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

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 dffculties 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?

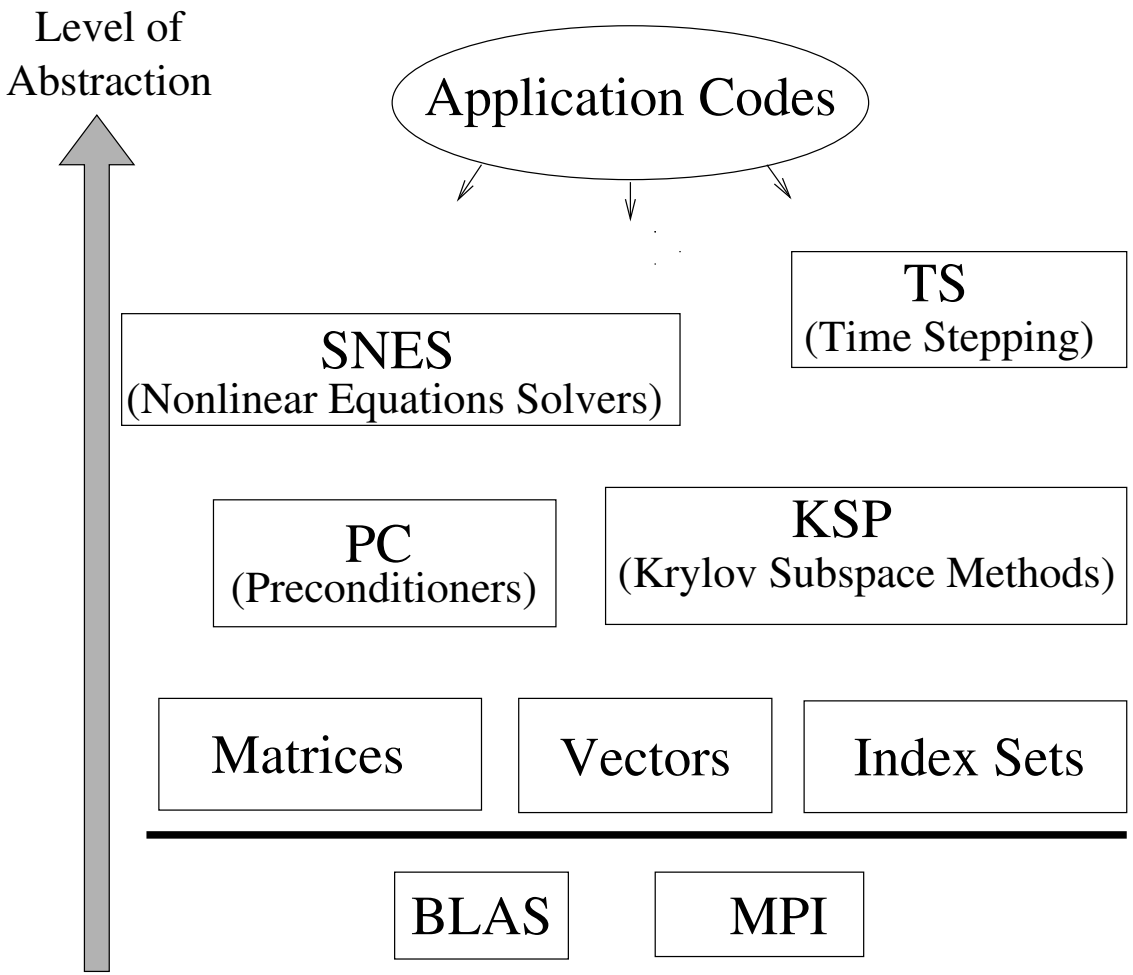
- Serial and Parallel
- Linear and nonlinear
- Finite difference and finite element
- Structured and unstructured

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
- PETSc on your local cluster: ask your local support
- General questions about PETSc: petsc-maint@mcs.anl.gov
- Example codes, found online, and in `$PETSC_DIR/src/mat/examples` et cetera
- Sometimes consult include files, for instance `$PETSC_DIR/include/petscmat.h`


Level of
Abstraction
Application Codes
TS
(Time Stepping)
SNES
(Nonlinear Equations Solvers)
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

