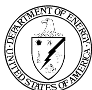
PETSc Tutorial

PETSc Team
Presented by Matthew Knepley

Mathematics and Computer Science Division
Argonne National Laboratory

ACTS Workshop
Lawrence Berkeley National Laboratory
August 23, 2006



Units

1. Getting Started with PETSc
2. Common PETSc Usage
3. PETSc Essentials
4. PETSc Integration
5. Advanced PETSc
6. PETSc Extensibility
7. PETSc Optimization
8. Future Plans

Part I

Getting Started with PETSc

Unit Objectives

- Introduce the
  Portable Extensible Toolkit for Scientific Computation
- Retrieve, Configure, Build, and Run a PETSc Example
- Empower students to learn more about PETSc

What I Need From You

- Tell me if you do not understand
- Tell me if an example does not work
- Suggest better wording or figures
- Followup problems at petsc-maint@mcs.anl.gov

How Can We Help?

- Provide documentation
- Answer email at petsc-maint@mcs.anl.gov

How Can We Help?

- Provide documentation
- Quickly answer questions

- Answer email at petsc-maint@mcs.anl.gov

How Can We Help?

- Provide documentation
- Quickly answer questions
- Help install

- Answer email at petsc-maint@mcs.anl.gov

How Can We Help?

- Provide documentation
- Quickly answer questions
- Help install
- Guide large scale code development
- Answer email at 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.

What is PETSc?

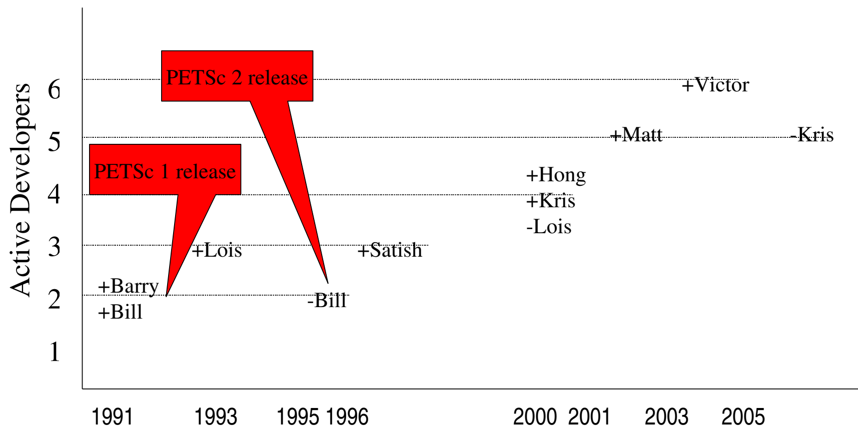
A freely available and supported research code

- Download from http://www.mcs.anl.gov/petsc
- Free for everyone, including industrial users
- Hyperlinked manual, examples, and manual pages for all routines
- Hundreds of tutorial-style examples
- Support via email: petsc-maint@mcs.anl.gov
- Usable from C, C++, Fortran 77/90, and Python

What is PETSc?

- Portable to any parallel system supporting MPI, including:
  - Tightly coupled systems
    - Cray T3E, SGI Origin, IBM SP, HP 9000, Sub Enterprise
  - Loosely coupled systems, such as networks of workstations
    - Compaq,HP, IBM, SGI, Sun, PCs running Linux or Windows
- PETSc History
  - Begun September 1991
  - Over 8,500 downloads since 1995 (version 2), currently 250 per month
- PETSc Funding and Support
  - Department of Energy
    - SciDAC, MICS Program
  - National Science Foundation
    - CIG, CISE, Multidisciplinary Challenge Program

Timeline



What Can We Handle?

- PETSc has run problems with over 500 million unknowns
  - http://www.scconference.org/sc2004/schedule/pdfs/pap111.pdf

What Can We Handle?

- PETSc has run problems with over 500 million unknowns
  - http://www.scconference.org/sc2004/schedule/pdfs/pap111.pdf
- PETSc has run on over 6,000 processors efficiently
  - ftp://info.mcs.anl.gov/pub/tech_reports/reports/P776.ps.Z

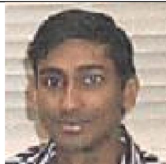
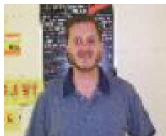
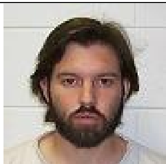
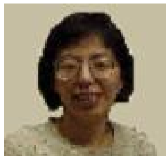
What Can We Handle?

- PETSc has run problems with over 500 million unknowns
  - http://www.scconference.org/sc2004/schedule/pdfs/pap111.pdf
- PETSc has run on over 6,000 processors efficiently
  - ftp://info.mcs.anl.gov/pub/tech_reports/reports/P776.ps.Z
- PETSc applications have run at 2 Teraflops
  - LANL PFLOTRAN code

Who Uses PETSc?

- Computational Scientists
  - PyLith (TECTON), Underworld, Columbia group
- Algorithm Developers
  - Iterative methods and Preconditioning researchers
- Package Developers
  - SLEPc, TAO, MagPar, StGermain, DealII

The PETSc Team

Bill Gropp	Barry Smith	 Satish Balay
Dinesh Kaushik	 Kris Buschelman	 Matt Knepley
 Hong Zhang	Victor Eijkhout	Lois McInnes

Downloading PETSc

- The latest tarball is on the PETSc site
  - ftp://ftp.mcs.anl.gov/pub/petsc/petsc.tar.gz
  - We no longer distribute patches (everything is in the distribution)
- There is a Debian package
- There is a FreeBSD Port
- There is a Mercurial development repository

Cloning PETSc

- The full development repository is open to the public
  - http://mercurial.mcs.anl.gov/petsc/petsc-dev
  - http://mercurial.mcs.anl.gov/petsc/BuildSystem
- Why is this better?
  - You can clone to any release (or any specific ChangeSet)
  - You can easily rollback changes (or releases)
  - You can get fixes from us the same day

Unpacking PETSc

- Just clone development repository
  - `hg clone http://mercurial.mcs.anl.gov/petsc/petsc-dev petsc-dev`
  - `hg clone -rRelease-2.3.1 petsc-dev petsc-2.3.1`

or

- Unpack the tarball
  - `tar xzf petsc.tar.gz`

Exercise 1

Download and Unpack PETSc!

Configuring PETSc

- Set $PETSC_DIR to the installtion root directory
- Run the configuration utility
  - $PETSC_DIR/config/configure.py
  - $PETSC_DIR/config/configure.py --help
  - $PETSC_DIR/config/configure.py --download-mpich
- There are many examples on the installation page
- Configuration files are placed in $PETSC_DIR/bmake/$PETSC_ARCH
  - $PETSC_ARCH has a default if not specified

Configuring PETSc

- You can easily reconfigure with the same options
  - `./bmake/$PETSC_ARCH/configure.py`
- Can maintain several different configurations
  - `./config/configure.py -PETSC_ARCH=linux-fast --with-debugging=0`
- All configuration information is in `configure.log`
  - ALWAYS send this file with bug reports

Automatic Downloads

- Starting in 2.2.1, some packages are automatically
  - Downloaded
  - Configured and Built (in `$PETSC_DIR/externalpackages`)
  - Installed in PETSc
- Currently works for
  - PETSc documentation utilities (Sowing, lgrind, c2html)
  - BLAS, LAPACK, BLACS, ScaLAPACK, PLAPACK
  - MPICH, MPE, LAM
  - ParMetis, Chaco, Jostle, Party, Scotch
  - MUMPS, Spooles, SuperLU, SuperLU_Dist, UMFPack
  - Prometheus, HYPRE, ML, SPAI
  - Sundials
  - Triangle, TetGen

Exercise 2

Configure the PETSc that you downloaded and unpacked.

Building PETSc

- Uses recursive make starting in `cd $PETSC_DIR`
  - `make`
  - Check build when done with `make test`
- Complete log for each build in `make_log_$PETSC_ARCH`
  - ALWAYS send this with bug reports
- Can build multiple configurations
  - `PETSC_ARCH=linux-fast make`
  - Libraries are in `$PETSC_DIR/lib/$PETSC_ARCH/`
- Can also build a subtree
  - `cd src/snes; make`
  - `cd src/snes; make ACTION=libfast tree`

Exercise 3

Build the PETSc that you configured.

Exercise 4

Reconfigure PETSc to use ParMetis.

