The Portable Extensible Toolkit for Scientific computing

Day 1: Usage and Algorithms

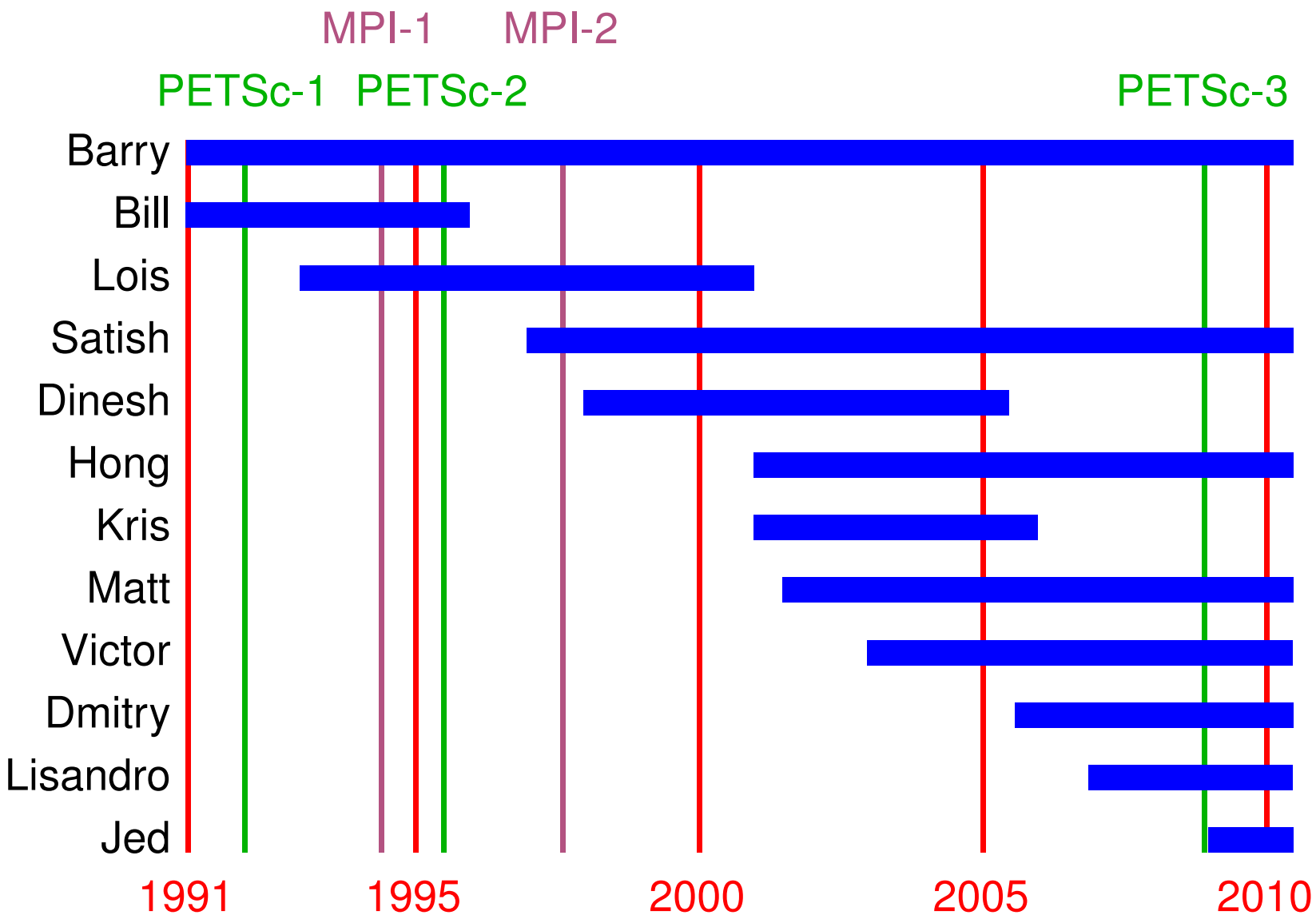
Jed Brown

CSCS 2010-05-10

Outline

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`


MPI-1
MPI-2
PETSc-1
PETSc-2
PETSc-3
Barry
Bill
Lois
Satish
Dinesh
Hong
Kris
Matt
Victor
Dmitry
Lisandro
Jed
1991
1995
2000
2005
2010