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
- $\Rightarrow$ you do not need to know much MPI when you use PETSc

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

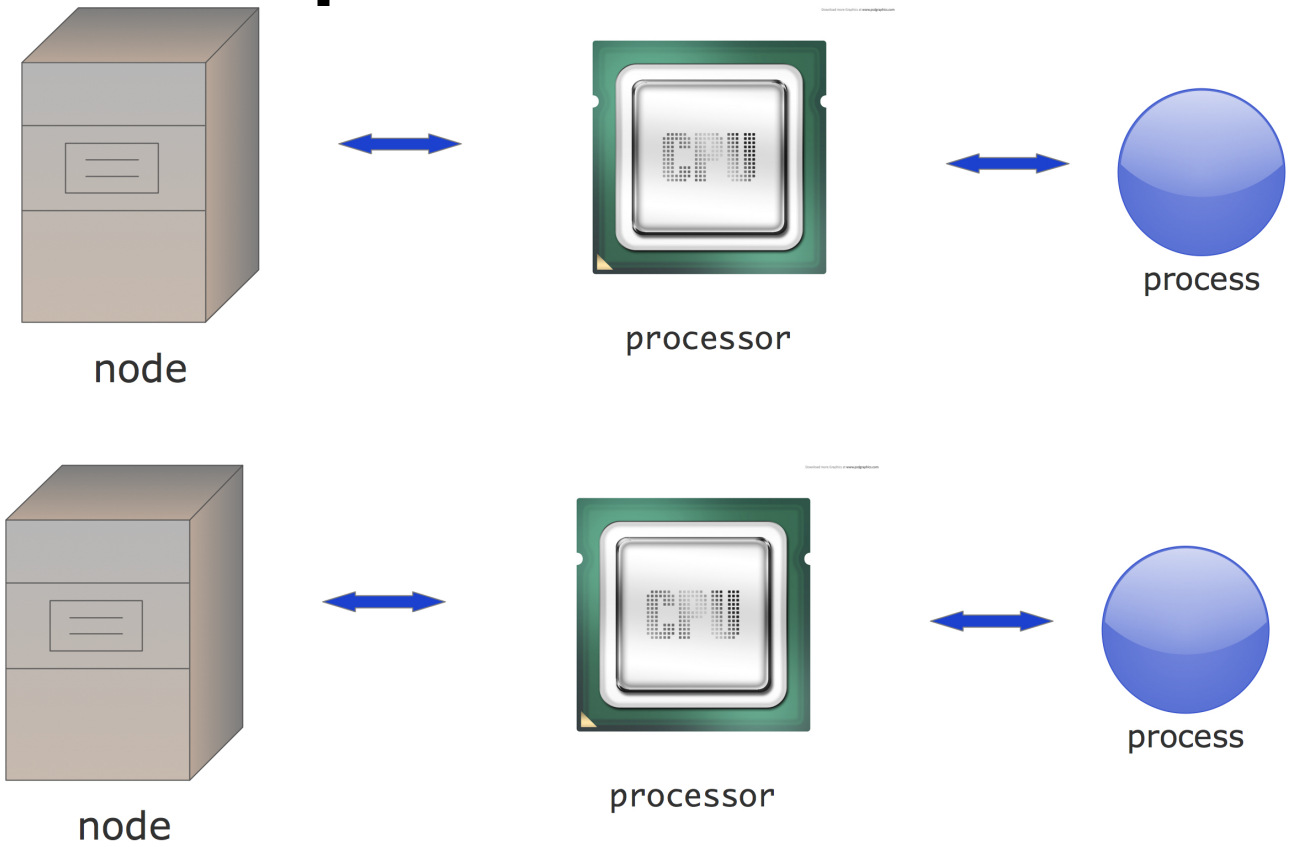
Data hiding

To support this uniform interface, the implementation is hidden:

```
MatSetValue(A,i,j,v,INSERT_VALUES); // A[i,j] <- v
```

