PETSc

Portable, Extensible Toolkit for Scientific Computation

Karl Rupp
rupp@mcs.anl.gov

Mathematics and Computer Science Division
Argonne National Laboratory

Tutorial at Segundo Encuentro Nacional de Computación
de Alto Rendimiento para Aplicaciones Científicas

May 7th, 2013

Argonne NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

Before we start...

Goal of this Workshop

You are here to learn new things about HPC

Ask Questions

Tell me if you do not understand

Ask for further details

Don’t be shy

Table of Contents

p-Bratu Equation

The $\mathfrak{p}$-Bratu Equation

The "Hello World of PDEs"

Poisson's Equation

$$-\nabla \cdot \left(\nabla u\right) = f,$$

Leads to symmetric, positive definite system matrices

Commonly used in numerical analysis (corner effects, etc.)

More General Form

With diffusivity tensor η:

$$-\nabla \cdot \left(\eta \nabla u\right) = f,$$

Typically: $\eta > \delta > 0$

η can be discontinous (material boundaries)

Reduced regularity of solution

The $\mathfrak{p}$-Bratu Equation

Additional Volume Term

Consider

$$-\nabla \cdot \left(\eta \nabla u\right) - \lambda e^{u} - f = 0 \ ,$$

Canonical nonlinear form

e^{u} has "wrong sign": turning point at λ_{crit}

Another Tweak

Diffusivity tensor η depends on u, e.g.:

$$\eta = \frac{1}{2}|\nabla u|^{2},$$

Singular or degenerate when $\nabla u = 0$.

The $\mathfrak{p}$-Bratu Equation

$\mathfrak{p}$-Bratu Equation

2-dimensional model problem

$$-\nabla \cdot \left(|\nabla u|^{\mathfrak{p}-2}\nabla u\right) - \lambda e^u - f = 0, \qquad 1 \leq \mathfrak{p} \leq \infty, \quad \lambda < \lambda_{\text{crit}}(\mathfrak{p})$$

Singular or degenerate when $\nabla u = 0$, turning point at λ_{crit}.

Regularized Variant

Remove singularity of η using a parameter ε:

$$-\nabla \cdot (\eta \nabla u) - \lambda e^u - f = 0$$

$$\eta(\gamma) = (\epsilon^2 + \gamma)^{\frac{\mathfrak{p}-2}{2}} \qquad \gamma(u) = \frac{1}{2}|\nabla u|^2$$

Physical interpretation: diffusivity tensor flattened in direction ∇u

Distributed Arrays

Distributed Array

Interface for topologically structured grids

Defines (topological part of) a finite-dimensional function space

Get an element from this space: `DMCreateGlobalVector()`

Provides parallel layout

Refinement and coarsening

`DMRefineHierarchy()`

Ghost value coherence

`DMGlobalToLocalBegin()`

Matrix preallocation

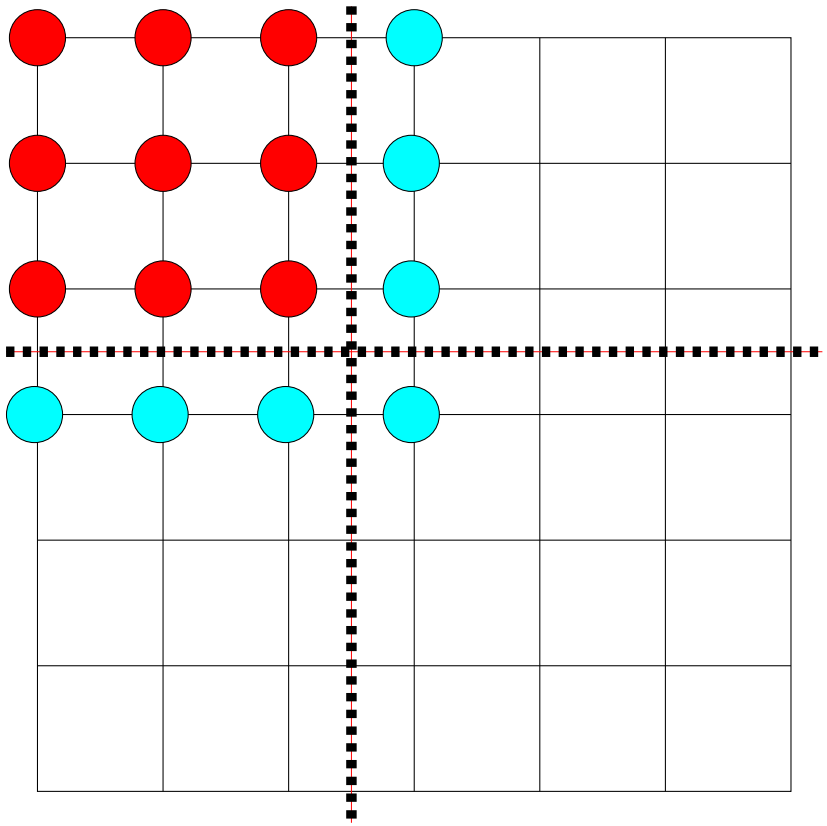
`DMCreateMatrix()` (formerly `DMGetMatrix()`)

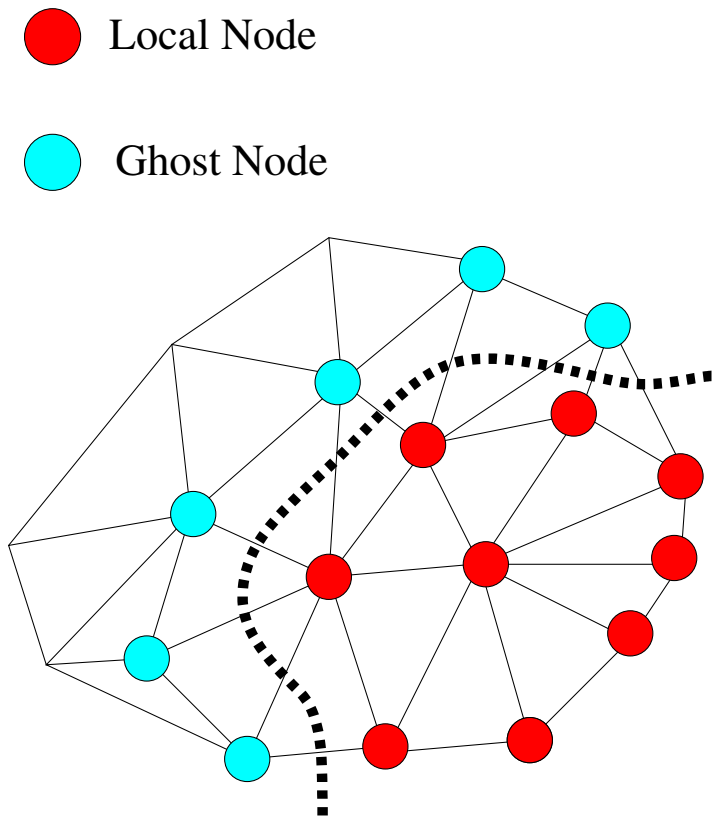
Ghost Values

To evaluate a local function $f(x)$, each process requires

its local portion of the vector x

its ghost values, bordering portions of x owned by neighboring processes





DMDA Global Numberings

Proc 2			Proc 3	
25	26	27	28	29
20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4
Proc 0			Proc 1	

Natural numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

PETSc numbering

DMDA Global vs. Local Numbering

Global: Each vertex has a unique id, belongs on a unique process

Local: Numbering includes vertices from neighboring processes

These are called ghost vertices

Proc 2			Proc 3	
X	X	X	X	X
X	X	X	X	X
12	13	14	15	X
8	9	10	11	X
4	5	6	7	X
0	1	2	3	X
Proc 0			Proc 1	

Local numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

Global numbering

DM Vectors

The DM object contains only layout (topology) information

All field data is contained in PETSc `Vec`s

Global vectors are parallel

Each process stores a unique local portion

```
DMCreateGlobalVector(DM dm, Vec *gvec)
```

Local vectors are sequential (and usually temporary)

Each process stores its local portion plus ghost values

```
DMCreateLocalVector(DM dm, Vec *lvec)
```

includes ghost values!