Portable Extensible Toolkit for Scientific computing

- Architecture
  - tightly coupled (XT5, BG/P, Earth Simulator, Sun Blade, SGI Altix)
  - 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)
- 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 (XT5, BG/P, Earth Simulator, Sun Blade, SGI Altix)
  - 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)
- 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, 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`

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

Downloading

- `http://mcs.anl.gov/petsc`, download tarball
- We will use Mecurial 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/petsc-releases/petsc-release-3.1`
  - `$ cd petsc-release-3.1`
  - `$ hg clone http://petsc.cs.iit.edu/petsc/`
    `BuildSystem-releases/BuildSystem-release-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, lgrind, 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

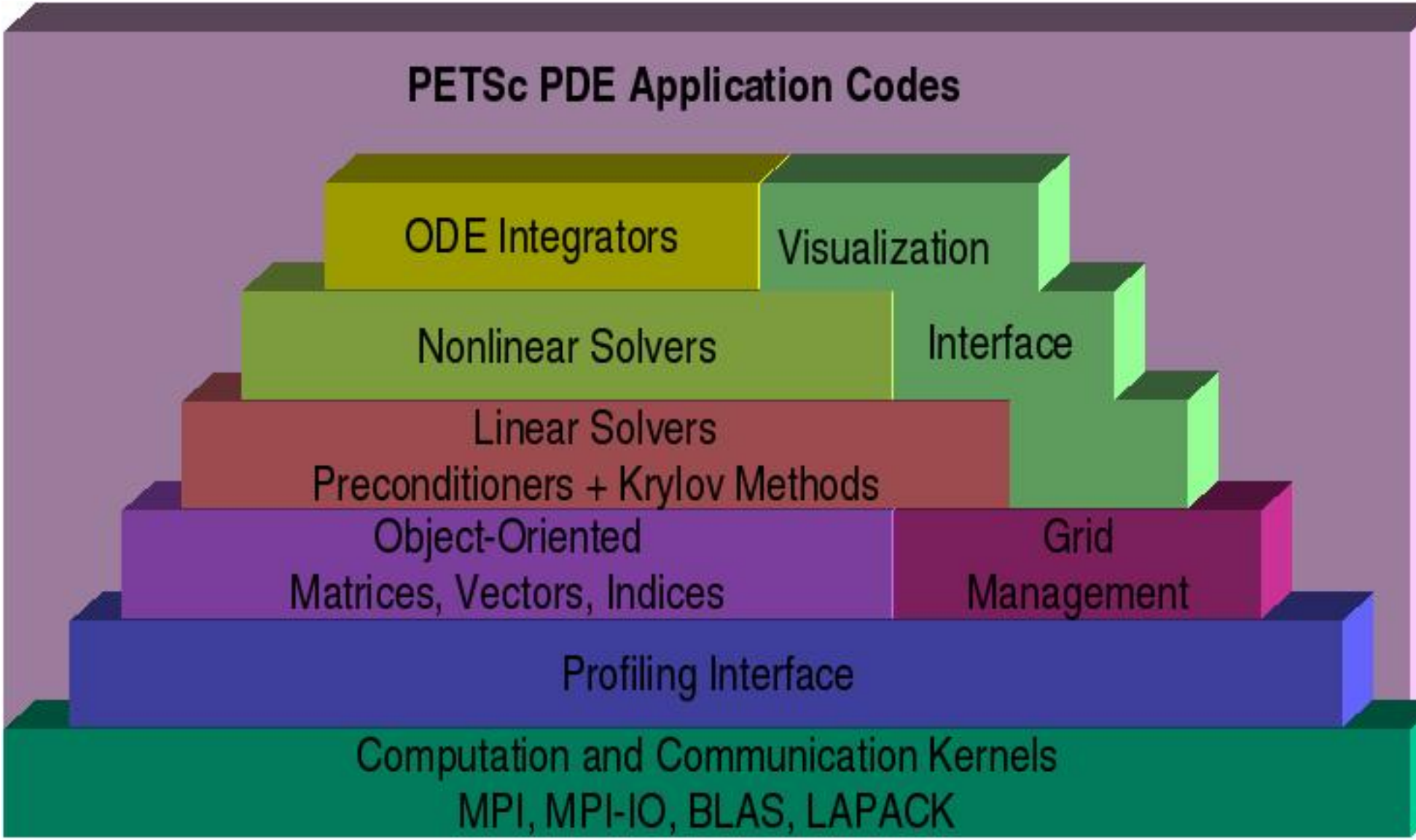
- `$ mpich-gcc-dbg/conf/reconfigure-mpich-gcc-dbg.py PETSC_ARCH=mpich-gcc-opt --with-debugging=0 && make PETSC_ARCH=mpich-gcc-opt`
- What does `--with-debugging=1` (default) do?
  - Keeps debugging symbols (of course)
  - Maintains a stack so that errors produce a full stack trace (even SEGV)
  - Does 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`

