# PETSc Tutorial

PETSc Team
Presented by Matthew Knepley

Mathematics and Computer Science Division
Argonne National Laboratory

ACTS Workshop 2007
Berkeley, CA
August 21–24, 2007



# Tutorial Goal

Enable students to develop new simulations with PETSc.

- Serial and Parallel

# Tutorial Goal

Enable students to develop new simulations with PETSc.

- Serial and Parallel
- Linear and Nonlinear

# Tutorial Goal

Enable students to develop new simulations with PETSc.

- Serial and Parallel
- Linear and Nonlinear
- Finite Difference and Finite Element

# Tutorial Goal

## Enable students to develop new simulations with PETSc.

- Serial and Parallel
- Linear and Nonlinear
- Finite Difference and Finite Element
- Structured and Unstructured

# Tutorial Goal

Enable students to develop new simulations with PETSc.

- Serial and Parallel
- Linear and Nonlinear
- Finite Difference and Finite Element
- Structured and Unstructured
- Triangles and Hexes

# Tutorial Goal

Enable students to develop new simulations with PETSc.

- Serial and Parallel
- Linear and Nonlinear
- Finite Difference and Finite Element
- Structured and Unstructured
- Triangles and Hexes
- Optimal Solvers

# Tutorial Goal

## Enable students to develop new simulations with PETSc.

- Serial and Parallel
- Linear and Nonlinear
- Finite Difference and Finite Element
- Structured and Unstructured
- Triangles and Hexes
- Optimal Solvers

Items in red not finished for tutorial

# Outline

# Outline

# How did PETSc Originate?

PETSc was developed as a Platform for Experimentation

We want to experiment with different

- Models
- Discretizations
- Solvers
- Algorithms (which blur these boundaries)

# 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 flexible 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.*

— Barry Smith

# What is PETSc?

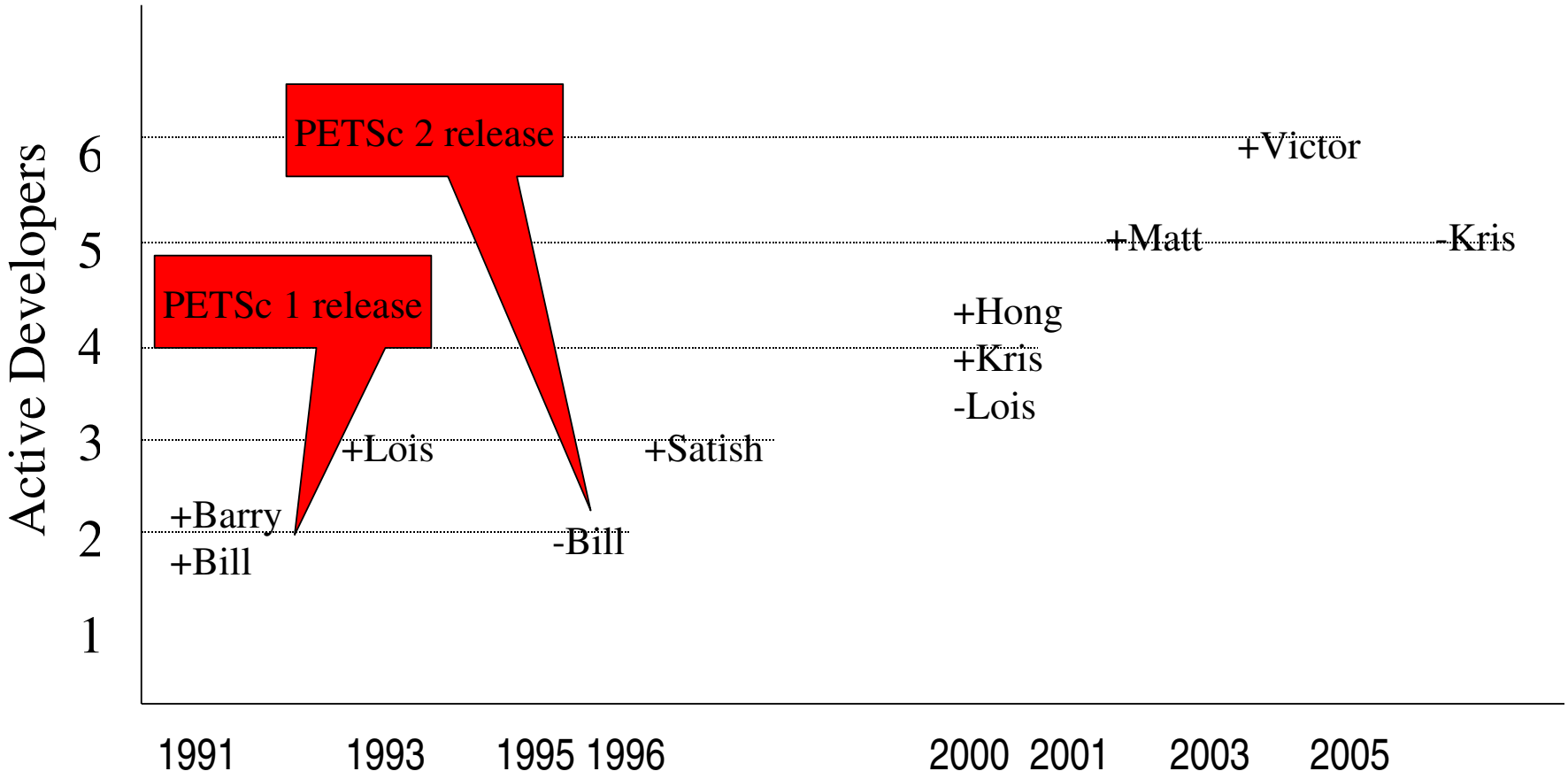
A freely available and supported research code

- 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 20,000 downloads since 1995 (version 2), currently 300 per month
- PETSc Funding and Support
  - Department of Energy
    - SciDAC, MICS Program, INL Reactor Program
  - National Science Foundation
    - CIG, CISE, Multidisciplinary Challenge Program

# Timeline


Active Developers
1
2
3
4
5
6
1991
1993
1995
1996
2000
2001
2003
2005
PETSc 2 release
PETSc 1 release
+Victor
+Matt
-Kris
+Hong
+Kris
-Lois
+Lois
+Satish
+Barry
+Bill
-Bill

# 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

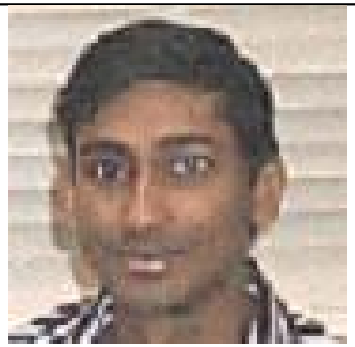
# Who Uses PETSc?

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

# The PETSc Team

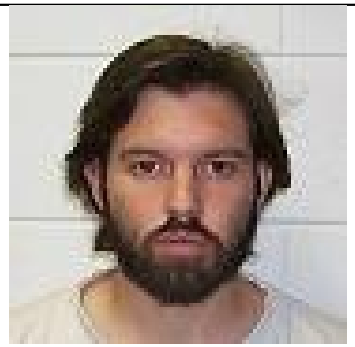

Bill Gropp


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://petsc.cs.iit.edu/petsc/petsc-dev
  - http://petsc.cs.iit.edu/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
- We also make release repositories available
  - http://petsc.cs.iit.edu/petsc/petsc-release-2.3.3

# Unpacking PETSc

- Just clone development repository
  - `hg clone http://petsc.cs.iit.edu/petsc/petsc-dev petsc-dev`
  - `hg clone -rRelease-2.3.3 petsc-dev petsc-2.3.3`

**or**

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

# Getting the Source

You will need the Developer copy of PETSc:

- Using Mercurial

```
hg clone http://petsc.cs.iit.edu/petsc/petsc-dev
cd petsc-dev/python
hg clone http://petsc.cs.iit.edu/petsc/BuildSystem
```

- Manual download

```
wget ftp://info.mcs.anl.gov/pub/petsc/petsc-dev.tar.gz .
```

and the tutorial source code:

- Using Mercurial

```
hg clone http://petsc.cs.iit.edu/petsc/ACTS07TutorialCode
```

- Manual download

```
wget ftp://info.mcs.anl.gov/pub/petsc/ACTS07TutorialCode.tar.gz .
```

# Configuring PETSc

- Set `$PETSC_DIR` to the installation root directory
- Run the configuration utility
  - `$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, Zoltan
  - MUMPS, Spooles, SuperLU, SuperLU_Dist, UMFPack, pARMS
  - BLOPEX, FFTW, SPRNG
  - Prometheus, HYPRE, ML, SPAI
  - Sundials
  - Triangle, TetGen
  - FIAT, FFC, Generator
  - Boost

# 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 `$PETSC_DIR/$PETSC_ARCH/lib/`
- Can also build a subtree
  - `cd src/snes; make`
  - `cd src/snes; make ACTION=libfast tree`

# 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_monitor_draw`
- Can disable dynamically
  - `-ksp_monitor_cancel`
- Does not display subsolvers
  - `-snes_monitor`
- Can use the true residual
  - `-ksp_monitor_true_residual`
- Can display different subobjects
  - `-snes_monitor_solution`, `-snes_monitor_solution_update`, `-snes_monitor_residual`
  - `-ksp_gmres_krylov_monitor`
- Can display the spectrum
  - `-ksp_monitor_singular_value`

# PETSc Example

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`

# User Example

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_SNES_LIB}`
   - `include ${PETSC_DIR}/conf/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
  - Lorena Barba
  - David Keyes
  - Xiao-Chuan Cai
  - Richard Katz

# Following the Tutorial

Update to each new checkpoint (r0):

- `hg clone -r0 ACTS07TutorialCode code-test`

**or**

- `hg update 0`

Build the executable with `make`, and then run:

- `make runbratu`
- `make debugbratu`
- `make valbratu`
- `make NP=2 runbratu`
- `make EXTRA_ARGS="-pc_type jacobi" runbratu`

# Code Update

## Update to Revision 0

# 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

# Command Line Processing

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

# 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

# Outline

# Higher Level Abstractions

The PETSc `DA` class is a topology and discretization interface.

- Structured grid interface
  - Fixed simple topology
- Supports stencils, communication, reordering
  - Limited idea of operators
- Nice for simple finite differences

The PETSc `Mesh` class is a topology interface.

- Unstructured grid interface
  - Arbitrary topology and element shape
- Supports partitioning, distribution, and global orders

# Higher Level Abstractions

The PETSc `DM` class is a hierarchy interface.

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

The PETSc `Section` class is a function interface.

