The Portable Extensible Toolkit for Scientific computing

Jed Brown

CSCS 2010-05-10
Part I

Usage and Algorithms
Day 1 Outline

1. Introduction
2. Installation
3. Programming model
   - Collective semantics
   - Options Database
4. Algorithms
   - Linear Algebra
   - SNES
   - DA
   - Preconditioning
   - Matrix Redux
   - Debugging
   - Correctness
Requests

- Tell me if you do not understand
- Tell me if an example does not work
- Suggest better wording, figures, organization
- Follow up:
  - Configuration issues, private: petsc-maint@mcs.anl.gov
  - Public questions: petsc-users@mcs.anl.gov
**Portable Extensible Toolkit for Scientific computing**

- Architecture
  - tightly coupled (e.g. XT5, BG/P, Earth Simulator)
  - loosely coupled such as network of workstations
- Operating systems (Linux, Mac, Windows, BSD, proprietary Unix)
- Any compiler
- Real/complex, single/double precision, 32/64-bit int
- Usable from C, C++, Fortran 77/90, and Python
- Free to everyone (BSD-style license), open development
- 500B unknowns, 75% weak scalability on Jaguar (225k cores) and Jugene (295k cores)
- Same code runs performantly on a laptop
- No iPhone support
Portable Extensible Toolkit for Scientific computing

- Architecture
  - tightly coupled (e.g. XT5, BG/P, Earth Simulator)
  - loosely coupled such as network of workstations
- Operating systems (Linux, Mac, Windows, BSD, proprietary Unix)
- Any compiler
- Real/complex, single/double precision, 32/64-bit int
- Usable from C, C++, Fortran 77/90, and Python
- Free to everyone (BSD-style license), open development
- 500B unknowns, 75% weak scalability on Jaguar (225k cores) and Jugene (295k cores)
- Same code runs performantly on a laptop
- **No** iPhone support
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

Algorithms, (parallel) debugging aids, low-overhead profiling

**Composability**

Try new algorithms by choosing from product space and composing existing algorithms (multilevel, domain decomposition, splitting).

**Experimentation**

- It is not possible to pick the solver *a priori*. What will deliver best/competitive performance for a given physics, discretization, architecture, and problem size?
- PETSc’s response: expose an algebra of composition so new solvers can be created at runtime.
- Important to keep solvers decoupled from physics and discretization because we also experiment with those.
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 ISICLES, MICS Program, INL Reactor Program
  - National Science Foundation
    - CIG, CISE, Multidisciplinary Challenge Program

- **Hundreds of tutorial-style examples**

- **Hyperlinked manual, examples, and manual pages for all routines**

- **Support from** [petsc-maint@mcs.anl.gov](mailto:petsc-maint@mcs.anl.gov)
The Role of PETSc

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
Installation

Downloading

- http://mcs.anl.gov/petsc, download tarball
- We will use Mercurial in this tutorial:
  - http://mercurial.selenic.com
  - Debian/Ubuntu: $ aptitude install mercurial
  - Fedora: $ yum install mercurial
- Get the PETSc release
  - $ hg clone http://petsc.cs.iit.edu/petsc/releases/petsc-3.1
  - $ cd petsc-3.1
  - $ hg clone http://petsc.cs.iit.edu/petsc/releases/BuildSystem-3.1 config/BuildSystem
  - Get the latest bug fixes with $ hg pull -update
Configuration

Basic configuration

- $ export PETSC_DIR=$PWD PETSC_ARCH=mpich-gcc-dbg
- $ ./configure --with-shared
  --with-blas-lapack-dir=/usr
  --download-{mpich,ml,hypre}
- $ make all test

Other common options

- --with-mpi-dir=/path/to/mpi
- --with-scalar-type=<real or complex>
- --with-precision=<single, double, longdouble>
- --with-64-bit-indices
- --download-{umfpack,mumps,scalapack,blacs,parmetis}

- reconfigure at any time with
  $ mpich-gcc-dbg/conf/reconfigure-mpich-gcc-dbg.py
    --new-options
Automatic Downloads

- Starting in 2.2.1, some packages can be automatically
  - Downloaded
  - Configured and Built (in $PETSC_DIR/externalpackages)
  - Installed with PETSc

- Currently works for
  - petsc4py
  - PETSc documentation utilities (Sowing, Igrind, c2html)
  - BLAS, LAPACK, BLACS, ScaLAPACK, PLAPACK
  - MPICH, MPE, Open MPI
  - ParMetis, Chaco, Jostle, Party, Scotch, Zoltan
  - MUMPS, Spooles, SuperLU, SuperLU_Dist, UMFPack, pARMS
  - PaStiX, BLOPEX, FFTW, SPRNG
  - Prometheus, HYPRE, ML, SPAI
  - Sundials
  - Triangle, TetGen, FIAT, FFC, Generator
  - HDF5, Boost

*Can also use* --with-xxx=/path/to/your/install
An optimized build

- $ mpich-gcc-dbg/conf/reconfigure-mpich-gcc-dbg.py
  PETSC_ARCH=mpich-gcc-opt
  --with-debugging=0 && make PETSC_ARCH=mpich-gcc-opt

- **What does** `--with-debugging=1 (default)` **do?**
  - Keeps debugging symbols (of course)
  - Maintains a stack so that errors produce a full stack trace (even SEGV)
  - Does lots of integrity checking of user input
  - Places sentinels around allocated memory to detect memory errors
  - Allocates related memory chunks separately (to help find memory bugs)
  - Keeps track of and reports unused options
  - Keeps track of and reports allocated memory that is not freed
    - `malloc_dump`
Compiling an example

- `$ hg clone http://petsc.cs.iit.edu/petsc/tutorials/CSCS10`
- `$ cd CSCS10`
- `$ hg update -r1`
- `$ make`
- `$ ./pbratu -help | less`
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

Diagram:
- **Main Routine**
  - Timestepping Solvers (TS)
  - Nonlinear Solvers (SNES)
  - Linear Solvers (KSP)
  - Preconditioners (PC)

- Application Initialization
- Function Evaluation
- Jacobian Evaluation
- Postprocessing

Jed Brown (ETH Zürich)
MPI

- Message Passing Interface
- Defines an interface, many implementations
- Can write and run multiprocess jobs without a multicore processor
- Highly optimized
  - Often bypasses kernel, IP stack
  - Network hardware can send/receive directly from user memory (no-copy)
  - Many nonblocking primitives, one-sided operations
  - Can use shared memory within a node
    - Sometimes faster to have network hardware do the copy
- Designed for library interoperability (e.g. attribute caching)
- Not very fault tolerant
  - Usually can’t recover if one process disappears, deadlock possible
- $ mpiexec -n 4 ./app -program -options
- Highly configurable runtime options (Open MPI, MPICH2)
- Rarely have to call MPI directly when using PETSc
MPI communicators