1. `./bmake/linux-gnu/configure.py`
   - `-PETSC_ARCH=linux-parmetis`
   - `-download-parmetis`
2. `PETSC_ARCH=linux-parmetis make`
3. `PETSC_ARCH=linux-parmetis make test`

Running PETSc

- Try running PETSc examples first
  - `cd $PETSC_DIR/src/snes/examples/tutorials`
- Build examples using make targets
  - `make ex5`
- Run examples using the make target
  - `make runex5`
- Can also run using MPI directly
  - `mpirun ./ex5 -snes_max_it 5`
  - `mpiexec ./ex5 -snes_monitor`

Using MPI

- The Message Passing Interface is:
  - a library for parallel communication
  - a system for launching parallel jobs (mpirun/mpiexec)
  - a community standard
- Launching jobs is easy
  - `mpiexec -np 4 ./ex5`
- You should never have to make MPI calls when using PETSc
  - Almost never

MPI Concepts

- Communicator
  - A context (or scope) for parallel communication ("Who can I talk to")
  - There are two defaults:
    - yourself (PETSC_COMM_SELF),
    - and everyone launched (PETSC_COMM_WORLD)
  - Can create new communicators by splitting existing ones
  - Every PETSc object has a communicator
- Point-to-point communication
  - Happens between two processes (like in `MatMult()`)
- Reduction or scan operations
  - Happens among all processes (like in `VecDot()`)

Alternative Memory Models

- Single process (address space) model
  - OpenMP and threads in general
  - Fortran 90/95 and compiler-discovered parallelism
  - System manages memory and (usually) thread scheduling
  - Named variables refer to the same storage
- Single name space model
  - HPF, UPC
  - Global Arrays
  - Titanium
  - Named variables refer to the coherent values (distribution is automatic)
- Distributed memory (shared nothing)
  - Message passing
  - Names variables in different processes are unrelated

Common Viewing Options

- Gives a text representation
  - `-vec_view`
- Generally views subobjects too
  - `-snes_view`
- Can visualize some objects
  - `-mat_view_draw`
- Alternative formats
  - `-vec_view_binary`, `-vec_view_matlab`, `-vec_view_socket`
- Sometimes provides extra information
  - `-mat_view_info`, `-mat_view_info_detailed`

Common Monitoring Options

- Display the residual
  - `-ksp_monitor`, graphically `-ksp_xmonitor`
- Can disable dynamically
  - `-ksp_cancelmonitors`
- Does not display subsolvers
  - `-snes_monitor`
- Can use the true residual
  - `-ksp_truemonitor`
- Can display different subobjects
  - `-snes_vecmonitor`, `-snes_vecmonitor_update`, `-snes_vecmonitor_residual`
  - `-ksp_gmres_krylov_monitor`
- Can display the spectrum
  - `-ksp_singmonitor`

Exercise 5

Run SNES Example 5 using come custom options.

1. `cd $PETSC_DIR/src/snes/examples/tutorials`
2. `make ex5`
3. `mpiexec ./ex5 -snes_monitor -snes_view`
4. `mpiexec ./ex5 -snes_type tr -snes_monitor -snes_view`
5. `mpiexec ./ex5 -ksp_monitor -snes_monitor -snes_view`
6. `mpiexec ./ex5 -pc_type jacobi -ksp_monitor -snes_monitor -snes_view`
7. `mpiexec ./ex5 -ksp_type bicg -ksp_monitor -snes_monitor -snes_view`

Exercise 6

Create a new code based upon SNES Example 5.

1. Create a new directory
   - `mkdir -p /home/knepley/proj/newsim/src`
2. Copy the source
   - `cp ex5.c /home/knepley/proj/newsim/src`
3. Create a PETSc makefile
   - Add a link target
   - `${CLINKER} -o $@ $^ ${PETSC_SNES_LIB}`
   - `${FLINKER} -o $@ $^ ${PETSC_FORTRAN_LIB} ${PETSC_SNES_LIB}`
   - `include ${PETSC_DIR}/bmake/common/base`

Getting More Help

- http://www.mcs.anl.gov/petsc
- Hyperlinked documentation
  - Manual
  - Manual pages for evey method
  - HTML of all example code (linked to manual pages)
- FAQ
- Full support at petsc-maint@mcs.anl.gov
- High profile users
  - David Keyes
  - Rich Martineau
  - Richard Katz
  - Charles Williams

Part II

Common PETSc Usage

Correctness Debugging

- Automatic generation of tracebacks
- Detecting memory corruption and leaks
- Optional user-defined error handlers

Interacting with the Debugger

- 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 khan.mcs.anl.gov:0.0`

Debugging Tips

- Putting a breakpoint in `PetscError()` can catch errors as they occur
- PETSc tracks memory overwrites at the beginning and end 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
  - Option `-trmalloc` will print unfreed memory on `PetscFinalize()`

Exercise 1

Use the debugger to find a SEGV
Locate a memory overwrite using CHKMEMQ.

- Get the example
  - `hg clone -r1.4 bk://mercurial.mcs.anl.gov/petsc/tutorialExercise1`
- Build the example `make`
- Run it `make run` and watch the fireworks
- Run it under the debugger `make debug` and correct the error
- Build it and run again `make ex1 run` to catch the memory overwrite
- Correct the error, build it and run again `make ex1 run`

Performance Debugging

- PETSc has integrated profiling
  - Option `-log_summary` prints a report on `PetscFinalize()`
- PETSc allows user-defined events
  - Events report time, calls, flops, communication, etc.
  - Memory usage is tracked by object
- Profiling is separated into stages
  - Event statistics are aggregated by stage

Using Stages and Events

- Use `PetscLogStageRegister()` to create a new stage
  - Stages are identifier by an integer handle
- Use `PetscLogStagePush/Pop()` to manage stages
  - Stages may be nested and will aggregate in a nested fashion
- Use `PetscLogEventRegister()` to create a new stage
  - Events also have an associated class
- Use `PetscLogEventBegin/End()` to manage events
  - Events may also be nested and will aggregate in a nested fashion
  - Can use `PetscLogFlops()` to log user flops

Adding A Logging Stage

```
int stageNum;

ierr = PetscLogStageRegister(&stageNum, "name");CHKERRQ(ierr);
ierr = PetscLogStagePush(stageNum);CHKERRQ(ierr);

  Code to Monitor

ierr = PetscLogStagePop();CHKERRQ(ierr);
```

Adding A Logging Event

```
static int USER_EVENT;

ierr = PetscLogEventRegister(&USER_EVENT, "name", CLASS_COOKIE);CH
ierr = PetscLogEventBegin(USER_EVENT,0,0,0,0);CHKERRQ(ierr);

  Code to Monitor

ierr = PetscLogFlops(user_event_flops);CHKERRQ(ierr);
ierr = PetscLogEventEnd(USER_EVENT,0,0,0,0);CHKERRQ(ierr);
```

Adding A Logging Class

```
static int CLASS_COOKIE;

ierr = PetscLogClassRegister(&CLASS_COOKIE,"name");CHKERRQ(ierr);
```

- Cookie identifies a class uniquely
- Initialization must happen before any objects of this type are created

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

Efficient Matrix Creation

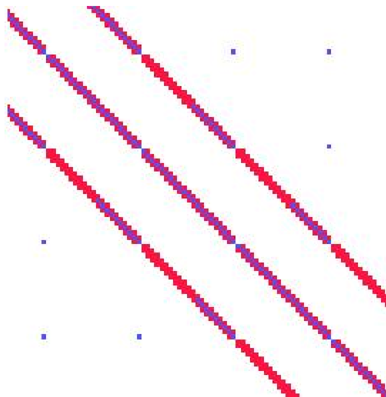
- Create matrix with `MatCreate()`
- Set type with `MatSetType()`
- Determine the number of nonzeros in each row
  - loop over the grid for finite differences
  - loop over the elements for finite elements
  - need only local+ghost information
- Preallocate matrix
  - `MatSeqAIJSetPreallocation()`
  - `MatMPIAIJSetPreallocation()`

Indicating Expected Nonzeros
Sequential Sparse Matrices

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

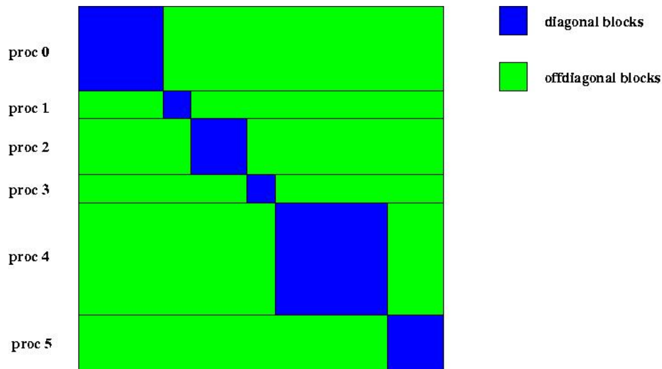
nz: expected number of nonzeros in any row