- Functions over unstructured grids
  - Arbitrary layout of degrees of freedom
- Support distribution and assembly

# Code Update

## Update to Revision 1

# 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

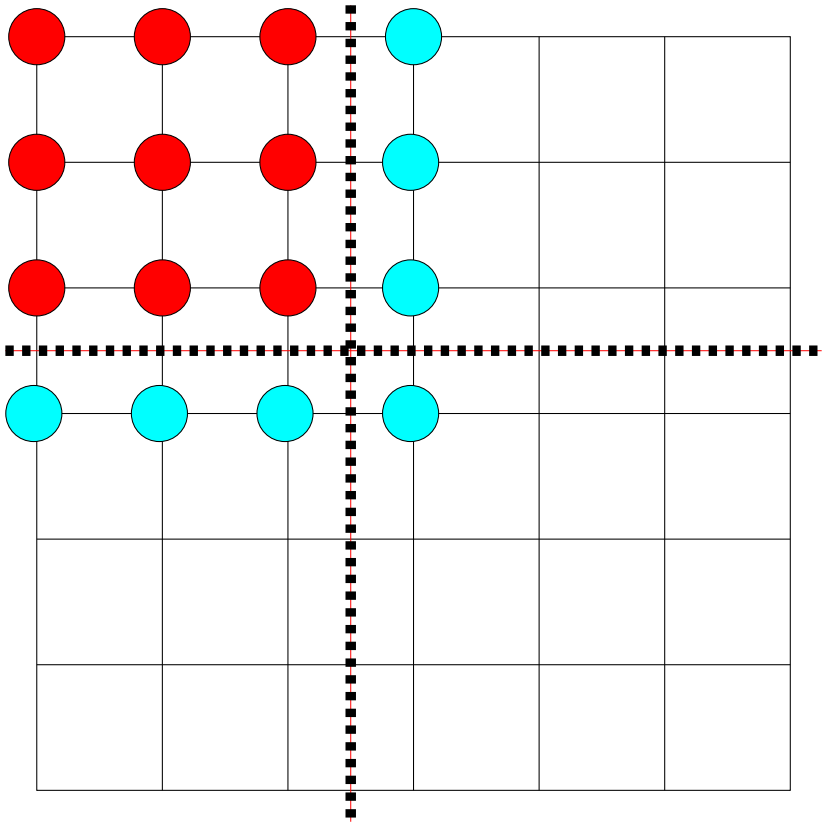
`s`: The stencil width

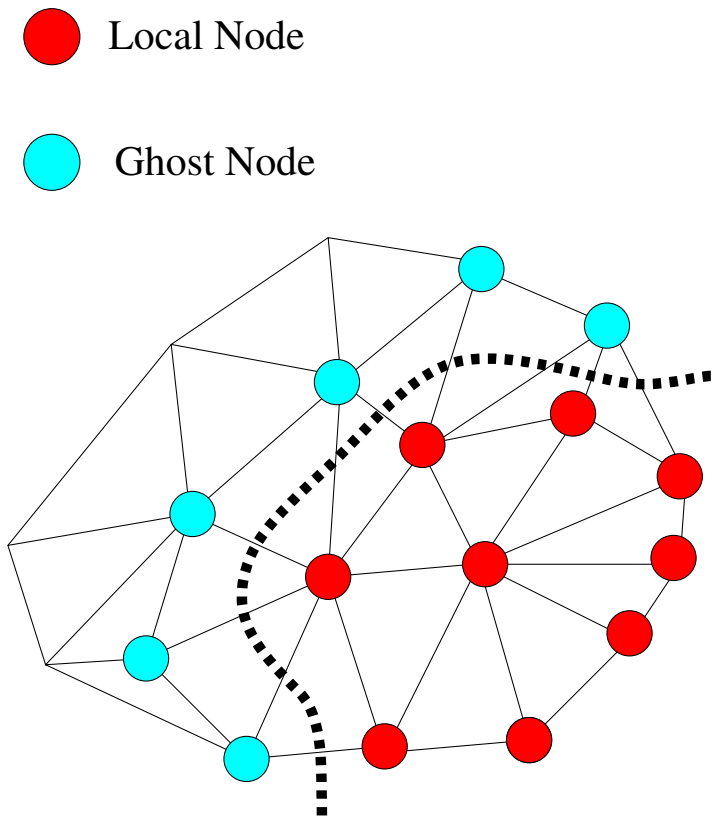
`lm/n`: Alternative array of local sizes
- Use `PETSC_NULL` for the default

# 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

# Viewing the DA

- `make NP=1 EXTRA_ARGS="-da_view_draw -draw_pause -1" runbratu`
- `make NP=1 EXTRA_ARGS="-da_grid_x 10 -da_grid_y 10 -da_view_draw -draw_pause -1" runbratu`
- `make NP=4 EXTRA_ARGS="-da_grid_x 10 -da_grid_y 10 -da_view_draw -draw_pause -1" runbratu`

# 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 `-malloc_dump` will print unfreed memory on `PetscFinalize()`
- Simply the best tool today is valgrind
  - It checks memory access, cache performance, memory usage, etc.
  - `http://www.valgrind.orghttp://www.valgrind.org`

# Memory Debugging

We can check for unfreed memory using:

```
make EXTRA_ARGS="-malloc_dump" runbratu
```

There is a leak!

All options can be seen using:

```
make EXTRA_ARGS="-help" runbratu
```

# Code Update

## Update to Revision 2

# Command Line Processing

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

# Code Update

## Update to Revision 3

# 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()`

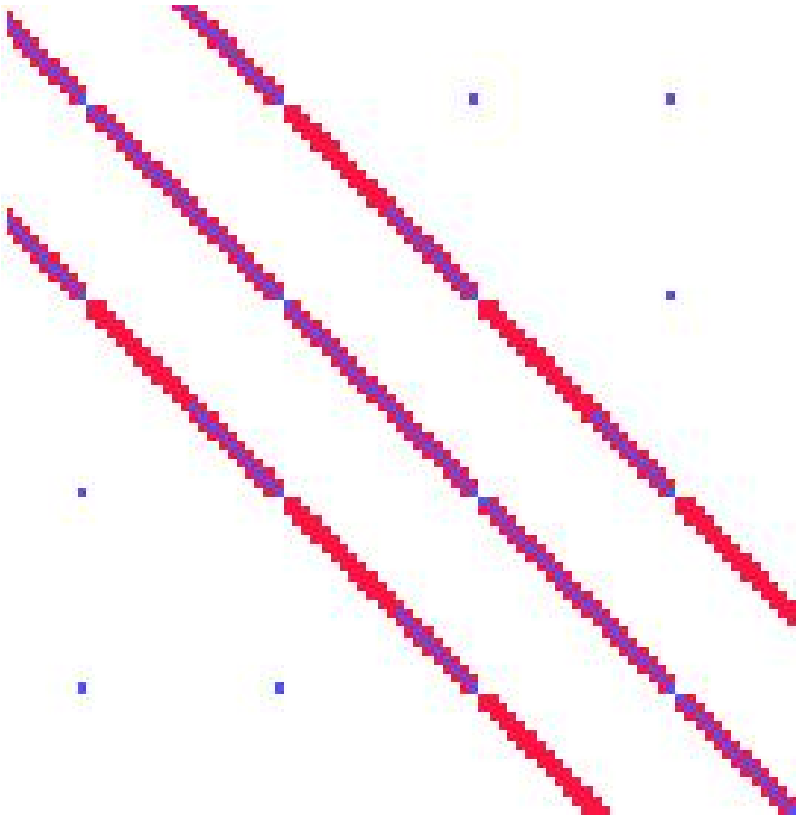
# Indicating Expected Nonzeros

## Sequential Sparse Matrices

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

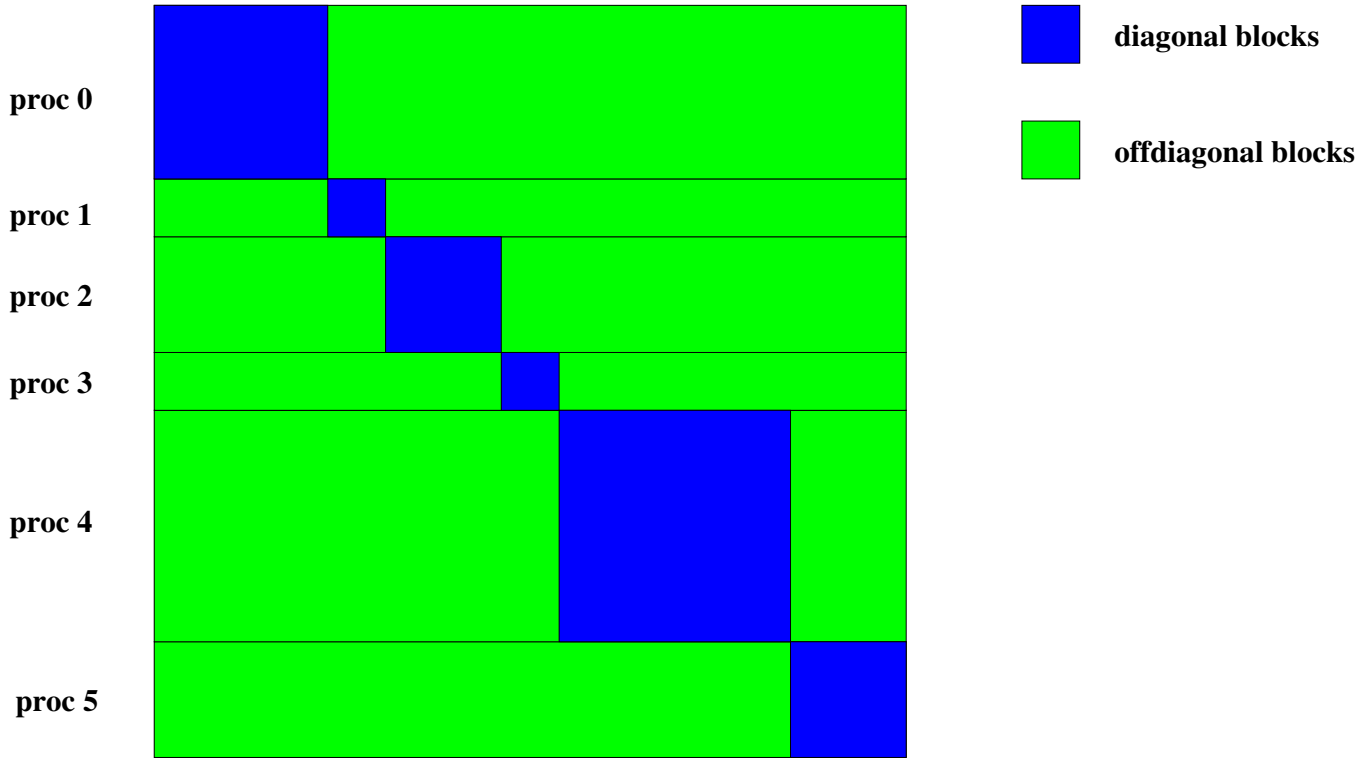
nz: expected number of nonzeros in any row