- Opaque object, defines process group and synchronization channel
- PETSc objects need an `MPI_Comm` in their constructor
  - `PETSC_COMM_SELF` for serial objects
  - `PETSC_COMM_WORLD` common, but *not* required
- Can split communicators, spawn processes on new communicators, etc
- Operations are either
  - Not Collective: `VecGetLocalSize()`, `MatSetValues()`
  - Collective: `VecNorm()`, `MatMult()`, `MatAssemblyBegin()`
- Deadlock if some process doesn’t participate (e.g. wrong order)
- Point-to-point communication between two processes e.g. `MatMult()`
- Global communication, like the reduction in `VecDot()`
Advice from Bill Gropp

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.
Objects

Mat A;
PetscInt m,n,M,N;
MatCreate(comm,&A);
MatSetSizes(A,m,n,M,N); /* or PETSC_DECIDE */
MatSetOptionsPrefix(A,"foo_" );
MatSetFromOptions(A);
/* Use A */
MatView(A,PETSC_VIEWER_DRAW_WORLD );
MatDestroy(A);

- **Mat** is an opaque object (pointer to incomplete type)
  - Assignment, comparison, etc, are cheap
- What’s up with this “Options” stuff?
  - Allows the type to be determined at runtime: -foo_mat_type sbaij
  - Inversion of Control similar to “service locator”,
    related to “dependency injection”
  - Other options (performance and semantics) can be changed at runtime
    under -foo_mat_
## Basic PetscObject Usage

Every object in PETSc supports a basic interface

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create()</td>
<td>create the object</td>
</tr>
<tr>
<td>Get/SetName()</td>
<td>name the object</td>
</tr>
<tr>
<td>Get/SetType()</td>
<td>set the implementation type</td>
</tr>
<tr>
<td>Get/SetOptionsPrefix()</td>
<td>set the prefix for all options</td>
</tr>
<tr>
<td>SetFromOptions()</td>
<td>customize object from the command line</td>
</tr>
<tr>
<td>SetUp()</td>
<td>preform other initialization</td>
</tr>
<tr>
<td>View()</td>
<td>view the object</td>
</tr>
<tr>
<td>Destroy()</td>
<td>cleanup object allocation</td>
</tr>
</tbody>
</table>

Also, all objects support the `-help` option.
Ways to set options

- Command line
- Filename in the third argument of `PetscInitialize()`
  - `~/.petscrc`
  - `${PWD}/.petscrc`
  - `${PWD}/petscrc`
- `PetscOptionsInsertFile()`
- `PetscOptionsInsertString()`
- `PETSC_OPTIONS` environment variable
- command line option `-options_file [file]`
Try it out

$ cd $PETSC_DIR/src/snes/examples/tutorials && make ex5

- $ ./ex5 -da_grid_x 10 -da_grid_y 10 -par 6.7 -snes_monitor -{ksp,snes}_converged_reason -snes_view
- $ ./ex5 -da_grid_x 10 -da_grid_y 10 -par 6.7 -snes_monitor -{ksp,snes}_converged_reason -snes_view -mat_view_draw -draw_pause 0.5
- $ ./ex5 -da_grid_x 10 -da_grid_y 10 -par 6.7 -snes_monitor -{ksp,snes}_converged_reason -snes_view -mat_view_draw -draw_pause 0.5 -pc_type lu -pc_factor_mat_ordering_type natural

Use --help to find other ordering types
In parallel

- $ mpiexec -n 4 ./ex5 -da_grid_x 10 -da_grid_y 10 -par 6.7 -snes_monitor -{ksp,snes}_converged_reason -snes_view -sub_pc_type lu

- How does the performance change as you
  - vary the number of processes (up to 32 or 64)?
  - increase the problem size?
  - use an inexact subdomain solve?
  - try an overlapping method: -pc_type asm -pc_asm_overlap 2
  - simulate a big machine: -pc_asm_blocks 512
  - change the Krylov method: -ksp_type ibcgs
  - use algebraic multigrid: -pc_type hypre
  - use smoothed aggregation multigrid: -pc_type ml
Newton iteration: foundation of SNES

- Standard form of a nonlinear system
  \[ F(u) = 0 \]

- Iteration
  \[
  \text{Solve: } \quad J(u)w = -F(u) \\
  \text{Update: } \quad u^+ \leftarrow u + w
  \]

- 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 \( J(u) \)

Example (Nonlinear Poisson)

\[
F(u) = 0 \quad \sim \quad -\nabla \cdot [(1 + u^2)\nabla u] - f = 0 \\
J(u)w \quad \sim \quad -\nabla \cdot [(1 + u^2)\nabla w + 2uw\nabla u]
\]
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).
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 $A = \sum_e A^e_x \otimes A^e_y \otimes A^e_z$
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 $Jy = \lim_{\varepsilon \to 0} \frac{F(x + \varepsilon y) - F(x)}{\varepsilon}$
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 $Jy = \lim_{\varepsilon \to 0} \frac{F(x+\varepsilon y)-F(x)}{\varepsilon}$
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 $A = \sum_e A^e_x \otimes A^e_y \otimes A^e_z$
9. Linearization of a few steps of an explicit integrator

- None of these matrices “have entries”
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
  - Existance of small polynomials $p_n(A) < \varepsilon$ where $p_n(0) = 1$.
  - Condition number $\kappa(A) = \|A\| \|A^{-1}\| = \sigma_{\text{max}} / \sigma_{\text{min}}$
  - Distribution of singular values, spectrum $\Lambda$, pseudospectrum $\Lambda_\varepsilon$
- 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^2 b, \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

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

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 \( \| p_n(A) \| < tol \) and \( p_n(0) = 1 \).
- Residual is monotonically decreasing, robust in practice
- Restarted variants are used to bound memory requirements
The $p$-Bratu equation

- 2-dimensional model problem

\[-\nabla \cdot (|\nabla u|^{p-2}\nabla u) - \lambda e^u - f = 0, \quad 1 \leq p \leq \infty, \quad \lambda < \lambda_{\text{crit}}(p)\]

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

- Regularized variant

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

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

- Jacobian

$$J(u)w \sim -\nabla \cdot [(\eta 1 + \eta' \nabla u \otimes \nabla u) \nabla w] - \lambda e^u w$$

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

Physical interpretation: conductivity tensor flattened in direction $\nabla u$
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**
SNES Paradigm