Coordinate vectors store the mesh geometry

```
DMDAGetCoordinates(DM dm, Vec *coords)
```

Can be manipulated with their own DMDA

```
DMDAGetCoordinateDA(DM dm,DM *cda)
```

Updating Ghosts

Two-step Process for Updating Ghosts

enables overlapping computation and communication

`DMGlobalToLocalBegin(dm, gvec, mode, lvec)`

`gvec` provides the data

`mode` is either `INSERT_VALUES` or `ADD_VALUES`

`lvec` holds the local and ghost values

`DMGlobalToLocalEnd(dm, gvec, mode, lvec)`

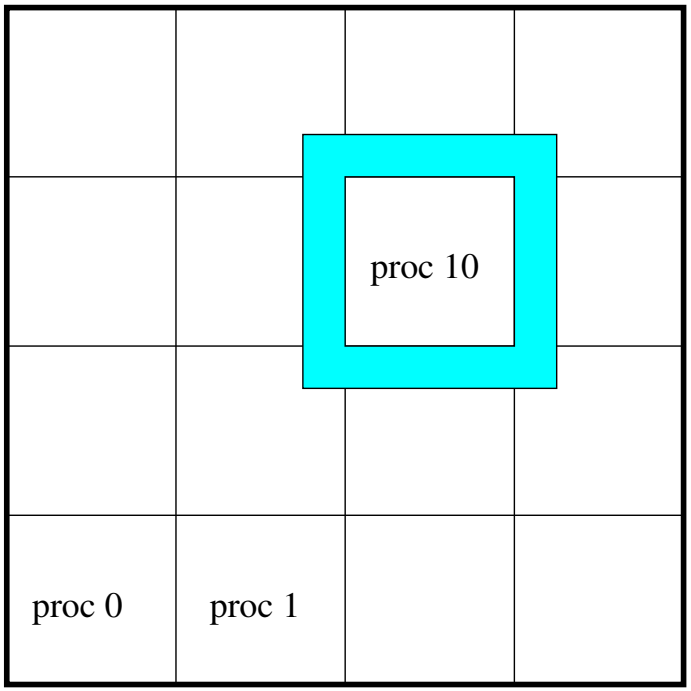
Finishes the communication

Reverse Process

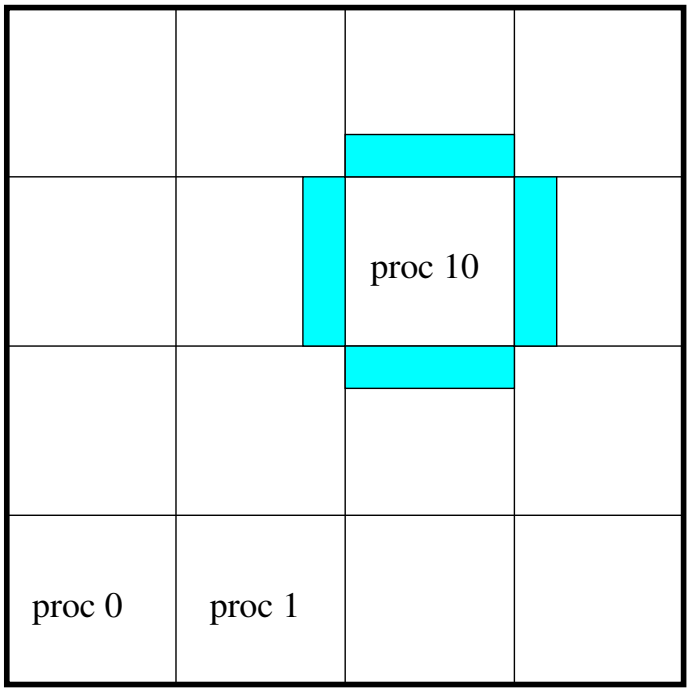
Via `DMLocalToGlobalBegin()` and `DMLocalToGlobalEnd()`.

DMDA Stencils

Available Stencils



Box Stencil



Star Stencil

Creating a DMDA

```
DMDACreate2d(comm, xbdy, ybdy, type, M, N, m, n,
             dof, s, lm[], ln[], DA *da)
```

`xbdy,ybdy`: Specifies periodicity or ghost cells

`DMDA_BOUNDARY_NONE`, `DMDA_BOUNDARY_GHOSTED`, `DMDA_BOUNDARY_MIRROR`, `DMDA_BOUNDARY_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

`lm,ln`

Alternative array of local sizes

Use `NULL` for the default

Working with the Local Form

Wouldn't it be nice if we could just write our code for the natural numbering?

Proc 2			Proc 3	
25	26	27	28	29
20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4
Proc 0			Proc 1	

Natural numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

PETSc numbering

Working with the Local Form

Wouldn't it be nice if we could just write our code for the natural numbering?

Yes, that's what `DMDAVecGetArray()` is for.