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



ParallelSparseMatrix

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



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

`start`: first locally owned row of global matrix

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

Indicating Expected Nonzeros

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

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

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

nnz(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!
```

Exercise 2

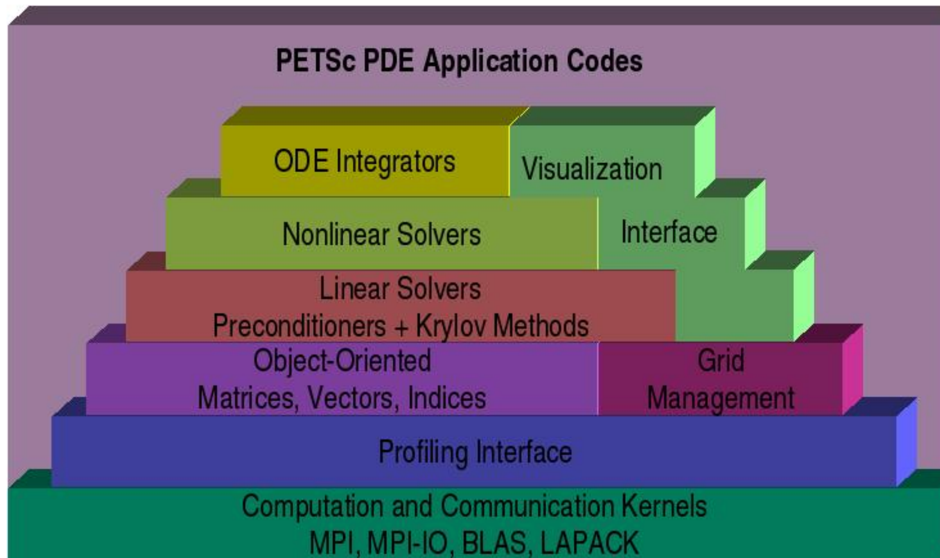
Return to Part 1:Execise 6 and add more profiling.

- Run it `make profile` and look at the profiling report
- Add a new stage for setup
- Add a new event for `FormInitialGuess()` and log the flops
- Run it again `make ex5 profile` and look at the profiling report

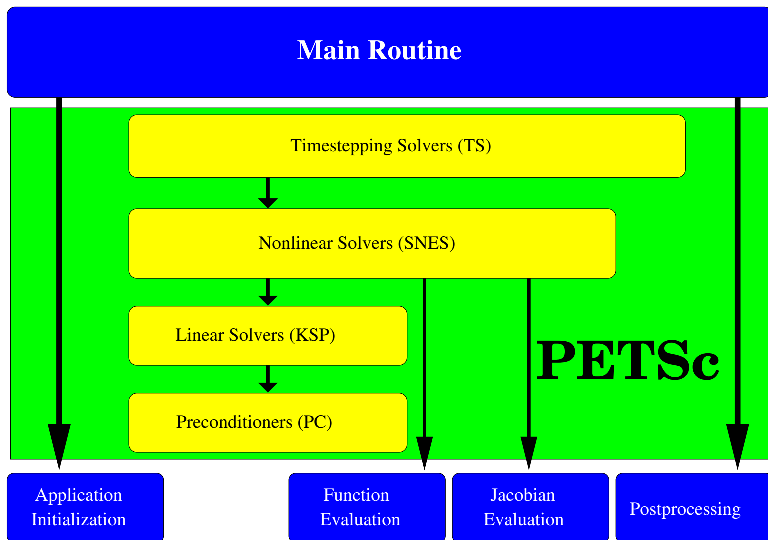
Part III

PETSc Essentials

PETSc Structure



Flow Control for a PETSc Application



Levels of Abstraction

In Mathematical Software

- Application-specific interface
  - Programmer manipulates objects associated with the application
- High-level mathematics interface
  - Programmer manipulates mathematical objects
    - Weak forms, boundary conditions, meshes
- Algorithmic and discrete mathematics interface
  - Programmer manipulates mathematical objects
    - Sparse matrices, nonlinear equations
  - Programmer manipulates algorithmic objects
    - Solvers
- Low-level computational kernels
  - BLAS-type operations, FFT

Object-Oriented Design

- Design based on operations you perform,
  - not on the data in the object
- Example: A vector is
  - **not** a 1d array of numbers
  - an object allowing addition and scalar multiplication
- The efficient use of the computer is an added difficulty

Symmetry Principle

Interfaces to mutable data must be symmetric.

- Creation and query interfaces are paired
  - "No get without a set"
- Fairness
  - "If you can do it, your users will want to do it"
- Openness
  - "If you can do it, your users will want to *undo* it"

Empiricism Principle

Interfaces must allow easy testing and comparison.

- Swapping different implementations
  - “You will not be smart enough to pick the solver”
- Commonly violated in FE code
  - Elements are hard coded
- Also avoid assuming structure outside of the interface
  - Making continuous fields have discrete structure
  - Temptation to put metadata in a different places

Experimentation is Essential!

Proof is not currrently enough to examine solvers

- N. M. Nachtigal, S. C. Reddy, and L. N. Trefethen, *How fast are nonsymmetric matrix iterations?*, SIAM J. Matrix Anal. Appl., **13**, pp.778–795, 1992.
- Anne Greenbaum, Vlastimil Ptak, and Zdenek Strakos, *Any Nonincreasing Convergence Curve is Possible for GMRES*, SIAM J. Matrix Anal. Appl., **17** (3), pp.465–469, 1996.

The PETSc Programming Model

- Goals
  - Portable, runs everywhere
  - High performance
  - Scalable parallelism
- Approach
  - Distributed memory ("shared-nothing")
  - No special compiler
  - Access to data on remote machines through MPI
  - Hide within objects the details of the communication
  - User orchestrates communication at a higher abstract level

Collectivity

- MPI communicators (`MPI_Comm`) specify collectivity
  - Processes involved in a computation
- Constructors are collective over a communicator
  - `VecCreate(MPI_Comm comm, Vec *x)`
  - Use `PETSC_COMM_WORLD` for all processes and `PETSC_COMM_SELF` for one
- Some operations are collective, while others are not
  - collective: `VecNorm()`
  - not collective: `VecGetLocalSize()`
- Sequences of collective calls must be in the same order on each process

What is not in PETSc?

- Higher level representations of PDEs
  - Unstructured mesh generation and manipulation
  - Discretizations, DealII
  - PETSc-CS and Sundance
- Load balancing
- Sophisticated visualization capabilities
  - MayaVi
- Eigenvalues
  - SLEPc and SIP
- Optimization and sensitivity
  - TAO and Veltisto

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.

Part IV

PETSc Integration

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 upgrading or seeking help
  - Automatic in PETSc

PETSc Integration

PETSc is a set a library interfaces

- We do not seize `main()`
- We do not control output
- We propagate errors from underlying packages
- We present the same interfaces in:
  - C
  - C++
  - F77
  - F90
  - Python

See Gropp in SIAM, OO Methods for Interop SciEng, '99

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
- Call `PetscFinalize()`
  - Calculates logging summary
  - Shutdown and release resources
- Checks compile and link

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

Command Line Processing

- Check for an option
  - `PetscOptionsHasName()`
- Retrieve a value
  - `PetscOptionsGetInt()`, `PetscOptionsGetIntArray()`
- Set a value
  - `PetscOptionsSetValue()`
- Clear, alias, reject, etc.

Vector Algebra

What are PETSc vectors?

- Fundamental objects for storing field solutions, right-hand sides, etc.
- Each process locally owns a subvector of contiguous global data

How do I create vectors?

- `VecCreate(MPI_Comm, Vec *)`
- `VecSetSizes(Vec, int n, int N)`
- `VecSetType(Vec, VecType typeName)`
- `VecSetFromOptions(Vec)`
  - Can set the type at runtime

Vector Algebra

A PETSc Vec

- Has a direct interface to the values
- Supports all vector space operations
  - `VecDot()`, `VecNorm()`, `VecScale()`
- Has unusual operations, e.g. `VecSqrt()`, `VecWhichBetween()`
- Communicates automatically during assembly
- Has customizable communication (scatters)

Creating a Vector

```
Vec x;
PetscInt N;
PetscErrorCode ierr;

ierr = PetscInitialize(&argc, &argv, PETSC_NULL, PETSC_NULL);CHKERR
ierr = PetscOptionsGetInt(PETSC_NULL, "-N", &N, PETSC_NULL);CHKER
ierr = VecCreate(PETSC_COMM_WORLD, &x);CHKERRQ(ierr);
ierr = VecSetSizes(x, PETSC_DECIDE, N);CHKERRQ(ierr);
ierr = VecSetType(x, "mpi");CHKERRQ(ierr);
ierr = VecSetFromOptions(x);CHKERRQ(ierr);
ierr = PetscFinalize();CHKERRQ(ierr);
```

Parallel Assembly
Vectors and Matrices

- Processes may set an arbitrary entry
  - Must use proper interface
- Entries need not be generated locally
  - Local meaning the process on which they are stored
- PETSc automatically moves data if necessary
  - Happens during the assembly phase

Vector Assembly

- A three step process
  - Each process sets or adds values
  - Begin communication to send values to the correct process
  - Complete the communication
- `VecSetValues(Vec v, int n, int rows[], PetscScalar values[], mode)`
  - `mode` is either INSERT_VALUES or ADD_VALUES
- Two phase assembly allows overlap of communication and computation
  - `VecAssemblyBegin(Vec v)`
  - `VecAssemblyEnd(Vec v)`

One Way to Set the Elements of a Vector

```
ierr = VecGetSize(x, &N);CHKERRQ(ierr);
ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank);CHKERRQ(ierr);
if (rank == 0) {
  for(i = 0, val = 0.0; i < N; i++, val += 10.0) {
    ierr = VecSetValues(x, 1, &i, &val, INSERT_VALUES);CHKERRQ(ierr);
  }
}
/* These routines ensure that the data is distributed to the other processes */
ierr = VecAssemblyBegin(x);CHKERRQ(ierr);
ierr = VecAssemblyEnd(x);CHKERRQ(ierr);
```

A Better Way to Set the Elements of a Vector

