PETSc
Portable, Extensible Toolkit for Scientific Computation

Karl Rupp
me@karlrupp.net
Freelance Computational Scientist

Workshop on Modern Scientific Computing
DTU Compute, Technical University of Denmark

March 2-3, 2016
Introducing Myself

Education

Master’s Degrees in Microelectronics and Mathematics
Doctoral Degree in Microelectronics
Home University: TU Wien

Interests

Efficient Numerics on Modern Hardware
High-level APIs
Semiconductor Device Simulation

Contact

Email: me@karlrupp.net
Web: http://www.karlrupp.net/
Find me at: Google+, Twitter, LinkedIn
Before we start...

Goal of this Workshop

**You** should learn new things about HPC

Ask Questions

Tell me if you do not understand
Ask for further details
Don’t be shy
**Schedule for Day 1**

**Morning: Lecture**
- About PETSc
- Vectors and matrices
- Linear solvers and preconditioners
- Distributed arrays

**Afternoon: Hands-On**
- PETSc configuration and installation
- Debugging your PETSc code
- Step-by-step implementation of a scalable Poisson solver
- Experimentation with preconditioners
About PETSc
PETSc was developed as a Platform for Experimentation

We want to experiment with different

Models
Discretizations
Solvers
Algorithms

These boundaries are often blurred...
Timeline

- Barry
- Bill
- Lois
- Satish
- Dinesh
- Hong
- Kris
- Matt
- Victor
- Dmitry
- Lisandro
- Jed
- Shri
- Peter
- Karl

**PETSc-1**
- 1991

**PETSc-2**
- 1995
- 2000

**PETSc-3**
- 2005
- 2010
- 2015
PETSc

Portable Extensible Toolkit for Scientific Computing

Architecture
- tightly coupled (e.g. XT5, BG/P, Earth Simulator)
- loosely coupled such as network of workstations
- GPU clusters (many vector and sparse matrix kernels)

Software Environment
- Operating systems (Linux, Mac, Windows, BSD, proprietary Unix)
- Any compiler
- Usable from C, C++, Fortran 77/90, Python, and MATLAB
- Real/complex, single/double/quad precision, 32/64-bit int

System Size
- 500B unknowns, 75% weak scalability on Jaguar (225k cores) and Jugene (295k cores)
- Same code runs performantly on a laptop

Free to everyone (BSD-style license), open development
Portable Extensible Toolkit for Scientific Computing

Philosophy: Everything has a plugin architecture

Vectors, Matrices, Coloring/ordering/partitioning algorithms
Preconditioners, Krylov accelerators
Nonlinear solvers, Time integrators
Spatial discretizations/topology

Example

Vendor supplies matrix format and associated preconditioner, distributes compiled shared library.
Application user loads plugin at runtime, no source code in sight.
Portable Extensible **Toolkit** for Scientific Computing

**Toolset**
- algorithms
- (parallel) debugging aids
- low-overhead profiling

**Composability**
- try new algorithms by choosing from product space
- composing existing algorithms (multilevel, domain decomposition, splitting)

**Experimentation**
- Impossible to pick the solver *a priori*
- PETSc’s response: expose an algebra of composition
- keep solvers decoupled from physics and discretization
Portable Extensible Toolkit for Scientific Computing

Computational Scientists
   PyLith (CIG), Underworld (Monash), Magma Dynamics (LDEO, Columbia), PFLOTRAN (DOE), SHARP/UNIC (DOE)

Algorithm Developers (iterative methods and preconditioning)

Package Developers
   SLEPc, TAO, Deal.II, Libmesh, FEniCS, PETSc-FEM, MagPar, OOFEM, FreeCFD, OpenFVM

Funding
   Department of Energy
      SciDAC, ASCR ISICLLES, MICS Program, INL Reactor Program
   National Science Foundation
      CIG, CISE, Multidisciplinary Challenge Program

Documentation and Support
   Hundreds of tutorial-style examples
   Hyperlinked manual, examples, and manual pages for all routines
   Support from petsc-maint@mcs.anl.gov
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 a black-box PDE solver, nor a silver bullet.