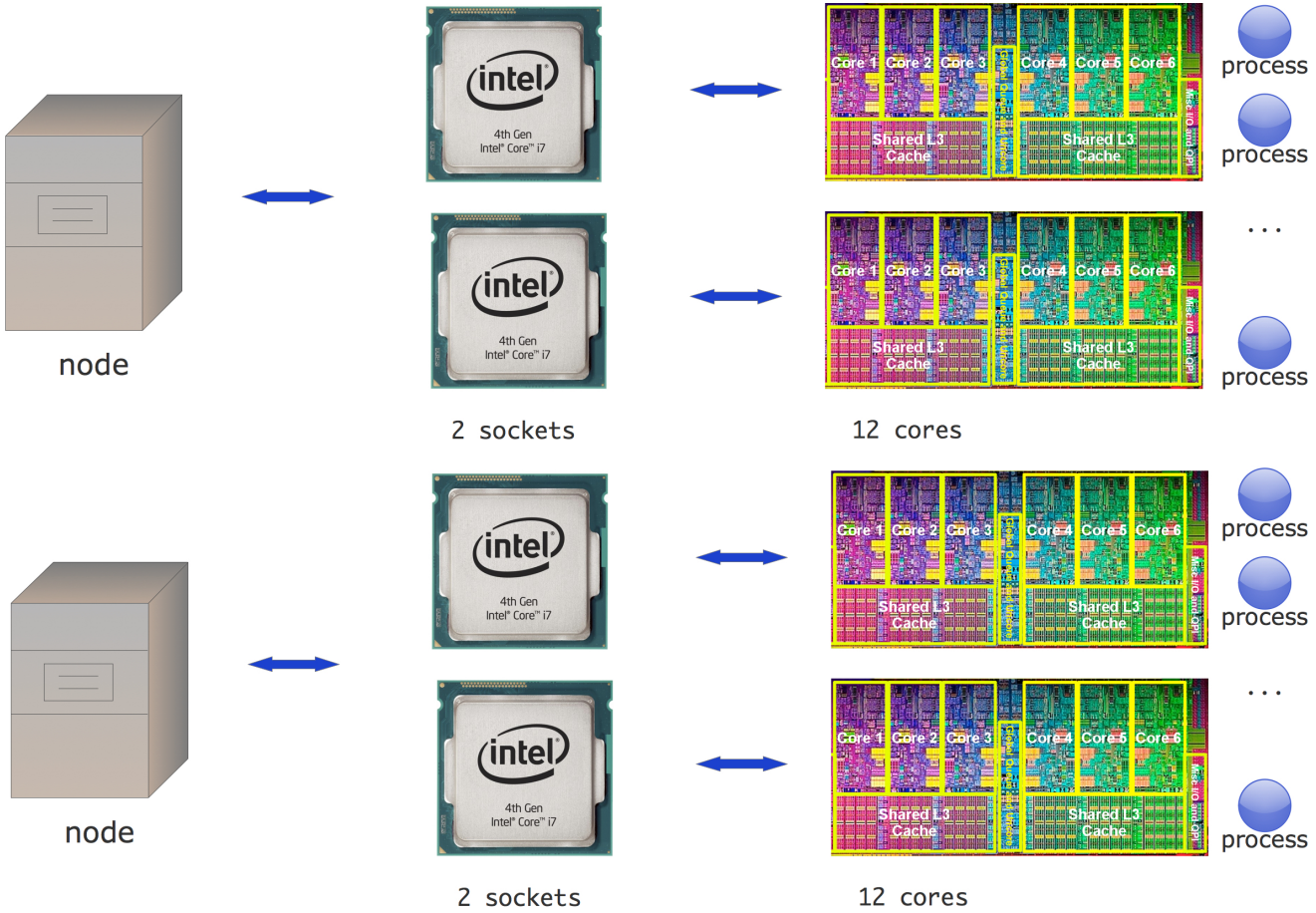
There are some direct access routines, but most of the time you don't need them.