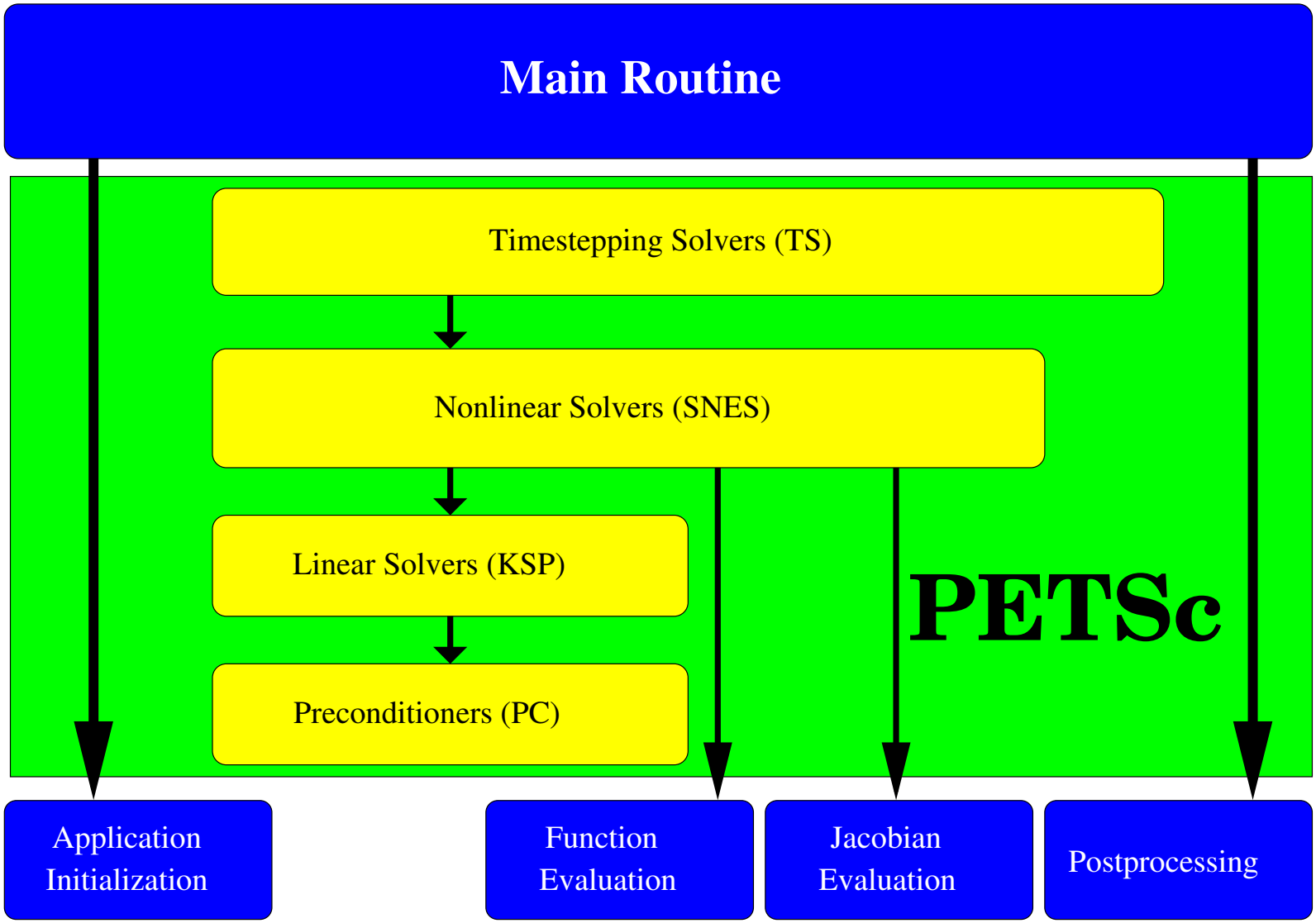
Compiling an example

- `$ hg clone http://petsc.cs.iit.edu/petsc/tutorials/CSCS10TutorialCode`
- `$ cd CSCS2010TutorialCode`
- `$ hg update -r1`
- `$ make`
- `$ ./pbratu -help | less`