The SNES interface is based upon callback functions

- `FormFunction()`, set by `SNESSetFunction()`
- `FormJacobian()`, set by `SNESGetJacobian()`

When PETSc needs to evaluate 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

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

The user provided function which calculates the Jacobian has signature

\[
\text{PetscErrorCode } (*\text{func})(\text{SNES } \text{snes}, \text{Vec } \text{x}, \text{Mat } *\text{J}, \text{Mat } *\text{M}, \text{MatStructure } *\text{flag}, \text{void } *\text{ctx})
\]

- **x**: The current solution
- **J**: The Jacobian
- **M**: The Jacobian preconditioning matrix (possibly J itself)
- **ctx**: The user context passed to \text{SNESetFunction}()
  - Use this to pass application information, e.g. physical constants
  - Possible MatStructure values are:
    - SAME_NONZERO_PATTERN
    - DIFFERENT_NONZERO_PATTERN

Alternatively, you can use

- a builtin sparse finite difference approximation
- automatic differentiation (ADIC/ADIFOR)
Distributed Array

- Interface for topologically structured grids
- Defines (topological part of) a finite-dimensional function space
  - Get an element from this space: `DACreateGlobalVector()`
- Provides parallel layout
- Refinement and coarsening
  - `DARefineHierarchy()`
- Ghost value coherence
  - `DAGlobalToLocalBegin()`
- Matrix preallocation:
  - `DAGetMatrix()`
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
**DA Global Numberings**

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

- **Proc 0** and **Proc 1** columns represent the **natural numbering**.
- **Proc 2** and **Proc 3** columns represent the **PETSc numbering**.
DA 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

![Diagram showing global and local numbering]
DA Vectors

- The DA 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
  - DACreateGlobalVector(DA da, Vec *gvec)
- Local vectors are sequential (and usually temporary)
  - Each process stores its local portion plus ghost values
  - DACreateLocalVector(DA da, Vec *lvec)
  - includes ghost values!
- Coordinate vectors store the mesh geometry
  - DAGetCoordinates(DA, Vec *coords)
  - Can be manipulated with their own DA
    DAGetCoordinateDA(DA, DA *cda)
Updating Ghosts

Two-step process enables overlapping computation and communication

- \( \text{DAGlobalToLocalBegin}(\text{da}, \text{gvec}, \text{mode}, \text{lvec}) \)
  - \( \text{gvec} \) provides the data
  - \( \text{mode} \) is either \text{INSERT\_VALUES} or \text{ADD\_VALUES}
  - \( \text{lvec} \) holds the local and ghost values
- \( \text{DAGlobalToLocalEnd}(\text{da}, \text{gvec}, \text{mode}, \text{lvec}) \)
  - Finishes the communication

The process can be reversed with \( \text{DALocalToGlobal}() \).
DA Stencils

Both the **box** stencil and **star** stencil are available.

Box Stencil

Star Stencil
Creating a DA

DACreate2d(comm, wrap, type, M, N, m, n, 
dof, s, lm[], ln[], DA *da)

**wrap**: Specifies periodicity
  - DA_NONPERIODIC, DA_XPERIODIC, DA_YPERIODIC, 
or DA_XYPERIODIC

**type**: Specifies stencil
  - DA_STENCIL_BOX or DA_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 PETSC_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?

- Yes, that’s what `DAVecGetArray()` is for.
- Also, the DA offers *local* callback functions
  - `FormFunctionLocal()`, set by `DASetLocalFunction()`
  - `FormJacobianLocal()`, set by `DASetLocalJacobian()`

- When PETSc needs to evaluate the nonlinear residual $F(x)$,
  - Each process evaluates the local residual
  - PETSc assembles the global residual automatically
    - Uses `DALocalToGlobal()` method
DA Local Function

The user provided function which calculates the nonlinear residual in 2D has signature

```c
PetscErrorCode (*lfunc)(DALocalInfo *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 `DASetLocalFunction`

The local DA function is activated by calling

```c
SNESSetFunction(snes, r, SNESDAFormFunction, ctx)
```
**Bratu Residual Evaluation**

\[-\Delta u - \lambda e^u = 0\]