Computers when MPI was designed



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

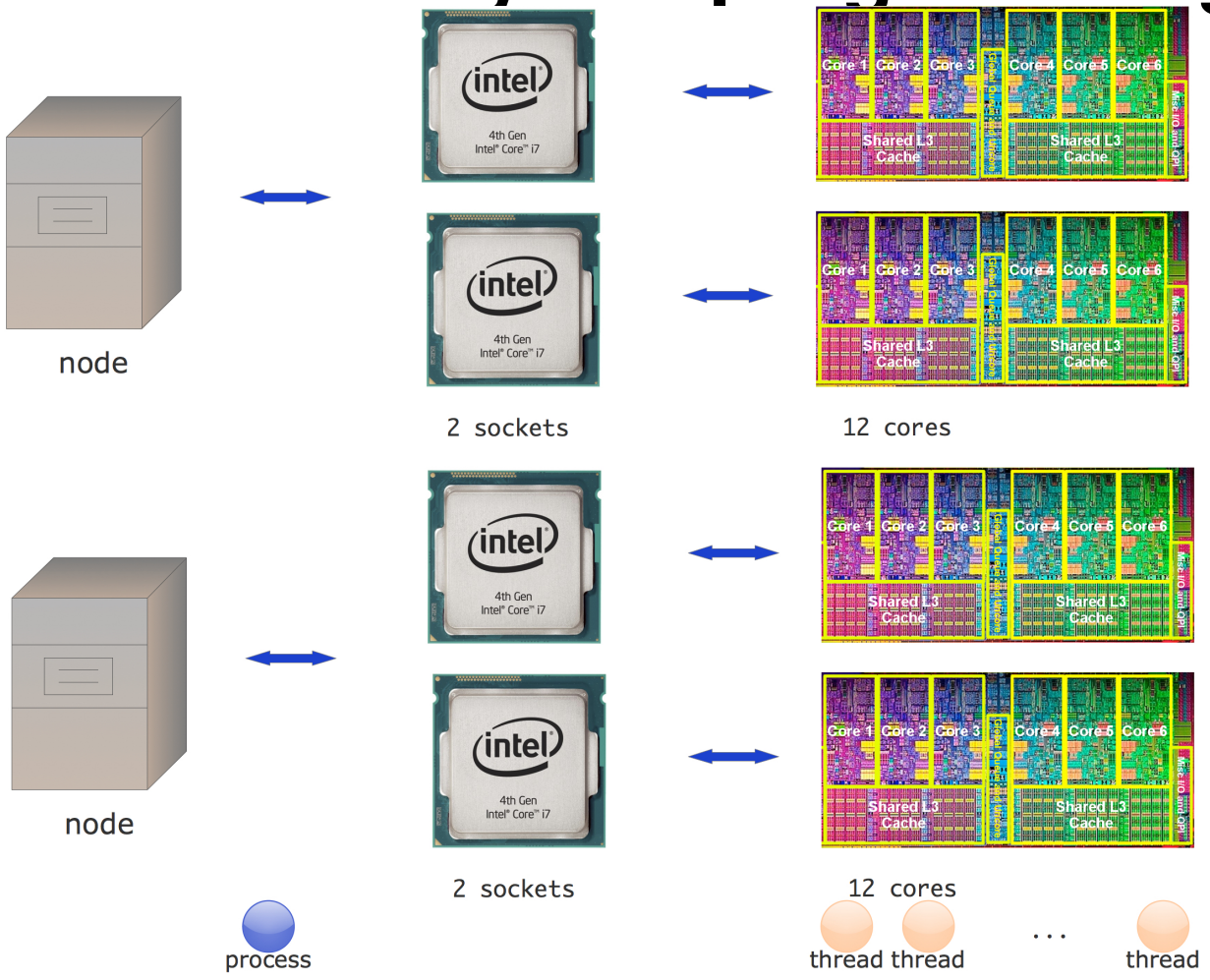
Pure MPI



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.