```
ierr = VecGetOwnershipRange(x, &low, &high);CHKERRQ(ierr);
for(i = low, val = low*10.0; i < high; i++, val += 10.0) {
  ierr = VecSetValues(x, 1, &i, &val, INSERT_VALUES);CHKERRQ(ierr);
}
/* These routines ensure that the data is distributed to the other processes */
ierr = VecAssemblyBegin(x);CHKERRQ(ierr);
ierr = VecAssemblyEnd(x);CHKERRQ(ierr);
```

Selected Vector Operations

Function Name	Operation
VecAXPY(Vec y, PetscScalar a, Vec x)	$y = y + a * x$
VecAYPX(Vec y, PetscScalar a, Vec x)	$y = x + a * y$
VecWAYPX(Vec w, PetscScalar a, Vec x, Vec y)	$w = y + a * x$
VecScale(Vec x, PetscScalar a)	$x = a * x$
VecCopy(Vec y, Vec x)	$y = x$
VecPointwiseMult(Vec w, Vec x, Vec y)	$w_i = x_i * y_i$
VecMax(Vec x, PetscInt *idx, PetscScalar *r)	$r = \max r_i$
VecShift(Vec x, PetscScalar r)	$x_i = x_i + r$
VecAbs(Vec x)	$x_i = \|x_i\|$
VecNorm(Vec x, NormType type, PetscReal *r)	$r = \|\|x\|\|$

Working With Local Vectors

It is sometimes more efficient to directly access local storage of a `Vec`.

- PETSc allows you to access the local storage with
  - `VecGetArray(Vec, double *[])`
- You must return the array to PETSc when you finish
  - `VecRestoreArray(Vec, double *[])`
- Allows PETSc to handle data structure conversions
  - Commonly, these routines are inexpensive and do not involve a copy

VecGetArray in C

```
Vec v;
PetscScalar *array;
PetscInt n, i;
PetscErrorCode ierr;

ierr = VecGetArray(v, &array);CHKERRQ(ierr);
ierr = VecGetLocalSize(v, &n);CHKERRQ(ierr);
ierr = PetscSynchronizedPrintf(PETSC_COMM_WORLD,
  "First element of local array is %f\n", array[0]);CHKERRQ(ierr);
ierr = PetscSynchronizedFlush(PETSC_COMM_WORLD);CHKERRQ(ierr);
for(i = 0; i < n; i++) {
  array[i] += (PetscScalar) rank;
}
ierr = VecRestoreArray(v, &array);CHKERRQ(ierr);
```

VecGetArray in F77

```
#include "petsc.h"
#include "petscvec.h"
      Vec v;
      PetscScalar array(1)
      PetscOffset offset
      PetscInt n, i
      PetscErrorCode ierr

      call VecGetArray(v, array, offset, ierr)
      call VecGetLocalSize(v, n, ierr)
      do i=1,n
        array(i+offset) = array(i+offset) + rank
      end do
      call VecRestoreArray(v, array, offset, ierr)
```

VecGetArray in F90

```
#include "petsc.h"
#include "petscvec.h"
      Vec v;
      PetscScalar pointer :: array(:)
      PetscInt n, i
      PetscErrorCode ierr

      call VecGetArrayF90(v, array, ierr)
      call VecGetLocalSize(v, n, ierr)
      do i=1,n
        array(i) = array(i) + rank
      end do
      call VecRestoreArrayF90(v, array, ierr)
```

Matrix Algebra

What are PETSc matrices?

- 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
- `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 m, 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

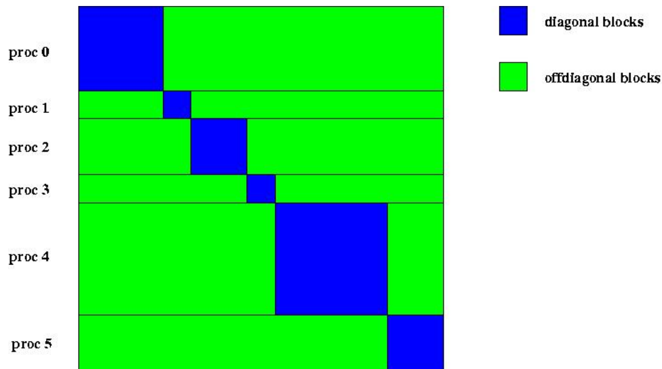
One Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

```
values[0] = -1.0; values[1] = 2.0; values[2] = -1.0;
if (rank == 0) { /* Only one process creates matrix */
  for(row = 0; row < N; row++) {
    cols[0] = row-1; cols[1] = row; cols[2] = row+1;
    if (row == 0) {
      ierr = MatSetValues(A, 1, &row, 2, &cols[1], &values[1], INSERT_VALU
    } else if (row == N-1) {
      ierr = MatSetValues(A, 1, &row, 2, cols, values, INSERT_VALUES);CHK
    } else {
      ierr = MatSetValues(A, 1, &row, 3, cols, values, INSERT_VALUES);CHK
    }
  }
}
ierr = MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
ierr = MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
```

ParallelSparseMatrix

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



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

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

A Better Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

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

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.
- Matrix decomposition in contiguous chunks is simple
  - Makes interoperation with other codes easier
  - For other ordering, PETSc provides “Application Orderings” (AO)

Solvers

- **Explicit**:
  - Field variables are updated using local neighbor information
- **Semi-implicit**:
  - Some subsets of variables are updated with global solves
  - Others with direct local updates
- **Implicit**:
  - Most or all variables are updated in a single global solve

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`

Nonlinear Solvers
Newton Methods

- Using PETSc linear algebra, just add:
  - `SNESSetFunction(SNES snes, Vec r, residualFunc, void *ctx)`
  - `SNESSetJacobian(SNES snes, Mat A, Mat M, jacFunc, void *ctx)`
  - `SNESSolve(SNES snes, Vec b, Vec x)`
- Can access subobjects
  - `SNESGetKSP(SNES snes, KSP *ksp)`
- Can customize subobjects from the cmd line
  - Set the subdomain preconditioner to ILU with `-sub_pc_type ilu`

Basic Solver Usage

We will illustrate basic solver usage with `SNES`.

- Use `SNESSetFromOptions()` so that everything is set dynamically
  - Use `-snes_type` to set the type or take the default
- Override the tolerances
  - Use `-snes_rtol` and `-snes_atol`
- View the solver to make sure you have the one you expect
  - Use `-snes_view`
- For debugging, monitor the residual decrease
  - Use `-snes_monitor`
  - Use `-ksp_monitor` to see the underlying linear solver

3rd Party Solvers in PETSc

1. Sequential LU
   - ILUDT (SPARSEKIT2, Yousef Saad, U of MN)
   - EUCLID & PILUT (Hypre, David Hysom, LLNL)
   - ESSL (IBM)
   - SuperLU (Jim Demmel and Sherry Li, LBNL)
   - Matlab
   - UMFPACK (Tim Davis, U. of Florida)
   - LUSOL (MINOS, Michael Saunders, Stanford)
2. Parallel LU
   - MUMPS (Patrick Amestoy, IRIT)
   - SPOOLES (Cleve Ashcroft, Boeing)
   - SuperLU_Dist (Jim Demmel and Sherry Li, LBNL)
3. Parallel Cholesky
   - DSCPACK (Padma Raghavan, Penn. State)
4. XYTlib - parallel direct solver (Paul Fischer and Henry Tufo, ANL)

3rd Party Preconditioners in PETSc

1. Parallel ICC
   - BlockSolve95 (Mark Jones and Paul Plassman, ANL)
2. Parallel ILU
   - BlockSolve95 (Mark Jones and Paul Plassman, ANL)
3. Parallel Sparse Approximate Inverse
   - Parasails (Hypre, Edmund Chow, LLNL)
   - SPAI 3.0 (Marcus Grote and Barnard, NYU)
4. Sequential Algebraic Multigrid
   - RAMG (John Ruge and Klaus Steuben, GMD)
   - SAMG (Klaus Steuben, GMD)
5. Parallel Algebraic Multigrid
   - Prometheus (Mark Adams, PPPL)
   - BoomerAMG (Hypre, LLNL)
   - ML (Trilinos, Ray Tuminaro and Jonathan Hu, SNL)

Higher Level Abstractions

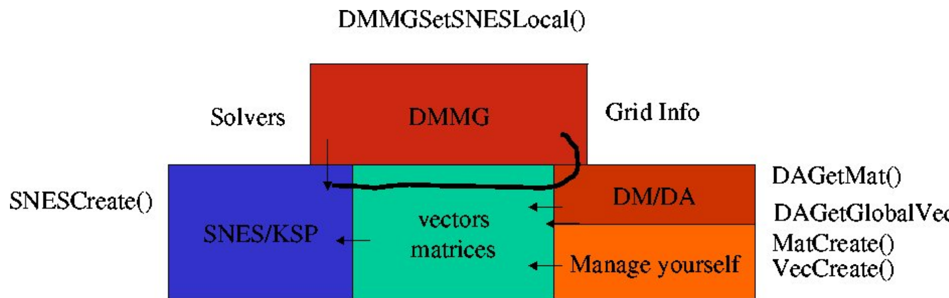
The PETSc `DA` class is a topology interface.

- Structured grid interface
  - Fixed simple topology
- Supports stencils, communication, reordering
  - No idea of operators
- Nice for simple finite differences

The PETSc `DM` class is a hierarchy interface.

- Supports multigrid
  - `DMMG` combines it with the `MG` preconditioner
- Abstracts the logic of multilevel methods

3 Ways To Use PETSc



- User manages all topology (just use Vec and Mat)
- PETSc manages single topology (use DA)
- PETSc manages a hierarchy (use DM)

Part V

Advanced PETSc

SNESCallbacks

The SNES interface is based upon callback functions

- `SNESSetFunction()`
- `SNESSetJacobian()`

When PETSc needs to evaluate the nonlinear residual $F(x)$, the solver calls the user's function inside the application.

The user function get 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 `MatrStructure` values are:
    - SAME_NONZERO_PATTERN, DIFFERENT_NONZERO_PATTERN, ...

Alternatively, you can use

- a builtin sparse finite difference approximation
- automatic differentiation
    - AD support via ADIC/ADIFOR (P. Hovland and B. Norris from ANL)

SNES Variants

- Line search strategies
- Trust region approaches
- Pseudo-transient continuation
- Matrix-free variants

Finite Difference Jacobians

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

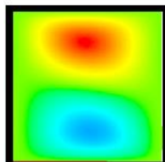
- Dense
  - Activated by `-snes_fd`
  - Computed by `SNESDefaultComputeJacobian()`
- Sparse via colorings
  - Coloring is created by `MatFDColoringCreate()`
  - Computed by `SNESDefaultComputeJacobianColor()`

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

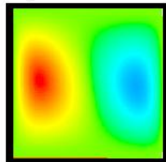
- Activated by `-snes_mf` without preconditioning
- Activated by `-snes_mf_operator` with user-defined preconditioning
  - Uses preconditioning matrix from `SNESSetJacobian()`

SNES Example: Driven Cavity

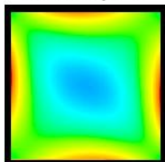
Solution Components



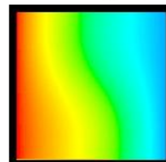
velocity: u



velocity: v



vorticity:



temperature: T

- Velocity-vorticity formulation
- Flow driven by lid and/or bouyancy
- Logically regular grid
  - Parallelized with `DA`
- Finite difference discretization
- Authored by David Keyes

$PETCS_DIR/src/snes/examples/tutorials/ex19.c

Driven Cavity Application Context

