Introduction to PETSc

Victor Eijkhout

Outline

Table of Contents

Sparse matrix from 2D PDE

Two-dimensional: $-u_{xx} - u_{yy} = f$ on unit square $[0,1]^2$

Difference equation:

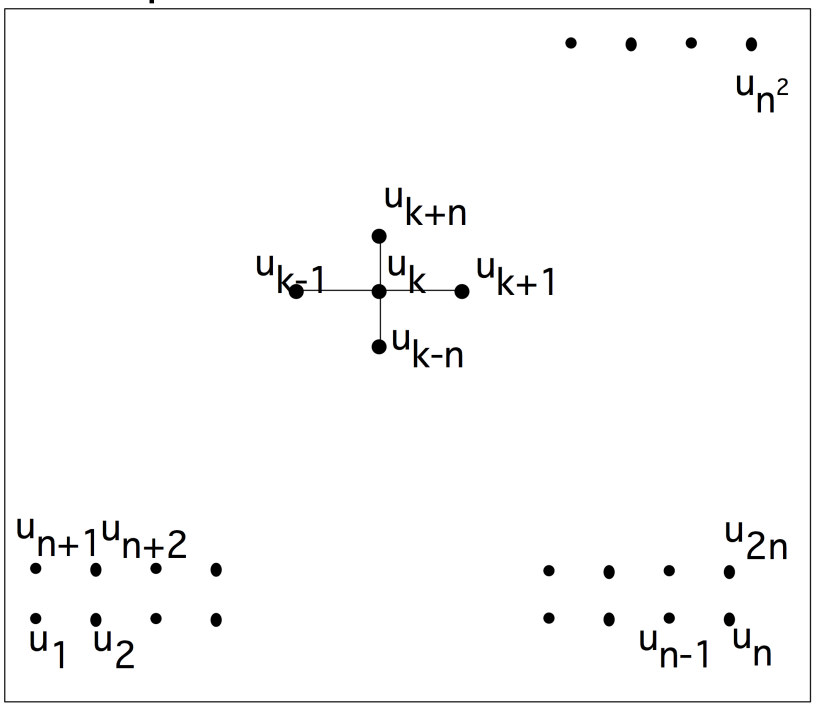
$$4u(x,y) - u(x+h,y) - u(x-h,y) - u(x,y+h) - u(x,y-h) = h^2 f(x,y)$$

$$4u_k - u_{k-1} - u_{k+1} - u_{k-n} - u_{k+n} = f_k$$

Consider a graph where $\{u_k\}_k$ are the edges
and (u_i, u_j) is an edge iff $a_{ij} \neq 0$.

The graph view of things

Poisson eq:



This is a graph!

This is the (adjacency) graph of a sparse matrix.

The matrix view of things

$$
\left(\begin{array}{cccccc|cccccc|ccc}
4 & -1 & & & & \emptyset & -1 & & & & & \emptyset & & & \\
-1 & 4 & 1 & & & & & -1 & & & & & & & \\
 & \ddots & \ddots & \ddots & & & & & \ddots & & & & & & \\
 & & \ddots & \ddots & -1 & & & & & \ddots & & & & & \\
\emptyset & & & -1 & 4 & & \emptyset & & & & & -1 & & & \\
\hline
-1 & & & & & \emptyset & 4 & -1 & & & & & -1 & & \\
 & -1 & & & & & -1 & 4 & -1 & & & & & -1 & \\
 & \uparrow & \ddots & & & & \uparrow & \uparrow & \uparrow & & & & & \uparrow & \\
 & k-n & & & & & k-1 & k & k+1 & & -1 & & & k+n & \\
 & & & & -1 & & & & & -1 & 4 & & & & \\
\hline
 & & & & & & \ddots & & & & & & \ddots & &
\end{array}\right)
$$