— Barry Smith
Obtaining PETSc

Linux Package Managers
Web: http://mcs.anl.gov/petsc, download tarball
Git: https://bitbucket.org/petsc/petsc
Mercurial: https://bitbucket.org/petsc/petsc-hg

Installing PETSc

```
$> cd /path/to/petsc/workdir
$> git clone https://bitbucket.org/petsc/petsc.git \
   --branch master --depth 1
$> cd petsc

$> export PETSC_DIR=$PWD PETSC_ARCH=mpich-gcc-dbg
$> ./configure --with-cc=gcc --with-fc=gfortran \
   --download-fblaslapack --download-{mpich,ml,hypre}
```
PETSc External Packages

Most packages can be automatically
  Downloaded
  Configured and Built (in $PETSC_DIR/externalpackages)
  Installed with PETSc

Works for (list incomplete)
  petsc4py
  PETSc documentation utilities (Sowing, Igrind, c2html)
  BLAS, LAPACK, BLACS, ScaLAPACK, PLAPACK
  MPICH, MPE, OpenMPI
  ParMetis, Chaco, Jostle, Party, Scotch, Zoltan
  MUMPS, Spooles, SuperLU, SuperLU_Dist, UMFPack, pARMS
  PaStiX, BLOPEX, FFTW, SPRNG
  Prometheus, HYPRE, ML, SPAl
  Sundials
  Triangle, TetGen, FIAT, FFC, Generator
  HDF5, Boost
PETSc Pyramid

PETSc Structure

- PETSc PDE Application Codes
  - ODE Integrators
  - Visualization
  - Nonlinear Solvers
  - Interface
  - Linear Solvers
    - Preconditioners + Krylov Methods
  - Grid Management
  - Object-Oriented Matrices, Vectors, Indices
  - Profiling Interface

- Computation and Communication Kernels
  - MPI, MPI-IO, BLAS, LAPACK
Flow Control for a PETSc Application

Main Routine

Timestepping Solvers (TS)

Nonlinear Solvers (SNES)

Linear Solvers (KSP)

Preconditioners (PC)

Application Initialization

Function Evaluation

Jacobian Evaluation

Postprocessing

PETSc
Vectors and Matrices
The Role of PETSc

You want to think about how you decompose your data structures, how you think about them globally. [...] 

If you were building a house, you’d start with a set of blueprints that give you a picture of what the whole house looks like. You wouldn’t start with a bunch of tiles and say. “Well I’ll put this tile down on the ground, and then I’ll find a tile to go next to it.”

But all too many people try to build their parallel programs by creating the smallest possible tiles and then trying to have the structure of their code emerge from the chaos of all these little pieces. You have to have an organizing principle if you’re going to survive making your code parallel.

— Bill Gropp

Parallel Vector Layout

```
VecCreate(PETSC_COMM_WORLD, &x);
VecSetSizes(x, PETSC_DECIDE, N);
VecSetFromOptions(x);
```
Vector Gather and Scatter

// y[iy[i]] = x[ix[i]]
VecScatterCreate(...);
VecScatterBegin(...);
VecScatterEnd(...);
PETSc Vectors

Vector Reductions

proc 0 proc 1 proc 2

VecNorm(...);
VecDot(...);
VecMax(...);
...

proc 0 proc 1 proc 2

VecNorm(...);
VecDot(...);
VecMax(...);
...
Local (Sequential) Operations

- Executed by an arbitrary subset of MPI ranks
- Usually involve `VecGetArray()` / `VecRestoreArray()`

Collective Operations

- Must be executed by all processes in the MPI communicator
- Involve MPI operations (scatter, gather, reduce, etc.)
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()
MatMPIAIJSetPreallocation()

