate2d
4     ( comm,
5         DM_BOUNDARY_NONE,DM_BOUNDARY_NONE,
6         DM_DA_STENCIL_STAR,
7         100,100,
8         PETSC_DECIDE,PETSC_DECIDE,
9         1,
10        1,
11        NULL,NULL,
12        &grid
13    ) );
14 PetscCall( DMSetFromOptions(grid) );
15 PetscCall( DMSetUp(grid) );
16 PetscCall( DMViewFromOptions(grid,NULL,"-dm_view") );
```

## Output:

```
1 ld: warning: dylib
      ↪(/Users/eijkhout/Ins
      ↪was built for
      ↪newer macOS
      ↪version (11.5)
      ↪than being
      ↪linked (11.0)
2 [0] Local = 0-50 x
      ↪0-50, halo =
      ↪0-51 x 0-51
3 [1] Local = 50-100 x
      ↪0-50, halo =
      ↪49-100 x 0-51
4 [2] Local = 0-50 x
      ↪50-100, halo =
      ↪0-51 x 49-100
5 [3] Local = 50-100 x
      ↪50-100, halo =
```

# 120. Associated vectors

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

```
1 Vec ghostvector;
2 PetscCall( DMGetLocalVector(grid,&ghostvector) );
3 PetscCall( DMGlobalToLocal(grid,xy,INSERT_VALUES,ghostvector) );
4 PetscReal ***xyarray,**gh;
5 PetscCall( DMDAVecGetArray(grid,xy,&xyarray) );
6 PetscCall( DMDAVecGetArray(grid,ghostvector,&gh) );
7 // computation on the arrays
8 PetscCall( DMDAVecRestoreArray(grid,xy,&xyarray) );
9 PetscCall( DMDAVecRestoreArray(grid,ghostvector,&gh) );
10 PetscCall( DMLocalToGlobal(grid,ghostvector,INSERT_VALUES,xy) );
11 PetscCall( DMRestoreLocalVector(grid,&ghostvector) );
```

# 121. Grid info

```
1  typedef struct {
2      PetscInt      dim,dof,sw;
3      PetscInt      mx,my,mz;      /* grid points in x,y,z */
4      PetscInt      xs,ys,zs;      /* starting point, excluding ghosts */
5      PetscInt      xm,ym,zm;      /* grid points, excluding ghosts */
6      PetscInt      gxs,gys,gzs; /* starting point, including ghosts */
7      PetscInt      gxm,gym,gzm; /* grid points, including ghosts */
8      DMBoundaryType bx,by,bz;    /* type of ghost nodes */
9      DMDAStencilType st;
10     DM              da;
11 } DMDALocalInfo;
```

## 122. Range over local subdomain

```
1  for (int j=info.ys; j<info.ys+info.ym; j++) {  
2      for (int i=info.xs; i<info.xs+info.xm; i++) {  
3          // actions on point i,j  
4      }  
5  }
```

# 123. Arrays of vectors

```
1 Vec ghostvector;
2 PetscCall( DMGetLocalVector(grid,&ghostvector) );
3 PetscCall( DMGlobalToLocal(grid,xy,INSERT_VALUES,ghostvector) );
4 PetscReal **xyarray,**gh;
5 PetscCall( DMDAVecGetArray(grid,xy,&xyarray) );
6 PetscCall( DMDAVecGetArray(grid,ghostvector,&gh) );
7 // computation on the arrays
8 PetscCall( DMDAVecRestoreArray(grid,xy,&xyarray) );
9 PetscCall( DMDAVecRestoreArray(grid,ghostvector,&gh) );
10 PetscCall( DMLocalToGlobal(grid,ghostvector,INSERT_VALUES,xy) );
11 PetscCall( DMRestoreLocalVector(grid,&ghostvector) );
```

# 124. Operating on arrays

```
1 for (int j=info.ys; j<info.ys+info.ym; j++) {  
2     for (int i=info.xs; i<info.xs+info.xm; i++) {  
3         if (info.gxs<info.xs && info.gys<info.ys)  
4             if (i-1>=info.gxs && i+1<=info.gxs+info.gxm &&  
5                 j-1>=info.gys && j+1<=info.gys+info.gym )  
6                 xyarray[j][i] =  
7                     ( gh[j-1][i] + gh[j][i-1] + gh[j][i+1] + gh[j+1][i] )  
8                     /4.;
```

# 125. Associated matrix

Matrix that has knowledge of the grid:

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

Set matrix values based on stencil:

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

(ordering of row/col variables too complicated for `MatSetValues`)

# 126. Set values by stencil

