



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

# Outline

- 1 Introduction
- 2 Getting started
- 3 A word about SPMD parallelism
- 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 Getting started
- 3 A word about SPMD parallelism
- 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

# 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

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.

# What does PETSc target?

- Serial and Parallel
- Linear and nonlinear
- Finite difference and finite element
- Structured and unstructured

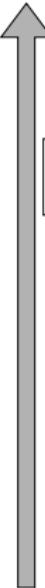
# What is in PETSc?

- 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

# Documentation and help

- Web page: <http://www.mcs.anl.gov/petsc/>
- PDF manual: <http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf>
- 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`

Level of  
Abstraction



Application Codes

SNES  
(Nonlinear Equations Solvers)

TS  
(Time Stepping)

PC  
(Preconditioners)

KSP  
(Krylov Subspace Methods)

Matrices

Vectors

Index Sets

---

BLAS

MPI

# 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

# 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
- **Eigenvalue solvers (including SVD):** SLEPC

# 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
- ⇒ you do not need to know much MPI when you use PETSc

# PETSc and parallelism

- All objects in Petsc are defined on a communicator;  
can only interact if on the same communicator
- Parallelism through MPI
- No OpenMP used in the library:  
user can use shared memory programming.
- Likewise, threading is kept outside of PETSc code: not thread-safe.
- Transparent: same code works sequential and parallel.

# 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

## 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.

# Running at TACC

Use two windows: one on the login node, one on a compute node.

Compute node for 6 hours, on 2 nodes with 20 cores total:

```
idev -t 2:0:0 -N 2 -n 20 -q normal-mic
```

(or whatever queue we run the training)

# Table of Contents

- 1 Introduction
- 2 Getting started
- 3 A word about SPMD parallelism
- 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

Eijkhout: PETSc intro

# Program header, C

```
#include "petsc.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **argv)
```

- Petsc include file: one at the top of the file
- Declare the name of each routine: helps with traceback
- Can also include `petscmat.h` if no higher functionality needed.

# Program header, F proper style

```
program basic
#include <petsc/finclude/petsc.h>
use petsc
implicit none
```

# Variable declarations, C

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

Note Scalar vs Real

# Variable declarations, F

```
KSP          :: solver  
Mat          :: A  
Vec          :: x,y  
PetscInt     :: j(3)  
PetscScalar   :: mv  
PetscReal    :: nrm
```

Much like in C; uses cpp

# Library setup, C

```
ierr = PetscInitialize(&argc,&argv,0,0); CHKERRQ(ierr);
// all the petsc work
ierr = PetscFinalize(); CHKERRQ(ierr);
```

Can replace MPI\_Init

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

## Library setup, F

```
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.

- **CHKERRA** in main program
- **CKKERRQ** in subprograms

## 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`
- Compile with `make hello`
- Part two: use `PetscSynchronizedPrintf`

## Note to self

```
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

## Routine start/end, C

Debugging support:

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

leads to informative tracebacks.

(Only in C, not in Fortran)

# Program parameters, C

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

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

Read commandline argument, print out from processor zero  
flag can be `PETSC_NULL`

# Program parameters, F

```
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)
```

*Compatibility note:* PETSC\_NULL\_BOOL will be introduced after 3.10.2

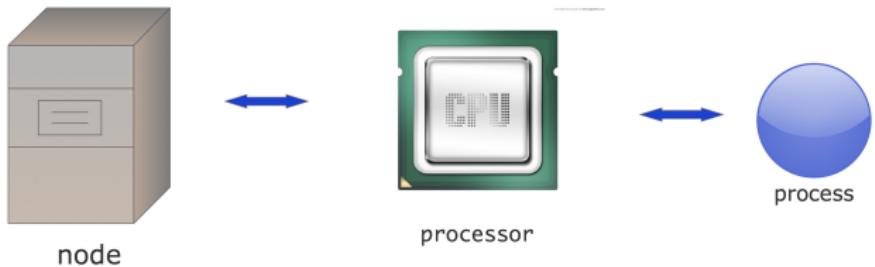
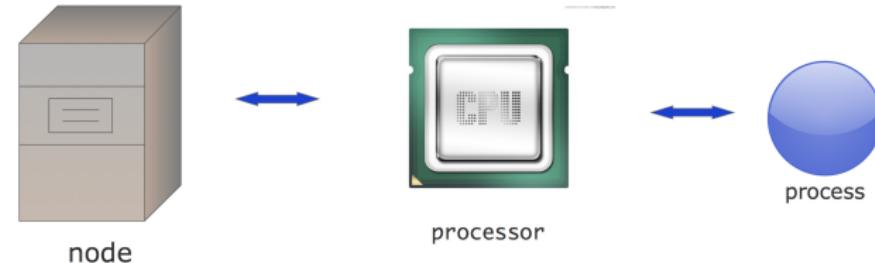
Less elegant than PetscPrintf in C