PETSc Structure



Flow Control for a PETSc Application



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

Function	Operation
`Create()`	create the object
`Get/SetName()`	name the object
`Get/SetType()`	set the implementation type
`Get/SetOptionsPrefix()`	set the prefix for all options
`SetFromOptions()`	customize object from the command line
`SetUp()`	preform other initialization
`View()`	view the object
`Destroy()`	cleanup object allocation

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

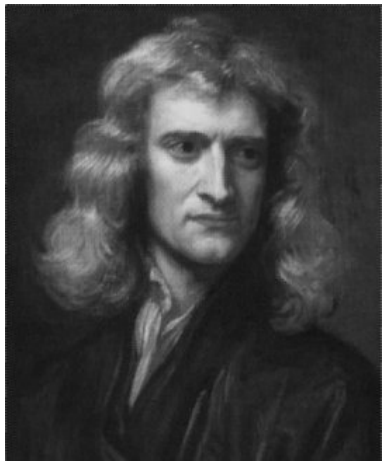
```
$ cd $PETSC_DIR/src/snes/examples/tutorials && make ex5
```

- `$ ./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

$$\begin{aligned} \text{Solve:} \quad & J(u)w = -F(u) \\ \text{Update:} \quad & u^+ \leftarrow u + w \end{aligned}$$

- Quadratically convergent near a root: $\left|u^{n+1} - u^*\right| \in \mathscr{O}\left(\left|u^n - u^*\right|^2\right)$
- Picard is the same operation with a different $J(u)$

Example (Nonlinear Poisson)

$$\begin{aligned} F(u) = 0 \quad &\sim \quad -\nabla \cdot \left[(1+u^2)\nabla u\right] - f = 0 \\ J(u)w \quad &\sim \quad -\nabla \cdot \left[(1+u^2)\nabla w + 2uw\nabla u\right] \end{aligned}$$

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 $Jy = \lim_{\varepsilon \to 0} \frac{F(x+\varepsilon y) - F(x)}{\varepsilon}$
4. Fourier transform $\mathscr{F}, \mathscr{F}^{-1}$
5. Other fast transforms, e.g. Fast Multipole Method
6. Low rank correction $B = A + uv^T$
7. Schur complement $S = D - CA^{-1}B$
8. Tensor product $A = \sum_e A_x^e \otimes A_y^e \otimes A_z^e$
9. Linearization of a few steps of an explicit integrator

Important matrices

1. Sparse (e.g. discretization of a PDE operator)
2. Inverse of *anything* interesting $B = A^{-1}$
3. Jacobian of a nonlinear function $Jy = \lim_{\varepsilon \to 0} \frac{F(x+\varepsilon y) - F(x)}{\varepsilon}$
4. Fourier transform $\mathscr{F}, \mathscr{F}^{-1}$
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 $\mathscr{F}$, $\mathscr{F}^{-1}$
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

- Thes 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 $\mathscr{F}, \mathscr{F}^{-1}$
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"

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

Krylov solvers for $Ax = b$

- Krylov subspace: $\{b, Ab, A^2b, A^3b, \dots\}$
- Convergence rate depends on the spectral properties of the matrix
  - Existance of small polynomials $p_n(A) < \varepsilon$ where $p_n(0) = 1$.
  - condition number $\kappa(A) = \|A\| \left\|A^{-1}\right\| = \sigma_{\max}/\sigma_{\min}$
  - distribution of singular values, spectrum Λ, pseudospectrum Λ_ε
- For any popular Krylov method $\mathscr{K}$, there is a matrix of size m, such that $\mathscr{K}$ outperforms all other methods by a factor at least $\mathscr{O}(\sqrt{m})$ [Nachtigal et. al., 1992]

Typically...

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

GMRES

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

1. Use Arnoldi to orthogonalize the nth subspace, producing

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

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

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

   using QR-decomposition, updated cheaply at each iteration.