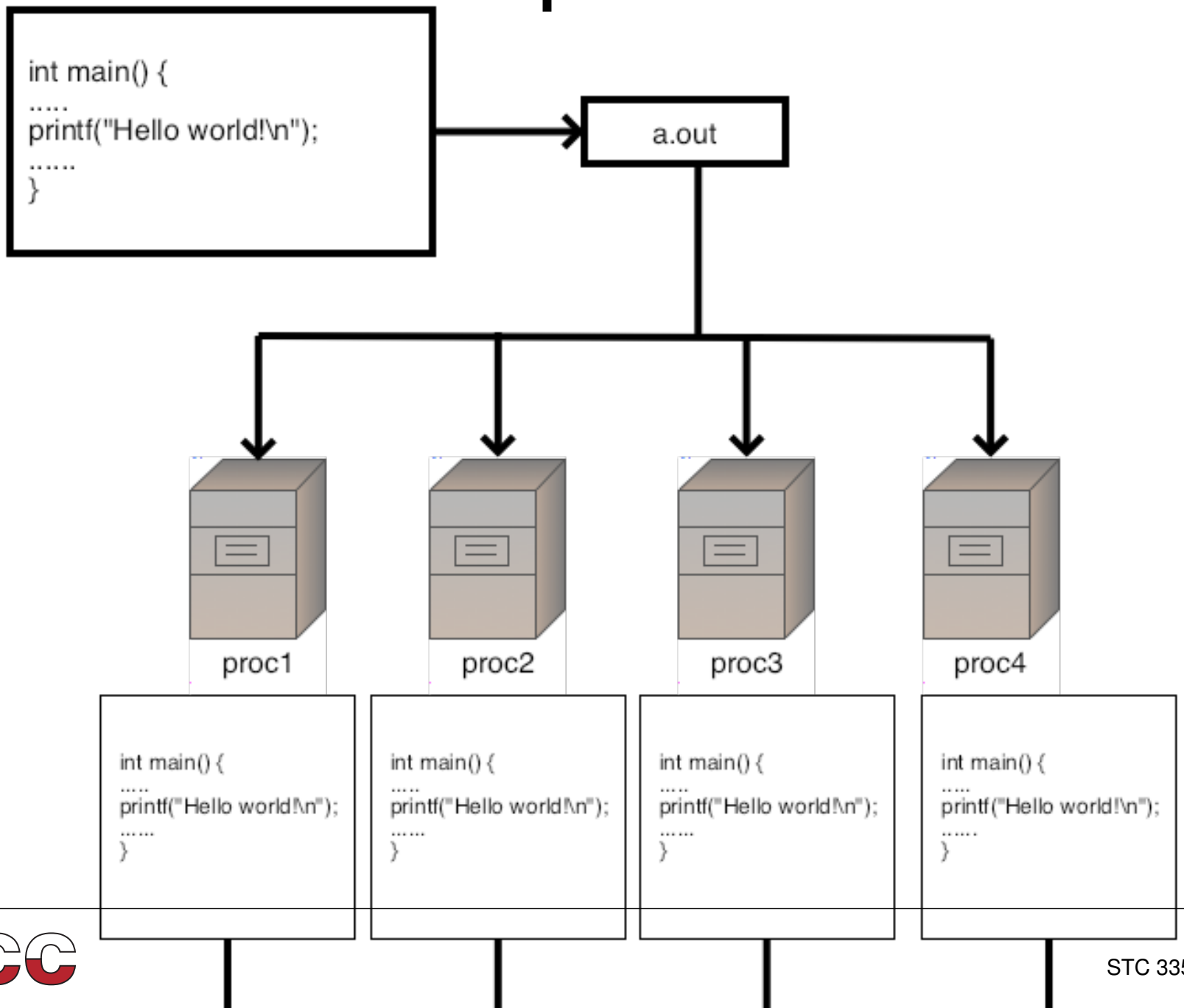
SPMD

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.