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

# Table of Contents

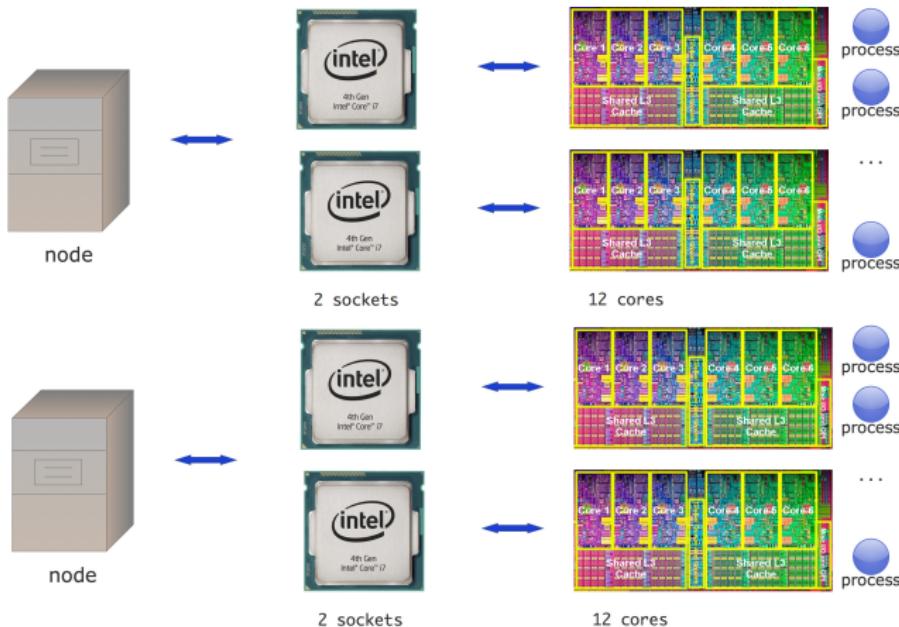
- 1 Introduction
- 2 Getting started
- 3 A word about SPMD parallelism
- 4 Vec datatype: vectors
- 5 Mat Datatype: matrix
- 6 KSP & PC: Iterative solvers
- 7 Grid manipulation
- 8 IS and VecScatter: irregular grids
- 9 SNES: Nonlinear solvers
- 10 TS: Time stepping
- 11 Profiling, debugging

# Computers when MPI was designed



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

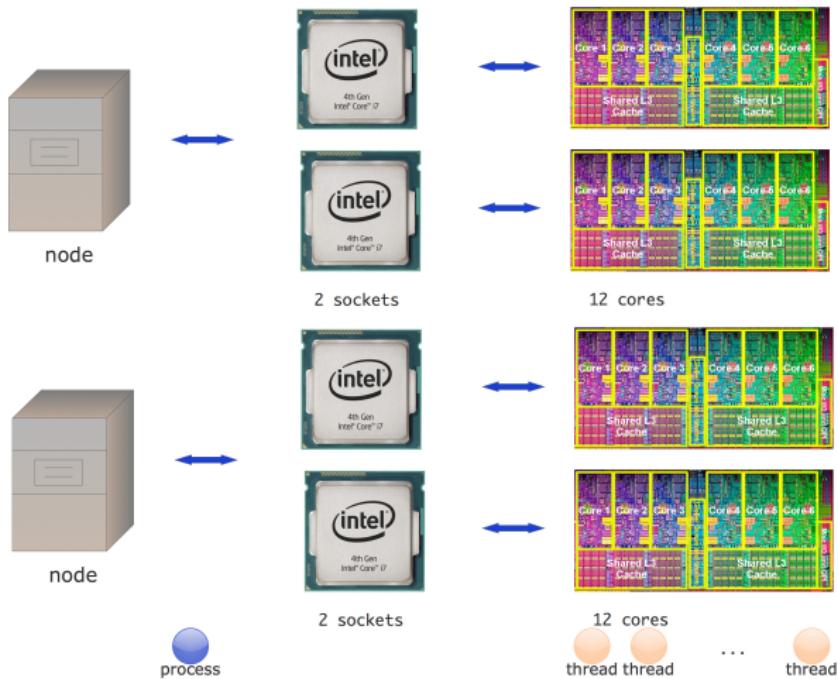
# Pure MPI



A node has multiple sockets, each with multiple cores.

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

# Hybrid programming



Hybrid programming puts a process per node or per socket; further parallelism comes from threading.

Not in this course...