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
PETSc Application Integration

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

![Diagram showing PETSc Application Integration]

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 26 27</td>
<td>28 29</td>
</tr>
<tr>
<td>20 21 22</td>
<td>23 24</td>
</tr>
<tr>
<td>15 16 17</td>
<td>18 19</td>
</tr>
<tr>
<td>10 11 12</td>
<td>13 14</td>
</tr>
<tr>
<td>5  6  7</td>
<td>8  9</td>
</tr>
<tr>
<td>0  1  2</td>
<td>3  4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>21 22 23</td>
<td>28 29</td>
</tr>
<tr>
<td>18 19 20</td>
<td>26 27</td>
</tr>
<tr>
<td>15 16 17</td>
<td>24 25</td>
</tr>
<tr>
<td>6   7   8</td>
<td>13 14</td>
</tr>
<tr>
<td>3   4   5</td>
<td>11 12</td>
</tr>
<tr>
<td>0   1   2</td>
<td>9  10</td>
</tr>
</tbody>
</table>

- **Proc 0**
- **Proc 1**

Natural numbering

PETSc numbering
Parallel Sparse Matrix

```c
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
Simple 3-point stencil for 1D Laplacian

```c
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

```c
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
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 $J_y = \lim_{\epsilon \to 0} \frac{F(x+\epsilon y) - F(x)}{\epsilon}$
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 $J_y = \lim_{\epsilon \to 0} \frac{F(x+\epsilon y)-F(x)}{\epsilon}$
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 $A = \sum \epsilon A_x^\epsilon \otimes A_y^\epsilon \otimes A_z^\epsilon$
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 \textit{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
What can we do with a matrix that doesn’t have entries?

Krylov solvers for $Ax = b$

Krylov subspace: \( \{ b, Ab, A^2 b, A^3 b, \ldots \} \)
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 $O(\sqrt{m})$ [Nachtigal et. al., 1992]

Typically...

The action $y \leftarrow Ax$ can be computed in $O(m)$
Aside from matrix multiply, the $n^{th}$ iteration requires at most $O(mn)$
GMRES

Brute force minimization of residual in \( \{b, Ab, A^2b, \ldots \} \)

1. Use Arnoldi to orthogonalize the \( n \)th subspace, producing

\[
AQ_n = Q_{n+1}H_n
\]

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

\[
H_ny_n = e_1^{(n+1)}
\]

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 \( \|p_n(A)\| < \text{tol} \) 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:

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

Can access subobjects

```c
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
Idea: improve the conditioning of the Krylov operator

Left preconditioning

\[(P^{-1}A)x = P^{-1}b\]

\[\{P^{-1}b, (P^{-1}A)P^{-1}b, (P^{-1}A)^2P^{-1}b, \ldots \} \]

Right preconditioning

\[(AP^{-1})Px = b\]

\[\{b, (P^{-1}A)b, (P^{-1}A)^2b, \ldots \} \]

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.
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 ($\sigma(A) \subset \{z \in \mathbb{C} : \text{Re}[z] > 0\}$)?
   - 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 ($\sigma(A) \subset \{z \in \mathbb{C} : \Re[z] > 0\}$)?
   - 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 $A$?
Split into lower, diagonal, upper parts: \( A = L + D + U \)

**Jacobi**

Cheapest preconditioner: \( P^{-1} = D^{-1} \)
Relaxation

Split into lower, diagonal, upper parts: \( A = L + D + U \)

**Jacobi**

Cheapest preconditioner: \( P^{-1} = D^{-1} \)

**Successive over-relaxation (SOR)**

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

\( P^{-1} = k \) 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
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
$O(mb^2)$ time and $O(mb)$ space.

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

Symbolic factorization is problematic in parallel
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 $O(mb^2)$ time and $O(mb)$ space.