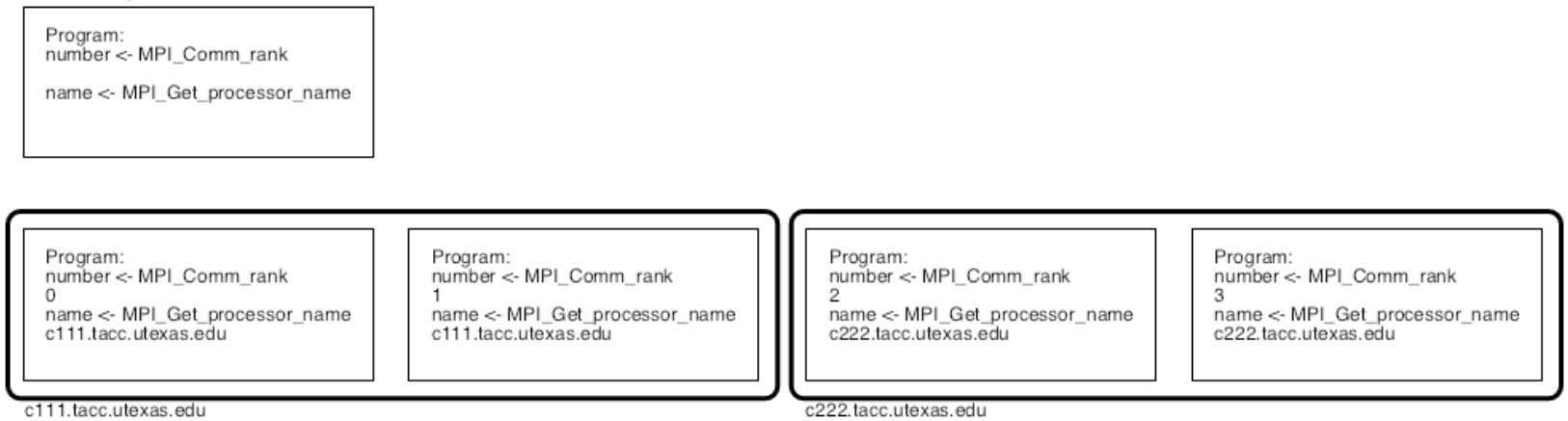
Communication calls do not see the cluster structure:
data sending/receiving is the same for all neighbours.

In a picture


int main() {
.....
printf("Hello world!\n");
.....
}
a.out
proc1
proc2
proc3
proc4
int main() {
.....
printf("Hello world!\n");
.....
}
int main() {
.....
printf("Hello world!\n");
.....
}
int main() {
.....
printf("Hello world!\n");
.....
}
int main() {
.....
printf("Hello world!\n");
.....
}

In a picture

Four processes on two nodes (`idev -N 2 -n 4`)



Sequential semantics

A PETSc program almost looks like sequential:

```
MatMult(A,x,y);       // y <- Ax
VecCopy(y,res);       // r <- y
VecAXPY(res,-1.,b);   // r <- r - b
```

but note all objects are distributed.

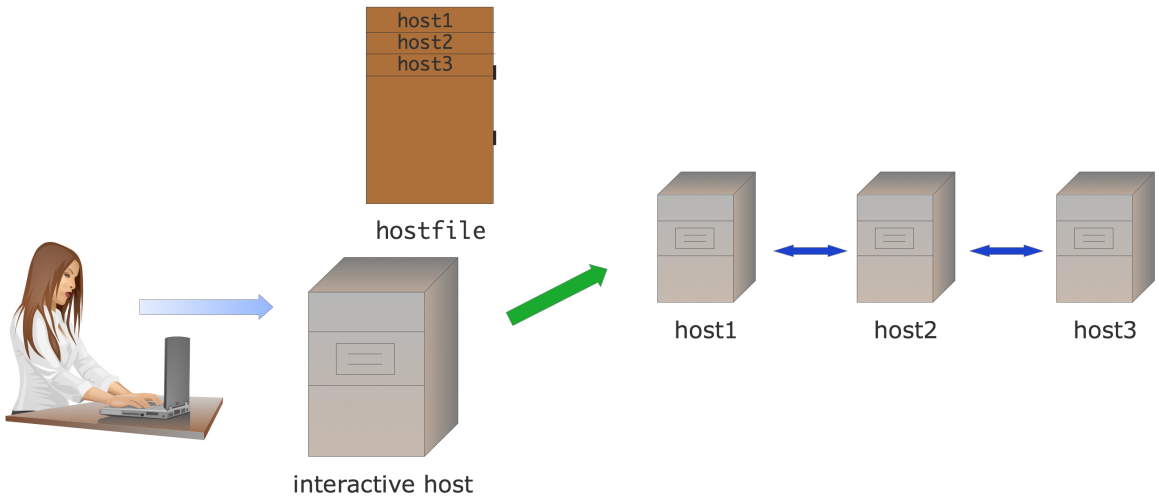
Do I need a supercomputer?