DMDA offers *local* callback functions

`FormFunctionLocal()`, set by `DMDASetLocalFunction()`

`FormJacobianLocal()`, set by `DMDASetLocalJacobian()`

Evaluating the nonlinear residual $F(x)$

Each process evaluates the local residual

PETSc assembles the global residual automatically

Uses `DMLocalToGlobal()` method

Thinking of Extensions

Multiple Unknowns per Grid Node

Example 1: Displacements u_x, u_y

Example 2: Velocity components, Pressure

Typical in a multiphysics setting

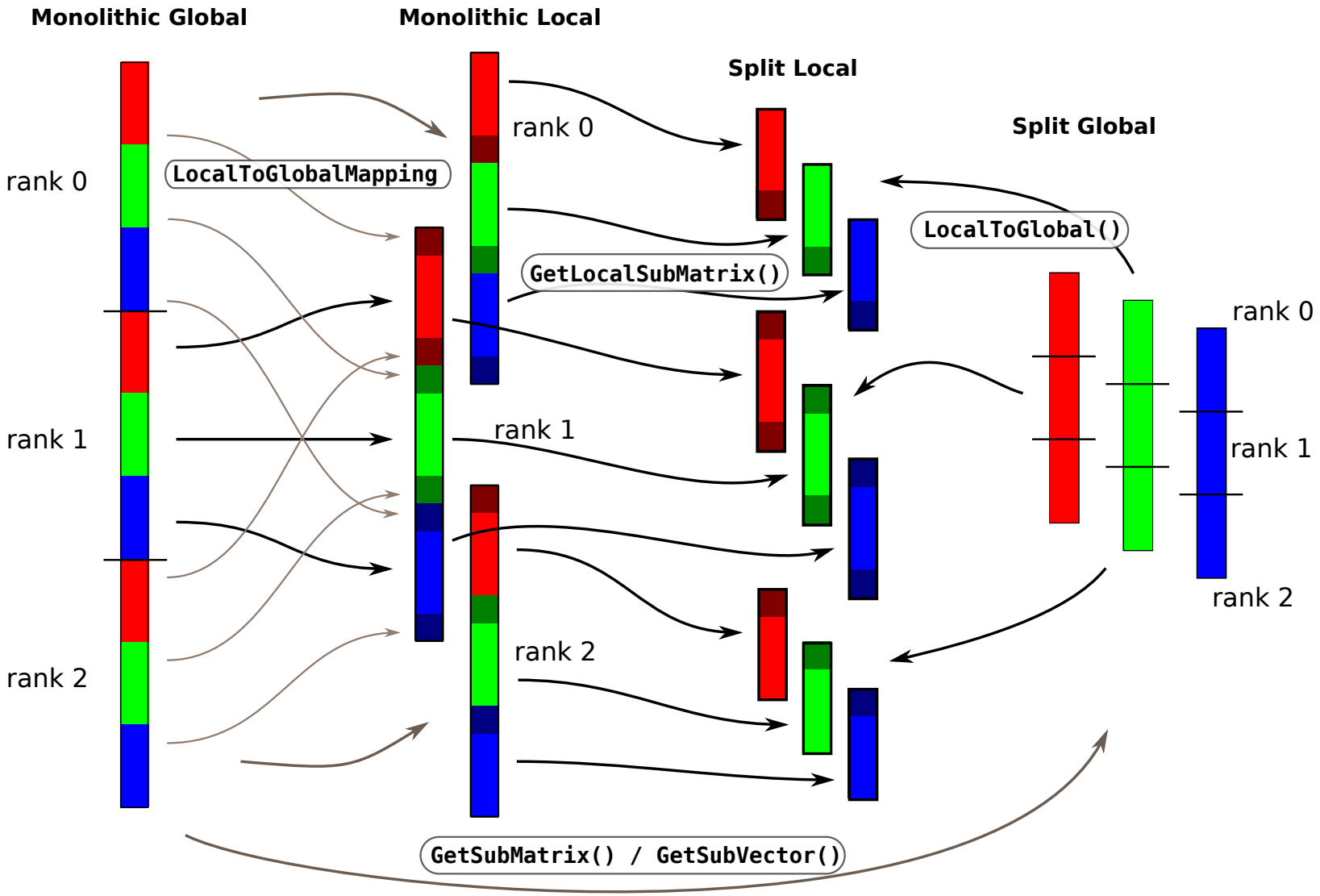
Multiple Unknowns in a Distributed Setting

Robust abstract concepts important

Lots of bookkeeping

All done by PETSc

Thinking of Extensions



DA Local Function

User-provided Function for Nonlinear Residual in 2D

```
PetscErrorCode (*lfunc)(DMDALocalInfo *info,
                        Field **x, Field **r,
                        void *ctx)
```

`info`	All layout and numbering information
`x`	The current solution *Notice that it is a multidimensional array*
`r`	The residual
`ctx`	The user context passed to `DMSetApplicationContext()` or to SNES

The local DMDA function is activated by calling

```
SNESSetDM(snes,dm)
SNESSetFunction(snes, r, SNESDAFormFunction, ctx)
```

Discretization

Mapping PDEs to a (un)structured Grid

- Can be arbitrarily complex (mathematically)
- Neverending area of research

Popular Discretization Schemes

- Finite Difference Method
- Finite Volume Method
- Finite Element Method

Finite Difference Methods: u'

Consider 1d-grid

Replace $u' \approx \frac{u[i+1]-u[i]}{h}$

or $u' \approx \frac{u[i]-u[i-1]}{h}$

or $u' \approx \frac{u[i+1]-u[i-1]}{2h}$

Finite Difference Methods: u''

Naive: $u'' \approx \frac{u'[i+1]-u'[i-1]}{2h} \approx \frac{u[i+2]-2u[i]+u[i-2]}{4h^2}$

Use 'virtual' grid nodes $u'[i+0.5]$, $u'[i-0.5]$ to obtain

$$u''(x_i) \approx \frac{u[i+1]-2u[i]+u[i-1]}{h^2}$$

Finite Volume Methods

- Suitable for unstructured grids
- Popular for conservation laws
- Integrate PDE over box, apply Gauss' theorem
- On regular grid: (Almost) same expression as finite differences

Finite Element Methods

Ansatz: $u \approx \sum_i u_i \varphi_i$

φ_i piecewise polynomials of degree p

Solve for u_i

Adaptivity: in h and/or p possible

Rich mathematical theory

$\mathfrak{p}$-Bratu Residual Evaluation

PETSc-User-Code for $\mathfrak{p}$-Bratu Residual Equation

$$-\Delta u - \lambda e^u = 0$$

```
BratuResidualLocal(DMDALocalInfo *info,
                   Field **x,Field **f,
                   UserCtx *user)
{
  /* Not Shown: Handle boundaries */
  /* Compute over the interior points */
  for(j = info->ys; j < info->ys+info->ym; j++) {
    for(i = info->xs; i < info->xs+info->xm; i++) {
      u         = x[j][i];
      u_xx      = (2.0*u - x[j][i-1] - x[j][i+1])*hydhx;
      u_yy      = (2.0*u - x[j-1][i] - x[j+1][i])*hxdhy;
      f[j][i] = u_xx + u_yy - hx*hy*lambda*exp(u);
    }
  }
}
```

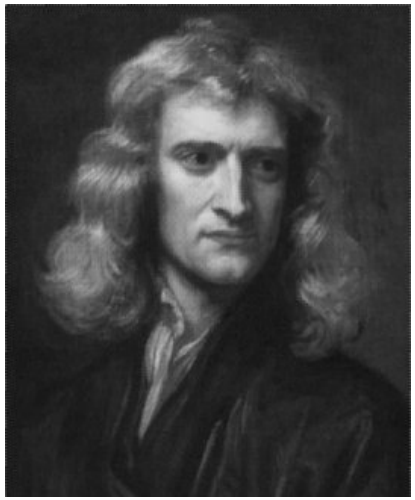
$PETSC_DIR/src/snes/examples/tutorials/ex15.c

Nonlinear Solvers

Newton iteration: Workhorse of SNES

Standard form of a nonlinear system

$$-\nabla \cdot \left(|\nabla u|^{\mathfrak{p}-2}\nabla u\right) - \lambda e^{u} = F(u) = 0$$



Iteration

Solve: $J(u)w = -F(u)$

Update: $u^{+} \leftarrow u + w$

Quadratically convergent near a root: $|u^{n+1} - u^{*}| \in \mathcal{O}\left(|u^{n} - u^{*}|^{2}\right)$

Picard is the same operation with a different $J(u)$

Jacobian Matrix for $\mathfrak{p}$-Bratu Equation

$$J(u)w \sim -\nabla\left[(\eta \mathbf{1} + \eta' \nabla u \otimes \nabla u)\nabla w\right] - \lambda e^{u} w$$

$$\eta' = \frac{\mathfrak{p}-2}{2}\eta/(\epsilon^{2} + \gamma)$$

Scalable Nonlinear Equation Solvers

Newton solvers: Line Search, Thrust Region

Inexact Newton-methods: Newton-Krylov

Matrix-Free Methods: With iterative linear solvers

How to get the Jacobian Matrix?