```
typedef struct {
  /*---  basic application data  ---*/
  double      lid_velocity;      /* Velocity of the lid */
  double      prandtl, grashof; /* Prandtl and Grashof numbers */
  int         mx, my;            /* Grid points in x and y */
  int         mc;                /* Degrees of freedom per node */
  PetscTruth draw_contours;  /* Flag for drawing contours */
  /*---  parallel data  ---*/
  MPI_Comm comm;             /* Communicator */
  DA          da;                /* Distributed array */
  Vec         localX, localF;    /* Local ghosted solution and residual */
} AppCtx;
```

$PETCS_DIR/src/snes/examples/tutorials/ex19.c

Driven Cavity Residual Evaluation

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

  /* Not Shown: Code to communicate nonlocal ghost point data (scatters) */
  ierr = VecGetArray(F, &f); CHKERRQ(ierr);
  /* Not Shown: Code to compute local function components */
  ierr = VecRestoreArray(F, &f); CHKERRQ(ierr);
  return 0;
}
```

$PETCS_DIR/src/snes/examples/tutorials/ex19.c

Better Driven Cavity Residual Evaluation

```
PetscErrorCode DrivenCavityFuncLocal(DALocalInfo *info,Field **x,Field **f,v
{
  /* Not Shown: Handle boundaries */
  /* Compute over the interior points */
  for(j = info->ys; j < info->xs+info->xm; j++) {
    for(i = info->xs; i < info->ys+info->ym; i++) {
      /* Not Shown: 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;
      f[j][i].u  = uxx + uyy - .5*(x[j+1][i].omega-x[j-1][i].omega)*hx;
      /* Not Shown: V velocity, Omega, Temperature */
    }
  }
}
```

$PETCS_DIR/src/snes/examples/tutorials/ex19.c

What is a DA?

`DA` is a topology interface handling parallel data layout on structured grids

- Handles local and global indices
  - `DAGetGlobalIndices()` and `DAGetAO()`
- Provides local and global vectors
  - `DAGetGlobalVector()` and `DAGetLocalVector()`
- Handles ghost values coherence
  - `DAGetGlobalToLocal()` and `DAGetLocalToGlobal()`

Creating a DA

```
DACreate1d(comm, DAPeriodicType wrap, M, dof, s, lm[], DA
*da)
```

`wrap`: Specifies periodicity
- DA_NONPERIODIC or DA_XPERIODIC

`M`: Number of grid points in x-direction

`dof`: Degrees of freedom per node

`s`: The stencil width

`lm`: Alternative array of local sizes
- Use PETSC_NULL for the default

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

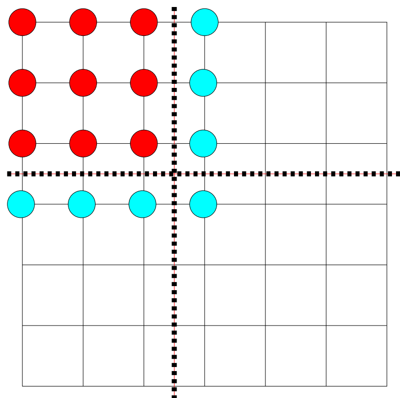
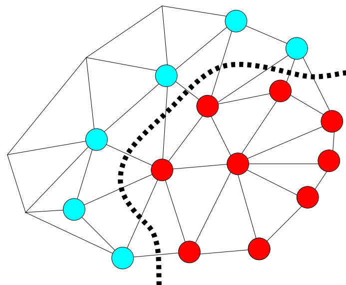
`s`: The stencil width

`lm/n`: Alternative array of local sizes
- Use PETSC_NULL for the default

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 belongs to a unique process and has a unique id
- **Local**: Numbering includes ghost vertices from neighboring processes

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 `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!

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
- `DAGlobalToLocal End(da, gvec, mode, lvec)`
    - Finishes the communication

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

DA Local Function

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

```
PetscErrorCode (*lfunc)(DALocalInfo *info, PetscScalar **x,
                        PetscScalar **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)
```

DA Local Jacobian

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

```
PetscErrorCode (*lfunc)(DALocalInfo *info, PetscScalar **x,
                        Mat J, void *ctx)
```

info: All layout and numbering information

x: The current solution

J: The Jacobian

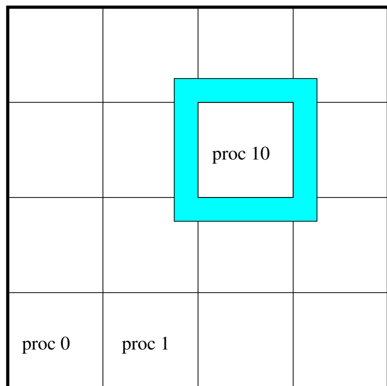
ctx: The user context passed to DASetLocalFunction()

The local DA function is activated by calling

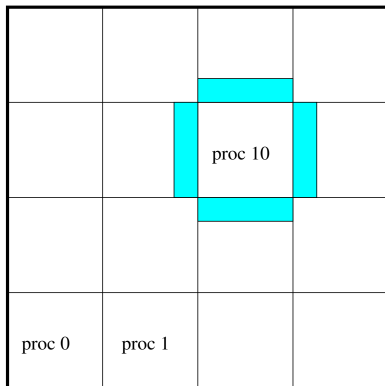
```
SNESSetJacobian(snes, J, J, SNESDAComputeJacobian, ctx)
```

DA Stencils

Both the box stencil and star stencil are available.



Box Stencil



Star Stencil

Setting Values for Finite Differences

PETSc provides

```
MatSetValuesStencil(Mat A, m, MatStencil idxm[], n,
        MatStencil idxn[], values[], mode)
```

- Each row or column is actually a `MatStencil`
  - This specifies grid coordinates and a component if necessary
  - Can imagine for unstructured grids, they are *vertices*
- The values are the same logically dense block in rows and columns

Mapping Between Global Numberings

- Natural global numbering
  - Convenient for visualization, boundary conditions, etc.
- Convert between global numbering schemes using `AO`
  - `DAGetAO(DA da, AO *ao)`
- Handled automatically by some utilities (e.g., VecView()) for `DA` vectors

DA Example: Bratu

- Create SNES and DA
- Use `DASetLocalFunction()` and `DASetLocalJacobian()` to set user callbacks
  - Use `DAGetMatrix()` to get DA matrix for SNES
- Use `SNESDAFormFunction()` and `SNESDAComputeJacobian()` for SNES callback
  - Could also use FormFunctionMatlab()
  - Could also use `SNESDefaultComputeJacobian()`

$PETCS_DIR/src/snes/examples/tutorials/ex5.c

DA Example: Bratu

```
PetscErrorCode LocalFunc(DALocalInfo *info, PetscScalar **x, PetscScalar **f
{
  for(j = info->ys; j < info->ys + info->ym; j++) {
    for(i = info->xs; i < info->xs + info->xm; i++) {
      if (i == 0 || j == 0 || i == info->mx-1 || j == info->my-1) {
        f[j][i] = x[j][i];
      } else {
        u       = x[j][i];
        u_xx    = -(x[j][i+1] - 2.0*u + x[j][i-1])*(hy/hx);
        u_yy    = -(x[j+1][i] - 2.0*u + x[j-1][i])*(hx/hy);
        f[j][i] = u_xx + u_yy - hx*hy*lambda*PetscExpScalar(u);
      }
    }
  }
}
```