# 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.

# 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.

# Compiling and running

MPI compilers are usually called `mpicc`, `mpif90`, `mpicxx`.

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

At TACC:

```
ibrun yourprog
```

the number of processes is determined by SLURM.

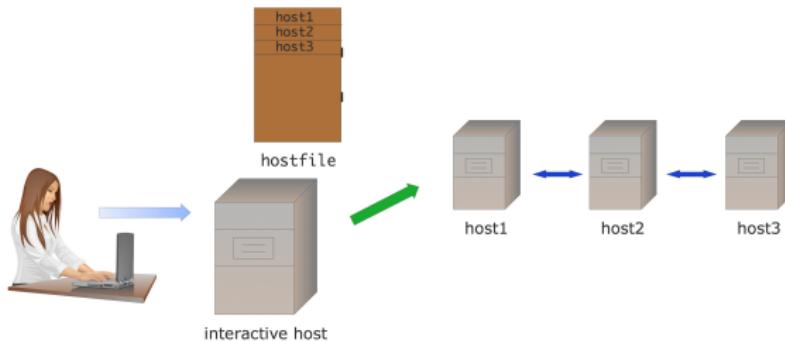
# Do I need a supercomputer?

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

# 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.

# How to make exercises

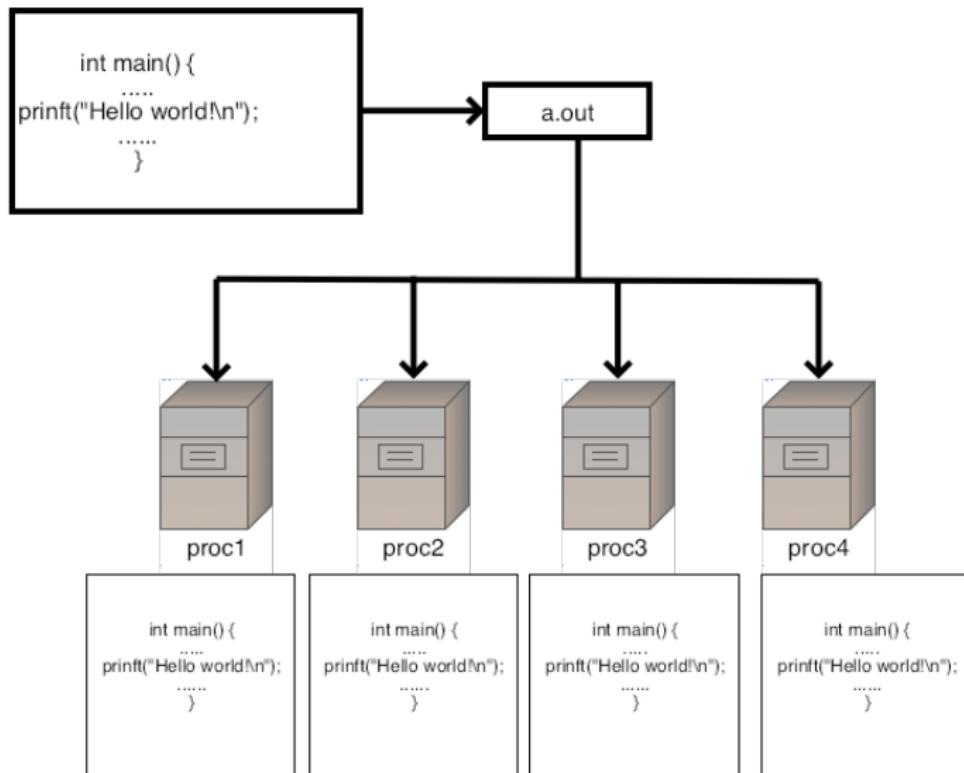
- Directory: exercises-3.10-c or f or (maybe) p
- Type `make exercisename` to compile it
- Python: setup once per session

```
module load python
```

No compilation needed. Run:

```
ibrun python yourprogram
```

# In a picture



# 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!

# About routine prototypes: C/C++

Prototype:

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

Use:

```
MPI_Comm comm = MPI_COMM_WORLD;  
int nprocs;  
int errorcode;  
errorcode = MPI_Comm_size( comm, &nprocs );
```

(but forget about that error code most of the time)

# About routine prototypes: Fortran

## Prototype

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

## Use:

```
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
integer :: size
CALL MPI_Comm_size( comm, size, ierr )
```

- Final parameter always error parameter. Do not forget!
- Most MPI\_... types are INTEGER.