```c
BratuResidualLocal (DALocalInfo *info, Field **x, Field **f) {
    /* 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);
        }
    }
}
```

$PETSC\_DIR/src/snes/examples/tutorials/ex5.c$
Start with 2-Laplacian plus Bratu nonlinearity

- Matrix-free Jacobians, no preconditioning \(-\text{snes\_mf}\)
- $\ hg \ update \ -r3$
- $\ ./\text{pbratu} \ -\text{da\_grid\_x} \ 10 \ -\text{da\_grid\_y} \ 10$
  \ -\lambda 6.7 \ -\text{snes\_mf} \ -\text{snes\_monitor}$
  \ -\text{ksp\_converged\_reason}$
- $\ ./\text{pbratu} \ -\text{da\_grid\_x} \ 20 \ -\text{da\_grid\_y} \ 20$
  \ -\lambda 6.7 \ -\text{snes\_mf} \ -\text{snes\_monitor}$
  \ -\text{ksp\_converged\_reason}$
- $\ ./\text{pbratu} \ -\text{da\_grid\_x} \ 40 \ -\text{da\_grid\_y} \ 40$
  \ -\lambda 6.7 \ -\text{snes\_mf} \ -\text{snes\_monitor}$
  \ -\text{ksp\_converged\_reason}$
- Watch linear and nonlinear convergence
Add $p$ nonlinearity

- Matrix-free Jacobians, no preconditioning $-$snes_mf
- $\$ ~hg~update~-r4$
- $\$ ~./pbratu~-da_grid_x~10~-da_grid_y~10
  \quad -\lambda 1 \quad -p~1.3 \quad -snes_mf \quad -snes_monitor
  \quad -ksp_converged_reason$
- $\$ ~./pbratu~-da_grid_x~20~-da_grid_y~20
  \quad -\lambda 1 \quad -p~1.3 \quad -snes_mf \quad -snes_monitor
  \quad -ksp_converged_reason$
- $\$ ~./pbratu~-da_grid_x~40~-da_grid_y~40
  \quad -\lambda 1 \quad -p~1.3 \quad -snes_mf \quad -snes_monitor
  \quad -ksp_converged_reason$
- Watch linear and nonlinear convergence
Preconditioning

Idea: improve the conditioning of the Krylov operator

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

- 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.

Definition (Preconditioner)

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 ($\sigma(A) \subset \{ z \in \mathbb{C} : \Re[z] > 0 \}$)?
   - 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 \( A \)?
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 $O(mb^2)$ time and $O(mb)$ space.
  - Bandwidth scales as $m^{\frac{d-1}{d}}$ in $d$-dimensions
- 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
- Very poor scaling in parallel, don’t bother beyond 8 PEs.
- No guarantees
1-level Domain decomposition

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

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

- Multilevel variants knock off the leading $\frac{L}{H}$
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
- Most theory applies to SPD systems
- Nonsymmetric (e.g. advection, shallow water, Euler) with low-order upwind discretization
- Good when coefficients in problem are smooth
  - Large jumps and anisotropy are harder
  - Build low-energy interpolants
  - Use stronger smoothers
- Aggressive coarsening is critical, especially in parallel
- Most theory uses SOR smoothers, ILU often more robust
- Coarsest component usually solved semi-redundantly with direct solver
- Multilevel Schwarz is an extreme case of aggressive coarsening and strong smoothers. Exotic interpolants for robustness.
Finite Difference Jacobians

PETSc can compute and explicitly store a Jacobian via 1st-order FD

- **Dense**
  - Activated by `-snes_fd`
  - Computed by `SNESDefaultComputeJacobian()`

- **Sparse via colorings**
  - Coloring is created by `MatFDColoringCreate()`
  - Computed by `SNESDefaultComputeJacobianColor()`

Can also use Matrix-free Newton-Krylov via 1st-order FD

- Activated by `-snes_mf` without preconditioning
- Activated by `-snes_mf_operator` with user-defined preconditioning
  - Uses preconditioning matrix from `SNESSetJacobian()`
Add finite difference Jacobian by coloring

- $\text{hg update } -r5$
- $\text{./pbratu } -\text{da_grid}_x \ 10 \ -\text{da_grid}_y \ 10$
  $\ -\text{lambda} \ 1 \ -p \ 1.3 \ -\text{snes_fd} \ -\text{snes_monitor}$
  $\ -\text{ksp_converged_reason}$
- $\text{./pbratu } -\text{da_grid}_x \ 10 \ -\text{da_grid}_y \ 10$
  $\ -\text{lambda} \ 1 \ -p \ 1.3 \ -\text{fd_jacobian} \ -\text{snes_monitor}$
  $\ -\text{ksp_converged_reason}$
- $\text{./pbratu } -\text{da_grid}_x \ 10 \ -\text{da_grid}_y \ 10$
  $\ -\text{lambda} \ 1 \ -p \ 1.3 \ -\text{fd_jacobian} \ -\text{snes_monitor}$
  $\ -\text{ksp_converged_reason}$
- Try some different preconditioners (jacobi, sor, asm, hypre, ml)
- Try changing the physical parameters
- May need $\text{-mat_fd_type ds}$
Matrices, redux

What are PETSc matrices?

- Linear operators on finite dimensional vector spaces.
- Fundamental objects for storing stiffness matrices and Jacobians
- Each process locally owns a contiguous set of rows
- Supports many data types
  - AIJ, Block AIJ, Symmetric AIJ, Block Diagonal, etc.
- Supports structures for many packages
  - MUMPS, Spooles, SuperLU, UMFPack, DSCPack
Matrices, redux

What are PETSc matrices?

- Linear operators on finite dimensional vector spaces.
- Fundamental objects for storing stiffness matrices and Jacobians.
- Each process locally owns a contiguous set of rows.
- Supports many data types:
  - AIJ, Block AIJ, Symmetric AIJ, Block Diagonal, etc.
- Supports structures for many packages:
  - MUMPS, Spooles, SuperLU, UMFPack, DSCPack.
How do I create matrices?

- `MatCreate(MPI_Comm, Mat *)`
- `MatSetSizes(Mat, int m, int n, int M, int N)`
- `MatSetType(Mat, MatType typeName)`
- `MatSetFromOptions(Mat)`
  - Can set the type at runtime
- `MatSeqAIJPreallocation(Mat, PetscInt nz, const PetscInt nnz[])`
- `MatMPIAIJPreallocation(Mat, PetscInt dnz, const PetscInt dnz[], PetscInt onz, const PetscInt onz[])`
- `MatSetValues(Mat, ...)`
  - **MUST** be used, but does automatic communication
Matrix Polymorphism

The PETSc Mat has a single user interface,

- Matrix assembly
  - MatSetValues()
- Matrix-vector multiplication
  - MatMult()
- Matrix viewing
  - MatView()

but multiple underlying implementations.

- AIJ, Block AIJ, Symmetric Block AIJ,
- Dense
- Matrix-Free
- etc.

A matrix is defined by its interface, not by its data structure.
Matrix Assembly

- A three step process
  - Each process sets or adds values
  - Begin communication to send values to the correct process
  - Complete the communication

- `MatSetValues(Mat A, m, rows[], n, cols[], values[], mode)`
  - `mode` is either `INSERT_VALUES` or `ADD_VALUES`
  - Logically dense block of values

- Two phase assembly allows overlap of communication and computation
  - `MatAssemblyBegin(Mat m, type)`
  - `MatAssemblyEnd(Mat m, type)`
  - `type` is either `MAT_FLUSH.Assembly` or `MAT_FINAL.Assembly`

- For vector problems
  - `MatSetValuesBlocked(Mat A, m, rows[], n, cols[], values[], mode)`
Matrix Assembly

- A three step process
  - Each process sets or adds values
  - Begin communication to send values to the correct process
  - Complete the communication
- MatSetValues(Mat A, m, rows[], n, cols[], values[], mode)
  - mode is either INSERT_VALUES or ADD_VALUES
  - Logically dense block of values
- Two phase assembly allows overlap of communication and computation
  - MatAssemblyBegin(Mat m, type)
  - MatAssemblyEnd(Mat m, type)
  - type is either MAT_FLUSH_ASSEMBLY or MAT_FINAL_ASSEMBLY
- For vector problems
  MatSetValuesBlocked(Mat A, m, rows[], n, cols[], values[], mode)
One Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

\[ v[0] = -1.0; \quad v[1] = 2.0; \quad v[2] = -1.0; \]

\[
\text{if} (\text{rank} == 0) \{ \\
\quad \text{for} (\text{row} = 0; \quad \text{row} < N; \quad \text{row}++) \{ \\
\quad\quad \text{cols}[0] = \text{row} - 1; \quad \text{cols}[1] = \text{row}; \quad \text{cols}[2] = \text{row} + 1; \\
\quad\quad \text{if} (\text{row} == 0) \{ \\