Implement it by hand

Let PETSc finite-difference it

Use Automatic Differentiation software

Nonlinear solvers in PETSc SNES

LS, TR	Newton-type with line search and trust region
NRichardson	Nonlinear Richardson, usually preconditioned
VIRS, VISS	reduced space and semi-smooth methods for variational inequalities
QN	Quasi-Newton methods like BFGS
NGMRES	Nonlinear GMRES
NCG	Nonlinear Conjugate Gradients
GS	Nonlinear Gauss-Seidel/multiplicative Schwarz sweeps
FAS	Full approximation scheme (nonlinear multigrid)
MS	Multi-stage smoothers, often used with FAS for hyperbolic problems
Shell	Your method, often used as a (nonlinear) preconditioner

SNES Paradigm

SNES Interface based upon Callback Functions

`FormFunction()`, set by `SNESSetFunction()`

`FormJacobian()`, set by `SNESSetJacobian()`

Evaluating the nonlinear residual $F(x)$

Solver calls the **user's** function

User function gets application state through the `ctx` variable

PETSc *never* sees application data

SNES Function

$$F(u) = 0$$

The user provided function which calculates the nonlinear residual has signature

```
PetscErrorCode (*func)(SNES snes,
                       Vec x,Vec r,
                       void *ctx)
```

`x` - The current solution

`r` - The residual

`ctx` - The user context passed to `SNESSetFunction()`

Use this to pass application information, e.g. physical constants

SNES Jacobian

User-provided function calculating the Jacobian Matrix

```
PetscErrorCode (*func)(SNES snes,Vec x,Mat *J,Mat *M,
                       MatStructure *flag,void *ctx)
```

`x` - The current solution

`J` - The Jacobian

`M` - The Jacobian preconditioning matrix (possibly J itself)

`ctx` - The user context passed to `SNESSetFunction()`

Use this to pass application information, e.g. physical constants

Possible `MatStructure` values are:

`SAME_NONZERO_PATTERN`
`DIFFERENT_NONZERO_PATTERN`

Alternatives

a builtin sparse finite difference approximation (“coloring”)

automatic differentiation (ADIC/ADIFOR)

Finite Difference Jacobians

PETSc can compute and explicitly store a Jacobian

- Dense
  - Activated by `-snes_fd`
  - Computed by `SNESDefaultComputeJacobian()`
- Sparse via colorings
  - Coloring is created by `MatFDColoringCreate()`
  - Computed by `SNESDefaultComputeJacobianColor()`

Also Matrix-free Newton-Krylov via 1st-order FD possible

- Activated by `-snes_mf` without preconditioning
- Activated by `-snes_mf_operator` with user-defined preconditioning
  - Uses preconditioning matrix from `SNESSetJacobian()`

Fusing Distributed Arrays and Nonlinear Solvers

Make DM known to SNES solver

```
SNESSetDM(snes,dm);
```

Attach residual evaluation routine

```
DMDASNESSetFunctionLocal(dm,INSERT_VALUES,
       (DMDASNESFunction)FormFunctionLocal,
                         &user);
```

Ready to Roll

First solver implementation completed

Uses finite-differencing to obtain Jacobian Matrix

Rather slow, but scalable!

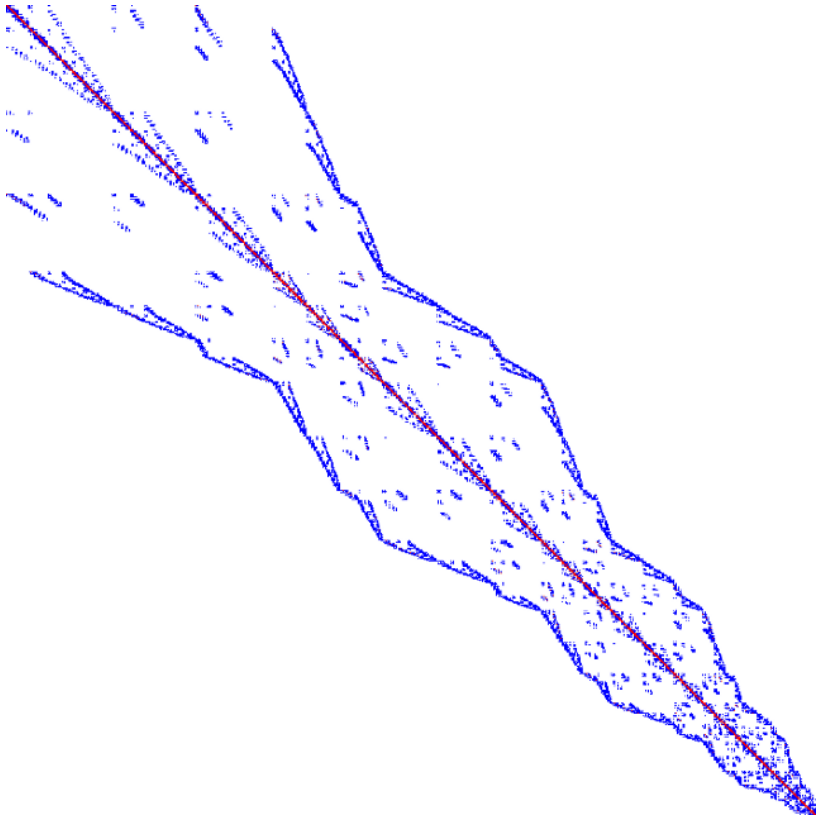
Matrices

Sparse Matrices

The important data type when solving PDEs

Two main phases:

- Filling with entries (assembly)
- Application of its action (e.g. SpMV)



Matrix Memory Preallocation

PETSc sparse matrices are dynamic data structures

- can add additional nonzeros freely

Dynamically adding many nonzeros

- requires additional memory allocations
- requires copies
- can kill performance

Memory preallocation provides

- the freedom of dynamic data structures
- good performance

Easiest solution is to replicate the assembly code

- Remove computation, but preserve the indexing code
- Store set of columns for each row

Call preallocation routines for all datatypes

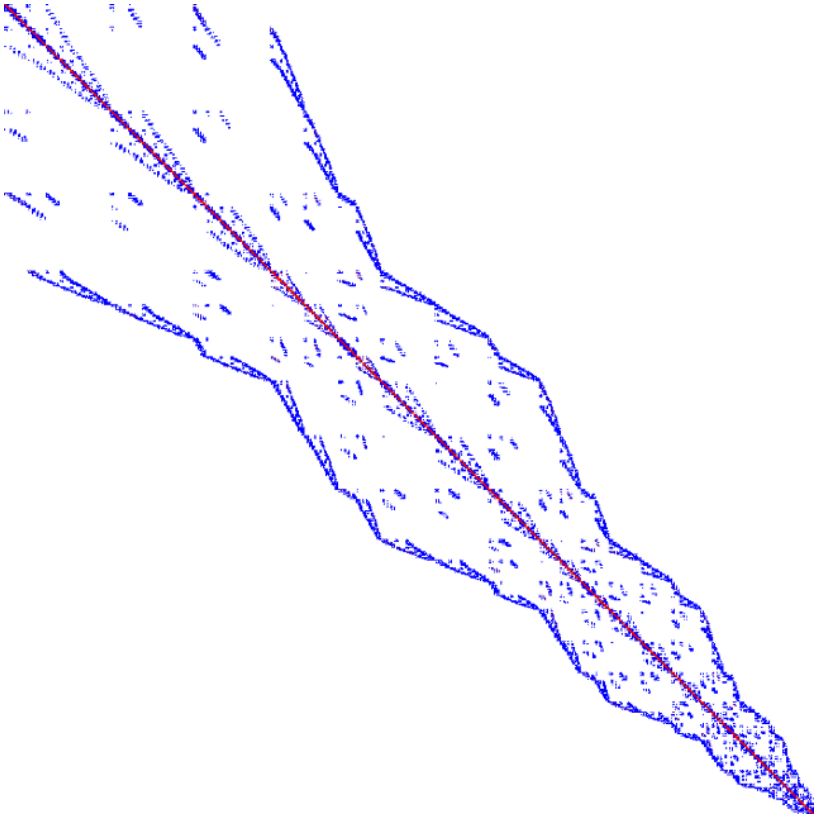
- `MatSeqAIJSetPreallocation()`
- `MatMPIBAIJSetPreallocation()`
- Only the relevant data will be used