LU of a sparse matrix

$$\begin{pmatrix} 4 & -1 & 0 & \dots & & -1 & & \\ -1 & 4 & -1 & 0 & \dots & 0 & -1 & \\ & \ddots & \ddots & \ddots & & & \ddots & \\ -1 & 0 & \dots & & & 4 & -1 & \\ 0 & -1 & 0 & \dots & & -1 & 4 & -1 \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} 4 & -1 & 0 & \dots & & -1 & & \\ & 4-\frac{1}{4} & -1 & 0 & \dots & -1/4 & -1 & \\ & \ddots & \ddots & \ddots & & & \ddots & \\ & -1/4 & \dots & & & 4-\frac{1}{4} & -1 & \\ & -1 & 0 & \dots & & -1 & 4 & -1 \end{pmatrix}$$

Fill-in during LU

2D BVP: Ω is $n \times n$, gives matrix of size $N = n^2$, with bandwidth n.

Matrix storage $O(N)$

LU storage $O(N^{3/2})$ (limited to band)

LU factorization work $O(N^2)$

Complicated recurrences hard to parallelize

Table of Contents

What is PETSc? Why should you use PETSc?

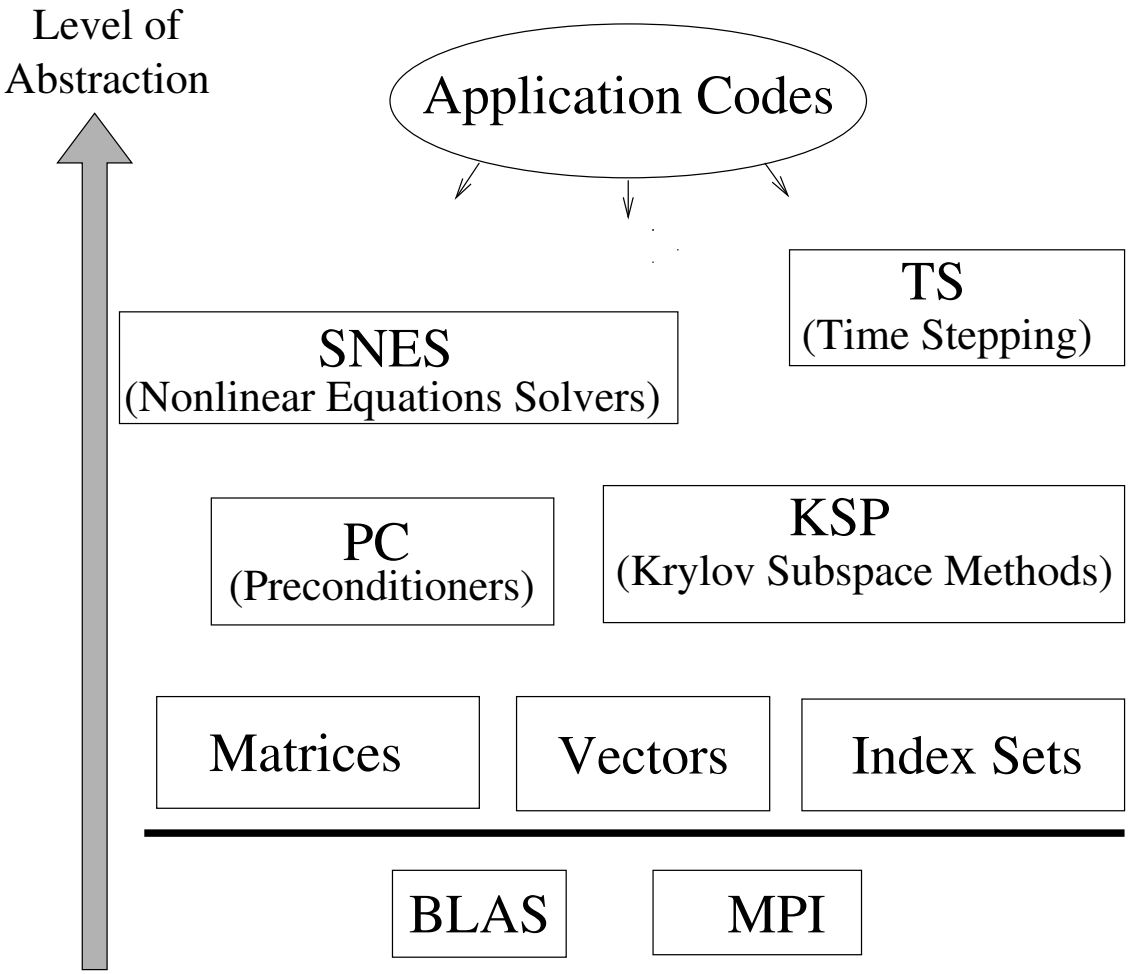
Portable Extendable Toolkit for Scientific Computations