\quad\quad\quad \text{MatSetValues}(A, 1, &\text{row}, 2, &\text{cols}[1], &v[1], \text{INSERT}\_\text{VALUES}); \\
\quad\quad\} \quad \text{else if} (\text{row} == N-1) \{ \\
\quad\quad\quad \text{MatSetValues}(A, 1, &\text{row}, 2, \text{cols}, v, \text{INSERT}\_\text{VALUES}); \\
\quad\quad\} \quad \text{else} \{ \\
\quad\quad\quad \text{MatSetValues}(A, 1, &\text{row}, 3, \text{cols}, v, \text{INSERT}\_\text{VALUES}); \\
\quad\quad\}\} \\
\}
\]

MatAssemblyBegin(A, MAT_FINAL\_ASSEMBLY);
MatAssemblyEnd(A, MAT_FINAL\_ASSEMBLY);
A Better Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

\[
\begin{align*}
v[0] &= -1.0; \quad v[1] = 2.0; \quad v[2] = -1.0; \\
\text{for} (\text{row} = \text{start}; \quad \text{row} < \text{end}; \quad \text{row}++) \{ \\
&\quad \text{cols}[0] = \text{row}-1; \quad \text{cols}[1] = \text{row}; \quad \text{cols}[2] = \text{row}+1; \\
&\quad \text{if} (\text{row} == 0) \{ \\
&\quad \quad \text{MatSetValues}(A,1,\&\text{row},2,\&\text{cols}[1],\&v[1],\text{INSERT_VALUES}); \\
&\quad \} \quad \text{else if} (\text{row} == N-1) \{ \\
&\quad \quad \text{MatSetValues}(A,1,\&\text{row},2,\text{cols},v,\text{INSERT_VALUES}); \\
&\quad \} \quad \text{else} \{ \\
&\quad \quad \text{MatSetValues}(A,1,\&\text{row},3,\text{cols},v,\text{INSERT_VALUES}); \\
&\quad \}
\}
\]

\text{MatAssemblyBegin}(A, \text{MAT_FINAL_ASSEMBLY});
\text{MatAssemblyEnd}(A, \text{MAT_FINAL_ASSEMBLY});
Why Are PETSc Matrices That Way?

- No one data structure is appropriate for all problems
  - Blocked and diagonal formats provide significant performance benefits
  - PETSc has many formats and makes it easy to add new data structures
- Assembly is difficult enough without worrying about partitioning
  - PETSc provides parallel assembly routines
  - Achieving high performance still requires making most operations local
  - However, programs can be incrementally developed.
  - MatPartitioning and MatOrdering can help
- Matrix decomposition in contiguous chunks is simple
  - Makes interoperation with other codes easier
  - For other ordering, PETSc provides “Application Orderings” (AO)
p-Bratu assembly

- Use `DAGetMatrix()` (can skip matrix preallocation details)
- Start by just assembling Bratu nonlinearity
- `$ hg update -r6`
- Watch `-snes_converged_reason`, what happens for $p \neq 2$?
- Solve exactly with the preconditioner `-pc_type lu`
- Try `-snes_mf_operator`
$p$-Bratu assembly

- We need to assemble the $p$ part

\[ J(u)w \sim -\nabla \cdot \left[ (\eta 1 + \eta' \nabla u \otimes \nabla u) \nabla w \right] \]

- Second part is scary, but what about just using $-\nabla \cdot (\eta \nabla w)$?
- `$hg update -r7`
- Solve exactly with the preconditioner `-pc_type lu`
- Try `-snes_mf_operator`
- Refine the grid, change $p$
- Try algebraic multigrid if available: `-pc_type [ml, hypre]`
Does the preconditioner need Newton linearization?

- The anisotropic part looks messy. Is it worth writing the code to assemble that part?
- Easy profiling: `log_summary`
- Observation: the Picard linearization uses a “star” (5-point) stencil while Newton linearization needs a “box” (9-point) stencil.
- Add support for reduced preallocation with a command-line option
- `$ hg update -r8`
- Compare performance (time, memory, iteration count) of
  - 5-point Picard-linearization assembled by hand
  - 5-point Newton-linearized Jacobian computed by coloring
  - 9-point Newton-linearized Jacobian computed by coloring
Maybe it’s not worth it, but let’s assemble it anyway

- $ hg update -r9
- Crash!
- You were using the debug PETSC_ARCH, right?
- Launch the debugger
  - `start_inDebugger [gdb, dbx, noxterm]`
  - `on_error_attachDebugger [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](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.
Memory error is gone now

- `$ hg update -r 10`
- Run with `-snes_mf_operator -pc_type lu`
- Do you see quadratic convergence?
  - Hmm, there must be a bug in that mess, where is it?
Memory error is gone now

- $ hg update -r10
- Run with -snes_mf_operator -pc_type lu
- Do you see quadratic convergence?
- Hmm, there must be a bug in that mess, where is it?
SNES Test

- PETSc can compute a finite difference Jacobian and compare it to yours
  - `--snes_type test`
    - Is the difference significant?
  - `--snes_type test --snes_test_display`
    - Are the entries in the star stencil correct?
- Find which line has the typo
- `$ hg update --rev 11`
- Check with `--snes_type test`
- and `--snes_mf_operator --pc_type lu`
Part II

Performance, scalability, and hard problems
Day 2 Outline

5 Algorithms (continued from yesterday)
   Hydrostatic Ice
   Driven cavity

6 Application Integration

7 Performance and Scalability
   Memory hierarchy
   Profiling

8 Hard problems
   Splitting
Hydrostatic equations for ice sheet flow

- Valid in the limit $w_x \ll u_z$, independent of basal friction
- Eliminate $p$ and $w$ by incompressibility:
  3D elliptic system for $u = (u, v)$

$$
-\nabla \cdot \left[ \eta \begin{pmatrix} 4u_x + 2v_y & u_y + v_x & u_z \\ u_y + v_x & 2u_x + 4v_y & v_z \end{pmatrix} \right] + \rho g \nabla s = 0
$$

$$
\eta(\gamma) = \frac{B}{2} (\varepsilon^2 + \gamma) \frac{1-n}{2n}, \quad n \approx 3
$$

$$
\gamma = u_x^2 + v_y^2 + u_x v_y + \frac{1}{4} (u_y + v_x)^2 + \frac{1}{4} u_z^2 + \frac{1}{4} v_z^2
$$

and slip boundary $\sigma \cdot \mathbf{n} = \beta^2 u$ where

$$
\beta^2(\gamma_b) = \beta_0^2 (\varepsilon_b^2 + \gamma_b)^\frac{m-1}{2}, \quad 0 < m \leq 1
$$

$$
\gamma_b = \frac{1}{2} (u^2 + v^2)
$$

- $Q_1$ FEM: src/snes/examples/tutorials/ex48.c
Algorithms (continued from yesterday)

Jed Brown (ETH Zürich)

PETSc

CSCS 2010-05-10 76 / 116
Some Multigrid Options

- `dmmg_grid_sequence`: [FALSE]
  Solve nonlinear problems on coarse grids to get initial guess
- `pc_mg_galerkin`: [FALSE]
  Use Galerkin process to compute coarser operators
- `pc_mg_type`: [FULL]
  (choose one of) MULTIPLICATIVE ADDITIVE FULL KASKADE
- `mg_coarse_{ksp,pc}*`
  control the coarse-level solver
- `mg_levels_{ksp,pc}*`
  control the smoothers on levels
- `mg_levels_3_{ksp,pc}*`
  control the smoother on specific level
- These also work with ML’s algebraic multigrid.
What is this doing?

- mpiexec -n 4 ./ex48 -M 16 -P 2 -da_refine_hierarchy_x 1,8,8 -da_refine_hierarchy_y 2,1,1 -da_refine_hierarchy_z 2,1,1 -dmmg_grid_sequence 1 -dmmg_view -log_summary -ksp_converged_reason -ksp_gmres_modifiedgramschmidt -ksp_monitor -ksp_rtol 1e-2 -pc_mg_type multiplicative -mg_coarse_pc_type lu -mg_levels_0_pc_type lu -mg_coarse_pc_factor_mat_solver_package mumps -mg_levels_0_pc_factor_mat_solver_package mumps -mg_levels_1_sub_pc_type cholesky -snes_converged_reason -snes_monitor -snes_stol 1e-12 -thi_L 80e3 -thi_alpha 0.05 -thi_friction_m 0.3 -thi_hom x -thi_nlevels 4

- What happens if you remove -dmmg_grid_sequence?
- What about solving with block Jacobi, ASM, or algebraic multigrid?
SNES Example

Driven Cavity

- Velocity-vorticity formulation
- Flow driven by lid and/or buoyancy
- Logically regular grid
  - Parallelized with DA
- Finite difference discretization
- Authored by David Keyes
SNES Example
Driven Cavity Application Context

/* Collocated at each node */
typedef struct {
    PetscScalar u, v, omega, temp;
} Field;

typedef struct {
    /* physical parameters */
    PassiveReal lidvelocity, prandtl, grashof;
    /* color plots of the solution */
    PetscTruth draw_contours;
} AppCtx;
SNES Example
Driven Cavity Residual Evaluation