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!
```

# 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.

# Creating the Mesh

- Generic object
  - `MeshCreate()`
  - `MeshSetMesh()`
- File input
  - `MeshCreatePCICE()`, `MeshCreatePyLith()`
  - `MeshCreateDolfin()`
- Generation
  - `MeshGenerate()`
  - `MeshRefine()`, `MeshCoarsen()`
  - `ALE::MeshBuilder::createSquareBoundary()`
- Representation
  - `ALE::SieveBuilder::buildTopology()`
  - `ALE::SieveBuilder::buildCoordinates()`
- Partitioning and Distribution
  - `MeshDistribute()`
  - `MeshDistributeByFace()`

# Code Update

## Update to Revision 4

# Viewing the Mesh

- `make NP=1 EXTRA_ARGS="-structured 0 -mesh_view_vtk" runbratu`
- `mayavi -d bratu.vtk -m SurfaceMap&`
- `make NP=4 EXTRA_ARGS="-structured 0 -mesh_view_vtk" runbratu`
- Viewable using Mayavi or Paraview

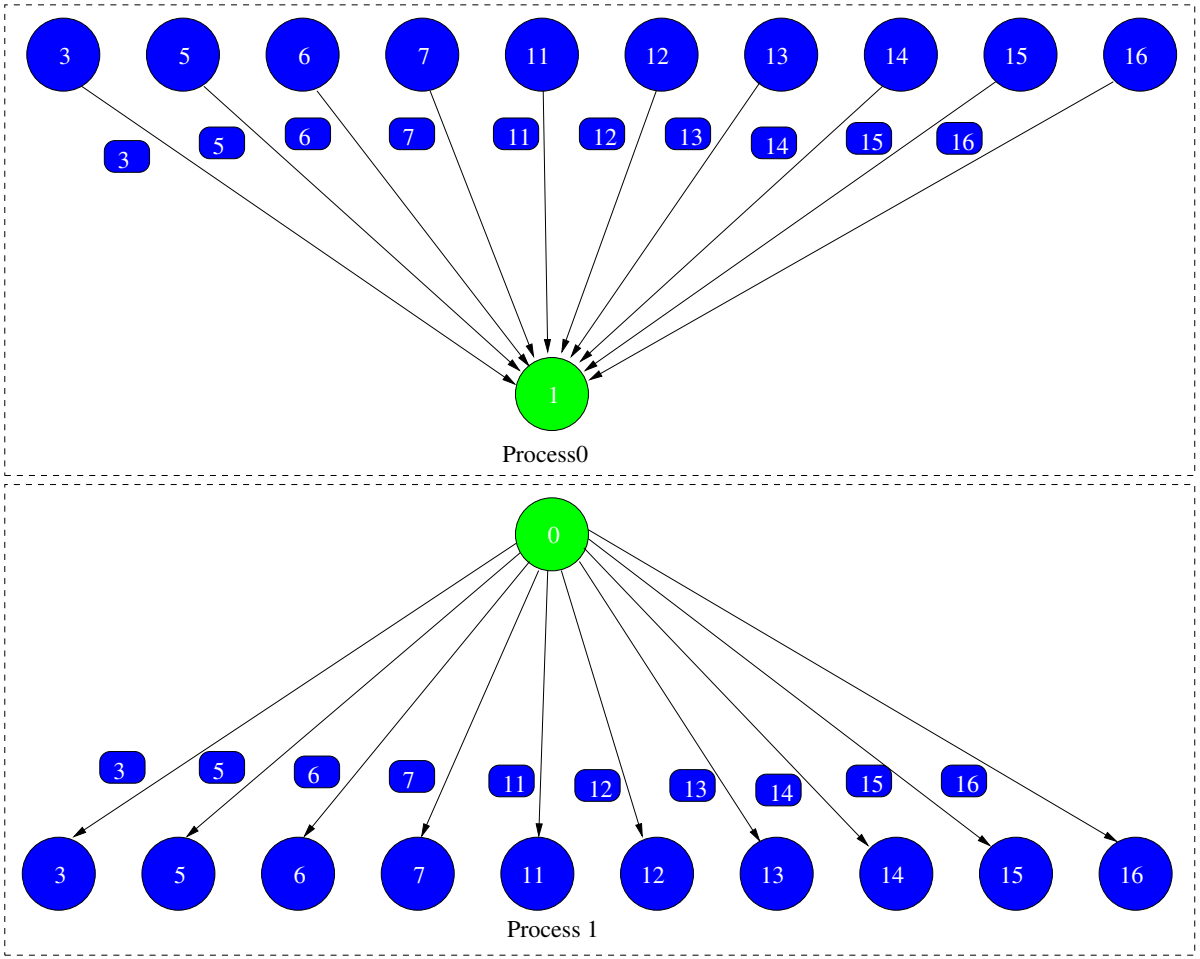
# Refining the Mesh

- `make NP=1 EXTRA_ARGS="-structured 0 -generate -mesh_view_vtk" runbratu`
- `make NP=1 EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.0625 -mesh_view_vtk" runbratu`
- `make NP=4 EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.0625 -mesh_view_vtk" runbratu`

# Parallel Sieves

- Sieves use *names*, not numberings
  - Allows independent adaptation
  - Demanding a global numbering can seriously impact memory scaling
  - Numberings can be constructed on demand
- `Overlaps` relate names on different processes
  - An Overlap can be encoded by a Sieve
- Distribution of a `Section` pushes forward along the Overlap
  - Sieves are distributed as "cone" sections

# Overlap for Distribution




- The send overlap is above the receive overlap
- Green points are remote process ranks
- Arrow labels indicate remote process names

# Code Update

## Update to Revision 5

# Viewing the 3d Mesh

- `make NP=1 EXTRA_ARGS="-dim 3 -da_view_draw -draw_pause -1" runbratu`
- `make NP=4 EXTRA_ARGS="-da_grid_x 5 -da_grid_y 5 -da_grid_z 5 -da_view_draw -draw_pause -1" runbratu`
- `make NP=1 EXTRA_ARGS="-dim 3 -structured 0 -generate -mesh_view_vtk" runbratu`
- `mayavi -d bratu.vtk -m SurfaceMap -f UserDefined:vtkExtractEdges`
- `make NP=4 EXTRA_ARGS="-dim 3 -structured 0 -generate -refinement_limit 0.01 -mesh_view_vtk" runbratu`

# Outline

# A DA is more than a Mesh

A DA contains topology, geometry, and an implicit Q1 discretization.

It is used as a template to create

- Vectors (functions)
- Matrices (linear operators)

# 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
- `DAGlobalToLocalEnd(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

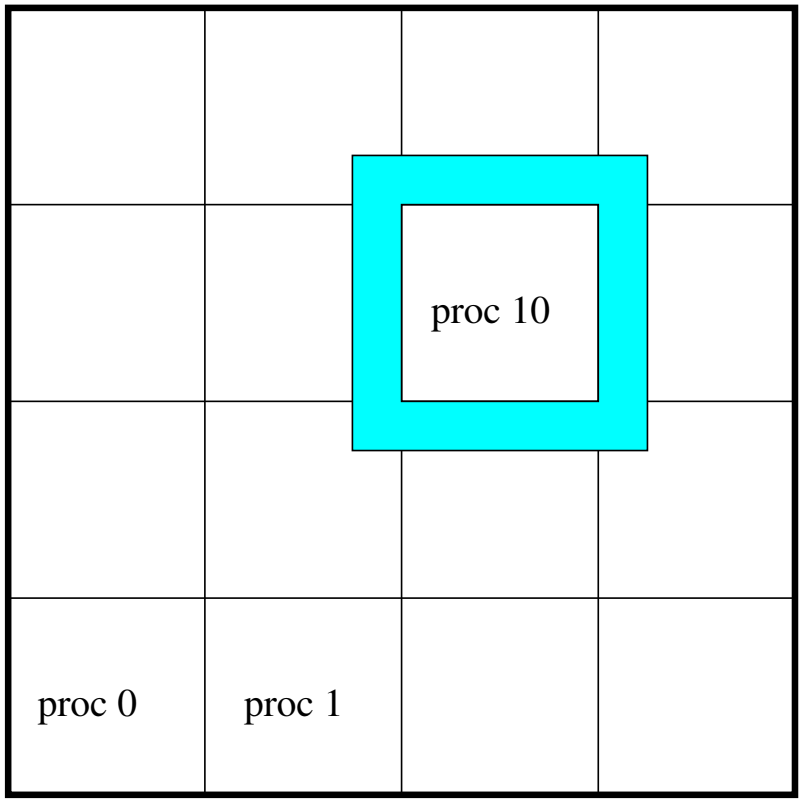
`ctx`: The user context passed to `DASetLocalFunction()`

The local DA function is activated by calling

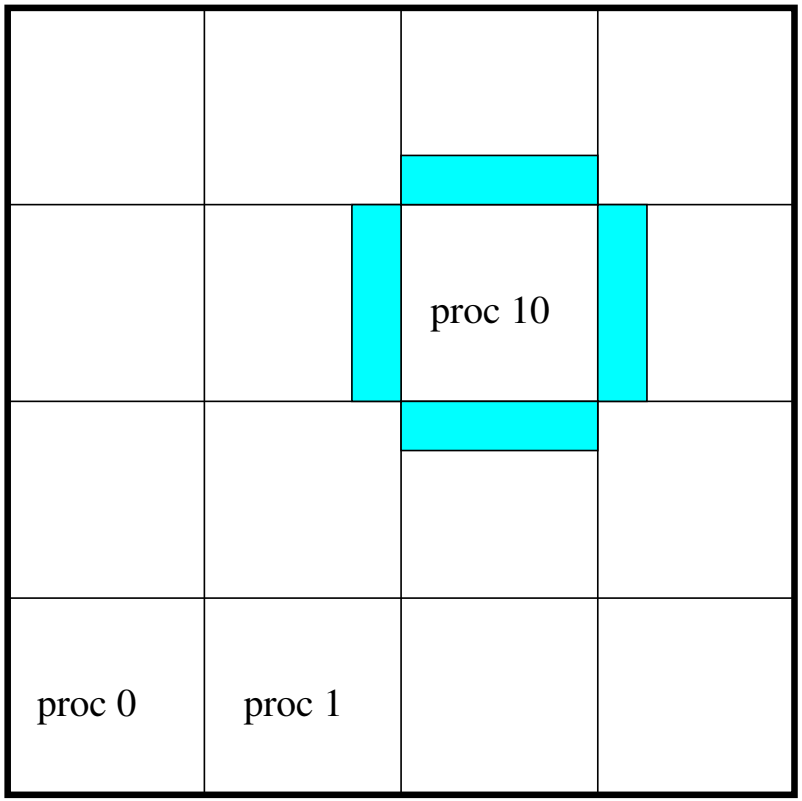
```
SNESSetFunction(snes, r, SNESDAFormFunction, ctx)
```

# DA Stencils

Both the box stencil and star stencil are available.




Box Stencil




Star Stencil

# Setting Values on Regular Grids

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

# Code Update

## Update to Revision 6

# Structured Functions

- Functions takes values at the DA vertices
- Used as approximations to functions on the continuous domain
  - Values are really coefficients of linear basis
- User only constructs the local portion
- `make NP=2 EXTRA_ARGS="-run test -vec_view_draw -draw_pause -1" runbratu`

# Sections

## *Sections* associate data to submeshes

- Name comes from section of a fiber bundle
  - Generalizes linear algebra paradigm
- Define `restrict()`,`update()`
- Define `complete()`
- Assembly routines take a `Sieve` and several `Sections`
  - This is called a `Bundle`

# Section Types

Section can contain arbitrary values

- C++ interface is templated over value type
- C interface has two value types
  - `SectionReal`
  - `SectionInt`

Section can have arbitrary layout

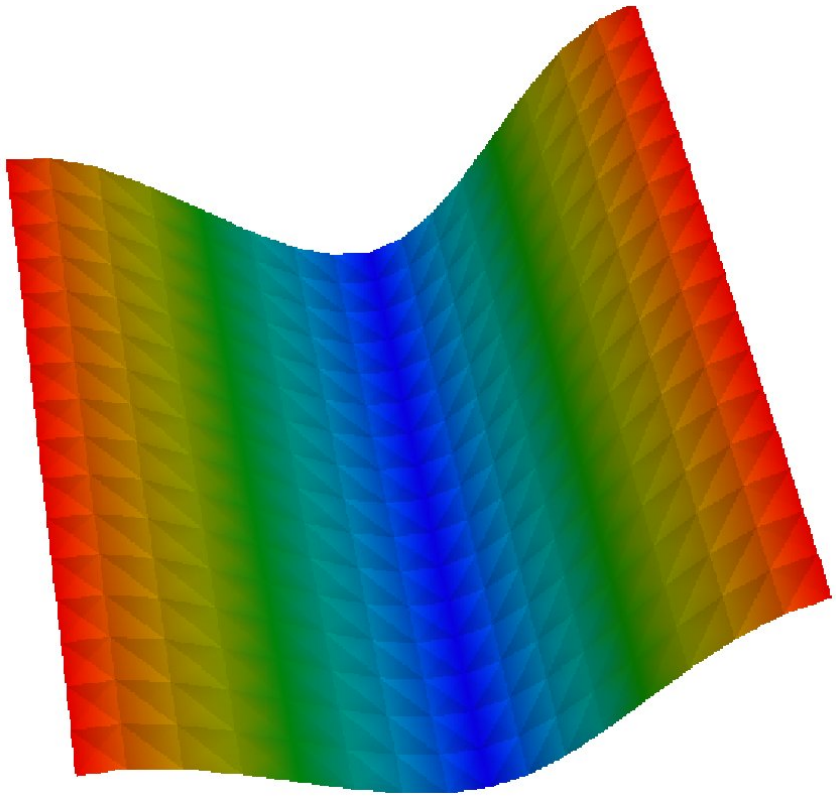
- C++ interface can place unknowns on any Mesh entity (Sieve point)
  - `Mesh::setupField()` parametrized by `Discretization` and `BoundaryCondition`
- C interface has default layouts
  - `MeshGetVertexSectionReal()`
  - `MeshGetCellSectionReal()`

# Code Update

## Update to Revision 7

# Viewing the Section

- `make EXTRA_ARGS="-run test -structured 0 -vec_view_vtk" runbratu`
  - Produces linear.vtk and cos.vtk
- Viewable with MayaVi, exactly as with the mesh.
- `make NP=2 EXTRA_ARGS="-run test -structured 0 -vec_view_vtk -generate -refinement_limit 0.003125" runbratu`
  - Use `mayavi -d cos.vtk -m SurfaceMap -f WarpScalar`



# Outline

# Weak Forms

A *weak form* is the pairing of a function with an element of the *dual space*.

- Produces a number (by definition of the dual)
- Can be viewed as a "function" of the dual vector
- Used to define finite element solutions
- Require a dual space and integration rules

For example, for $f \in V$, we have the weak form

$$\int_\Omega \phi(\mathbf{x}) f(\mathbf{x}) dx \qquad \phi \in V^*$$

# FIAT

Finite Element Integrator and Tabulator by Rob Kirby

`http://www.fenics.org/fiat`

FIAT understands

- Reference element shapes (line, triangle, tetrahedron)
- Quadrature rules
- Polynomial spaces
- Functionals over polynomials (dual spaces)
- Derivatives

User can build arbitrary elements by specifying the Ciarlet triple $(K, P, P')$

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

# Code Update

## Update to Revision 8

# FIAT Integration

The `quadrature.fiat` file contains:

- An element (usually a family and degree) defined by FIAT
- A quadrature rule

It is run

- automatically by `make`, or
- independently by the user

It can take arguments

- `--element_family` and `--element_order`, or
- `make` takes variables `ELEMENT` and `ORDER`

Then `make` produces `quadrature.h` with:

- Quadrature points and weights
- Basis function and derivative evaluations at the quadrature points
- Integration against dual basis functions over the cell
- Local dofs for Section allocation

# Boundary Conditions

Dirichlet conditions may be expressed as

Neumann conditions may be expressed as

# Boundary Conditions

Dirichlet conditions may be expressed as

$$u|_\Gamma = g$$

Neumann conditions may be expressed as

# Boundary Conditions

Dirichlet conditions may be expressed as

$$u|_\Gamma = g$$

and implemented by constraints on dofs in a Section

Neumann conditions may be expressed as

# Boundary Conditions

Dirichlet conditions may be expressed as

$$u|_\Gamma = g$$

and implemented by constraints on dofs in a Section

- The user provides a function.

Neumann conditions may be expressed as

# Boundary Conditions

Dirichlet conditions may be expressed as

$$u|_\Gamma = g$$

and implemented by constraints on dofs in a Section

- The user provides a function.

Neumann conditions may be expressed as

$$\nabla u \cdot \hat{n}|_\Gamma = h$$

# Boundary Conditions

Dirichlet conditions may be expressed as

$$u|_\Gamma = g$$

and implemented by constraints on dofs in a Section

- The user provides a function.

Neumann conditions may be expressed as

$$\nabla u \cdot \hat{n}|_\Gamma = h$$

and implemented by explicit integration along the boundary

# Boundary Conditions

Dirichlet conditions may be expressed as

$$u|_\Gamma = g$$

and implemented by constraints on dofs in a Section

- The user provides a function.

Neumann conditions may be expressed as

$$\nabla u \cdot \hat{n}|_\Gamma = h$$

and implemented by explicit integration along the boundary

- The user provides a weak form.

# Dirichlet Conditions (Essential BC)

- Explicit limitation of the approximation space
- Idea:
  - Maintain the same FEM interface (`restrict()`, `update()`)
  - Allow direct access to reduced problem (contiguous storage)
- Implementation
  - Ignored by `size()` and `update()`, but `restrict()` works normally
  - Use `updateBC()` to define the boundary values