Properties

- Converges in n steps for all right hand sides if there exists a polynomial of degree n such that $\|p_n(A)\| < tol$ and $p_n(0) = 1$.
- Residual is monotonically decreasing, robust in practice
- Restarted variants are used to bound memory requirements

The $\mathfrak{p}$-Bratu equation

- 2-dimensional model problem

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

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

- Regularized variant

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

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

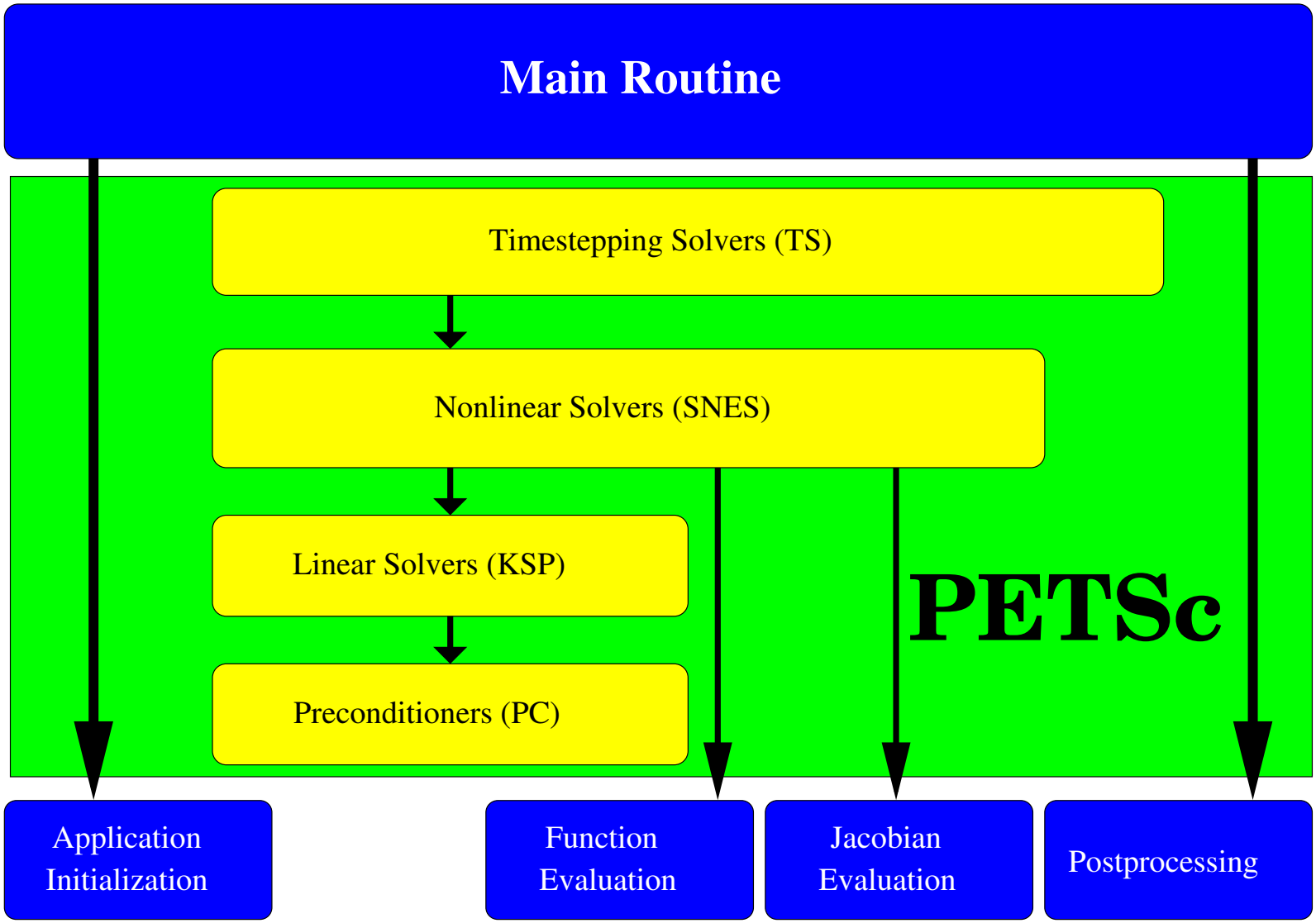
- Jacobian

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

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

  Physical interpretation: conductivity tensor flattened in direction ∇u

Flow Control for a PETSc Application



SNES Paradigm

The SNES interface is based upon callback functions

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

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

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

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