```c
DrivenCavityFunction(SNES snes, Vec X, Vec F, void *ptr) {
  AppCtx *user = (AppCtx *) ptr;
  /* local starting and ending grid points */
  PetscInt istart, iend, jstart, jend;
  PetscScalar *f;          /* local vector data */
  PetscReal grashof = user->grashof;
  PetscReal prandtl = user->prandtl;
  PetscErrorCode ierr;

  /* Code to communicate nonlocal ghost point data */
  VecGetArray(F, &f);
  /* Code to compute local function components */
  VecRestoreArray(F, &f);
  return 0;
}
```
SNES Example

Better Driven Cavity Residual Evaluation

PetscErrorCode DrivenCavityFuncLocal(DALocalInfo *info, Field **x, Field **f, void *ctx) {

    /* 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++) {
            /* convective coefficients for upwinding */
            /* U velocity */
            u = x[j][i].u;
            uxx = (2.0*u - x[j][i-1].u - x[j][i+1].u)*hydhx;
            uyy = (2.0*u - x[j-1][i].u - x[j+1][i].u)*hxdhy;
            upw = 0.5*(x[j+1][i].omega - x[j-1][i].omega)*hx;
            f[j][i].u = uxx + uyy - upw;
            /* V velocity, Omega, Temperature */
        }
    }
}

$PETSC_DIR/src/snes/examples/tutorials/ex19.c
Running the driven cavity

- ./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16 -da_grid_y 16 -snex_moniter -dmmg_view -nlevels 3
- ./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snex_moniter -dmmg_view -nlevels 3
- ./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snex_moniter -dmmg_view -nlevels 3

Uh oh, we have convergence problems
Run with -snex_moniter_convergence
Does -dmmg_grid_sequence help?
Running the driven cavity

- ./ex19 -lidvelocity 100 -grashof 1e2 -da_grid_x 16 -da_grid_y 16 -snes_monitor -dmmg_view -nlevels 3
- ./ex19 -lidvelocity 100 -grashof 1e4 -da_grid_x 16 -da_grid_y 16 -snes_monitor -dmmg_view -nlevels 3
- ./ex19 -lidvelocity 100 -grashof 1e5 -da_grid_x 16 -da_grid_y 16 -snes_monitor -dmmg_view -nlevels 3

- Uh oh, we have convergence problems
- Run with -snes_monitor_convergence
- Does -dmmg_grid_sequence help?
Why isn’t SNES converging?

- The Jacobian is wrong (maybe only in parallel)
  - Check with `-snes_type test` and `-snes_mf_operator`
  - `-pc_type lu`
- The linear system is not solved accurately enough
  - Check with `-pc_type lu`
  - Check `-ksp_monitor_true_residual`, try right preconditioning
- The Jacobian is singular with inconsistent right side
  - Use `MatNullSpace` to inform the KSP of a known null space
  - Use a different Krylov method or preconditioner
- The nonlinearity is just really strong
  - Run with `-info` or `-snes_ls_monitor` (petsc-dev) to see line search
  - Try using trust region instead of line search `-snes_type tr`
  - Try grid sequencing if possible
  - Use a continuation
Globalizing the lid-driven cavity

Pseudotransient continuation continuation ($\Psi tc$)

- Do linearly implicit backward-Euler steps, driven by steady-state residual
- Clever way to adjust step sizes to retain quadratic convergence it terminal phase

- Implemented in `src/snes/examples/tutorials/ex27.c`
- `$ $ make runex27`
- Make the method linearly implicit: `-snes_max_it 1`
  - Compare required number of linear iterations
- Try increasing `-lidvelocity`, `-grashof`, and problem size
Application Integration

- Be willing to experiment with algorithms
  - No optimality without interplay between physics and algorithmics
- Adopt flexible, extensible programming
  - Algorithms and data structures not hardwired
- Be willing to play with the real code
  - Toy models are rarely helpful
- If possible, profile before integration
  - Automatic in PETSc
Incorporating PETSc into existing codes

- PETSc does not seize `main()`, does not control output
- Propogates errors from underlying packages, flexible error handling
- Nothing special about `MPI_COMM_WORLD`
- Can wrap existing data structures/algorithms
  - `MatShell`, `PCShell`, full implementations
  - `VecCreateMPIWithArray()`
  - `MatCreateSeqAIJWithArrays()`
  - Use an existing semi-implicit solver as a preconditioner
  - Usually worthwhile to use native PETSc data structures unless you have a good reason not to
- Uniform interfaces across languages: C, C++, Fortran 77/90, Python
- Do not have to use high level interfaces
  - but PETSc can offer more if you do, like MFFD and SNES Test
Integration Stages

• Version Control
  • It is impossible to overemphasize

• Initialization
  • Linking to PETSc

• Profiling
  • Profile before changing
  • Also incorporate command line processing

• Linear Algebra
  • First PETSc data structures

• Solvers
  • Very easy after linear algebra is integrated
Initialization

- **Call** `PetscInitialize()`
  - Setup static data and services
  - Setup MPI if it is not already
  - Can set `PETSC_COMM_WORLD` to use your communicator (can always use subcommunicators for each object)
- **Call** `PetscFinalize()`
  - Calculates logging summary
  - Can check for leaks/unused options
  - Shutdown and release resources
- Can only initialize PETSc once
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

\texttt{MatSeqAIJPreallocation(Mat A, int nz, int nnz[])}

- \texttt{nz}: expected number of nonzeros in any row
- \texttt{nnz(i)}: expected number of nonzeros in row \textit{i}
Parallel Sparse Matrix

- Each process locally owns a submatrix of contiguous global rows
- Each submatrix consists of diagonal and off-diagonal parts