  - Use `updateAll()` to define both boundary and regular values
  - Points have a negative fiber dimension **or**
  - Dof are specified as constrained

# Dirichlet Values

- Topological boundary is marked during generation
- Cells bordering boundary are marked using `markBoundaryCells()`
- To set values:
  1. Loop over boundary cells
  2. Loop over the element closure
  3. For each boundary point $i$, apply the functional $N_i$ to the function $g$
- The functionals are generated with the quadrature information
- Section allocation applies Dirichlet conditions automatically
  - Values are stored in the Section
  - `restrict()` behaves normally, `update()` ignores constraints

# Dual Basis Application

We would like the action of a dual basis vector (functional)

$$< \mathcal{N}_i, f >= \int_{\text{ref}} N_i(x) f(x) dV$$

- Projection onto $\mathcal{P}$
- Code is generated from FIAT specification
  - Python code generation package inside PETSc
- Common interface for all elements

# Maps

We are interested in nonlinear maps $F : \mathbb{R}^n \longrightarrow \mathbb{R}^n$.

- Can contain the action of differential operators
- Encapsulated in `Rhs_*()` methods
- Will later be used to form the residual of our system

# Code Update

## Update to Revision 9

# Section Assembly

First we do local operations:

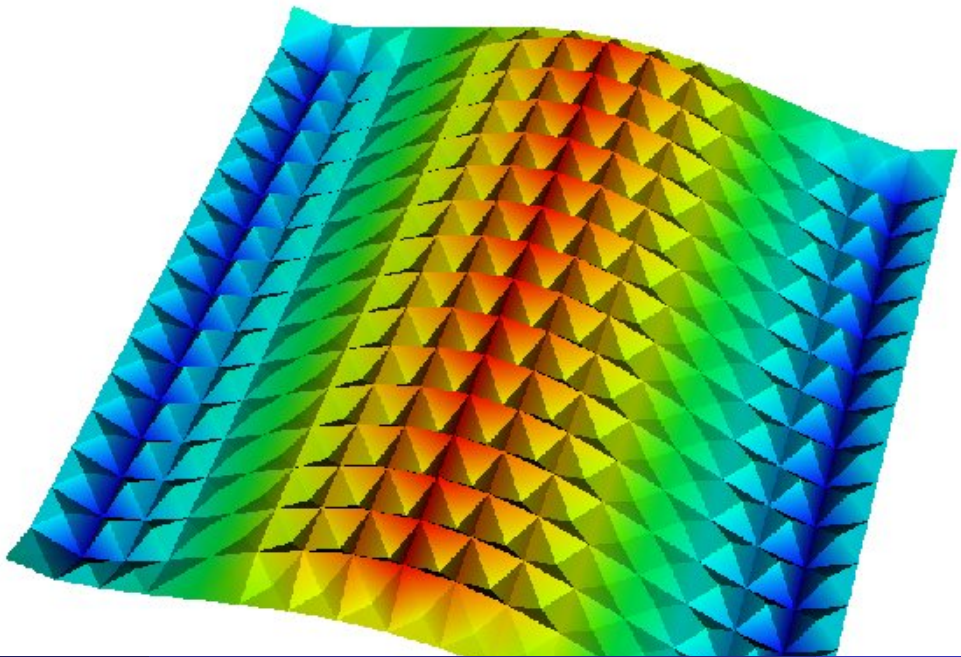
- Loop over cells
- Compute cell geometry
- Integrate each basis function to produce an element vector
- Call `SectionUpdateAdd()`
  - Note that this updates the *closure* of the cell

Then we do global operations:

- `SectionComplete()` exchanges data across overlap
  - C just adds nonlocal values (C++ is flexible)
- C++ also allows completion over arbitrary overlaps

# Viewing a Mesh Weak Form

- We use finite elements and a *Galerkin* formulation
  - We calculate the residual $F(u) = -\Delta u - f$
  - Correct basis/derivatives table chosen by `setupQuadrature()`
  - Could substitute exact integrals for quadrature
- `make NP=2 EXTRA_ARGS="-run test -structured 0 -vec_view_vtk -generate -refinement_limit 0.003125" runbratu`
- `make EXTRA_ARGS="-run test -dim 3 -structured 0 -generate -vec_view_vtk" runbratu`



# Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions

# Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions

Largely dim dependent
(e.g. quadrature)

# Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions

Largely dim dependent
(e.g. quadrature)

Global (topological)

- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies

# Global and Local

Local (analytical)

- Discretization/Approximation
  - FEM integrals
  - FV fluxes
- Boundary conditions

Largely dim dependent
(e.g. quadrature)

Global (topological)

- Data management
  - Sections (local pieces)
  - Completions (assembly)
- Boundary definition
- Multiple meshes
  - Mesh hierarchies

Largely dim independent
(e.g. mesh traversal)

# Difference Approximations

With finite differences, we approximate differential operators with difference quotients,

$$\begin{aligned} \frac{\partial u(x)}{\partial x} &\approx \frac{u(x+h)-u(x-h)}{2h} \\ \frac{\partial^2 u(x)}{\partial x^2} &\approx \frac{u(x+h)-2u(x)+u(x-h)}{h^2} \end{aligned}$$

The important property for the approximation is *consistency*, meaning

$$\lim_{h \to 0} \frac{\partial u(x)}{\partial x} - \frac{u(x+h) - u(x-h)}{2h} = 0$$

and in fact,

$$\frac{\partial^2 u(x)}{\partial x^2} - \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} \in \mathcal{O}(h^2)$$

# Code Update

## Update to Revision 10

# Viewing FD Operator Actions

We cannot currently visualize the 3D results,

- `make EXTRA_ARGS="-run test -vec_view_draw -draw_pause -1" runbratu`
- `make EXTRA_ARGS="-run test -da_grid_x 10 -da_grid_y 10 -vec_view_draw -draw_pause -1" runbratu`
- `make EXTRA_ARGS="-run test -dim 3 -vec_view" runbratu`

but can check the ASCII output if necessary.

# Debugging Assembly

On two processes, I get a SEGV!

So we try running with:

- `make NP=2 EXTRA_ARGS="-run test -vec_view_draw -draw_pause -1" debugbratu`

# Debugging Assembly

On two processes, I get a SEGV!

So we try running with:

- `make NP=2 EXTRA_ARGS="-run test -vec_view_draw -draw_pause -1" debugbratu`
- Spawns one debugger window per process

# Debugging Assembly

On two processes, I get a SEGV!

So we try running with:

- `make NP=2 EXTRA_ARGS="-run test -vec_view_draw -draw_pause -1" debugbratu`
- Spawns one debugger window per process
- SEGV on access to ghost coordinates

# Debugging Assembly

On two processes, I get a SEGV!

So we try running with:

- `make NP=2 EXTRA_ARGS="-run test -vec_view_draw -draw_pause -1" debugbratu`
- Spawns one debugger window per process
- SEGV on access to ghost coordinates
- Fix by using a local ghosted vector
  - Update to Revision 11

# Debugging Assembly

On two processes, I get a SEGV!

So we try running with:

- `make NP=2 EXTRA_ARGS="-run test -vec_view_draw -draw_pause -1" debugbratu`
- Spawns one debugger window per process
- SEGV on access to ghost coordinates
- Fix by using a local ghosted vector
  - Update to Revision 11
- Notice
  - we already use ghosted assembly (completion) for FEM
  - FD does not need ghosted assembly

# Representations of the Error

- A single number, the norm itself
- A number per element, the element-wise norm
- Injection into the finite element space

$$e = \sum_i e_i \phi_i(x) \tag{1}$$

  - We calculate $e_i$ by least-squares projection into $\mathcal{P}$

# Interpolation Pitfalls

Comparing solutions on different meshes can be problematic.

- Picture our solutions as functions defined over the entire domain
  - For FEM, $\hat{u}(x) = \sum_i u_i \phi_i(x)$
- After interpolation, the interpolant might not be the same function
- We often want to preserve thermodynamic bulk properties
  - Energy, stress energy, incompressibility, ...
- Can constrain interpolation to preserve desirable quantities
  - Usually produces a saddlepoint system

# Calculating the $L_2$ Error

We begin with a continuum field $u(x)$ and a finite element approximation

$$\hat{u}(x) = \sum_i \hat{u}_i \phi_i(x) \tag{2}$$

The FE theory predicts a convergence rate for the quantity

$$||u - \hat{u}||_2^2 = \sum_T \int_T dA(u - \hat{u})^2 \tag{3}$$

$$= \sum_T \sum_q w_q |J| \left( u(q) - \sum_j \hat{u}_j \phi_j(q) \right)^2 \tag{4}$$

$$\tag{5}$$

The estimate for linear elements is

$$||u - \hat{u}_h|| < Ch||u|| \tag{6}$$

# Code Update

## Update to Revision 12

# Calculating the Error

- Added `CreateProblem()`
  - Define the global section
  - Setup exact solution and boundary conditions
- Added `CreateExactSolution()` to project the solution function
- Added `CheckError()` to form the error norm
  - Finite differences calculates a pointwise error
  - Finite elements calculates a normwise error
- Added `CheckResidual()` which uses our previous functionality

# Checking the Error

- `make NP=2 EXTRA_ARGS="-run full -da_grid_x 10 -da_grid_y 10" runbratu`
- `make EXTRA_ARGS="-run full -dim 3" runbratu`
- `make EXTRA_ARGS="-run full -structured 0 -generate" runbratu`
- `make NP=2 EXTRA_ARGS="-run full -structured 0 -generate" runbratu`
- `make EXTRA_ARGS="-run full -structured 0 -generate -refinement_limit 0.03125" runbratu`
- `make EXTRA_ARGS="-run full -dim 3 -structured 0 -generate -refinement_limit 0.01" runbratu`

Notice that the FE error does not always vanish, since we are using information across the entire element. We can enrich our FE space:

- `rm quadrature.h; make ORDER=2`
- `make EXTRA_ARGS="-run full -structured 0 -generate -refinement_limit 0.03125" runbratu`
- `make EXTRA_ARGS="-run full -dim 3 -structured 0 -generate -refinement_limit 0.01" runbratu`

# Outline

# DA Local Jacobian

The user provided function which calculates the Jacboian 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)
```