# About routine prototypes: Python

Prototype:

```
# object method  
MPI.Comm.Send(self, buf, int dest, int tag=0)  
# class method  
MPI.Request.Waitall(type cls, requests, statuses=None)
```

Use:

```
from mpi4py import MPI  
comm = MPI.COMM_WORLD  
comm.Send(sendbuf, dest=other)  
MPI.Request.Waitall(requests)
```

# Table of Contents

- 1 Introduction
- 2 Getting started
- 3 A word about SPMD parallelism
- 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

Eijkhout: PETSc intro

# Create calls

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

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

```
VecDestroy(Vec *v);
```

```
/* C */
```

```
Vec V;
```

```
VecCreate(MPI_COMM_SELF, &V);
```

```
VecDestroy(&V);
```

```
! Fortran
```

```
Vec V
```

```
call VecCreate(MPI_COMM_SELF, V, e)
```

```
call VecDestroy(V, e)
```

Note: in Fortran there are no “star” arguments

## More about vectors

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

The vector object is not completely created in one call:

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

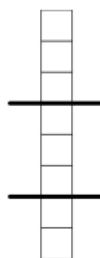
Other ways of creating: make more vectors like this one:

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

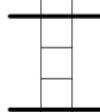
# Parallel layout up to PETSc

```
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)
```



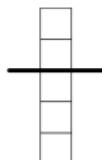
```
VecSetSizes(V,PETSC_DECIDE,8)
```

# Parallel layout specified

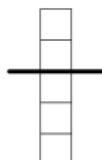
Local or global size in

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

Global size can be specified as PETSC\_DECIDE.



VecSetSizes(V,2,5)



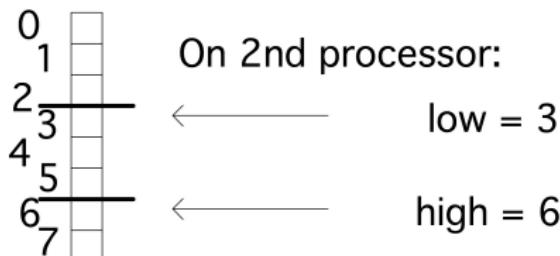
VecSetSizes(V,3,PETSC\_DECIDE)

VecSetSizes(V,3,PETSC\_DECIDE)

# Query parallel layout

Query vector layout:

```
VecGetSize(Vec, PetscInt *globalsize)  
VecGetLocalSize(Vec, PetscInt *localsize)  
VecGetOwnershipRange(Vec x, PetscInt *low, PetscInt *high)  
VecGetOwnershipRange(x, low, high, ierr) ! F
```



Query general layout:

```
PetscSplitOwnership(MPI_Comm comm, Petsc
```

```
    )  
EIJOUT PetscSplitOwnership(comm, n, N, ierr) ! F
```

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

```
i = 1; v = 3.14;  
VecSetValue(x,i,v,INSERT_VALUES);  
ii[0] = 1; ii[1] = 2; vv[0] = 2.7; vv[1] = 3.1;  
VecSetValues(x,2,ii,vv,INSERT_VALUES);
```

```
call VecSetValue(x,i,v,INSERT_VALUES,ierr,e)  
ii(1) = 1; ii(2) = 2; vv(1) = 2.7; vv(2) = 3.1  
call VecSetValues(x,2,ii,vv,INSERT_VALUES,i
```

# Setting values

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

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

# Basic operations

```
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, double *r);
VecSum(Vec x, PetscScalar *r);
VecCopy(Vec x, Vec y);
VecSwap(Vec x, Vec y);
VecPointwiseMult(Vec w,Vec x,Vec y);
VecPointwiseDivide(Vec w,Vec x,Vec y);
VecMAXPY(Vec y,int n, PetscScalar *a, Vec x[]);
VecMax(Vec x, int *idx, double *r);
VecMin(Vec x, int *idx, double *r);
VecAbs(Vec x);
VecReciprocal(Vec x);
VecShift(Vec x,PetscScalar s);
```

## Exercise 2 (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:

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

# Getting 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?

- Create vector from user provided array:

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

- Get the internal array (local only; see `VecScatter` for more general mechanism):

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

# Getting values example

```
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

# Array handling in F90

```
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 Getting started
- 3 A word about SPMD parallelism
- 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

Eijkhout: PETSc intro

# Matrix creation

The usual create/destroy calls:

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

Several more aspects to creation:

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

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

## If you already have a CRS matrix

```
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...

# 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:

```
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
```

# Sequential matrix structure

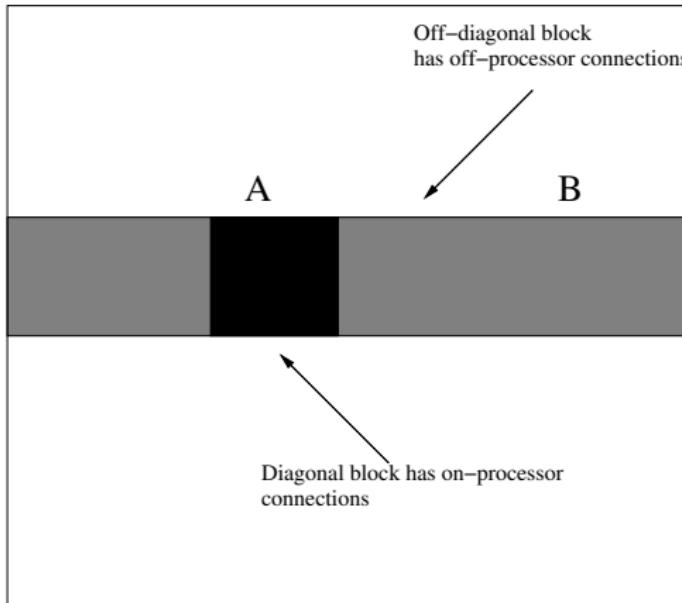
MatSeqAIJSetPreallocation

```
(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

# Parallel matrix structure



## (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.

# Parallel matrix structure description

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

MatMPIAIJSetPreallocation

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

In Fortran use PETSC\_NULL\_INTEGER if not specifying arrays

# Querying parallel structure

Matrix partitioned by block rows:

```
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

# Setting values

Set one value:

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

where insert mode is INSERT\_VALUES, ADD\_VALUES

Set block of values:

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

(v is row-oriented)

Special case of the general case:

```
MatSetValues(A,1,&i,1,&j,&v,INSERT_VALUES); // C  
MatSetValues(A,1,i,1,j,v,INSERT_VALUES,e); ! F
```

## Assembling the matrix

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

```
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 3 (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.)

## Getting values (C)

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

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

Note: for inspection only; possibly expensive.

## Getting values (F)

```
MatGetRow (A, row, ncols, cols, vals, ierr)
```

```
MatRestoreRow (A, row, ncols, cols, vals, ierr)
```

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

## Exercise 4 (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.)

## Other matrix types

MATBAIJ : blocked matrices (dof per node)

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

Dense:

```
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)
```

fg

Data argument optional: PETSC\_NULL or PETSC\_NULL\_SCALAR causes allocation

# Matrix operations

Main operations are matrix-vector:

MatMult (Mat A, Vec in, Vec out);

MatMultAdd

MatMultTranspose

MatMultTransposeAdd

Simple operations on matrices:

MatNorm

MatScale

MatDiagonalScale

# Some matrix-matrix operations

```
MatMatMult (Mat,Mat,MatReuse,PetscReal,Mat*) ;
```

```
MatPtAP (Mat,Mat,MatReuse,PetscReal,Mat*) ;
```

```
MatMatMultTranspose (Mat,Mat,MatReuse,PetscReal,Mat*) ;
```

```
MatAXPY (Mat,PetscScalar,Mat,MatStructure) ;
```

# Matrix viewers

```
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)

# 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.

```
PetscViewer fd;  
PetscViewerCreate( comm, &fd );  
PetscViewerSetType( fd,PETSCVIEWERVTK );  
MatView( A,fd );  
PetscViewerDestroy(fd);
```

# Shell matrices

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

```
MatSetType(A,MATSHELL); /* or */  
MatCreateShell(MPI_Comm comm,  
    int m,int n,int M,int N,void *ctx,Mat *mat);  
  
PetscErrorCode UserMult(Mat mat,Vec x,Vec y);  
  
MatShellSetOperation(Mat mat,MatOperation MATOP_MULT,  
    (void(*)(void)) PetscErrorCode (*UserMult)(Mat,Vec,Vec));
```

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

# Shell matrix context

Shell matrices need custom data

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

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

User program sets context, matmult routine accesses it

## Shell matrix example

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

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

# Submatrices

Extract one parallel submatrix:

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

Extract multiple single-processor matrices:

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

Collective call, but different index sets per processor

# Load balancing

MatPartitioningCreate

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

Various packages for creating better partitioning: Chaco, Parmetis

# Table of Contents

- 1 Introduction
- 2 Getting started
- 3 A word about SPMD parallelism
- 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

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

# Topics

- All linear solvers in PETSc are iterative, even the direct ones
- Preconditioners
- Fargoin control through commandline options
- Tolerances, convergence and divergence reason
- Custom monitors and convergence tests

# Iterative solver basics

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

**Optional:** KSPSetup(solver)

**Reuse** Amat **or** Pmat: KSPGetOperators **and** PetscObjectReference.

## Solver type

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

KSP can be controlled from the commandline:

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

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

- **type:** `-ksp_type gmres -ksp_gmres_restart 20`
- `-ksp_view`

# 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?