```
1 // grid2d.c
2 for (int j=info.ys; j<info.ys+info.ym; j++) {
3     for (int i=info.xs; i<info.xs+info.xm; i++) {
4         MatStencil row,col[5];
5         PetscScalar v[5];
6         PetscInt     ncols = 0;
7         row.j        = j; row.i = i;
8         /***** local connection: diagonal element ****/
9         col[ncols].j = j; col[ncols].i = i; v[ncols++] = 4.;
10        /* boundaries: top and bottom row */
11        if (i>0)          {col[ncols].j = j;   col[ncols].i = i-1;
12           ↪v[ncols++] = -1.;}
13        if (i<info.mx-1) {col[ncols].j = j;   col[ncols].i = i+1;
14           ↪v[ncols++] = -1.;}
15        /* boundary left and right */
16        if (j>0)          {col[ncols].j = j-1; col[ncols].i = i;
17           ↪v[ncols++] = -1.;}
18        if (j<info.my-1) {col[ncols].j = j+1; col[ncols].i = i;
19           ↪v[ncols++] = -1.;}

20        PetscCall(
21           ↪MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES) );
22    }
```

## 127. DM Plex

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

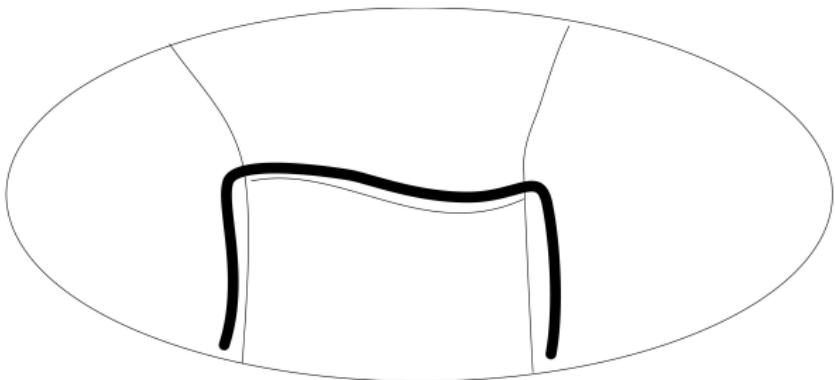
This is complicated and under-documented.

# 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

## 128. 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

# 129. 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)
```

# 130. IS: index set

Index Set is a set of indices

```
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`

# 131. 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.

## 132. index sets for this example

```
1 // oddeven.c
2 IS oddeven;
3 if (procid==0) {
4 PetscCall( ISCreateStride(comm,Nglobal/2,0,2,&oddeven) );
5 } else {
6 PetscCall( ISCreateStride(comm,Nglobal/2,1,2,&oddeven) );
7 }
```

# 133. scatter for this example

```
1 VecScatter separate;
2 PetscCall( VecScatterCreate
3     (in,oddeven,out,NULL,&separate) );
4 PetscCall( VecScatterBegin
5     (separate,in,out,INSERT_VALUES,SCATTER_FORWARD) );
6 PetscCall( VecScatterEnd
7     (separate,in,out,INSERT_VALUES,SCATTER_FORWARD) );
```

# Exercise 9 (oddeven)

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

Process [0]

10.  
8.  
6.  
4.  
2.  
0.

Process [1]

11.  
9.  
7.  
5.  
3.  
1.

## 134. 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);
```

# 136. Example: even and odd indices

```
1 // oddeven.c
2 IS oddeven;
3 if (procid==0) {
4 PetscCall( ISCreateStride(comm,Nglobal/2,0,2,&oddeven) );
5 } else {
6 PetscCall( ISCreateStride(comm,Nglobal/2,1,2,&oddeven) );
7 }
```

# 137. scattering odd and even

```
1 VecScatter separate;
2 PetscCall( VecScatterCreate
3     (in,oddeven,out,NULL,&separate) );
4 PetscCall( VecScatterBegin
5     (separate,in,out,INSERT_VALUES,SCATTER_FORWARD) );
6 PetscCall( VecScatterEnd
7     (separate,in,out,INSERT_VALUES,SCATTER_FORWARD) );
```