# Code Update

## Update to Revision 13

# DA Operators

- Evaluate only the local portion
  - No nice local array form without copies
- Use `MatSetValuesStencil()` to convert `(i,j,k)` to indices
- `make NP=2 EXTRA_ARGS="-run test -da_grid_x 10 -da_grid_y 10 -mat_view_draw -draw_pause -1" runbratu`
- `make NP=2 EXTRA_ARGS="-run test -dim 3 -da_grid_x 5 -da_grid_y 5 -da_grid_z 5 -mat_view_draw -draw_pause -1" runbratu`

# Mesh Operators

- We evaluate the local portion just as with functions
- Notice we use $J^{-1}$ to convert derivatives
- Currently `updateOperator()` uses `MatSetValues()`
  - We need to call `MatAssembleyBegin/End()`
  - We should properly have `OperatorComplete()`
  - Also requires a Section, for layout, and a global variable order for PETSc index conversion
- `make EXTRA_ARGS="-run test -structured 0 -mat_view_draw -draw_pause -1 -generate" runbratu`
- `make NP=2 EXTRA_ARGS="-run test -structured 0 -mat_view_draw -draw_pause -1 -generate -refinement_limit 0.03125" runbratu`
- `make EXTRA_ARGS="-run test -dim 3 -structured 0 -mat_view_draw -draw_pause -1 -generate" runbratu`

# Code Update

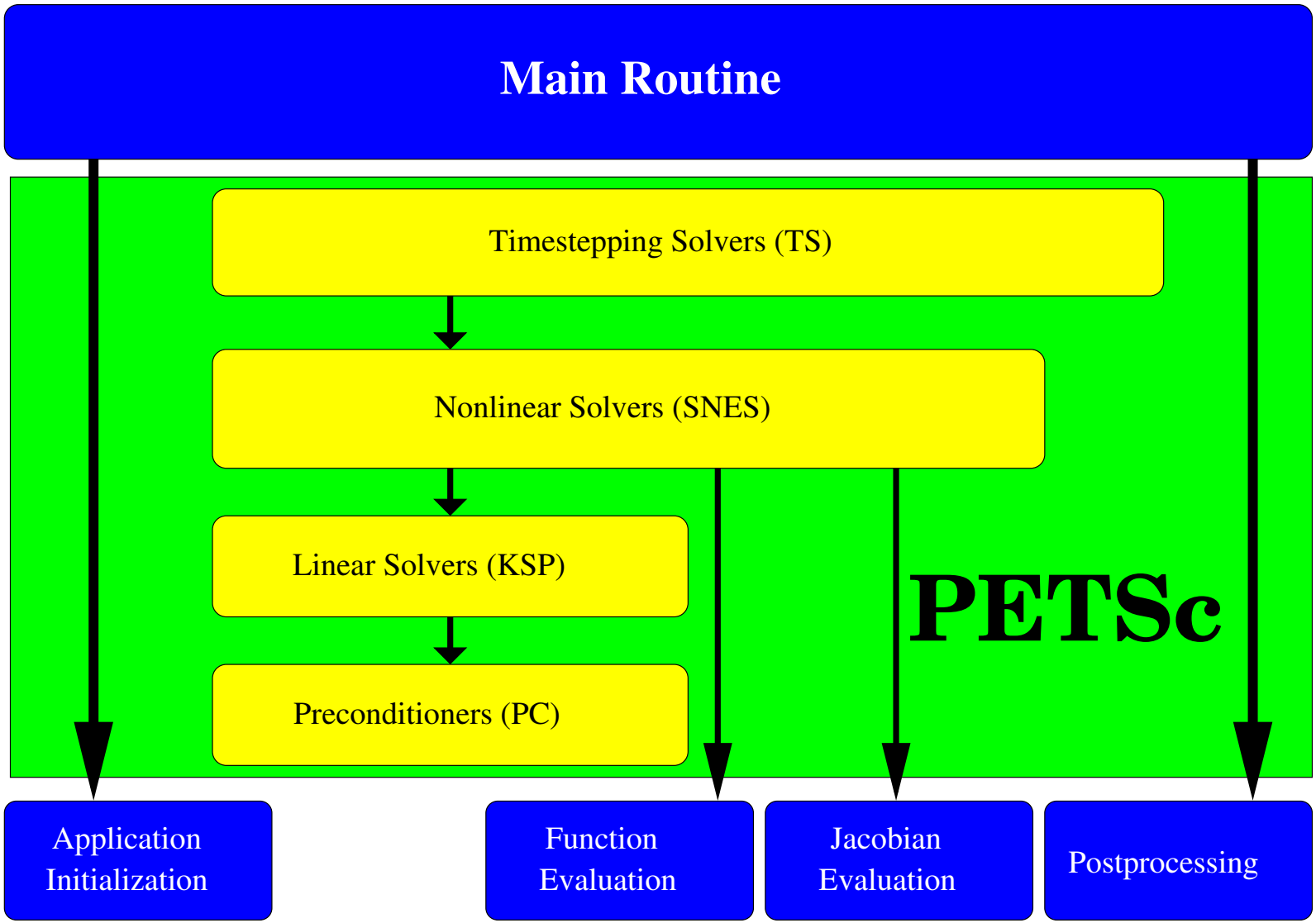
Update to Revision 14

# Operator Assembly

- `Full` assembly
  - Aggregate along all element interfaces
  - Global sparse matrix
- `Stored` assembly
  - No aggregation along element interfaces
  - Store element matrices
  - Use a `MATSHELL` in the solve
- `Calculated` assembly
  - No aggregation along element interfaces
  - Calculate element matrices on the fly
  - Use a `MATSHELL` in the solve
- Other alternatives...
  - Aggregation along some element interfaces (local?)
  - Custom overlaps
  - Partial calculation

# Outline

# Flow Control for a PETSc Application



# 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()`

# Code Update

## Update to Revision 15

# DMMG Integration with SNES

- DMMG supplies global residual and Jacobian to SNES
  - User supplies local version to DMMG
  - The `Rhs_*()` and `Jac_*()` functions in the example
- Allows automatic parallelism
- Allows grid hierarchy
  - Enables multigrid once interpolation/restriction is defined
- Paradigm is developed in unstructured work
  - Notice we have to scatter into contiguous global vectors (initial guess)
- Handle Neumann BC using `DMMGSetNullSpace()`

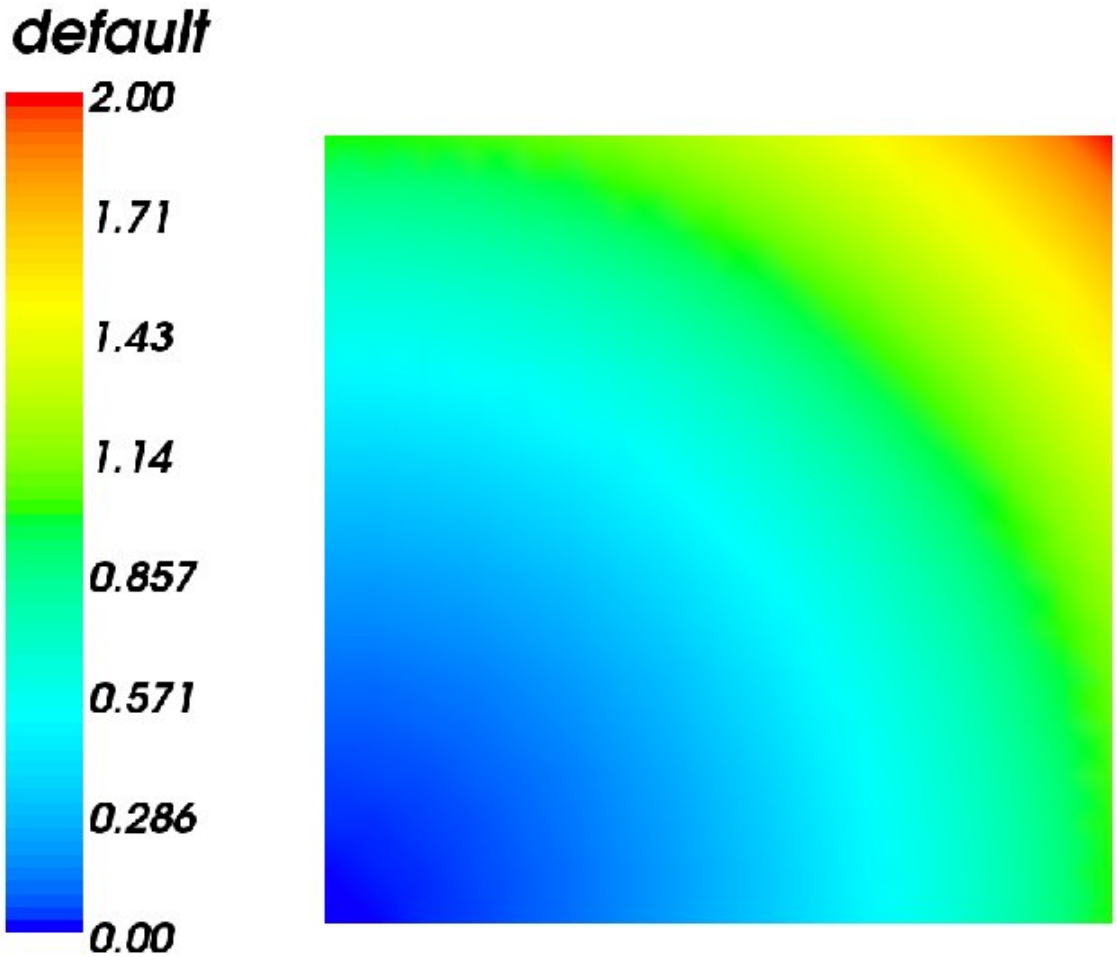
# DM Interface

- Allocation and layout
  - `createglobalvector(DM, Vec *)`
  - `createlocalvector(DM, Vec *)`
  - `getmatrix(DM, MatType, Mat *)`
- Intergrid transfer
  - `getinterpolation(DM, DM, Mat *, Vec *)`
  - `getaggregates(DM, DM, Mat *)`
  - `getinjection(DM, DM, VecScatter *)`
- Grid creation
  - `refine(DM, MPI_Comm, DM *)`
  - `coarsen(DM, MPI_Comm, DM *)`
  - `refinehierarchy(DM, PetscInt, DM **)`
  - `coarsenhierarchy(DM, PetscInt, DM **)`
- Mapping (completion)
  - `globaltolocalbegin/end(DM, Vec, InsertMode, Vec)`
  - `localtoglobal(DM, Vec, InsertMode, Vec)`