```
proc 5
proc 4
proc 3
proc 2
proc 1
proc 0
diagonal blocks
offdiagonal blocks
```

- `MatGetOwnershipRange(Mat A, int *start, int *end)`
  - `start`: first locally owned row of global matrix
  - `end-1`: last locally owned row of global matrix
Parallel Sparse Matrices

MatMPIAIJPreallocation(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 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 routine
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routine
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
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 bicgstab`
Nonlinear Solvers
Newton and Picard Methods

- Using PETSc linear algebra, just add:
  - \texttt{SNESSetFunction(SNES\ snes, \ Vec\ r, \ residualFunc, \ void* ctx)}
  - \texttt{SNESSetJacobian(SNES\ snes, \ Mat\ A, \ Mat\ M, \ jacFunc, \ void* ctx)}
  - \texttt{SNES Solve(SNES\ snes, \ Vec\ b, \ Vec\ x)}

- Can access subobjects
  - \texttt{SNESGetKSP(SNES\ snes, \ KSP* ksp)}

- Can customize subobjects from the cmd line
  - Set the subdomain preconditioner to ILU with \texttt{-sub_pc_type ilu}
Bottlenecks of (Jacobian-free) Newton-Krylov

- Matrix assembly
  - integration/fluxes: FPU
  - insertion: memory/branching
- Preconditioner setup
  - coarse level operators
  - overlapping subdomains
  - (incomplete) factorization
- Preconditioner application
  - triangular solves/relaxation: memory
  - coarse levels: network latency
- Matrix multiplication
  - Sparse storage: memory
  - Matrix-free: FPU
- Globalization
Intel Clowertown

- 75 Gflop/s
- 21 GB/s bandwidth
- thread + instruction level parallelism
- vector instructions (SSE)

AMD Opteron

- 17 Gflop/s
- 21 GB/s bandwidth
- thread + instruction level parallelism
- vector instructions (SSE)
Hardware capabilities

Floating point unit

Recent Intel: each core can issue

- 1 packed add (latency 3)
- 1 packed mult (latency 5)
- One can include a read
- Out of Order execution
- Peak: 10 Gflop/s (double)

Memory

- \( \sim 250 \) cycle latency
- 5.3 GB/s bandwidth
- 1 double load / 3.7 cycles
- Pay by the cache line (32/64 B)
- L2 cache: \( \sim 10 \) cycle latency
Performance and Scalability

Memory hierarchy

- Model faster cores by commenting out the inner kernel calls, but still performing all DMAs
- Enabled 1x1 BCOO
- ~16% improvement

(Olik et al. Multi-core Optimization of Sparse Matrix Vector Multiplication, 2008)
Sparse Mat-Vec performance model

Compressed Sparse Row format (AIJ)

For \( m \times n \) matrix with \( N \) nonzeros

- \( a_i \) row starts, length \( m + 1 \)
- \( a_j \) column indices, length \( N \), range \([0, n - 1)\)
- \( a_{a} \) nonzero entries, length \( N \), scalar values

\[
\text{for} \ (i = 0; i < m; i++)
\]
\[
y \leftarrow y + Ax
\]
\[
\text{for} \ (j = a_i[i]; j < a_i[i + 1]; j++)
\]
\[
y[i] += a_{a}[j] \times a_{j}[j];
\]

- One add and one multiply per inner loop
- Scalar \( a_{a}[j] \) and integer \( a_{j}[j] \) only used once
- Must load \( a_{j}[j] \) to read from \( x \), may not reuse cache well
Memory Bandwidth

- **Stream Triad benchmark (GB/s):** \( w \leftarrow \alpha x + y \)

<table>
<thead>
<tr>
<th>Threads per Node</th>
<th>Cray XT5 Total</th>
<th>Cray XT5 Per Core</th>
<th>BlueGene/P Total</th>
<th>BlueGene/P Per Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8448</td>
<td>8448</td>
<td>2266</td>
<td>2266</td>
</tr>
<tr>
<td>2</td>
<td>10112</td>
<td>5056</td>
<td>4529</td>
<td>2264</td>
</tr>
<tr>
<td>4</td>
<td>10715</td>
<td>2679</td>
<td>8903</td>
<td>2226</td>
</tr>
<tr>
<td>6</td>
<td>10482</td>
<td>1747</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- **Sparse matrix-vector product:** 6 bytes per flop

<table>
<thead>
<tr>
<th>Machine</th>
<th>Peak MFlop/s per core</th>
<th>Bandwidth (GB/s)</th>
<th>Ideal MFlop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Required</td>
<td>Measured</td>
</tr>
<tr>
<td>Blue Gene/P</td>
<td>3,400</td>
<td>20.4</td>
<td>2.2</td>
</tr>
<tr>
<td>XT5</td>
<td>10,400</td>
<td>62.4</td>
<td>1.7</td>
</tr>
</tbody>
</table>
Optimizing Sparse Mat-Vec

- Order unknowns so that vector reuses cache (Reverse Cuthill-McKee)
  - Optimal: \(\frac{(2 \text{ flops})(\text{bandwidth})}{\text{sizeof}(\text{Scalar})+\text{sizeof}(\text{Int})}\)
  - Usually improves strength of ILU and SOR

- Coalesce indices for adjacent rows with same nonzero pattern (Inodes)
  - Optimal: \(\frac{(2 \text{ flops})(\text{bandwidth})}{\text{sizeof}(\text{Scalar})+\text{sizeof}(\text{Int})/i}\)
  - Can do block SOR (much stronger than scalar SOR)
  - Default in PETSc, turn off with \(-\text{mat}_\text{no}_\text{inode}\)
  - Requires ordering unknowns so that fields are interlaced, this is (much) better for memory use anyway

- Use explicit blocking, hold one index per block (BAIJ format)
  - Optimal: \(\frac{(2 \text{ flops})(\text{bandwidth})}{\text{sizeof}(\text{Scalar})+\text{sizeof}(\text{Int})/b^2}\)
  - Block SOR and factorization
  - Symbolic factorization works with blocks (much cheaper)