Alternatively, you can use

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

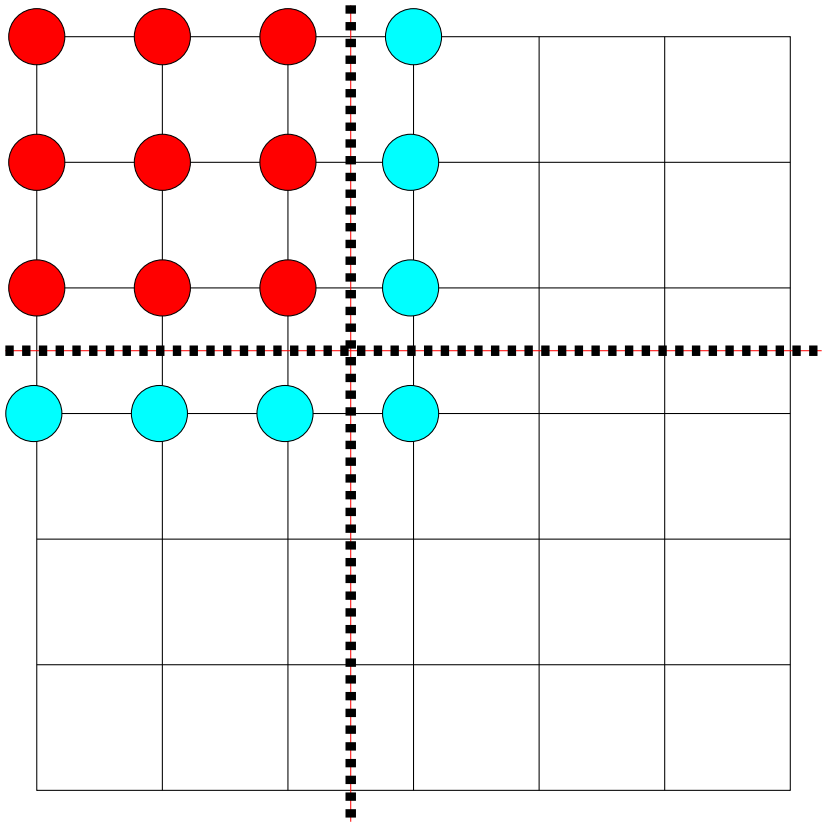
Distributed Array

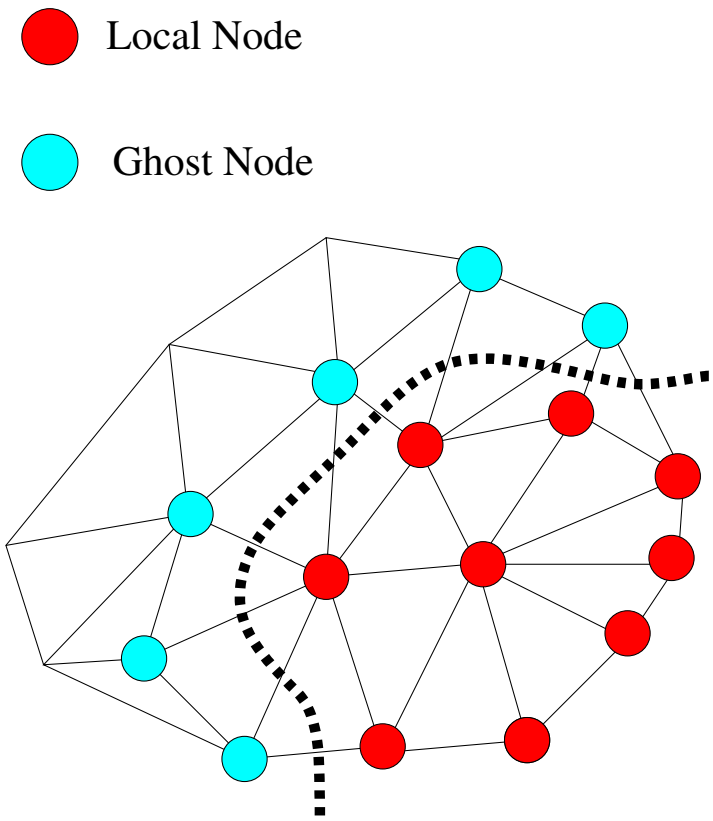
- Interface for topologically structured grids
- Defines (topological part of) a finite-dimensional function space
  - Get an element from this space: `DACreateGlobalVector()`
- Provides parallel layout
- Refinement and coarsening
  - `DARefineHierarchy()`
- Ghost value coherence
  - `DAGlobalToLocalBegin()`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

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

Natural numbering

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

PETSc numbering

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

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

Local numbering

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

Global numbering

DA Vectors

- The DA object contains only layout (topology) information
  - All field data is contained in PETSc `Vec`s
- Global vectors are parallel
  - Each process stores a unique local portion
  - `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