# Solving the Dirichlet Problem: $P_1$

- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -vec_view_vtk" runbratu`
- `make EXTRA_ARGS="-dim 3 -structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -vec_view_vtk" runbratu`
- The linear basis cannot represent the quadratic solution exactly
- `make EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.00125 -ksp_monitor -snes_monitor -vec_view_vtk -ksp_rtol 1.0e-9" runbratu`
- The error decreases with $h$
- `make NP=2 EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.00125 -ksp_monitor -snes_monitor -vec_view_vtk -ksp_rtol 1.0e-9" runbratu`
- `make EXTRA_ARGS="-dim 3 -structured 0 -generate -refinement_limit 0.00125 -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -vec_view_vtk" runbratu`
- Notice that the preconditioner is weaker in parallel

# Solving the Dirichlet Problem: $P_1$



# Solving the Dirichlet Problem: $P_2$

- `rm quadrature.h; make ORDER=2`
- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9" runbratu`
- `make EXTRA_ARGS="-dim 3 -structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9" runbratu`
- Here we get the exact solution
- `make EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.00125 -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9" runbratu`
- Notice that the solution is only as accurate as the KSP tolerance
- `make NP=2 EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.00125 -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9" runbratu`
- `make EXTRA_ARGS="-dim 3 -structured 0 -generate -refinement_limit 0.00125 -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9" runbratu`
- Again the preconditioner is weaker in parallel
- Currently we have no system for visualizing higher order solutions

# Alternative Assembly

- `make EXTRA_ARGS="-structured 0 -generate -ksp_monitor -snes_monitor -ksp_rtol 1.0e-9 -assembly_type full -pc_type none" runbratu`
- Since we cannot precondition without a matrix, we turn it off for comparison
- `make EXTRA_ARGS="-structured 0 -generate -ksp_monitor -snes_monitor -ksp_rtol 1.0e-9 -assembly_type stored -pc_type none" runbratu`
- Here we store all the element matrices
- `make EXTRA_ARGS="-structured 0 -generate -ksp_monitor -snes_monitor -ksp_rtol 1.0e-9 -assembly_type calculated -pc_type none" runbratu`
- This reduces storage, but increases computation

# Solving the Dirichlet Problem: FD

- `make EXTRA_ARGS="-snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -vec_view_draw -draw_pause -1" runbratu`
- Notice that we converge at the vertices, despite the quadratic solution
- `make EXTRA_ARGS="-snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -da_grid_x 40 -da_grid_y 40 -vec_view_draw -draw_pause -1" runbratu`
- `make NP=2 EXTRA_ARGS="-snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -da_grid_x 40 -da_grid_y 40 -vec_view_draw -draw_pause -1" runbratu`
- Again the preconditioner is weaker in parallel
- `make NP=2 EXTRA_ARGS="-dim 3 -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -da_grid_x 10 -da_grid_y 10 -da_grid_z 10" runbratu`

# Solving the Neumann Problem: $P_1$

- `make EXTRA_ARGS="-structured 0 -generate -bc_type neumann -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -vec_view_vtk" runbratu`
- `make EXTRA_ARGS="-dim 3 -structured 0 -generate -bc_type neumann -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -vec_view_vtk" runbratu`
- `make EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.00125 -bc_type neumann -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -vec_view_vtk" runbratu`
- The error decreases with $h$
- `make NP=2 EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.00125 -bc_type neumann -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -vec_view_vtk" runbratu`

# Solving the Neumann Problem: $P_3$

- `rm bratu_quadrature.h; make ORDER=3`
- `make EXTRA_ARGS="-structured 0 -generate -bc_type neumann -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9" runbratu`
- Here we get the exact solution
- `make EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.00125 -bc_type neumann -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9" runbratu`
- `make NP=2 EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.00125 -bc_type neumann -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9" runbratu`

# The Louisville-Bratu-Gelfand Problem

$$-\Delta u - \lambda e^u = f \tag{7}$$

- Simplification of the Solid-Fuel Ignition Problem
- Also a nonlinear eigenproblem
- Exhibits a bifurcation at $\lambda \approx 6.8$
- We will use Dirichlet conditions

# Nonlinear Equations

We will have to alter

- The residual calculation, `Rhs_*()`
- The Jacobian calculation, `Jac_*()`
- The forcing function to match our chosen solution, `CreateProblem()`

# Code Update

## Update to Revision 16

# Solving the Bratu Problem: FD

- `make EXTRA_ARGS="-snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -vec_view_draw -draw_pause -1 -lambda 0.4" runbratu`
- Notice that we converge at the vertices, despite the quadratic solution
- `make NP=2 EXTRA_ARGS="-da_grid_x 40 -da_grid_y 40 -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -vec_view_draw -draw_pause -1 -lambda 6.8" runbratu`
- Notice the problem is more nonlinear near the bifurcation
- `make NP=2 EXTRA_ARGS="-dim 3 -da_grid_x 10 -da_grid_y 10 -da_grid_z 10 -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 6.8" runbratu`

# Finding Problems

We switch to quadratic elements so that our FE solution will be exact

- `rm bratu_quadrature.h; make ORDER=2`
- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 0.4" runbratu`

# Finding Problems

We switch to quadratic elements so that our FE solution will be exact

- `rm bratu_quadrature.h; make ORDER=2`
- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 0.4" runbratu`

We do not converge!

# Finding Problems

We switch to quadratic elements so that our FE solution will be exact

- `rm bratu_quadrature.h; make ORDER=2`
- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 0.4" runbratu`

We do not converge!

- Residual is zero, so the Jacobian could be wrong (try FD)
- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 0.4 -snes_fd" runbratu`

# Finding Problems

We switch to quadratic elements so that our FE solution will be exact

- `rm bratu_quadrature.h; make ORDER=2`
- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 0.4" runbratu`

We do not converge!

- Residual is zero, so the Jacobian could be wrong (try FD)
- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 0.4 -snes_fd" runbratu`

It works!

# Finding Problems

We switch to quadratic elements so that our FE solution will be exact

- `rm bratu_quadrature.h; make ORDER=2`
- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 0.4" runbratu`

We do not converge!

- Residual is zero, so the Jacobian could be wrong (try FD)
- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 0.4 -snes_fd" runbratu`

It works!

- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 0.4 -snes_max_it 3 -mat_view" runbratu`
- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 0.4 -snes_fd -mat_view" runbratu`
- Entries are too big, we forgot to initialize the matrix

# Code Update

## Update to Revision 17

# Solving the Bratu Problem: $P_2$

- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 0.4" runbratu`
- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 6.8" runbratu`
- `make NP=2 EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.00125 -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 6.8" runbratu`
- `make EXTRA_ARGS="-dim 3 -structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 6.8" runbratu`
- `make EXTRA_ARGS="-dim 3 -structured 0 -generate -refinement_limit 0.00125 -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 6.8" runbratu`

# Solving the Bratu Problem: $P_1$

- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 0.4" runbratu`
- `make EXTRA_ARGS="-structured 0 -generate -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 6.8" runbratu`
- `make NP=2 EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.00125 -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 6.8" runbratu`
- `make EXTRA_ARGS="-dim 3 -structured 0 -generate -refinement_limit 0.01 -snes_monitor -ksp_monitor -ksp_rtol 1.0e-9 -lambda 6.8" runbratu`

# Outline

# What Is Optimal?

I will define *optimal* as an $\mathcal{O}(N)$ solution algorithm

These are generally hierarchical, so we need

- hierarchy generation
- assembly on subdomains
- restriction and prolongation

# Multigrid

Multigrid is *optimal* in that is does $\mathcal{O}(N)$ work for $||r|| < \epsilon$

- Brandt, Briggs, Chan & Smith
- Constant work per level
  - Sufficiently strong solver
  - Need a constant factor decrease in the residual
- Constant factor decrease in dof
  - Log number of levels

# Structured Meshes

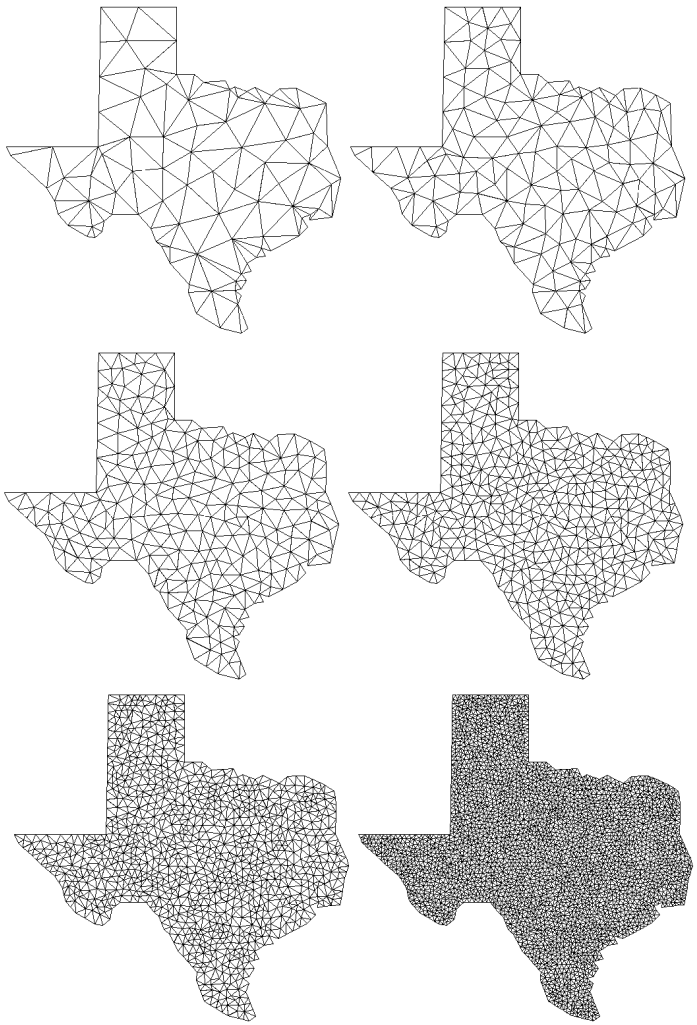
The DMMG allows multigrid which some simple options

- `-dmmg_nlevels`, `-dmmg_view`
- `-pc_mg_type`, `-pc_mg_cycle_type`
- `-mg_levels_1_ksp_type`, `-dmmg_levels_1_pc_type`
- `-mg_coarse_ksp_type`, `-mg_coarse_pc_type`