Sequential Sparse Matrices

```
MatSeqAIJSetPreallocation(Mat A, int nz, int nnz[])
```

nz: expected number of nonzeros in any row

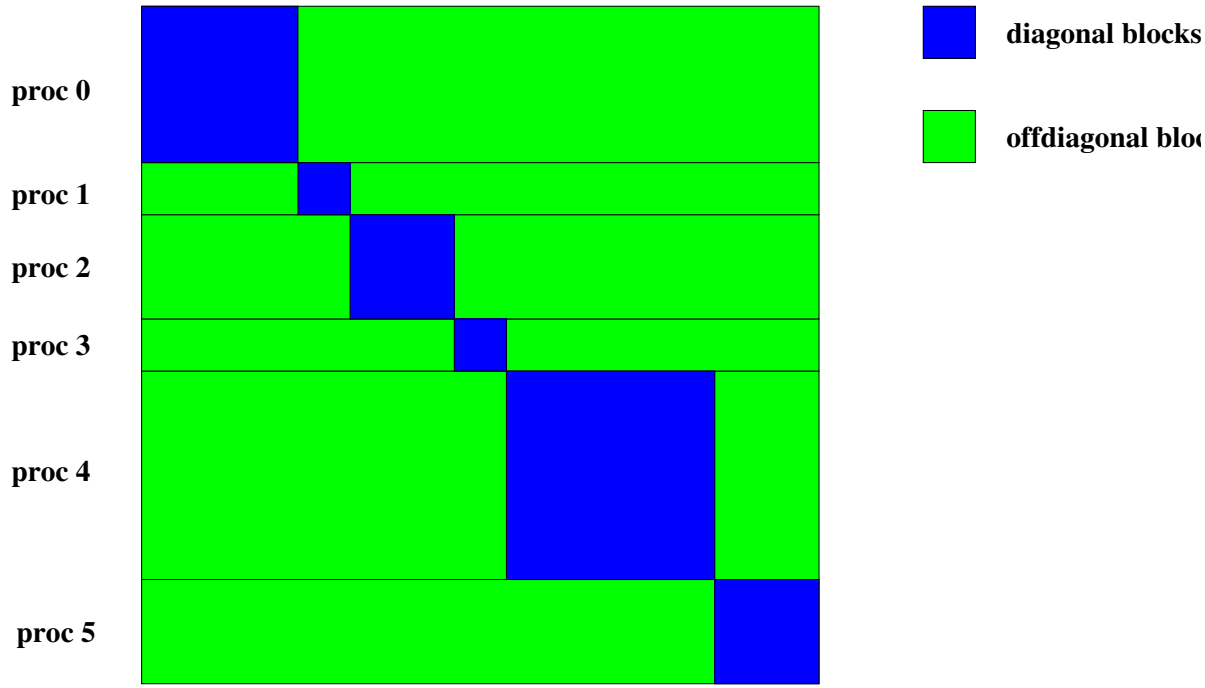
nnz(i): expected number of nonzeros in row i



Parallel Sparse Matrix

Each process locally owns a submatrix of contiguous global rows

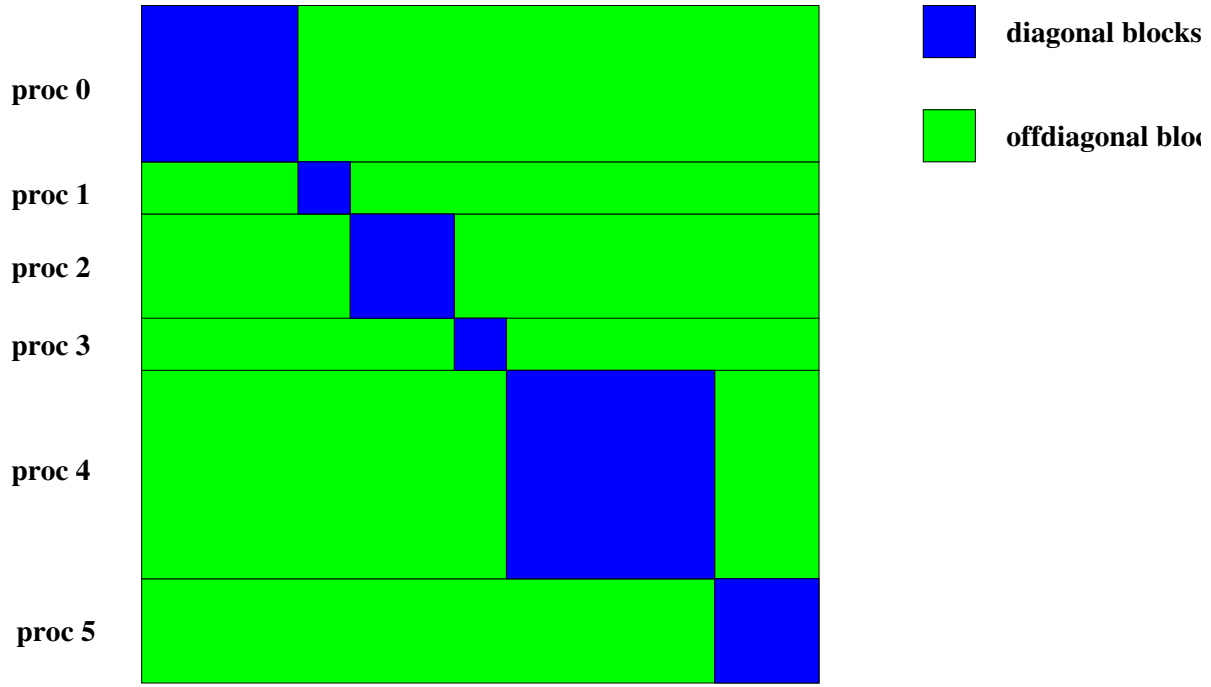
Each submatrix consists of diagonal and off-diagonal parts




```
MatGetOwnershipRange(Mat A,int *start,int *end)
```

`start`: first locally owned row of global matrix
`end-1`: last locally owned row of global matrix

PETSc Application Integration



Proc 2			Proc 3	
25	26	27	28	29
20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4
Proc 0			Proc 1	

Natural numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

PETSc numbering

Parallel Sparse Matrix

```
MatMPIAIJSetPreallocation(Mat A, int dnz, int dnnz[],
                                 int onz, int onnz[]
```

dnz: expected number of nonzeros in any row in the diagonal block

dnnz(i): expected number of nonzeros in row i in the diagonal block

onz: expected number of nonzeros in any row in the offdiagonal portion

onnz(i): expected number of nonzeros in row i in the offdiagonal portion

Verifying Preallocation

Use runtime options

```
-mat_new_nonzero_location_err
-mat_new_nonzero_allocation_err
```

Use runtime option

```
-info
```

Output:

```
[proc #] Matrix size: %d X %d; storage space: %d unneeded, %d used
[proc #] Number of mallocs during MatSetValues( )  is %d
```

```
[merlin] mpirun ex2 -log_info
[0]MatAssemblyEnd_SeqAIJ:Matrix size: 56 X 56; storage space:
[0]   310 unneeded, 250 used
[0]MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues() is 0
[0]MatAssemblyEnd_SeqAIJ:Most nonzeros in any row is 5
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routines
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routines
Norm of error 0.000156044 iterations 6
[0]PetscFinalize:PETSc successfully ended!
```