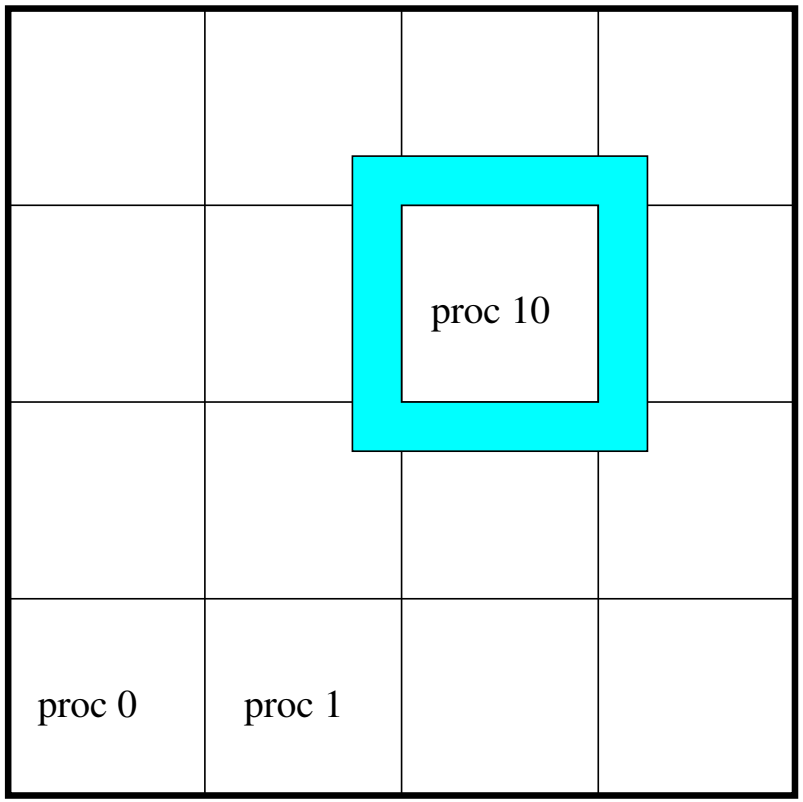
Two-step process enables overlapping computation and communication

- `DAGlobalToLocalBegin(da, gvec, mode, lvec)`
  - `gvec` provides the data
  - `mode` is either `INSERT_VALUES` or `ADD_VALUES`
  - `lvec` holds the local and ghost values
- `DAGlobalToLocalEnd(da, gvec, mode, lvec)`
  - Finishes the communication

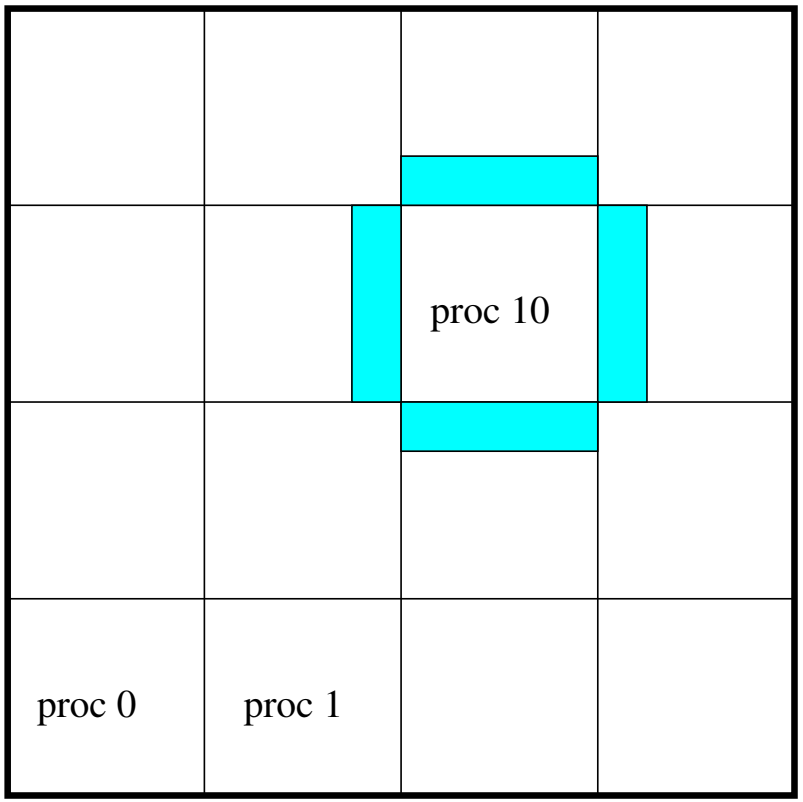
The process can be reversed with `DALocalToGlobal()`.