```
KSPSolve(solver,B,X);
KSPGetConvergedReason(solver,&reason);
if (reason<0) {
    PetscPrintf(comm,
        "Failure to converge after %d iterations; reason %s\n",
        its,KSPConvergedReasons[reason]);
} else {
    KSPGetIterationNumber(solver,&its);
    PetscPrintf(comm,
        "Convergence in %d iterations.\n",its);
}
```

# 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

# PC basics

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

```
KSP solver; PC precon;  
KSPCreate(comm, &solver);  
KSPGetPC(solver, &precon);  
PCSetType(precon, PCJACOBI);
```

- 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
```

# 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`

# 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

# Simple preconditioners

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

- 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:  $M = (I + (\omega^{-1}D_A)^{-1} + L_A)(\omega^{-1}D_A + U_A)$

# 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:

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

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

Factorization preconditioners are sequential

# ILU

PCICC: symmetric, PCILU: nonsymmetric

many options:

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

Prevent indefinite preconditioners:

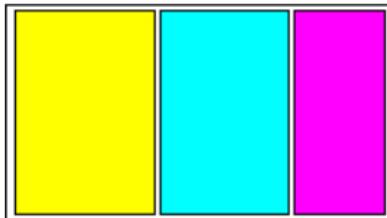
```
PCFactorSetShiftType(PC pc, MatFactorShiftType type);
```

**value** MAT\_SHIFT\_POSITIVE\_DEFINITE **et cetera**

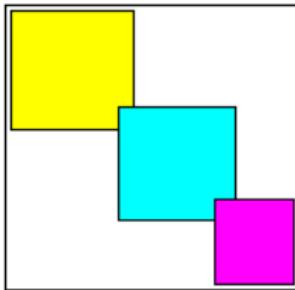
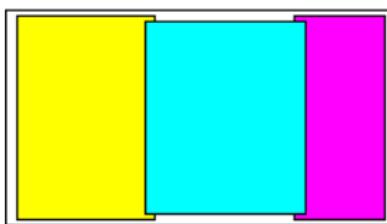
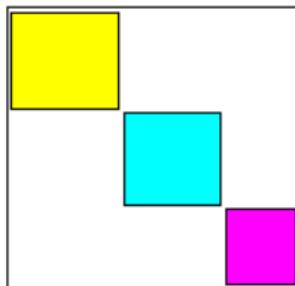
# Block Jacobi and Additive Schwarz

- Factorization preconditioners are sequential;
- can be made parallel by use in Block Jacobi or Additive Schwarz methods
- each processor has its own block(s) to work with

Domain partitioning



Matrix blocks



# 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

## Block Jacobi and Additive Schwarz, coding

```
KSP *ksps; int nlocal,firstlocal; PC pc;  
PCBJacobiGetSubKSP(pc,&nlocal,&firstlocal,&ksps);  
for (i=0; i<nlocal; i++) {  
    KSPSetType( ksps[i], KSPGMRES );  
    KSPGetPC( ksps[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 5 (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.

# Monitors and convergence tests

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

Monitors can be set in code, but simple cases:

- -ksp\_monitor
- -ksp\_monitor\_true\_residual

# Custom monitors and convergence tests

```
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*))
```

# Example of convergence tests

```
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);
}
```

## Advanced options

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

KSPSetInitialGuessNonzero

KSPGMRESSetRestart

KSPSetPreconditionerSide

KSPSetNormType

Many options easier through commandline.

# Null spaces

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

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

# Matrix-free solvers

Shell matrix requires shell preconditioner in `KSPSetOperators`):

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

similar idea to shell matrices

Alternative: use different operator for preconditioner

# 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

Easy case: all fields are the same size

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

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

```
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: 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

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

# Fieldsplit example

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

# Global preconditioners: MG

```
PCSetType(PC pc,PCMG);  
PCMGSetsLevels(pc,int levels,MPI Comm *comms);  
PCMGSetsType(PC pc,PCMGTMode mode);  
PCMGSetsCycleType(PC pc,PCMGCycleType ctype);  
PCMGSetsNumberSmoothUp(PC pc,int m);  
PCMGSetsNumberSmoothDown(PC pc,int n);  
PCMGGetsCoarseSolve(PC pc,KSP *ksp);  
PCMGSetsInterpolation(PC pc,int level,Mat P); and  
PCMGSetsRestriction(PC pc,int level,Mat R);  
PCMGSetsResidual(PC pc,int level,PetscErrorCode  
(*residual)(Mat,Vec,Vec,Vec),Mat mat);
```