Block and Symmetric Formats

BAIJ

Like AIJ, but uses static block size

Preallocation is like AIJ, but just one index per block

SBAIJ

Only stores upper triangular part

Preallocation needs number of nonzeros in upper triangular parts of on- and off-diagonal blocks

MatSetValuesBlocked()

Better performance with blocked formats

Also works with scalar formats, if `MatSetBlockSize()` was called

Variants `MatSetValuesBlockedLocal()`, `MatSetValuesBlockedStencil()`

Change matrix format at runtime, don't need to touch assembly code

One Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

```
v[0] = -1.0; v[1] = 2.0; v[2] = -1.0;
if (rank == 0) {
  for(row = 0;  row < N; row++) {
    cols[0] = row-1; cols[1] = row; cols[2] = row+1;
    if (row == 0) {
      MatSetValues(A,1,&row,2,&cols[1],&v[1],
                   INSERT_VALUES);
    } else if (row == N-1) {
      MatSetValues(A,1,&row,2,cols,v,INSERT_VALUES);
    } else {
      MatSetValues(A,1,&row,3,cols,v,INSERT_VALUES);
    }
  }
}
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

A Better Way to Set the Elements of a Matrix

A More Efficient Way

```
v[0] = -1.0; v[1] = 2.0; v[2] = -1.0;
for(row = start;  row < end; row++) {
  cols[0] = row-1; cols[1] = row; cols[2] = row+1;
  if (row == 0) {
    MatSetValues(A,1,&row,2,&cols[1],&v[1],
                 INSERT_VALUES);
  } else if (row == N-1) {
    MatSetValues(A,1,&row,2,cols,v,INSERT_VALUES);
  } else {
    MatSetValues(A,1,&row,3,cols,v,INSERT_VALUES);
  }
}
MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
```

Advantages

All ranks busy: Scalable!

Amount of code essentially unchanged

Matrices

Definition (Matrix)

A matrix is a linear transformation between finite dimensional vector spaces.

Definition (Forming a matrix)

Forming or assembling a matrix means defining it's action in terms of entries (usually stored in a sparse format).

Important Matrices

1. Sparse (e.g. discretization of a PDE operator)
2. Inverse of *anything* interesting $B = A^{-1}$
3. Jacobian of a nonlinear function $Jy = \lim_{\epsilon \to 0} \frac{F(x+\epsilon y) - F(x)}{\epsilon}$
4. Fourier transform $\mathcal{F}, \mathcal{F}^{-1}$
5. Other fast transforms, e.g. Fast Multipole Method
6. Low rank correction $B = A + uv^T$
7. Schur complement $S = D - CA^{-1}B$
8. Tensor product $A = \sum_e A_x^e \otimes A_y^e \otimes A_z^e$
9. Linearization of a few steps of an explicit integrator

Important Matrices

1. Sparse (e.g. discretization of a PDE operator)
2. Inverse of *anything* interesting $B = A^{-1}$
3. Jacobian of a nonlinear function $Jy = \lim_{\epsilon \to 0} \frac{F(x+\epsilon y) - F(x)}{\epsilon}$
4. Fourier transform $\mathcal{F}, \mathcal{F}^{-1}$
5. Other fast transforms, e.g. Fast Multipole Method
6. Low rank correction $B = A + uv^T$
7. Schur complement $S = D - CA^{-1}B$
8. Tensor product $A = \sum_e A_x^e \otimes A_y^e \otimes A_z^e$
9. Linearization of a few steps of an explicit integrator

These matrices are dense. Never form them.

Important Matrices

1. Sparse (e.g. discretization of a PDE operator)
2. Inverse of *anything* interesting $B = A^{-1}$
3. Jacobian of a nonlinear function $Jy = \lim_{\epsilon \to 0} \frac{F(x+\epsilon y) - F(x)}{\epsilon}$
4. Fourier transform $\mathcal{F}, \mathcal{F}^{-1}$
5. Other fast transforms, e.g. Fast Multipole Method
6. Low rank correction $B = A + uv^T$
7. Schur complement $S = D - CA^{-1}B$
8. Tensor product $A = \sum_e A_x^e \otimes A_y^e \otimes A_z^e$
9. Linearization of a few steps of an explicit integrator

These are not very sparse. Don't form them.

Important Matrices

1. Sparse (e.g. discretization of a PDE operator)
2. Inverse of *anything* interesting $B = A^{-1}$
3. Jacobian of a nonlinear function $Jy = \lim_{\epsilon \to 0} \frac{F(x+\epsilon y) - F(x)}{\epsilon}$
4. Fourier transform $\mathcal{F}, \mathcal{F}^{-1}$
5. Other fast transforms, e.g. Fast Multipole Method
6. Low rank correction $B = A + uv^T$
7. Schur complement $S = D - CA^{-1}B$
8. Tensor product $A = \sum_e A_x^e \otimes A_y^e \otimes A_z^e$
9. Linearization of a few steps of an explicit integrator

None of these matrices “have entries”

Iterative Solvers

Matrices

What can we do with a matrix that doesn't have entries?

Krylov solvers for $Ax = b$

Krylov subspace: $\{b, Ab, A^2b, A^3b, \dots\}$

Convergence rate depends on the spectral properties of the matrix

For any popular Krylov method $\mathcal{K}$, there is a matrix of size m, such that $\mathcal{K}$ outperforms all other methods by a factor at least $\mathcal{O}(\sqrt{m})$ [Nachtigal et. al., 1992]

Typically...

The action $y \leftarrow Ax$ can be computed in $\mathcal{O}(m)$

Aside from matrix multiply, the n^{th} iteration requires at most $\mathcal{O}(mn)$

GMRES

Brute force minimization of residual in $\{b, Ab, A^2b, \dots\}$

1. Use Arnoldi to orthogonalize the nth subspace, producing

$$AQ_n = Q_{n+1}H_n$$

2. Minimize residual in this space by solving the overdetermined system

$$H_n y_n = e_1^{(n+1)}$$

using QR-decomposition, updated cheaply at each iteration.

Properties

Converges in n steps for all right hand sides if there exists a polynomial of degree n such that $\|p_n(A)\| < \mathit{tol}$ and $p_n(0) = 1$.

Residual is monotonically decreasing, robust in practice

Restarted variants are used to bound memory requirements

Linear Solvers - Krylov Methods

Using PETSc linear algebra, just add:

```
KSPSetOperators(KSP ksp, Mat A, Mat M, MatStructure flag)
KSPSolve(KSP ksp, Vec b, Vec x)
```

Can access subobjects