$PETCS_DIR/src/snes/examples/tutorials/ex5.c

DA Example: Bratu

```
int LocalJac(DALocalInfo *info, PetscScalar **x, Mat jac, void *ctx)
{
  for(j = info->ys; j < info->ys + info->ym; j++) {
    for(i = info->xs; i < info->xs + info->xm; i++) {
      row.j = j; row.i = i;
      if (i == 0 || j == 0 || i == info->mx-1 || j == info->my-1) {
        v[0] = 1.0;
        ierr = MatSetValuesStencil(jac, 1, &row, 1, &row, v, INSERT_VALUES
      } else {
        v[0] = -(hx/hy); col[0].j = j-1; col[0].i = i;
        v[1] = -(hy/hx); col[1].j = j;  col[1].i = i-1;
        v[2] = 2.0*(hy/hx+hx/hy) - hx*hy*lambda*PetscExpScalar(x[j][i]);
        v[3] = -(hy/hx); col[3].j = j;  col[3].i = i+1;
        v[4] = -(hx/hy); col[4].j = j+1; col[4].i = i;
        ierr = MatSetValuesStencil(jac, 1, &row, 5, col, v, INSERT_VALUES);
} } } }
```

$PETCS_DIR/src/snes/examples/tutorials/ex5.c

Part VI

PETSc Extensibility

Extending Configure

- See `python/PETSc/packages/*.py` for examples
- Add module with class `Configure` derived from `config.base.Configure`
  - Can also derive from `PETSc.package.Package`
  - Implement `configure()` and `configureHelp()`
- Customize PETSc through the `make` system
  - `addDefine()`
  - `addTypedef()`, `addPrototype()`
  - `addMakeMacro()`, `addMakeRule()`
  - Deprecated `addSubstitution()`

Linking to PETSc

- Nothing but the libraries
  - User can custom link
- Using only PETSc build variables
  - Include `bmake/common/variables`
- Also use PETSc build rules
  - Include `bmake/common/base`
- Also makes available 3rd party packages

Layering over PETSc

- SLEPc, TAO, and MagPar
  - Infrastructure and linear algebra
- Use PETSc object structure
  - Dynamic dispatch
- Use dynamic linking facilities
  - Runtime type selection
- Use debugging and profiling tools
  - Memory management, runtime type checking

Adding an Implementation

- See `src/ksp/pc/impls/jacobi/jacobi.c`
- Implement the interface methods
  - For Jacobi, `PCSetUp()`, `PCApply()`,...
- Define a constructor
  - Allocate and initialize the class structure
  - Fill in the function table
  - Must have C linkage
- Register the constructor
  - See `src/ksp/ksp/interface/dlregis.c`
  - Maps a string (class name) to the constructor
  - Usually uses `PetscFListAdd()`

Adding a New Wrapper

- See `src/ts/impls/implicit/pvode/petscpvode.c`
- Just like an Implementation
  - Methods dispatch to 3rd party software
- Need to alter local makefile
  - Add a `requirespackage` line
  - Add include variable to `CPPFLAGS`
- Usually requires configure additions

Adding a New Subtype

- See `src/mat/impls/aij/seq/umfpack/umfpack.c`
- Have to virtualize methods by hand
- Define a constructor
  - Change type name first to correct name
  - Call `MatSetType()` for base type
  - Replace (and save) overidden methods
  - Construct any specific data
- Must also define a conversion to the base type
  - Only called in destructor right now

Adding a New Type

- See `src/ksp/ksp/kspimpl.h`
- Define a methods structure (interface)
  - A list of function pointers
- Define a type structure
  - First member is `PETSCHEADER(struct _Ops)`
  - Possibly other data members common to the type
  - A `void *data` for implementation structures

Adding a New Type

- See `src/ksp/ksp/interface/dlregis.c`
- Define a package initializer (`PetscDLLibraryRegister`)
  - Called when the DLL is loaded
    - Also called from generic create if linking statically
  - Registers classes and events (see below)
  - Registers any default implementation constructors
  - Setup any info or summary exclusions

Adding a New Type

- See `src/ksp/ksp/interface/itcreate.c`
- Define a generic create
    - Call package initializer if linking statically
    - Call `PetscHeaderCreate()`
    - Initialize any default data members
- Define a `setType()` method
    - Call the destructor of any current implementation
    - Call the constructor of the given implementation
    - Set the type name

Adding a New Type

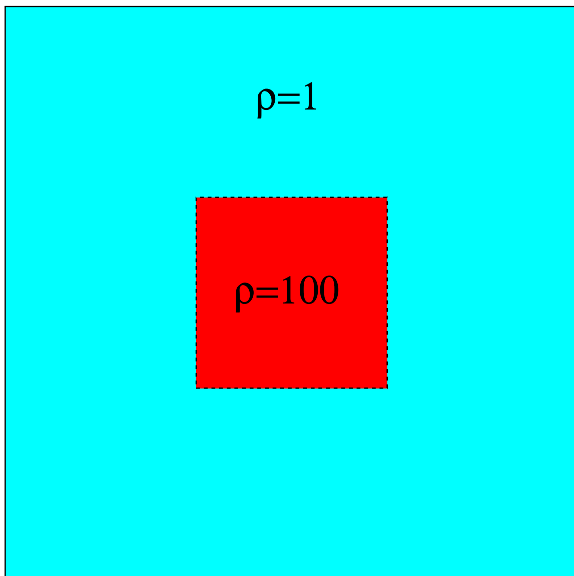
Things Swept Under The Rug

- Need `setFromOptions()` which allows implementation selection at runtime
- Have to manage the database of registered constructors
- View and destroy functions handled a little funny due to historical reasons

Part VII

PETSc Optimization

Problem Domain



Problem Statement

Inhomogeneous Laplacian in 2D. Modeled by the partial differential equation

$$\nabla \cdot (\rho \nabla \mathbf{u}) = \mathbf{f} \qquad \text{on} \qquad \Omega = [0,1] \times [0,1],$$

with forcing function

$$\mathbf{f}(x,y) = e^{-(1-x)^2/\nu} e^{-(1-y)^2/\nu},$$

with Dirichlet boundary conditions

$$\mathbf{u} = \mathbf{f}(x,y) \qquad \text{on} \qquad \partial\Omega,$$

or homogeneous Neumman boundary conditions

$$\hat{n} \cdot \nabla \mathbf{u} = 0 \qquad \text{on} \qquad \partial\Omega.$$

Revision 1.1

For the initial try, we modify a common PETSc `DMMG` skeleton:

- Use a simple FD 5-point stencil discretization
- Use a structured grid (`DA`)
- Use a hierarchical method (`DMMG`)
- Only implement Dirichlet BC (simple masking)

Revision 1.2

Now utilize some more PETSc features:

- Add `UserContext` structure to hold ν and the BC type
  - Need to set the context at each `DM` level
- Add Neumann BC using a `MatNullSpace`
  - Used to project onto the orthogonal complement
  - `KSPSetNullSpace()`
- Set parameters from the command line
  - `PetscOptionsBegin()`, `PetscOptionsEnd()`
  - `PetscOptionsScalar()`, `PetscOptionsString()`
  - By hand, `PetscOptionsGetScalar()`, `PetscOptionsGetString()`
- Fixed scaling for anisotropic grids

Revision 1.6

Barry fixed the example to converge nicely:

- Set nullspace on all `DM` levels
  - Actually set in the smoother (`KSP`)
  - Same idea as the user context
  - Now completely handled by `DMMGSetNullSpace()`
- Remove the null space component of the rhs
  - `MatNullSpaceRemove()`
  - Usually handled by the model
- Add a shift to the coarse grid LU for Neumann BC
  - System is singular so augment with the identity
  - One extra step if coarse solve is redundant
- Fix weighting for center point of Neumann condition
  - Depends on the number of missing points
- Also use `PetscOptionsElist()` to set BC
  - Can provide a nice listbox using the GUI

DMMG Grids

The use specifies the coarse grid, and then DMMG successively refines it.

- In our problem, we being with a 3×3 grid
  - We LU factor a 9×9 matrix
- By level 10, we have a 1025×1025 grid
  - Our final solution has 1,050,625 unknowns
- Set the initial grid using `-da_grid_x` and `-da_grid_y`
- Set the number of levels using `-dmmg_nlevels`

Mesh Independence

The iteration number should be independent of the mesh size, or the number of levels.

Levels	Unknowns	KSP Iterations
2	25	2
4	289	2
6	4225	3
8	66049	2
10	1050625	2

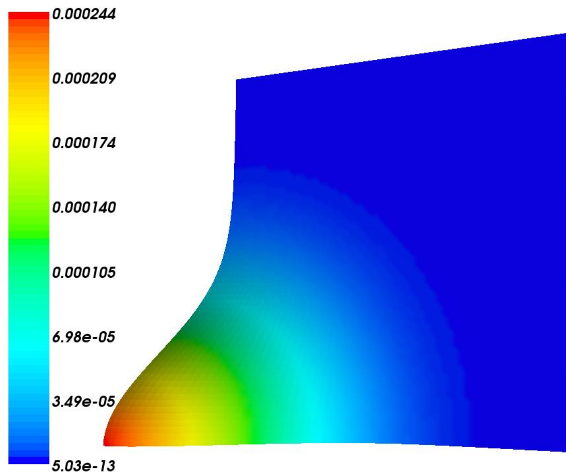
We have used Dirichlet BC above

Viewing the DA

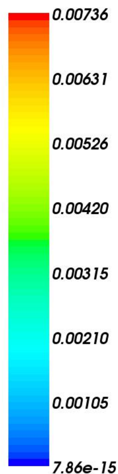
- `-da_view` outputs an ascii description
- `-da_view_draw` display the grid
  - First the grid itself is drawn
  - Global variable numbers are then provided
  - Finally ghost variable numbers are shown for error checking
- `-vec_view_draw` draws a contour plot for DA vectors
  - The contour grid can be shown with `-draw_contour_grid`