# 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`

# 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
```

## Other external PCs

If installed, other parallel preconditioner are available:

- From Hypre: PCHYPRE with subtypes boomeramg, parasails, euclid, pilut:  
PCHYPRESetType (pc,parasails)  
or -pc\_hypre\_type parasails
- PCSPAI for Sparse Approximate Inverse
- PCPROMETHEUS
- External packages' existence can be tested:

```
%% grep hypre $PETSC_DIR/$PETSC_ARCH/include/petscconf.h
#ifndef PETSC_HAVE_HYPRE
#define PETSC_HAVE_HYPRE 1
#ifndef PETSC_HAVE_LIBHYPRE
#define PETSC_HAVE_LIBHYPRE 1
```

# Table of Contents

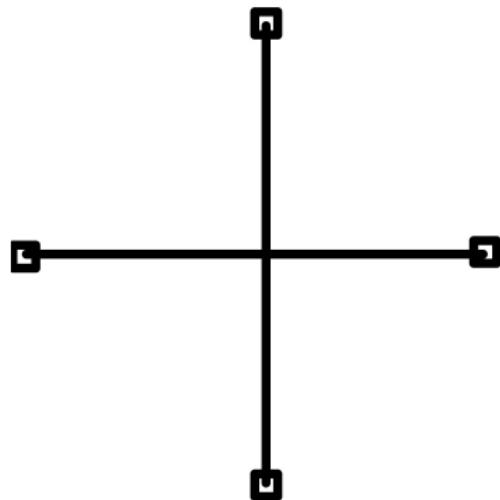
- 1 Introduction
- 2 Getting started
- 3 A word about SPMD parallelism
- 4 Vec datatype: vectors
- 5 Mat Datatype: matrix
- 6 KSP & PC: Iterative solvers
- 7 Grid manipulation
- 8 IS and VecScatter: irregular grids
- 9 SNES: Nonlinear solvers
- 10 TS: Time stepping
- 11 Profiling, debugging

# Regular grid: DMDA

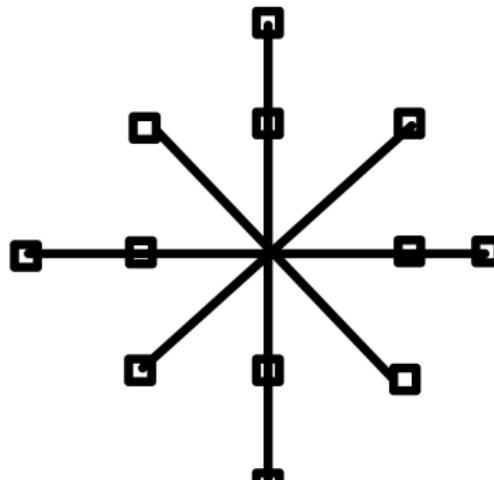
DMDAs are for storing vector field, not matrix.

Support for different stencil types:

Star stencil

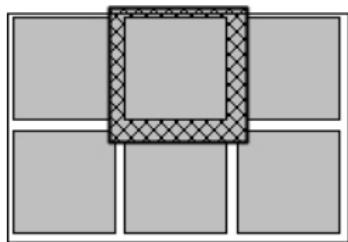


Box stencil



# 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".



# DMDA construction

```
DMDACreate2d(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)

# Associated vectors

```
DMCreateGlobalVector(DM grid,Vec *g);  
DMCreateLocalVector(DM grid,Vec *l);
```

global -> local

```
DMGlobalToLocalBegin/End  
(DMDA da,Vec g,InsertMode iora,Vec l);
```

local -> global

```
DMLocalToGlobalBegin/End  
(DMDA da,Vec l,InsertMode mode,Vec g);
```

local -> global -> local :

```
DMLocalToLocalBegin/End  
(DMDA da,Vec l1,InsertMode iora,Vec l2);
```

## Associated matrix

Matrix that has knowledge of the grid:

```
DMCreateMatrix(DM grid, Mat *J)
```

Set matrix values based on stencil:

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

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

Loop over local part of the grid:

```
DMDAGetCorners(DMDA da,  
    PetscInt *x, PetscInt *y, PetscInt *z,
```

# Grid info

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

# DM Plex

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

This is still preliminary (and therefore under-documented).