  - Very regular memory access, unrolled dense kernels
  - Faster insertion: \texttt{MatSetValuesBlocked()}
Optimizing unassembled Mat-Vec

- High order spatial discretizations do more work per node
  - Dense tensor product kernel (like small BLAS3)
  - Cubic ($Q_3$) elements in 3D can achieve $> 60\%$ of peak FPU
    (compare to $< 6\%$ for assembled operators on multicore)
  - Can store Jacobian information at quadrature points
    (usually pays off for $Q_2$ and higher in 3D)
  - Spectral methods
  - Often still need an assembled operator for preconditioning

- Boundary element methods
  - Dense kernels
  - Fast Multipole Method (FMM)
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`

- **Time (sec):**
  - Max: 1.548e+02
  - Max/Min: 1.00122
  - Avg: 1.547e+02
  - Total: 1.028e+03

- **Objects:**
  - Max: 1.028e+03
  - Max/Min: 1.00000
  - Avg: 1.028e+03

- **Flops:**
  - Max: 1.519e+10
  - Max/Min: 1.01953
  - Avg: 1.505e+10
  - Total: 1.204e+11

- **Flops/sec:**
  - Max: 9.814e+07
  - Max/Min: 1.01829
  - Avg: 9.727e+07
  - Total: 7.782e+08

- **MPI Messages:**
  - Max: 8.854e+03
  - Max/Min: 1.00556
  - Avg: 8.819e+03
  - Total: 7.055e+04

- **MPI Message Lengths:**
  - Max: 1.936e+08
  - Max/Min: 1.00950
  - Avg: 2.185e+04
  - Total: 1.541e+09

- **MPI Reductions:**
  - Max: 2.799e+03
  - Max/Min: 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**
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
### Scalability definitions

#### Strong scalability
- Fixed problem size
- Execution time $T$ inversely proportional to number of processors $p$

#### Weak scalability
- Fixed problem size per processor
- Execution time constant as problem size increases
Multigrid advice

• Rapid coarsening is essential for weak scalability
  • Push the algorithm towards “multilevel domain decomposition”

• Energy minimizing interpolants
  • Similar to exotic Schwarz methods, see Dohrmann and Widlund 2008, 2009
  • Closely related to FETI-DP/BDDC coarse spaces

• Interpolation operators must be compatible with physics (e.g. inf-sup conditions)

• Ordering of unknowns can make incomplete factorization behave similar to line smoothers

• Nonlinear multigrid (FAS) is worth trying if pointwise or block residuals are cheap, or globalization is especially challenging

• Monotone multigrid (Kornhuber) for variational inequalities

• Boundary conditions in subdomain problems (“optimized Schwarz”)
Splitting for Multiphysics

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

- **Relaxation**: \(-pc\_fieldsplit\_type\)  
  \[\text{[additive, multiplicative, symmetric\_multiplicative]}\]

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

- Gauss-Seidel inspired, works when fields are loosely coupled
- **Factorization**: \(-pc\_fieldsplit\_type\) \text{ schur}\n
\[
\begin{bmatrix}
1 \\
CA^{-1} & 1
\end{bmatrix}
\begin{bmatrix}
A & B \\
S &
\end{bmatrix},
\quad S = D - CA^{-1}B
\]

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

- Smooth all components together
  - Block SOR is the most popular
  - Vanka smoothers for indefinite problems
- Interpolate in a compatible way
- Scaling between fields is critical
- First-order upwind for transport
- Coarse space and subdomain problems should be compatible with inf-sup condition
- Open research area
Anisotropy and Heterogeneity

- Anisotropy
  - Semi-coarsening
  - Line smoothers
  - Order unknowns so that incomplete factorization “includes” a line smoother

- Heterogeneity
  - Make coarse grids align
  - Strong smoothers
  - Energy-minimizing interpolants
Physics-based preconditioners (semi-implicit method)

Shallow water with stiff gravity wave

$h$ is hydrostatic pressure, $u$ is velocity, $\sqrt{gh}$ is fast wave speed

\[
\begin{align*}
    h_t - (uh)_x &= 0 \\
    (uh)_t + (u^2 h + \frac{1}{2} gh^2)_x &= 0
\end{align*}
\]

Semi-implicit method

Suppress spatial discretization, discretize in time, implicitly for the terms contributing to the gravity wave

\[
\begin{align*}
    \frac{h^{n+1} - h^n}{\Delta t} + (uh)_{x}^{n+1} &= 0 \\
    \frac{(uh)^{n+1} - (uh)^n}{\Delta t} + (u^2 h)_x^n + gh^n h_x^{n+1} &= 0
\end{align*}
\]

Rearrange, eliminating $(uh)^{n+1}$

\[
\begin{align*}
    \frac{h^{n+1} - h^n}{\Delta t} - \Delta t(g h^n h_x^{n+1})_x &= -S^n_x
\end{align*}
\]
Delta form

- Preconditioner should work like the Newton step
  \[-F(x) \mapsto \delta x\]

- Recast semi-implicit method in delta form
  \[
  \frac{\delta h}{\Delta t} + (\delta uh)_x = -F_0, \\
  \frac{\delta uh}{\Delta t} + gh^n(\delta h)_x = -F_1
  \]

- Eliminate \(\delta uh\)
  \[
  \frac{\delta h}{\Delta t} - \Delta t(gh^n(\delta h)_x)_x = -F_0 + (\Delta tF_1)_x
  \]

- Solve for \(\delta h\), then evaluate
  \[
  \delta uh = -\Delta t[gh^n(\delta h)_x - F_1]
  \]

- Fully implicit solver
  - Is nonlinearly consistent (no splitting error)
  - Can be high-order in time
  - Leverages existing code (the semi-implicit method)
References