```
KSPGetPC(KSP ksp, PC *pc)
```

Preconditioners must obey PETSc interface

- Basically just the KSP interface

Can change solver dynamically from the command line, `-ksp_type`

Linear solvers in PETSc KSP (Excerpt)

- Richardson
- Chebychev
- Conjugate Gradient
- BiConjugate Gradient
- Generalized Minimum Residual Variants
- Transpose-Free Quasi-Minimum Residual
- Least Squares Method
- Conjugate Residual

Preconditioners

Preconditioning

Idea: improve the conditioning of the Krylov operator

Left preconditioning

$$(P^{-1}A)x = P^{-1}b$$

$$\{P^{-1}b, (P^{-1}A)P^{-1}b, (P^{-1}A)^2P^{-1}b, \dots\}$$

Right preconditioning

$$(AP^{-1})Px = b$$

$$\{b, (P^{-1}A)b, (P^{-1}A)^2b, \dots\}$$

The product $P^{-1}A$ or AP^{-1} is *not* formed.

A *preconditioner* $\mathcal{P}$ is a method for constructing a matrix (just a linear function, not assembled!) $P^{-1} = \mathcal{P}(A, A_p)$ using a matrix A and extra information A_p, such that the spectrum of $P^{-1}A$ (or AP^{-1}) is well-behaved.

Preconditioning

Definition (Preconditioner)

A *preconditioner* $\mathcal{P}$ is a method for constructing a matrix $P^{-1} = \mathcal{P}(A, A_p)$ using a matrix A and extra information A_p, such that the spectrum of $P^{-1}A$ (or AP^{-1}) is well-behaved.

P^{-1} is dense, P is often not available and is not needed

A is rarely used by $\mathcal{P}$, but $A_p = A$ is common

A_p is often a sparse matrix, the "preconditioning matrix"

Matrix-based: Jacobi, Gauss-Seidel, SOR, ILU(k), LU

Parallel: Block-Jacobi, Schwarz, Multigrid, FETI-DP, BDDC

Indefinite: Schur-complement, Domain Decomposition, Multigrid

Questions to ask when you see a matrix

1. What do you want to do with it?
   - Multiply with a vector
   - Solve linear systems or eigen-problems
2. How is the conditioning/spectrum?
   - distinct/clustered eigen/singular values?
   - symmetric positive definite ($\sigma(A) \subset \mathbb{R}^+$)?
   - nonsymmetric definite ($\sigma(A) \subset \{z \in \mathbb{C} : \mathrm{Re}[z] > 0\}$)?
   - indefinite?
3. How dense is it?
   - block/banded diagonal?
   - sparse unstructured?
   - denser than we'd like?
4. Is there a better way to compute Ax?
5. Is there a different matrix with similar spectrum, but nicer properties?
6. How can we precondition A?

Questions to ask when you see a matrix

1. What do you want to do with it?
   - Multiply with a vector
   - Solve linear systems or eigen-problems
2. How is the conditioning/spectrum?
   - distinct/clustered eigen/singular values?
   - symmetric positive definite ($\sigma(A) \subset \mathbb{R}^+$)?
   - nonsymmetric definite ($\sigma(A) \subset \{z \in \mathbb{C} : \mathrm{Re}[z] > 0\}$)?
   - indefinite?
3. How dense is it?
   - block/banded diagonal?
   - sparse unstructured?
   - denser than we'd like?
4. Is there a better way to compute Ax?
5. Is there a different matrix with similar spectrum, but nicer properties?
6. How can we precondition A?

Relaxation

Split into lower, diagonal, upper parts: $A = L + D + U$

Jacobi

Cheapest preconditioner: $P^{-1} = D^{-1}$

Successive over-relaxation (SOR)

$$\left(L + \frac{1}{\omega}D\right) x_{n+1} = \left[\left(\frac{1}{\omega} - 1\right) D - U\right] x_n + \omega b$$

$P^{-1} = k$ iterations starting with $x_0 = 0$

Implemented as a sweep

$\omega = 1$ corresponds to Gauss-Seidel

Very effective at removing high-frequency components of residual

Factorization

Two phases

symbolic factorization: find where fill occurs, only uses sparsity pattern
numeric factorization: compute factors

LU decomposition

Ultimate preconditioner
Expensive, for $m \times m$ sparse matrix with bandwidth b, traditionally requires $\mathcal{O}(mb^2)$ time and $\mathcal{O}(mb)$ space.

- Bandwidth scales as $m^{\frac{d-1}{d}}$ in d-dimensions
- Optimal in 2D: $\mathcal{O}(m \cdot \log m)$ space, $\mathcal{O}(m^{3/2})$ time
- Optimal in 3D: $\mathcal{O}(m^{4/3})$ space, $\mathcal{O}(m^2)$ time

Symbolic factorization is problematic in parallel

Incomplete LU

Allow a limited number of levels of fill: ILU(k)
Only allow fill for entries that exceed threshold: ILUT
Usually poor scaling in parallel
No guarantees

1-level Domain decomposition

Domain size L, subdomain size H, element size h

Overlapping/Schwarz

Solve Dirichlet problems on overlapping subdomains

No overlap: *its* $\in \mathcal{O}\left(\frac{L}{\sqrt{Hh}}\right)$

Overlap δ: *its* $\in \left(\frac{L}{\sqrt{H\delta}}\right)$

Neumann-Neumann

Solve Neumann problems on non-overlapping subdomains

its $\in \mathcal{O}\left(\frac{L}{H}(1+\log\frac{H}{h})\right)$

Tricky null space issues (floating subdomains)

Need subdomain matrices, net globally assembled matrix.

Multilevel variants knock off the leading $\frac{L}{H}$

Both overlapping and nonoverlapping with this bound

Multigrid

Hierarchy: Interpolation and restriction operators

$$\mathcal{I}^{\uparrow} : X_{\text{coarse}} \to X_{\text{fine}} \qquad \mathcal{I}^{\downarrow} : X_{\text{fine}} \to X_{\text{coarse}}$$

Geometric: define problem on multiple levels, use grid to compute hierarchy
Algebraic: define problem only on finest level, use matrix structure to build hierarchy

Galerkin approximation

Assemble this matrix: $A_{\text{coarse}} = \mathcal{I}^{\downarrow} A_{\text{fine}} \mathcal{I}^{\uparrow}$

Application of multigrid preconditioner (V-cycle)

Apply pre-smoother on fine level (any preconditioner)
Restrict residual to coarse level with $\mathcal{I}^{\downarrow}$
Solve on coarse level $A_{\text{coarse}} x = r$
Interpolate result back to fine level with $\mathcal{I}^{\uparrow}$
Apply post-smoother on fine level (any preconditioner)

Multigrid convergence properties

- Textbook: $P^{-1}A$ is spectrally equivalent to identity
  - Constant number of iterations to converge up to discretization error
- Most theory applies to SPD systems
  - variable coefficients (e.g. discontinuous): low energy interpolants
  - mesh- and/or physics-induced anisotropy: semi-coarsening/line smoothers
  - complex geometry: difficult to have meaningful coarse levels
- Deeper algorithmic difficulties
  - nonsymmetric (e.g. advection, shallow water, Euler)
  - indefinite (e.g. incompressible flow, Helmholtz)
- Performance considerations
  - Aggressive coarsening is critical in parallel
  - Most theory uses SOR smoothers, ILU often more robust
  - Coarsest level usually solved semi-redundantly with direct solver
- Multilevel Schwarz is essentially the same with different language
  - assume strong smoothers, emphasize aggressive coarsening

Splitting for Multiphysics

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}$$

Relaxation: `-pc_fieldsplit_type [additive,multiplicative,symmetric_multiplicative]`

$$\begin{bmatrix} A & \\ & D \end{bmatrix}^{-1} \qquad \begin{bmatrix} A & \\ C & D \end{bmatrix}^{-1} \qquad \begin{bmatrix} A & \\ & \mathbf{1} \end{bmatrix}^{-1} \left(\mathbf{1} - \begin{bmatrix} A & B \\ & \mathbf{1} \end{bmatrix} \begin{bmatrix} A & \\ C & D \end{bmatrix}^{-1} \right)$$

Gauss-Seidel inspired, works when fields are loosely coupled

Factorization: `-pc_fieldsplit_type schur`

$$\begin{bmatrix} A & B \\ & S \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{1} & \\ CA^{-1} & \mathbf{1} \end{bmatrix}^{-1}, \qquad S = D - CA^{-1}B$$

robust (exact factorization), can often drop lower block
how to precondition S which is usually dense?

interpret as differential operators, use approximate commutators

Debugging and Profiling

PETSc Debugging

By default, a debug build is provided

Launch the debugger