# 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

# 138. 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.

# 139. Basic SNES usage

User supplies function and Jacobian:

```
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

# 140. SNES specification: function evaluation

```
1 PetscErrorCode (*FunctionEvaluation)(SNES,Vec,Vec,void*);  
2 VecCreate(PETSC_COMM_WORLD,&r);  
3 SNESSetFunction(snes,r,FunctionEvaluation,*ctx);
```

# 141. SNES specification: jacobian evaluation

```
1 PetscErrorCode (*FormJacobian)(SNES,Vec,Mat,Mat,void*);  
2 MatCreate(PETSC_COMM_WORLD,&J);  
3 SNESSetJacobian(snes,J,J,FormJacobian,*ctx);
```

# 142. SNES solution

```
1 SNESolve(snes, /* rhs= */ PETSC_NULL, x)
2 SNESGetConvergedReason(snes, &reason)
3 SNESGetIterationNumber(snes, &its)
```

# 143. 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);
```

# 144. I

In the user function:

```
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);
}
```

# 145. 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);  
SNESSet Jacobian  
(snes,J,J,MatMFFDCompute Jacobian,(void*)&user);
```

# 146. Further possibilities

```
1 SNESSetTolerances  
2 (SNES snes,double atol,double rtol,double stol,  
3 int its,int fcts);
```

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

adaptive convergence: `-snes_ksp_ew_conv` (Eisenstat Walker)

## 147. 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)
```

# 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

# 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

# 148. Basic profiling

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

```
%% 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

# 149. Log summary: overall

	Max	Max/Min	Avg	Total
Time (sec):	5.493e-01	1.00006	5.493e-01	
Objects:	2.900e+01	1.00000	2.900e+01	
Flops:	1.373e+07	1.00000	1.373e+07	2.746e+07
Flops/sec:	2.499e+07	1.00006	2.499e+07	4.998e+07
Memory:	1.936e+06	1.00000		3.871e+06
MPI Messages:	1.040e+02	1.00000	1.040e+02	2.080e+02
MPI Msg Lengths:	4.772e+05	1.00000	4.588e+03	9.544e+05
MPI Reductions:	1.450e+02	1.00000		

# 150. Log summary: details

	Max	Ratio	Max	Ratio	Avg	len	%T	%F	%M	%L	%R	%T	%F	%M	%L	%R	Mflop/s	
MatMult	100	1.0	3.4934e-02	1.0	1.28e+08	1.0	8.0e+02	6	32	96	17	0	6	32	96	17	0	255
MatSolve	101	1.0	2.9381e-02	1.0	1.53e+08	1.0	0.0e+00	5	33	0	0	0	5	33	0	0	0	305
MatLUFactorNum	1	1.0	2.0621e-03	1.0	2.18e+07	1.0	0.0e+00	0	0	0	0	0	0	0	0	0	0	43
MatAssemblyBegin	1	1.0	2.8350e-03	1.1	0.00e+00	0.0	1.3e+05	0	0	3	83	1	0	0	3	83	1	0
MatAssemblyEnd	1	1.0	8.8258e-03	1.0	0.00e+00	0.0	4.0e+02	2	0	1	0	3	2	0	1	0	3	0
VecDot	101	1.0	8.3244e-03	1.2	1.43e+08	1.2	0.0e+00	1	7	0	0	35	1	7	0	0	35	243
KSPSetup	2	1.0	1.9123e-02	1.0	0.00e+00	0.0	0.0e+00	3	0	0	0	2	3	0	0	0	2	0
KSPSolve	1	1.0	1.4158e-01	1.0	9.70e+07	1.0	8.0e+02	26100	96	17	92	26100	96	17	92	194		

# 151. User events

```
1 #include "petsclog.h"
2 int USER_EVENT;
3 PetscLogEventRegister(&USER_EVENT,"User event name",0);
4 PetscLogEventBegin(USER_EVENT,0,0,0,0);
5 /* application code segment to monitor */
6 PetscLogFlops(number of flops for this code segment);
7 PetscLogEventEnd(USER_EVENT,0,0,0,0);
```

# 152. Program stages

```
1 PetscLogStagePush(int stage); /* 0 <= stage <= 9 */
2 PetscLogStagePop();
3 PetscLogStageRegister(int stage, char *name)
```

# 153. 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`