- 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 is in PETSc?

- Linear system solvers (sparse/dense, iterative/direct)
- Nonlinear system solvers
- Tools for distributed matrices
- Support for profiling, debugging, graphical output


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

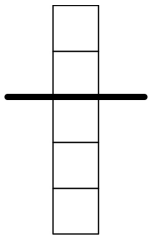
Table of Contents

Parallel layout

Local or global size in

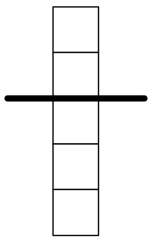
```
VecSetSizes(Vec v, int m, int M);
```

Global size can be specified as `PETSC_DECIDE`.



VecSetSizes(V,2,5)

VecSetSizes(V,3,5)



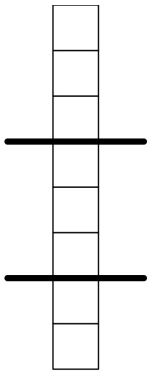
VecSetSizes(V,2,PETSC_DECIDE)

VecSetSizes(V,3,PETSC_DECIDE)

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)

Setting values

Set vector to constant value:

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

Set individual elements (global indexing!):

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

i = 1; v = 3.14;
VecSetValues(x,1,&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 VecSetValues(x,1,i,v,INSERT_VALUES,ierr,e)
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);
```

`Mat` Datatype: matrix

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)

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
- (run your code with `-memory_info`, `-malloc_log`)

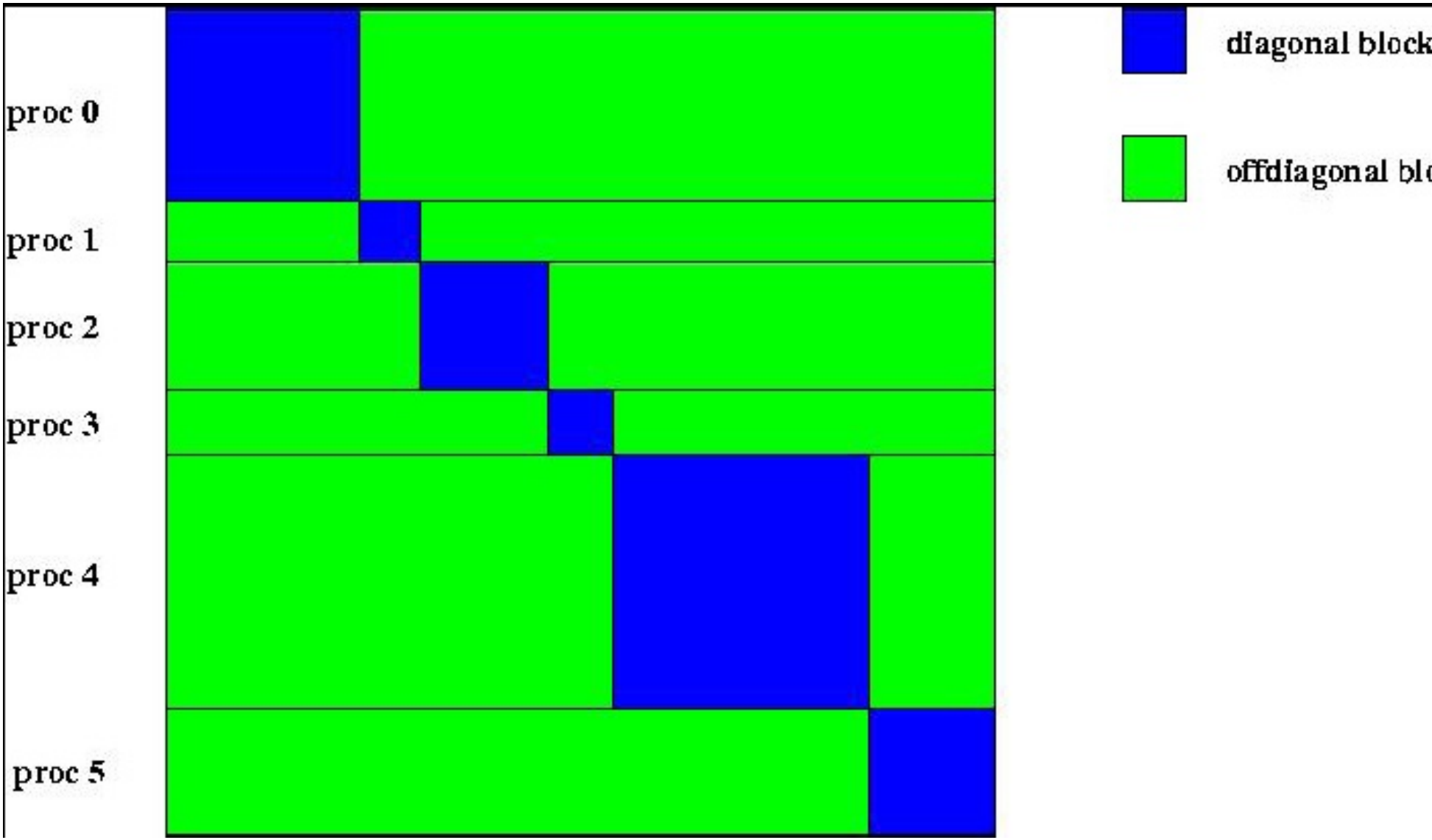
malloc is very expensive:
tell PETSc the matrix' sparsity structure
(do construction loop twice: once counting, once making)

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!

Parallel matrix structure



Parallel matrix structure description

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

- `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

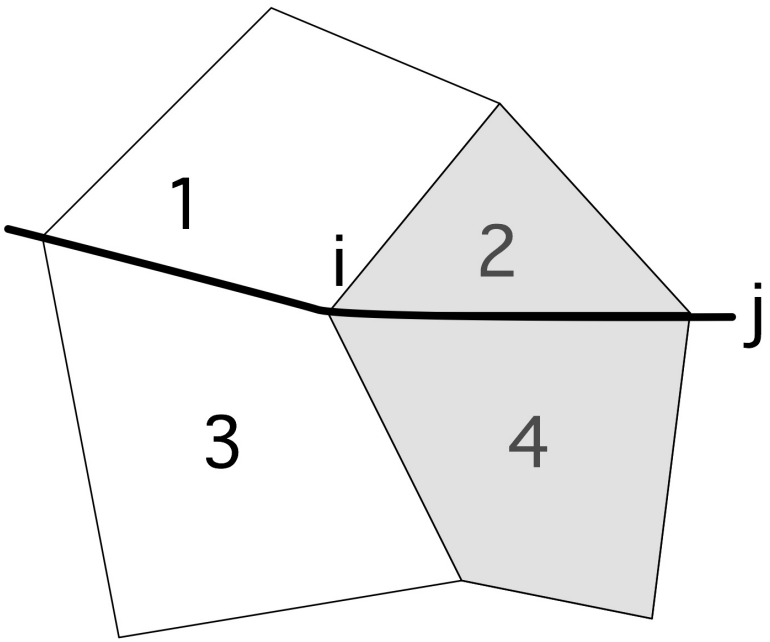
In Fortran use `PETSC_NULL_INTEGER` if not specifying arrays

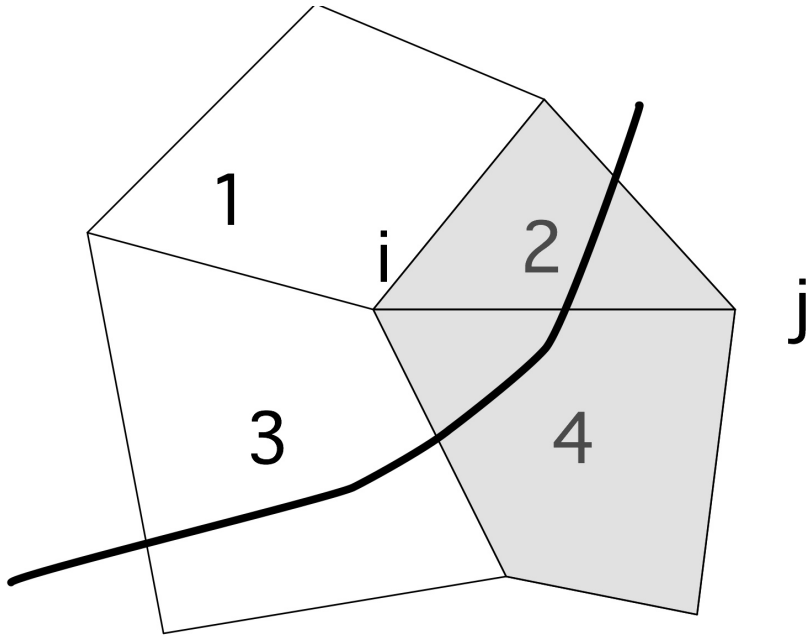
Setting matrix elements

▸ Recall laplace graph