# Solving with Structured Multigrid

- `make EXTRA_ARGS="-dmmg_nlevels 2 -dmmg_view -snes_monitor -ksp_monitor -ksp_rtol 1e-9" runbratu`
- Notice that the solver on each level can be customized
- number of KSP iterations is approximately constant
- `make EXTRA_ARGS="-da_grid_x 10 -da_grid_y 10 -dmmg_nlevels 8 -dmmg_view -snes_monitor -ksp_monitor -ksp_rtol 1e-9" runbratu`
  - Notice that there are over 1 million unknowns!
- Coarsening is not currently implemented

# Coarsening



- Users want to control the mesh
- Developed efficient, topological coarsening
  - Miller, Talmor, Teng algorithm
- Provably well-shaped hierarchy

# Mesh Coarsening

- Easy in structured case, but unstructured is more subtle
- Delaunay coarsening is popular
  - $M_{coarse}$ is a nonadjacent vertex subset of $M_{fine}$
  - Reduces to maximal independent set over edges
  - Enforces a spacing increase for well-shaped meshes
  - Mesh degradation from repeated coarsenings

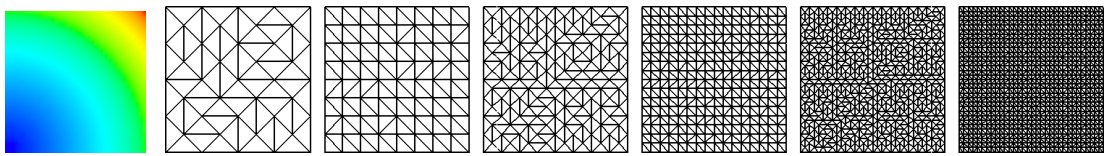
# Function-Based Coarsening

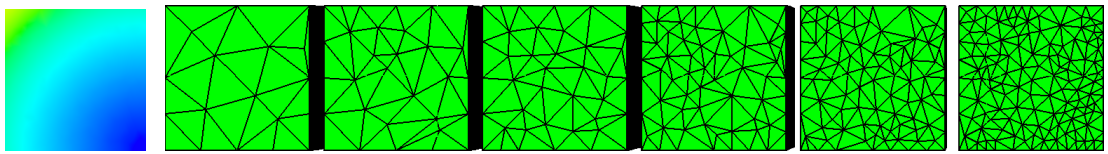
**G. Miller, D. Talmor, S.-H. Teng**, *Optimal Coarsening of Unstructured Meshes*, J. Algorithms, 31 (1999), pp. 29-65

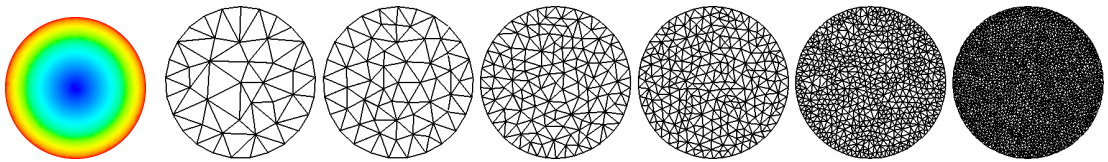
- Vertex *spacing function*
  - For example, nearest neighbor distance
- Expand the spacing function by some factor $C$
- Prune the mesh until expanded function is satisfied
  - Remove nodes until spheres of diameter $C * dist_{NN}$ are disjoint
- Guaranteed vertex spacing and cell shape
- Works in any dimension

# Convex Domains

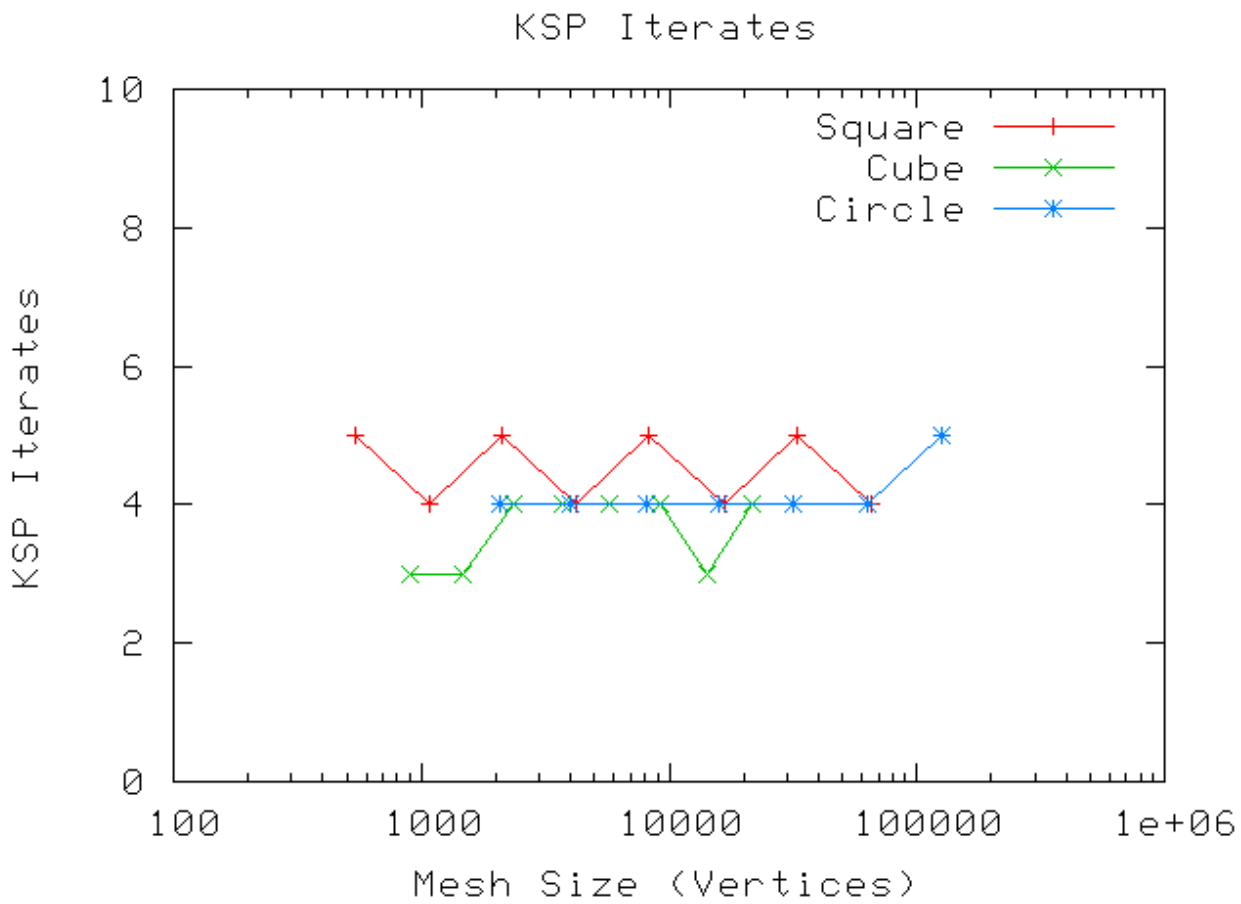
- $\Omega_{square} = [0,1] \times [0,1] (\times [0,1])$
- $\Omega_{circle} = \{p(x,y) : x^2 + y^2 <= 1\}$
- $\Delta u = f$
- $f(x,y) = -4$
- Exact Solution: $u(x,y) = x^2 + y^2$