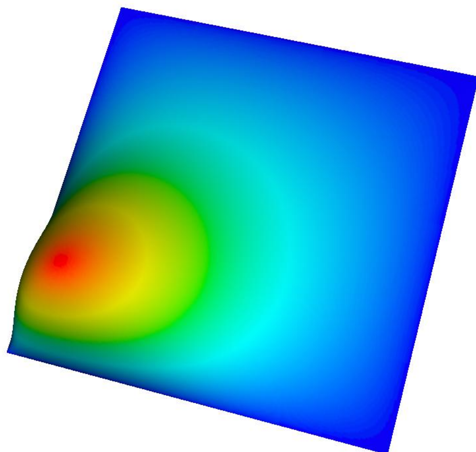
Forcing Function

The rhs of our linear system drives the solution:

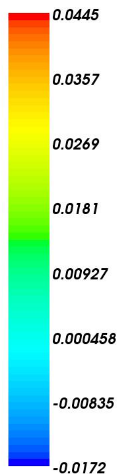


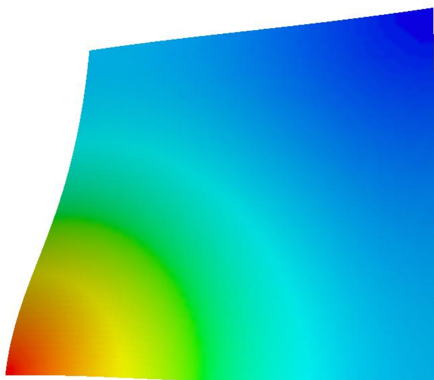
Dirichlet Solution





Neumann Solution





Multigrid Options

- Choose V-cycle or W-cycle using `-pc_mg_cycles`
- Can set the iteration method using `-pc_mg_type`
  - `MULTIPLICATIVE`, `ADDITIVE`, `FULL`, `KASKADE`
- Choose the number of steps in both the up and down smoothers
  - `-pc_mg_smoothup`, `-pc_mg_smoothdown`
- The coarse problem has prefix `mg_coarse_`
  - `-mg_coarse_pc_type`, `-mg_coarse_ksp_maxit`
- Each level k has prefix `mg_levels_`k`_`
  - `-mg_levels_1_ksp_type`, `-mg_levels_2_pc_ilu_fill`
- Can automatically form coarse operators with the Galerkin process
  - `-pc_mg_galerkin`
  - `DMMG` provides these automatically by interpolation

Part VIII

Future Plans

What is New?

New classes and tools will be added to the existing PETSc framework:

- Unstructured mesh generation, refinement, and coarsening
- Unstructured multilevel algorithms
- A high-level language for specification of weak forms (with FFC)
- Automatic generation of arbitrary finite elements (with FIAT)
- Platform independent, system for configure, build, and distribution

PETSc Parts

To make the new functionality easier for users to understand, PETSc will be divided conceptually into two parts:

- **PETSc-AS** will contain the components related to algebraic solvers, which is the current PETSc distribution.
- **PETSc-CS** will contain the support for continuum problems phrased in weak form. These modules will make use of the PETSc-AS modules in our implementation, but this is not strictly necessary.

However, the framework will remain integrated:

- Single release version, version control, and build system
- Each module is self-contained (install only what is necessary)

BuildSystem

- Pure Python and freely available
- Currently handles configure and build
  - needs install and distribution
- Handles generated code through ASE
  - can uses md5, timestamp, diff, etc.
- Handles shared libraries for many architectures
  - Linux, Mac OSX, Windows

Mesh Capabilities

- 2D Delaunay generation and refinement
  - Triangle
- 3D Delaunay generation and refinement
  - TetGen
- 3D Delaunay generation, coarsening, and refinement
  - Allows quadratic Bezier elements
  - Allows efficient, dynamic mesh update
  - TUMBLE

How does Sieve Work?

Application codes should only rarely use low-level operations.

- `create`, `distribute`

How does Sieve Work?

Application codes should only rarely use low-level operations.

- `create`, `distribute`
  - Can create from a file or boundary

How does Sieve Work?

Application codes should only rarely use low-level operations.

- `create`, `distribute`
  - Can create from a file or boundary
- `refine`, `coarsen`

How does Sieve Work?

Application codes should only rarely use low-level operations.

- `create`, `distribute`
  - Can create from a file or boundary
- `refine`, `coarsen`
  - Can refine or coarsen no matter the mesh source

How does Sieve Work?

Application codes should only rarely use low-level operations.

- `create`, `distribute`
  - Can create from a file or boundary
- `refine`, `coarsen`
  - Can refine or coarsen no matter the mesh source
- `overlap`, `fusion`

How does Sieve Work?

Application codes should only rarely use low-level operations.

- `create`, `distribute`
  - Can create from a file or boundary
- `refine`, `coarsen`
  - Can refine or coarsen no matter the mesh source
- `overlap`, `fusion`
  - Allow fine grain parallelism, and enable Field operations

How does Sieve Work?

Application codes should only rarely use low-level operations.

- `create`, `distribute`
  - Can create from a file or boundary
- `refine`, `coarsen`
  - Can refine or coarsen no matter the mesh source
- `overlap`, `fusion`
  - Allow fine grain parallelism, and enable Field operations
  - Implemented by low-level Sieve operation

How does Field Work?

Application codes should use Fields for any data organized over a mesh

- `setChart`, `setFiberDimension`

How does Field Work?

Application codes should use Fields for any data organized over a mesh

- `setChart`, `setFiberDimension`
    - Setup aribtrary data layout over a mesh

How does Field Work?

Application codes should use Fields for any data organized over a mesh

- `setChart`, `setFiberDimension`
    - Setup aribtrary data layout over a mesh
- `restrict`, `update`

How does Field Work?

Application codes should use Fields for any data organized over a mesh

- `setChart`, `setFiberDimension`
  - Setup aribtrary data layout over a mesh
- `restrict`, `update`
  - Control data flow between levels of a hierarchy

How does Field Work?

Application codes should use Fields for any data organized over a mesh

- `setChart`, `setFiberDimension`
  - Setup aribtrary data layout over a mesh
- `restrict`, `update`
  - Control data flow between levels of a hierarchy
  - Heart of FEM, Multigrid, Domain Decomposition

How does Field Work?

Application codes should use Fields for any data organized over a mesh

- `setChart`, `setFiberDimension`
  - Setup aribtrary data layout over a mesh
- `restrict`, `update`
  - Control data flow between levels of a hierarchy
  - Heart of FEM, Multigrid, Domain Decomposition
- `distributeSection`, `distributeSieve`

How does Field Work?

Application codes should use Fields for any data organized over a mesh

- `setChart`, `setFiberDimension`
  - Setup aribtrary data layout over a mesh
- `restrict`, `update`
  - Control data flow between levels of a hierarchy
  - Heart of FEM, Multigrid, Domain Decomposition
- `distributeSection`, `distributeSieve`
  - Persistent communication structures for irregular data

Advantages

Sieve/Field operations enable algorithms which are:

- Dimension independent

Advantages

Sieve/Field operations enable algorithms which are:

- Dimension independent
  - Same partitioning code with in 1D, 2D, and 3D

Advantages

Sieve/Field operations enable algorithms which are:

- Dimension independent
  - Same partitioning code with in 1D, 2D, and 3D
- Shape independent

Advantages

Sieve/Field operations enable algorithms which are:

- Dimension independent
  - Same partitioning code with in 1D, 2D, and 3D
- Shape independent
  - Same assembly code works for tets and quads

Advantages

Sieve/Field operations enable algorithms which are:

- Dimension independent
  - Same partitioning code with in 1D, 2D, and 3D
- Shape independent
  - Same assembly code works for tets and quads
- Discretization independent

Advantages

Sieve/Field operations enable algorithms which are:

- Dimension independent
  - Same partitioning code with in 1D, 2D, and 3D
- Shape independent
  - Same assembly code works for tets and quads
- Discretization independent
  - Same assembly code works for any finite element

Advantages

Sieve/Field operations enable algorithms which are:

- Dimension independent
  - Same partitioning code with in 1D, 2D, and 3D
- Shape independent
  - Same assembly code works for tets and quads
- Discretization independent
  - Same assembly code works for any finite element
  - Only variation is in local integration routine (tied to weak form)

Advantages

Sieve/Field operations enable algorithms which are:

- Dimension independent
  - Same partitioning code with in 1D, 2D, and 3D
- Shape independent
  - Same assembly code works for tets and quads
- Discretization independent
  - Same assembly code works for any finite element
  - Only variation is in local integration routine (tied to weak form)
- Transparent to parallelism

What is Sieve Good For?

- Parallelization

What is Sieve Good For?

- Parallelization
- Hierarchical, unstructured data layout

What is Sieve Good For?

- Parallelization
- Hierarchical, unstructured data layout
  - Solution fields, esp. higher order, adaptive, hybrid

What is Sieve Good For?

- Parallelization
- Hierarchical, unstructured data layout
  - Solution fields, esp. higher order, adaptive, hybrid
  - Auxiliary fields, e.g. viscosity, energy density

What is Sieve Good For?