```
-start_in_debugger [gdb,dbx,noxterm]
-on_error_attach_debugger [gdb,dbx,noxterm]
```

Attach the debugger only to some parallel processes

```
-debugger_nodes 0,1
```

Set the display (often necessary on a cluster)

```
-display :0
```

Debugging Tips

Put a breakpoint in `PetscError()` to catch errors as they occur

PETSc tracks memory overwrites at both ends of arrays

- The `CHKMEMQ` macro causes a check of all allocated memory
- Track memory overwrites by bracketing them with `CHKMEMQ`

PETSc checks for leaked memory

- Use `PetscMalloc()` and `PetscFree()` for all allocation
- Print unfreed memory on `PetscFinalize()` with `-malloc_dump`

Simply the best tool today is Valgrind

- It checks memory access, cache performance, memory usage, etc.
- http://www.valgrind.org
- Pass `-malloc 0` to PETSc when running under Valgrind
- Might need `--trace-children=yes` when running under MPI
- `--track-origins=yes` handy for uninitialized memory

Profiling

- Use `-log_summary` for a performance profile
  - Event timing
  - Event flops
  - Memory usage
  - MPI messages
- Call `PetscLogStagePush()` and `PetscLogStagePop()`
  - User can add new stages
- Call `PetscLogEventBegin()` and `PetscLogEventEnd()`
  - User can add new events
- Call `PetscLogFlops()` to include your flops

Reading -log_summary

	Max	Max/Min	Avg	Total
Time (sec):	1.548e+02	1.00122	1.547e+02	
Objects:	1.028e+03	1.00000	1.028e+03	
Flops:	1.519e+10	1.01953	1.505e+10	1.204e+11
Flops/sec:	9.814e+07	1.01829	9.727e+07	7.782e+08
MPI Messages:	8.854e+03	1.00556	8.819e+03	7.055e+04
MPI Message Lengths:	1.936e+08	1.00950	2.185e+04	1.541e+09
MPI Reductions:	2.799e+03	1.00000		

- Also a summary per stage
- Memory usage per stage (based on when it was allocated)
- Time, messages, reductions, balance, flops per event per stage
- Always send `-log_summary` when asking performance questions on mailing list

PETSc Profiling

```
Event                Count      Time (sec)     Flops                             --- Global ---  --- S
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %F %M %L %R  %T %l
------------------------------------------------------------------------------------------------------
--- Event Stage 1: Full solve
VecDot                43 1.0 4.8879e-02 8.3 1.77e+06 1.0 0.0e+00 0.0e+00 4.3e+01  0  0  0  0  0   0  0
VecMDot             1747 1.0 1.3021e+00 4.6 8.16e+07 1.0 0.0e+00 0.0e+00 1.7e+03  0  1  0  0 14   1  1
VecNorm             3972 1.0 1.5460e+00 2.5 8.48e+07 1.0 0.0e+00 0.0e+00 4.0e+03  0  1  0  0 31   1  1
VecScale            3261 1.0 1.6703e-01 1.0 3.38e+07 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0
VecScatterBegin     4503 1.0 4.0440e-01 1.0 0.00e+00 0.0 6.1e+07 2.0e+03 0.0e+00  0  0 50 26  0   0  0
VecScatterEnd       4503 1.0 2.8207e+00 6.4 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0
MatMult             3001 1.0 3.2634e+01 1.1 3.68e+09 1.1 4.9e+07 2.3e+03 0.0e+00 11 22 40 24  0  22 44
MatMultAdd           604 1.0 6.0195e-01 1.0 5.66e+07 1.0 3.7e+06 1.3e+02 0.0e+00  0  0  3  0  0   0  1
MatMultTranspose     676 1.0 1.3220e+00 1.6 6.50e+07 1.0 4.2e+06 1.4e+02 0.0e+00  0  0  3  0  0   1  1
MatSolve            3020 1.0 2.5957e+01 1.0 3.25e+09 1.0 0.0e+00 0.0e+00 0.0e+00  9 21  0  0  0  18 41
MatCholFctrSym         3 1.0 2.8324e-04 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0
MatCholFctrNum        69 1.0 5.7241e+00 1.0 6.75e+08 1.0 0.0e+00 0.0e+00 0.0e+00  2  4  0  0  0   4  5
MatAssemblyBegin     119 1.0 2.8250e+00 1.5 0.00e+00 0.0 2.1e+06 5.4e+04 3.1e+02  1  0  2 24  2   2  0
MatAssemblyEnd       119 1.0 1.9689e+00 1.4 0.00e+00 0.0 2.8e+05 1.3e+03 6.8e+01  1  0  0  0  1   1  0
SNESSolve              4 1.0 1.4302e+02 1.0 8.11e+09 1.0 6.3e+07 3.8e+03 6.3e+03 51 50 52 50 50  99100
SNESLineSearch        43 1.0 1.5116e+01 1.0 1.05e+08 1.1 2.4e+06 3.6e+03 1.8e+02  5  1  2  2  1  10  1
SNESFunctionEval      55 1.0 1.4930e+01 1.0 0.00e+00 0.0 1.8e+06 3.3e+03 8.0e+00  5  0  1  1  0  10  0
SNESJacobianEval      43 1.0 3.7077e+01 1.0 7.77e+06 1.0 4.3e+06 2.6e+04 3.0e+02 13  0  4 24  2  26  0
KSPGMRESOrthog      1747 1.0 1.5737e+00 2.9 1.63e+08 1.0 0.0e+00 0.0e+00 1.7e+03  1  1  0  0 14   1  2
KSPSetup             224 1.0 2.1040e-02 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 3.0e+01  0  0  0  0  0   0  0
KSPSolve              43 1.0 8.9988e+01 1.0 7.99e+09 1.0 5.6e+07 2.0e+03 5.8e+03 32 49 46 24 46  62 9
PCSetUp              112 1.0 1.7354e+01 1.0 6.75e+08 1.0 0.0e+00 0.0e+00 8.7e+01  6  4  0  0  1  12  1
PCSetUpOnBlocks     1208 1.0 5.8182e+00 1.0 6.75e+08 1.0 0.0e+00 0.0e+00 8.7e+01  2  4  0  0  1   4  1
PCApply              276 1.0 7.1497e+01 1.0 7.14e+09 1.0 5.2e+07 1.8e+03 5.1e+03 25 44 42 20 41  49 88
```

Communication Costs

Reductions: usually part of Krylov method, latency limited

- `VecDot`
- `VecMDot`
- `VecNorm`
- `MatAssemblyBegin`

Change algorithm (e.g. IBCGS)

Point-to-point (nearest neighbor), latency or bandwidth

- `VecScatter`
- `MatMult`
- `PCApply`
- `MatAssembly`
- `SNESFunctionEval`
- `SNESJacobianEval`

Compute subdomain boundary fluxes redundantly
Ghost exchange for all fields at once
Better partition

Conclusions

PETSc can help You

solve algebraic and DAE problems in your application area

rapidly develop efficient parallel code, can start from examples

develop new solution methods and data structures

debug and analyze performance

advice on software design, solution algorithms, and performance

`petsc-{users,dev,maint}@mcs.anl.gov`

You can help PETSc

report bugs and inconsistencies, or if you think there is a better way

tell us if the documentation is inconsistent or unclear

consider developing new algebraic methods as plugins, contribute if your idea works