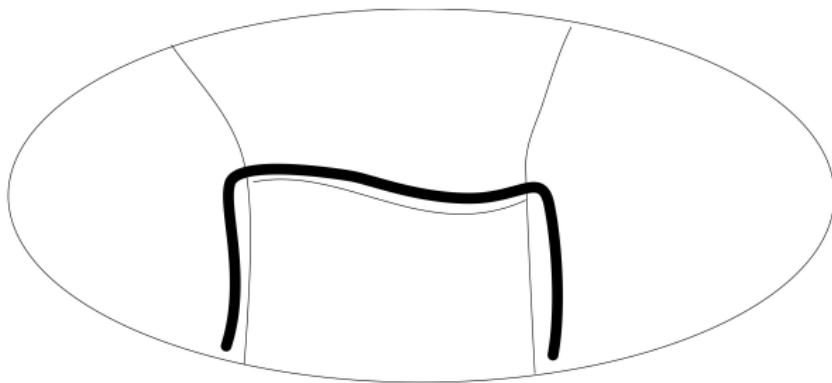
# Table of Contents

- 1 Introduction
- 2 Getting started
- 3 A word about SPMD parallelism
- 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

Eijkhout: PETSc intro

# 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

# 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)
```

# 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, ISIInvertPermutations et cetera.

Also ISGetIndices / ISRestoreIndices / ISGetSize

## 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);
```

## Example: even and odd indices

```
if (mytid==0) {
    ierr = ISCreateStride
        (MPI_COMM_SELF, (N+1)/2, 0, 2, &gindices); // even
} else {
    ierr = ISCreateStride
        (MPI_COMM_SELF, N/2, 1, 2, &gindices); // odd
}
ierr = ISGetSize(gindices, &localsize);
/* create a vector with the requisite local size */
ierr = VecCreate(MPI_COMM_WORLD, &in);
ierr = VecSetType(in, VECMPI);
ierr = VecSetSizes(in, localsize, N);
```

## Create local target vectors:

```
/* create IS objects for the target locations */
ierr = ISCreateStride
      (MPI_COMM_SELF,localsize,0,1,&lindices);
/* output vector is local */
ierr = VecCreate(MPI_COMM_SELF,&out);
ierr = VecSetType(out,VECSEQ);
ierr = VecSetSizes(out,localsize,localsize);
```

## Create and apply scatter:

```
/* create a scatter from the source indices to target */
ierr = VecScatterCreate(in,gindices,out,lindices,&scatter);
/* index sets are no longer needed */
ierr = ISDestroy(&gindices);
ierr = ISDestroy(&gindices);

ierr = VecScatterBegin
(scatter,in,out,INSERT_VALUES,SCATTER_FORWARD);
ierr = VecScatterEnd
(scatter,in,out,INSERT_VALUES,SCATTER_FORWARD);
VecView(out,0);
```

```
Vector Object: 1 MPI processes
  type: seq
0
2
4
6
8
10
12
14
16
18
20
Vector Object: 1 MPI processes
  type: seq
1
3
5
7
9
11
13
15
17
19
21
```

## Same, but distributed output vector

```
/* create IS objects for the target locations */
ierr = VecGetOwnershipRange(in,&myfirst,PETSC_NULL);
ierr = ISCreateStride
      (MPI_COMM_SELF,localsize,myfirst,1,&lindices);
/* output vectors will have the same size & layout as input */
ierr = VecDuplicate(in,&out);
```

# Table of Contents

- 1 Introduction
- 2 Getting started
- 3 A word about SPMD parallelism
- 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

# 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.

# 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

# SNES specification

```
VecCreate(PETSC_COMM_WORLD, &r)
SNESSetFunction(snes, r, FormFunction, *ctx)

MatCreate(PETSC_COMM_WORLD, &J)
SNESSet Jacobian(snes, J, J, Form Jacobian, *ctx)

SNES Solve(snes, /* rhs= */ PETSC_NULL, x)
SNESGetConvergedReason(snes, &reason)
SNESGetIterationNumber(snes, &its)
```

# User functions

```
PetscErrorCode FormFunction
  (SNES snes,Vec x,Vec f,void *ctx)
{
// set elements of residual f
return 0;
}
```

```
PetscErrorCode FormJacobian
  (SNES snes,Vec x,Mat *jac,Mat *prec,MatStructure *flag,
   void *ctx)
{
// compute the jacobian
return 0;
}
```

# 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);
```

## 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);
}
```

# 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) ;
```

## Further possibilities

```
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)

# 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 Getting started
- 3 A word about SPMD parallelism
- 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 Getting started
- 3 A word about SPMD parallelism
- 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

# 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

# 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		

# Log summary: details

	Max	Ratio	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	26100	96	194	

# User events

```
#include "petsclog.h"
int USER EVENT;
PetscLogEventRegister(&USER EVENT, "User event name", 0);
PetscLogEventBegin(USER EVENT, 0, 0, 0, 0);
/* application code segment to monitor */
PetscLogFlops(number of flops for this code segment);
PetscLogEventEnd(USER EVENT, 0, 0, 0, 0);
```

# Program stages

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

# 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`