```
MatGetOwnershipRange(A,&low,&high);
for ( i=0; i<m; i++ ) {
  for ( j=0; j<n; j++ ) {
    I = j + n*i;
    if (I>=low && I<high) { // my row!
      J = I-1; v = -1.0;
      if (i>0) MatSetValues
          (A,1,&I,1,&J,&v,INSERT_VALUES);
      J = I+1 // et cetera
    }
  }
}
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

Finite Element Matrix assembly





Finite Element Matrix assembly

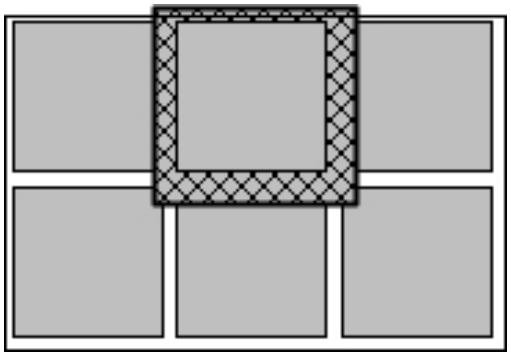
```
for (e=myfirstelement; e<mylastelement; e++) {
  for (i=0; i<nlocalnodes; i++) {
    I = localtoglobal(e,i);
    for (j=0; j<nlocalnodes; j++) {
      J = localtoglobal(e,j);
      v = integration(e,i,j);
      MatSetValues
          (mat,1,&I,1,&J,&v,ADD_VALUES);
      ....
    }
  }
}
MatAssemblyBegin(mat,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(mat,MAT_FINAL_ASSEMBLY);
```

Matrix usage

Very important:

```
MatMult(Mat mat,Vec x,Vec y)
MatMultTranspose(Mat mat,Vec x,Vec y)
MatMultAdd(Mat mat,Vec v1,Vec v2,Vec v3)
```