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

Overlap $\delta$: $its \in \left(\frac{L}{\sqrt{H\delta}}\right)$
1-level Domain decomposition

Domain size $L$, subdomain size $H$, element size $h$

Overlapping/Schwarz

Solve Dirichlet problems on overlapping subdomains
No overlap: $its \in O\left(\frac{L}{\sqrt{Hh}}\right)$
Overlap $\delta$: $its \in \left(\frac{L}{\sqrt{H\delta}}\right)$

Neumann-Neumann

Solve Neumann problems on non-overlapping subdomains
$its \in O\left(\frac{L}{H}(1 + \log \frac{H}{h})\right)$
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

\[ \mathcal{I}^\uparrow : X_{\text{coarse}} \rightarrow X_{\text{fine}} \quad \mathcal{I}^\downarrow : X_{\text{fine}} \rightarrow X_{\text{coarse}} \]

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]

\[
\begin{bmatrix}
A & D
\end{bmatrix}^{-1}
\begin{bmatrix}
A & B \\
C & D
\end{bmatrix}^{-1}
\begin{bmatrix}
A & 1
\end{bmatrix}^{-1}
\left(1 - \begin{bmatrix}
A & B \\
1 & C & D
\end{bmatrix}^{-1}\right)
\]

Gauss-Seidel inspired, works when fields are loosely coupled

Factorization: \(-pc_fieldsplit_type\) schur

\[
\begin{bmatrix}
A & B \\
S
\end{bmatrix}^{-1}
\begin{bmatrix}
1 & & \\
CA^{-1} & 1
\end{bmatrix}^{-1}
, \\
S = D - CA^{-1}B
\]

robust (exact factorization), can often drop lower block  
how to precondition \(S\) which is usually dense?  
interpret as differential operators, use approximate commutators
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()`)

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

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 26 27</td>
<td>28 29</td>
</tr>
<tr>
<td>20 21 22</td>
<td>23 24</td>
</tr>
<tr>
<td>15 16 17</td>
<td>18 19</td>
</tr>
<tr>
<td>10 11 12</td>
<td>13 14</td>
</tr>
<tr>
<td>5 6 7</td>
<td>8 9</td>
</tr>
<tr>
<td>0 1 2</td>
<td>3 4</td>
</tr>
</tbody>
</table>

**Natural numbering**

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>21 22 23</td>
<td>28 29</td>
</tr>
<tr>
<td>18 19 20</td>
<td>26 27</td>
</tr>
<tr>
<td>15 16 17</td>
<td>24 25</td>
</tr>
<tr>
<td>6 7 8</td>
<td>13 14</td>
</tr>
<tr>
<td>3 4 5</td>
<td>11 12</td>
</tr>
<tr>
<td>0 1 2</td>
<td>9 10</td>
</tr>
</tbody>
</table>

**PETSc 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

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local numbering</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>X</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>26</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global numbering</td>
<td></td>
</tr>
</tbody>
</table>
The DM object contains only layout (topology) information

All field data is contained in PETSc Vecs

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)
```
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().
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
    DM_BOUNDARY_NONE, DM_BOUNDARY_GHOSTED, DM_BOUNDARY_MIRROR,
    DM_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
Wouldn’t it be nice if we could just write our code for the natural numbering?

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>Proc 0</strong></td>
<td><strong>Proc 1</strong></td>
</tr>
</tbody>
</table>

Natural numbering

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>Proc 0</strong></td>
<td><strong>Proc 1</strong></td>
</tr>
</tbody>
</table>

PETSc numbering
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

Monolithic Global

Monolithic Local

Split Local

Split Global

LocalToGlobalMapping

GetLocalSubMatrix()

GetSubMatrix() / GetSubVector()

rank 0

rank 1

rank 2

GetSubMatrix() / GetSubVector()
DA Local Function

User-provided Function for Nonlinear Residual in 2D

```c
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

```c
SNESSetDM(snes, dm)
SNESSetFunction(snes, r, SNESDAFormFunction, ctx)
```
End of Morning Session

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

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

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