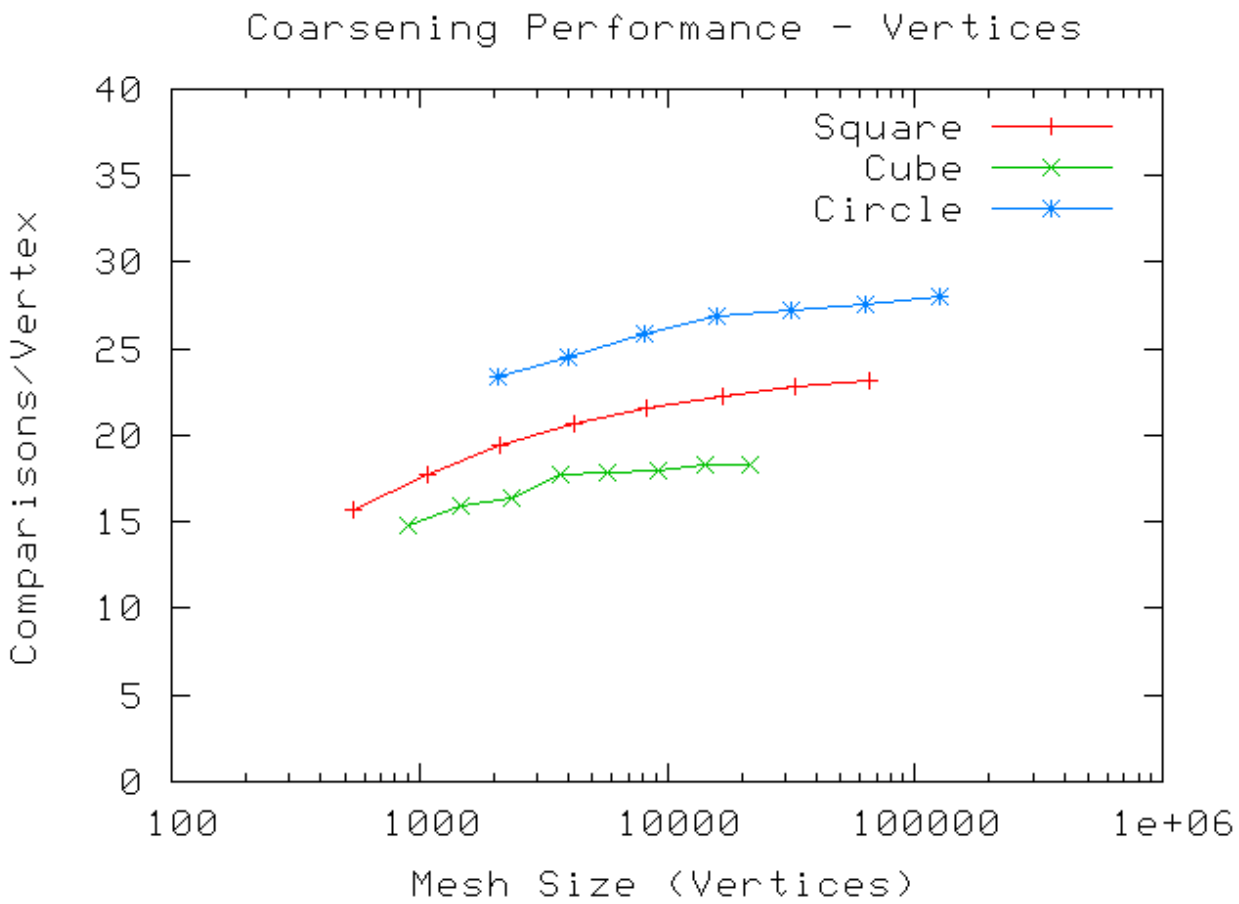




# KSP Performance

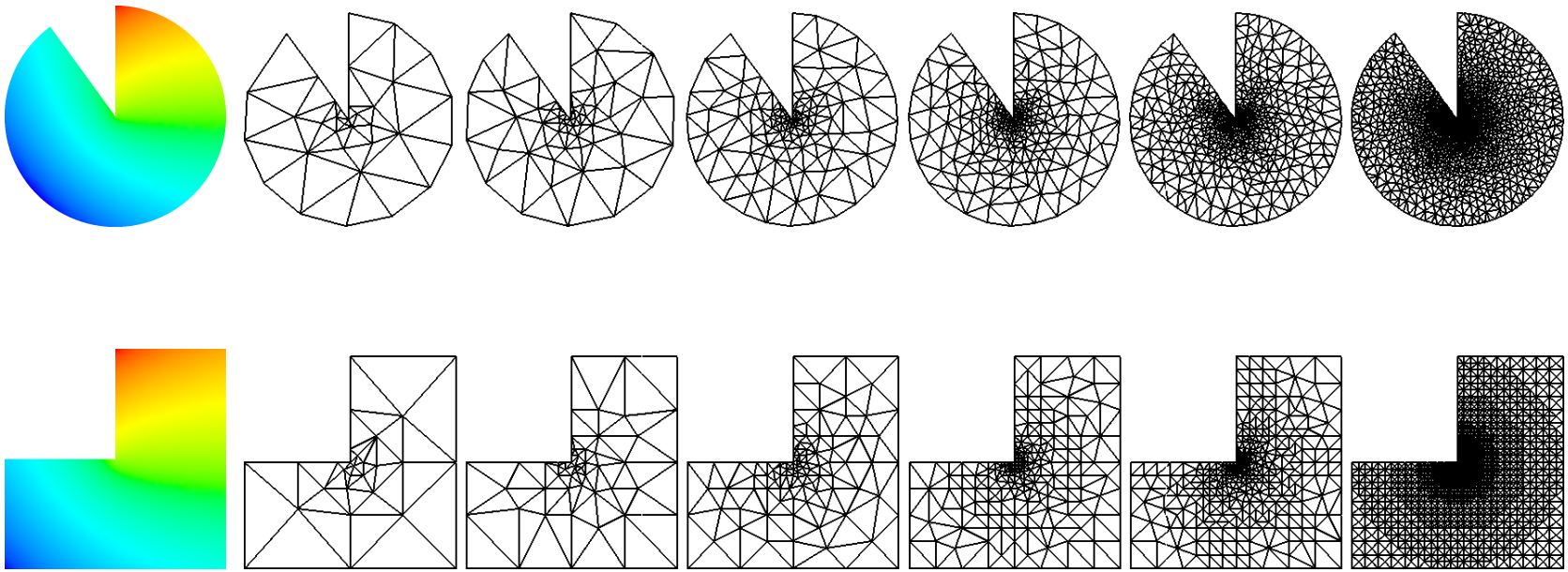


# Coarsening Performance

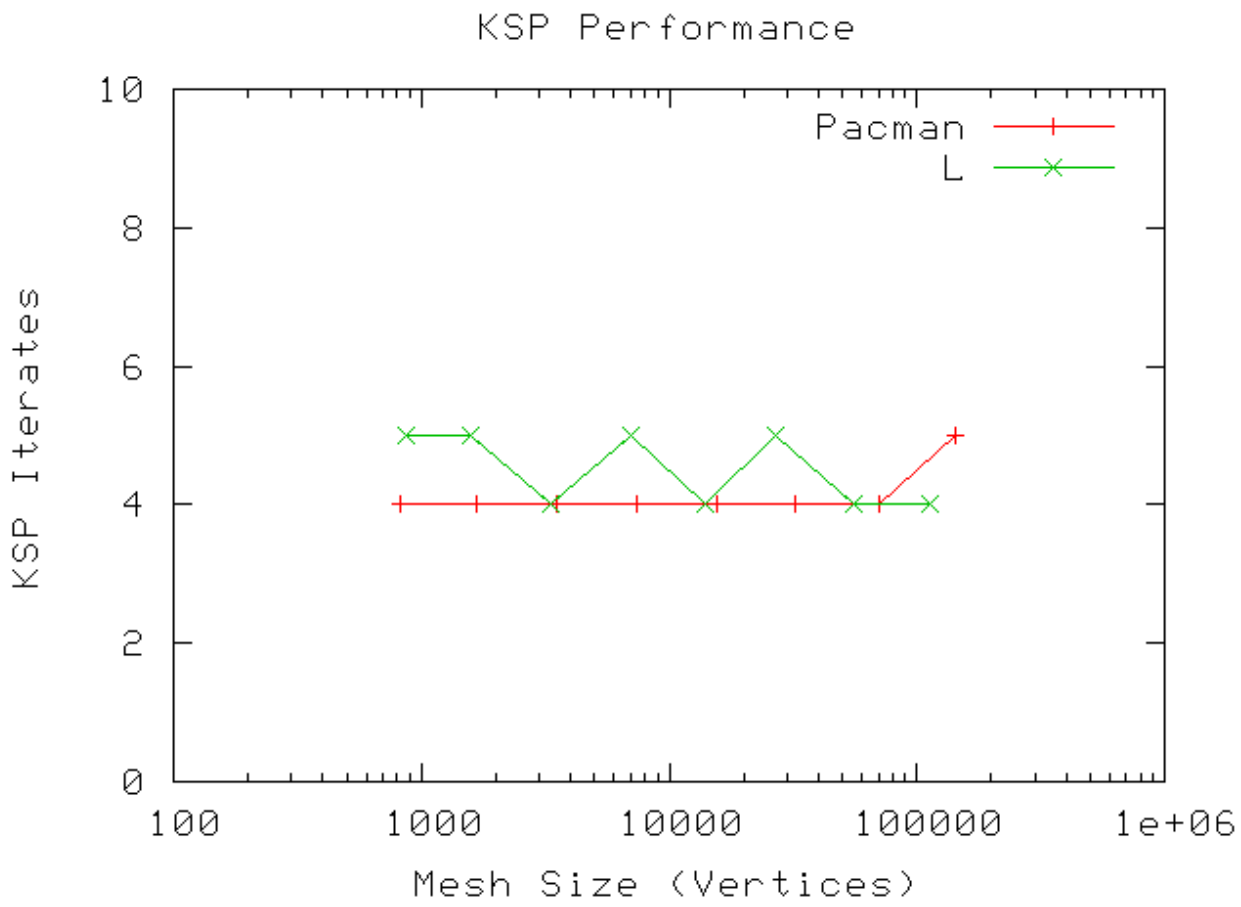


# Domains with Reentrant Corners and Refinement

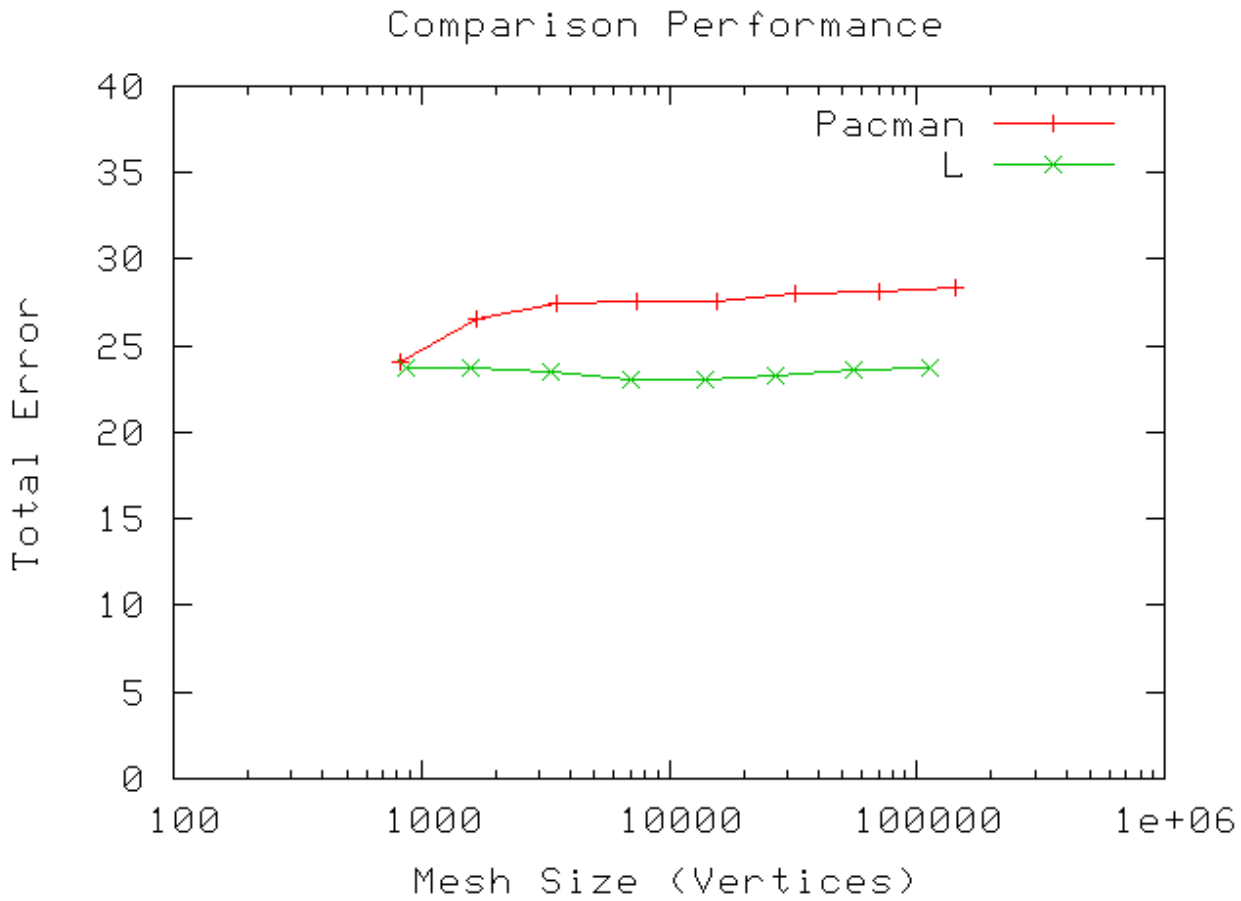
- $\Omega_{pacman} = \{p(x,y) \rightarrow p(r,\theta) : [0,1] \times [0, .9 * 2\pi]\}$
- $\Omega_L = [0,1] \times [-1,1] \setminus [-1,0] \times [-1,0]$
- $\Delta u = f$
- $f(x,y) = 0$
- Exact Solution: $u(x,y) = r^{\frac{2}{3}} sin(\frac{2}{3}\theta)$



# KSP Performance



# Coarsening Performance



# Unstructured Meshes

- Same `DMMG` options as the structured case
- Mesh refinement
  - Ruppert algorithm in Triangle and TetGen
- Mesh coarsening
  - Talmor-Miller algorithm in PETSc
- More advanced options
  - `-dmmg_refine`
  - `-dmmg_hierarchy`
- Current version only works for linear elements

# Solving with Unstructured Multigrid

- `make EXTRA_ARGS="-structured 0 -generate -bc_type neumann -dmmg_nlevels 2 -dmmg_view -snes_monitor -ksp_monitor -ksp_rtol 1e-9 -vec_view" runbratu`
- Compare to explicitly refined solution
- `make EXTRA_ARGS="-structured 0 -generate -bc_type neumann -snes_monitor -ksp_monitor -ksp_rtol 1e-9 -refinement_limit 0.0625 -vec_view" runbratu`
- We would really like to coarsen an existing mesh
- `make EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.03125 -bc_type neumann -dmmg_nlevels 3 -dmmg_refine 0 -dmmg_hierarchy -dmmg_view -snes_monitor -ksp_monitor -ksp_rtol 1e-9 -vec_view" runbratu`
- `make EXTRA_ARGS="-structured 0 -generate -refinement_limit 0.03125 -bc_type neumann -snes_monitor -ksp_monitor -ksp_rtol 1e-9 -vec_view" runbratu`
- Notice that here we refine both meshes to the same level

# Outline

# What We Have Not Covered

- Unstructured hexes
  - Structured hex FEM
- *a posteriori* Error Estimation
- Exotic elements
- Semi-Lagrangian Schemes

# What We Have Not Focused On

- Linear and Nonlinear Solvers
  - MANY other PETSc tutorials on this
- Unstructured mesh framework
  - Several preprints on Sieve architecture
- Structure of multilevel methods
  - Barry's talk from SIAM PP 2006
- Preconditioning
  - Very problem dependent (best left to applications?)
- Scalability and Performance
  - Coming soon...

# 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

# 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.