- 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 such as SLURM.

Running at TACC

First:

```
module load petsc/3.12-debug
# later: module load petsc/3.12
```

Compute node for 2 hours, on 2 nodes with 12 cores total:

```
idev -t 2:0:0 -N 2 -n 12
```

If you do this during an official TACC training, there will probably be a reservation in place.

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
  ```

Minimal MPI knowledge

With PETSc you barely need to know MPI.
Here is just about the only thing:

```
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`.

Include files

C:

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

Fortran:

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

Python:

```
from petsc4py import PETSc
```

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(ier)
// all the petsc work
  call PetscFinalize(ierr); CHKERRQ(ier)
```

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`

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)
```

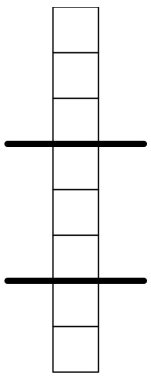
Use:

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

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)

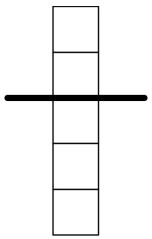
VecSetSizes(V,PETSC_DECIDE,8)

Parallel layout specified

Local or global size in

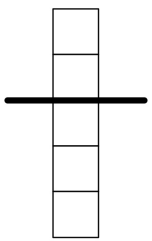
```
VecSetSizes(Vec v, int m, int M);
```

Global size can be specified as `PETSC_DECIDE`.



VecSetSizes(V,2,5)

VecSetSizes(V,3,5)



VecSetSizes(V,2,PETSC_DECIDE)

VecSetSizes(V,3,PETSC_DECIDE)

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,ierr,e)
```

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:

$$p \leftarrow x^t y$$
$$x \leftarrow x/p$$
$$n \leftarrow \|x\|_2$$

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
  $\Rightarrow$ 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
```

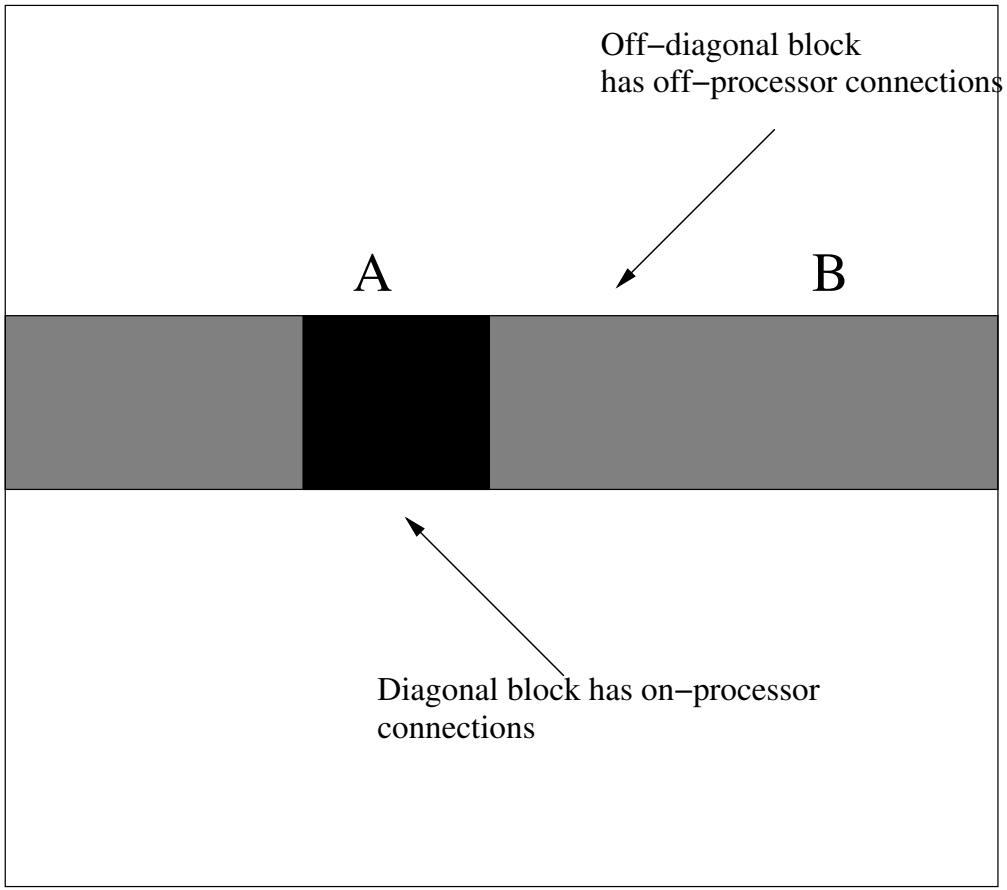
Sequential matrix structure

```
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.)

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);
```