- Parallelization
- Hierarchical, unstructured data layout
  - Solution fields, esp. higher order, adaptive, hybrid
  - Auxiliary fields, e.g. viscosity, energy density
- Complex topologies

What is Sieve Good For?

- Parallelization
- Hierarchical, unstructured data layout
  - Solution fields, esp. higher order, adaptive, hybrid
  - Auxiliary fields, e.g. viscosity, energy density
- Complex topologies
  - Sphere, torus

What is Sieve Good For?

- Parallelization
- Hierarchical, unstructured data layout
  - Solution fields, esp. higher order, adaptive, hybrid
  - Auxiliary fields, e.g. viscosity, energy density
- Complex topologies
  - Sphere, torus
  - Fault systems

What is Sieve Good For?

- Parallelization
- Hierarchical, unstructured data layout
  - Solution fields, esp. higher order, adaptive, hybrid
  - Auxiliary fields, e.g. viscosity, energy density
- Complex topologies
  - Sphere, torus
  - Fault systems
  - Block material models

What is Sieve Good For?

- Parallelization
- Hierarchical, unstructured data layout
  - Solution fields, esp. higher order, adaptive, hybrid
  - Auxiliary fields, e.g. viscosity, energy density
- Complex topologies
  - Sphere, torus
  - Fault systems
  - Block material models
- Complex communication patterns

What is Sieve Good For?

- Parallelization
- Hierarchical, unstructured data layout
  - Solution fields, esp. higher order, adaptive, hybrid
  - Auxiliary fields, e.g. viscosity, energy density
- Complex topologies
  - Sphere, torus
  - Fault systems
  - Block material models
- Complex communication patterns
  - Automatic discovery (Δ)

What is Sieve Good For?

- Parallelization
- Hierarchical, unstructured data layout
  - Solution fields, esp. higher order, adaptive, hybrid
  - Auxiliary fields, e.g. viscosity, energy density
- Complex topologies
  - Sphere, torus
  - Fault systems
  - Block material models
- Complex communication patterns
  - Automatic discovery (Δ)
  - Arbitrary domains and data layout

What is Sieve Good For?

- Parallelization
- Hierarchical, unstructured data layout
  - Solution fields, esp. higher order, adaptive, hybrid
  - Auxiliary fields, e.g. viscosity, energy density
- Complex topologies
  - Sphere, torus
  - Fault systems
  - Block material models
- Complex communication patterns
  - Automatic discovery (Δ)
  - Arbitrary domains and data layout
  - Persistent communication structures (PETSc VecScatter)

Integration

Two main integration paradigms for Sieve

- Parallelizaion of legacy code

Integration

Two main integration paradigms for Sieve

- Parallelizaion of legacy code
  - Model existing data structures

Integration

Two main integration paradigms for Sieve

- Parallelizaion of legacy code
  - Model existing data structures
  - Example: PyLith

Integration

Two main integration paradigms for Sieve

- Parallelizaion of legacy code
  - Model existing data structures
  - Example: PyLith
- Reorganization for new development

Integration

Two main integration paradigms for Sieve

- Parallelizaion of legacy code
  - Model existing data structures
  - Example: PyLith
- Reorganization for new development
  - Better describe common FEM operations

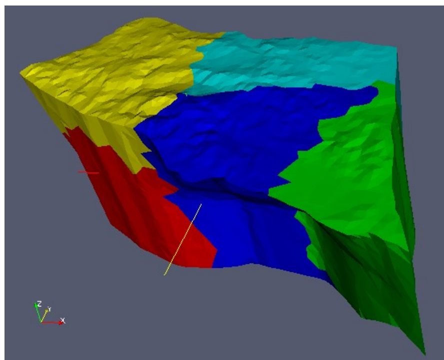
Integration

Two main integration paradigms for Sieve

- Parallelizaion of legacy code
  - Model existing data structures
  - Example: PyLith
- Reorganization for new development
  - Better describe common FEM operations
  - Example: EqSim/PyLith integration

PyLith in Parallel

PyLith can now handle moderately sized meshes in parallel:



Mesh courtesy of Carl Gable, LANL

Sieve and PETSc

- Sieve is the unstructured mesh component of PETSc
    - Wrapper obeys the DM semantics
    - Works with PETSc Viewers

Sieve and PETSc

- Sieve is the unstructured mesh component of PETSc
  - Wrapper obeys the DM semantics
  - Works with PETSc Viewers
- Sieve can be installed separately (almost)
  - FEniCS Project component

Sieve and PETSc

- Sieve is the unstructured mesh component of PETSc
  - Wrapper obeys the DM semantics
  - Works with PETSc Viewers
- Sieve can be installed separately (almost)
  - FEniCS Project component
- Includes support for global, contiguous vectors
  - Global and local numberings
  - Automatic, persistent scatters

Rhs Evaluation

```
const patch_type          patch    = 0;
const Obj<label_sequence>& elements = topology->heightStratum(patch, 0);
PetscScalar               elementVec[NUM_BASIS];

for(iterator e_iter = elements->begin(); e_iter != elements->end(); ++e_iter)
  ElementGeometry(mesh, *e_iter, v0, Jac, PETSC_NULL, &detJ);
  PetscMemzero(elementVec, NUM_BASIS*sizeof(PetscScalar));
  for(int q = 0; q < NUM_QUADRATURE_POINTS; q++) {
    xi  = points[q*2+0] + 1.0;
    eta = points[q*2+1] + 1.0;
    x_q = Jac[0]*xi + Jac[1]*eta + v0[0];
    y_q = Jac[2]*xi + Jac[3]*eta + v0[1];
    for(i = 0; i < NUM_BASIS; i++) {
      elementVec[i] += Basis[q*NUM_BASIS+i]*f(x_q, y_q)*weights[q]*detJ;
    }
  }
  field->updateAdd(*e_iter, elementVec);
}
```

Jacobian Evaluation

```
PetscScalar elementMat[NUM_BASIS*NUM_BASIS];

for(iterator e_iter = elements->begin(); e_iter != elements->end(); ++e_iter)
  ElementGeometry(mesh, *e_iter, v0, Jac, Jinv, &detJ);
  PetscMemzero(elementMat, NUM_BASIS*NUM_BASIS*sizeof(PetscScalar));
  for(int q = 0; q < NUM_QUADRATURE_POINTS; q++) {
    for(i = 0; i < NUM_BASIS; i++) {
      testDer[0] = Jinv[0]*BasisDers[(q*NUM_BASIS+i)*2+0] + Jinv[2]*BasisI
      testDer[1] = Jinv[1]*BasisDers[(q*NUM_BASIS+i)*2+0] + Jinv[3]*BasisI
      for(j = 0; j < NUM_BASIS; j++) {
        basisDer[0] = Jinv[0]*BasisDers[(q*NUM_BASIS+j)*2+0] + Jinv[2]*Ba
        basisDer[1] = Jinv[1]*BasisDers[(q*NUM_BASIS+j)*2+0] + Jinv[3]*Ba
        elementMat[i*NUM_BASIS+j] +=
          (testDer[0]*basisDer[0] + testDer[1]*basisDer[1])*weights[q]*detJ;
  }}}
  updateOperator(jac, field, globalOrder, *e_iter, elementMat, ADD_VALUES);
}
```

FIAT Integration

Finite Element Integrator and Tabulator by Rob Kirby

In order to generate a quadrature routines, we need:

- A differential form to integrate
- An element (usually a family and degree) using FIAT
- A quadrature rule

We then produce

- A C integration routines
- A Python module wrapper
- Optional Ferari optimized routines
- Optional element assembly loop

FIAT is part of the FEniCS project, as is the PETSc Sieve module

A Language for Weak Forms

In collaboration with

- Anders Logg of Simula
- Rob Kirby of Texas Tech

We have developed a small language for weak forms, based directly on an AST representation.

The Fenics Form Compiler (FFC) processes each form algebraically, allowing some simplification and optimization, before passing it on to the integration generation routines.

Input Language

We have a simple text language for input, incorporating:

- Arithmetic, +, -, *, /, ^, () abs(x)
- Coordinate functions, cos(x) exp(x)
- Continuum fields (known and unknown)
- Dual pairing, $\langle\ ,\ \rangle$
- Matrix operations, TRANS(u) DET(u) VEC(u)
- Differential operators, GRAD(u) DIV(u) CURL(u)

We must use expression graphs for efficiency.

Examples

- Poisson Equation

```
<grad t, grad u> - <t, 2*y*(2-y) + 2*x*(2-x)>
```

- Vector Poisson Equation

```
<grad vec t, grad vec u>
  - <vec t, {4, 2*y*(2-y) + 2*x*(2-x)}>
```

- Linear Elasticity

```
<grad vec t, (grad vec u) + trans (grad vec u)>
  - <vec t, {6, 6}>
```

- Stokes Equation

```
<grad t, grad u> - <t, grad p> + <q, div u>
  - <vec t, {4, -4}> + <q, 0>
```

References

- Documentation: http://www.mcs.anl.gov/petsc/docs
  - PETSc Users manual
  - Manual pages
  - Many hyperlinked examples
  - FAQ, Troubleshooting info, installation info, etc.
- Publications: http://www.mcs.anl.gov/petsc/publications
  - Research and publications that make use PETSc
- MPI Information: http://www.mpi-forum.org
- **Using MPI** (2nd Edition), by Gropp, Lusk, and Skjellum
- **Domain Decomposition**, by Smith, Bjorstad, and Gropp