Sparse matrix vector product induces halo region;



Product $y \leftarrow Ax$ becomes

$$y \leftarrow A_{\text{local}} x_{\text{local}} + A_{\text{remote}} x_{\text{remote}}$$

▸ Recall matrix structure

- Start communication for halo
- Do local computation
- Wait for completion of halo transfer
- Do remote computation

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);
```

Table of Contents

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 $B \approx A^{-1}$

General form of iterative methods

$$\begin{cases} r_i = Ax_i - b \\ x_{i+1}\gamma_{i+1,i} = K^{-1} r_i + \sum_{j \le i} x_j \gamma_{ji} \\ r_{i+1}\gamma_{i+1,i} = AK^{-1} r_i + \sum_{j \le i} r_j \gamma_{ji} \end{cases}$$

where $\gamma_{i+1,i} = \sum_{j \le i} \gamma_{ji}$.

Conjugate Gradients

Basic idea:

$$r_i^t K^{-1} r_j = 0 \quad \text{if } i \neq j.$$

Split recurrences:

$$\begin{cases} x_{i+1} = x_i - \delta_i p_i \\ r_{i+1} = r_i - \delta_i A p_i \\ p_i = K^{-1} r_i + \gamma_i p_{i-1}, \end{cases}$$

where

$$\delta_i = \frac{r_i^t K^{-1} r_i}{p_i^t A p_i}, \quad \gamma_i = \frac{r_i^t K^{-1} r_i}{r_{i-1}^t K^{-1} r_{i-1}}$$

Components of every iterative method

- Matrix-vector product
- Preconditioner (construction and application)
- Inner products
- Other vector operations.

Basic concepts

- All linear solvers in PETSc are iterative (see below)
- Object oriented: solvers only need matrix action, so can handle shell matrices
- Preconditioners
- Fargoing control through commandline options
- Tolerances, convergence and divergence reason
- Custom monitors and convergence tests

KSP: Krylov Space objects

Iterative solver basics

```
KSPCreate(comm,&solver);
KSPSetOperators(solver,A,B);
/* optional */ KSPSetup(solver);
KSPSolve(solver,rhs,sol);
KSPDestroy(solver);
```

Settings in general

- Other settings, both as command and runtime option
- option `-ksp_view`
- (preconditioners discussed below)

Solver type and tolerances

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

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`
- tolerances: `-ksp_max_it 50`

Convergence

Iterative solvers can fail

- Solve call itself gives no feedback: solution may be completely wrong
- `KSPGetConvergedReason(solver,&reason)` :
  positive is convergence, negative divergence
  (`${PETSC_DIR}/include/petscksp.h` for list)
- `KSPGetIterationNumber(solver,&nits)` : after how many iterations did the method stop?

PC: Preconditioner objects

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);
```

- `PCJACOBI`, `PCILU` (only sequential), `PCASM`, `PCBJACOBI`, `PCMG`, et cetera
- Controllable through commandline options:

```
-pc_type ilu -pc_factor_levels 3
```

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_solve_package mumps
```

Table of Contents

Running

Parallel invocation. On your own machine:
`mpirun -np 3 petscprog <bunch of runtime options>`
On stampede and such: using `ibrun`

Petsc has lots of runtime options.

- `-log_summary` : give runtime statistics
- `-malloc_dump -memory_info` : memory statistics
- `-start_in_debugger` : parallel debugging (not on our clusters, but very convenient on your laptop)
- `-options_left` : check for mistyped options
- `-ksp_type gmres` (et cetera) : program control

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

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

Commandline options, C

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

Read commandline argument, print out from processor zero

Commandline options, F

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

Note the `PETSC_NULL_CHARACTER`, note that `PetscPrintf` has only one string argument

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	26	100	96	17	92	26	100	96	17	92	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);
```