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
- Fargoing 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`

Exercise 4 (`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`

More KSP topics

- Custom monitors and convergence tests
- Method-specific options (especially GMRES)
- Null spaces

Fieldsplit preconditioners

If a problem contains multiple physics, seperate 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

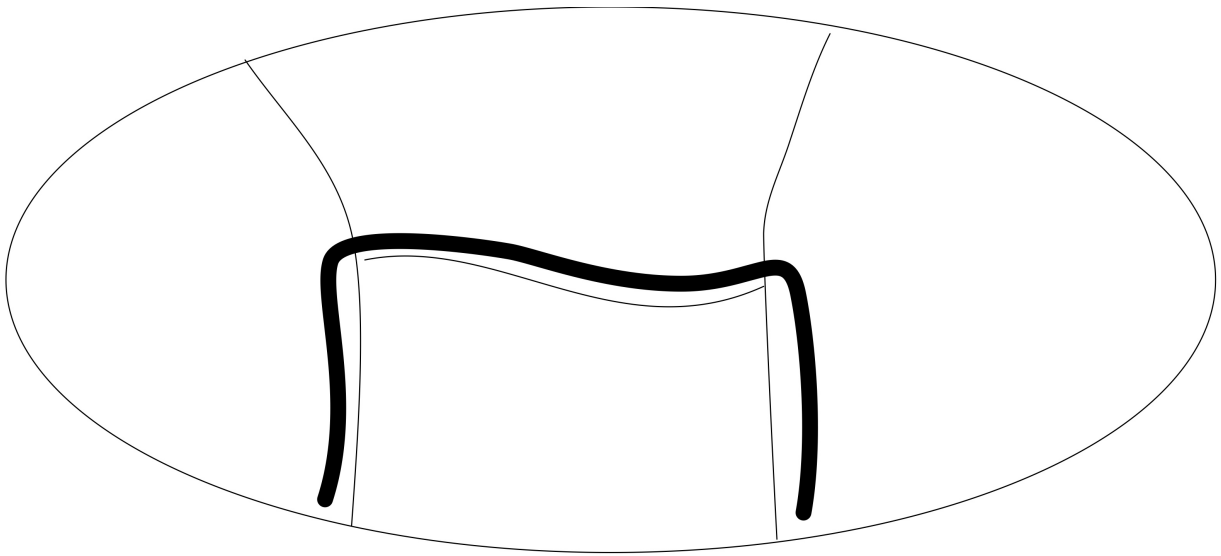
```
PCSetType(PC pc,PCMG);
PCMGSetLevels(pc,int levels,MPI Comm *comms);
PCMGSetType(PC pc,PCMGType mode);
PCMGSetCycleType(PC pc,PCMGCycleType ctype);
PCMGSetNumberSmoothUp(PC pc,int m);
PCMGSetNumberSmoothDown(PC pc,int n);
PCMGGetCoarseSolve(PC pc,KSP *ksp);
PCMGSetInterpolation(PC pc,int level,Mat P); and
PCMGSetRestriction(PC pc,int level,Mat R);
PCMGSetResidual(PC pc,int level,PetscErrorCode
(*residual)(Mat,Vec,Vec,Vec),Mat mat);
```

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`

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`, `ISInvertPermutations` 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);
```

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	26	100	96	17	92	26	100	96	17	92	194

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`