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

```
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

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

Bratu Residual Evaluation

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

```
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 `-snes_mf`
- `$ hg update -r3`
- `$ ./pbratu -da_grid_x 10 -da_grid_y 10 -lambda 6.7 -snes_mf -snes_monitor -ksp_converged_reason`
- `$ ./pbratu -da_grid_x 20 -da_grid_y 20 -lambda 6.7 -snes_mf -snes_monitor -ksp_converged_reason`
- `$ ./pbratu -da_grid_x 40 -da_grid_y 40 -lambda 6.7 -snes_mf -snes_monitor -ksp_converged_reason`
- Watch linear and nonlinear convergence

Add $\mathfrak{p}$ nonlinearity

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

Preconditioning

Idea: improve the conditioning of the Krylov operator

- Left preconditioning

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

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

- Right preconditioning

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

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

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

Definition (Preconditioner)

A *preconditioner* $\mathscr{P}$ is a method for constructing a matrix (just a linear function, not assembled!) $P^{-1} = \mathscr{P}(A, A_p)$ 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* $\mathscr{P}$ is a method for constructing a matrix $P^{-1} = \mathscr{P}(A, A_p)$ 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 $\mathscr{P}$, 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)

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

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

- Implemented as a sweep
- $\omega = 1$ corresponds to Gauss-Seidel
- Very effective at removing high-frequency components of residual

Factorization

Two phases

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

LU decomposition

- Ultimate preconditioner
- Expensive, for $m \times m$ sparse matrix with bandwidth b, traditionally requires $\mathcal{O}(mb^2)$ time and $\mathcal{O}(mb)$ space.
  - Bandwidth scales as $m^{\frac{d-1}{d}}$ in d-dimensions
- 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

Overlapping/Schwarz

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

Neumann-Neumann

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

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

Multigrid

Hierarchy: Interpolation and restriction operators

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

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

Galerkin approximation

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

Application of multigrid preconditioner (V-cycle)

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

- `$ hg update -r5`
- `$ ./pbratu -da_grid_x 10 -da_grid_y 10 -lambda 1 -p 1.3 -snes_fd -snes_monitor -ksp_converged_reason`
- `$ ./pbratu -da_grid_x 10 -da_grid_y 10 -lambda 1 -p 1.3 -fd_jacobian -snes_monitor -ksp_converged_reason`
- `$ ./pbratu -da_grid_x 10 -da_grid_y 10 -lambda 1 -p 1.3 -fd_jacobian -snes_monitor -ksp_converged_reason`
- Try some different preconditioners (`jacobi`, `sor`, `asm`, `hypre`, `ml`)
- Try changing the physical parameters
- May need `-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
- `MatMPIBAIJSetPreallocation(Mat,...)`
  - important for assembly performance, more tomorrow
- `MatSetBlockSize(Mat, int bs)`
  - for vector problems
- `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; 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

Simple 3-point stencil for 1D Laplacian

```
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);
```

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)

$\mathfrak{p}$-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`

$\mathfrak{p}$-Bratu assembly

- We need to assemble the $\mathfrak{p}$ part

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

- 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 $\mathfrak{p}$
- 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 the debug PETSC_ARCH, right?
- Launch the debugger
  - `-start_in_debugger [gdb,dbx,noxterm]`
  - `-on_error_attach_debugger [gdb,dbx,noxterm]`
- Attach the debugger only to some parallel processes
  - `-debugger_nodes 0,1`
- Set the display (often necessary on a cluster)
  - `-display :0`

Debugging Tips

- Put a breakpoint in `PetscError()` to catch errors as they occur
- PETSc tracks memory overwrites at both ends of arrays
  - The `CHKMEMQ` macro causes a check of all allocated memory
  - Track memory overwrites by bracketing them with `CHKMEMQ`
- PETSc checks for leaked memory
  - Use `PetscMalloc()` and `PetscFree()` for all allocation
  - Print unfreed memory on `PetscFinalize()` with `-malloc_dump`
- Simply the best tool today is Valgrind
  - It checks memory access, cache performance, memory usage, etc.
  - http://www.valgrind.org
  - Pass `-malloc 0` to PETSc when running under Valgrind
  - Might need `--trace-children=yes` when running under MPI
  - `--track-origins=yes` handy for uninitialized memory

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?

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 -r11`
- Check with `-snes_type test`
- and `-snes_mf_operator -pc_type lu`