Domain Decomposition with PETSc

William Gropp and David Keyes

Argonne National Laboratory

and

Columbia University

Introduction

- What and why is PETSc?
  - PETSc is a portable library for solving linear and nonlinear systems of equations in parallel
  - PETSc was originally designed to provide a library for experimentation in domain decomposition algorithms
- What is Domain Decomposition?
  - DD is a algorithmic technique for dividing problems into subproblems and combining the results to solve (or approximate) the solution
  - DD is a natural method for effective parallel algorithms for distributed memory computers

PETSc Team

Satish Balay

Bill Gropp

Lois Curfman McInnes

Kris Buschelman

Dinesh Kaushik

Barry Smith

Victor Eijkhout

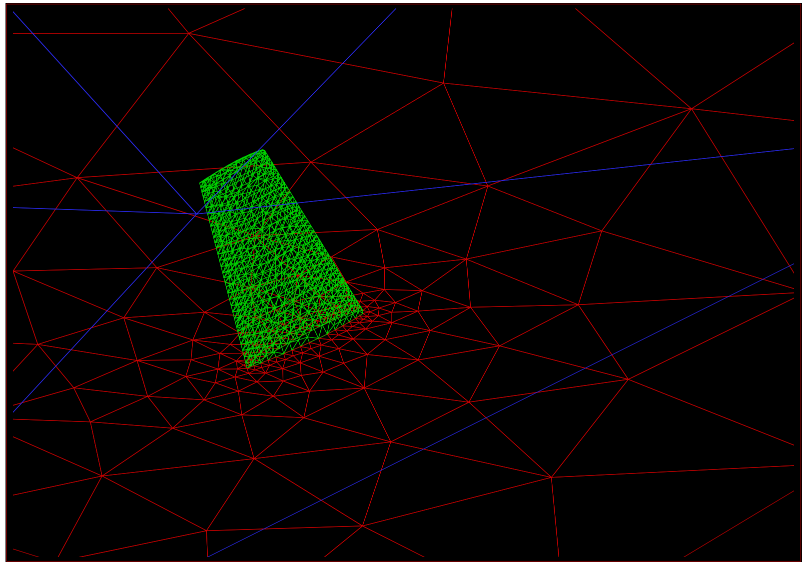
Matt Knepley

Hong Zhang

Plus many users and contributors

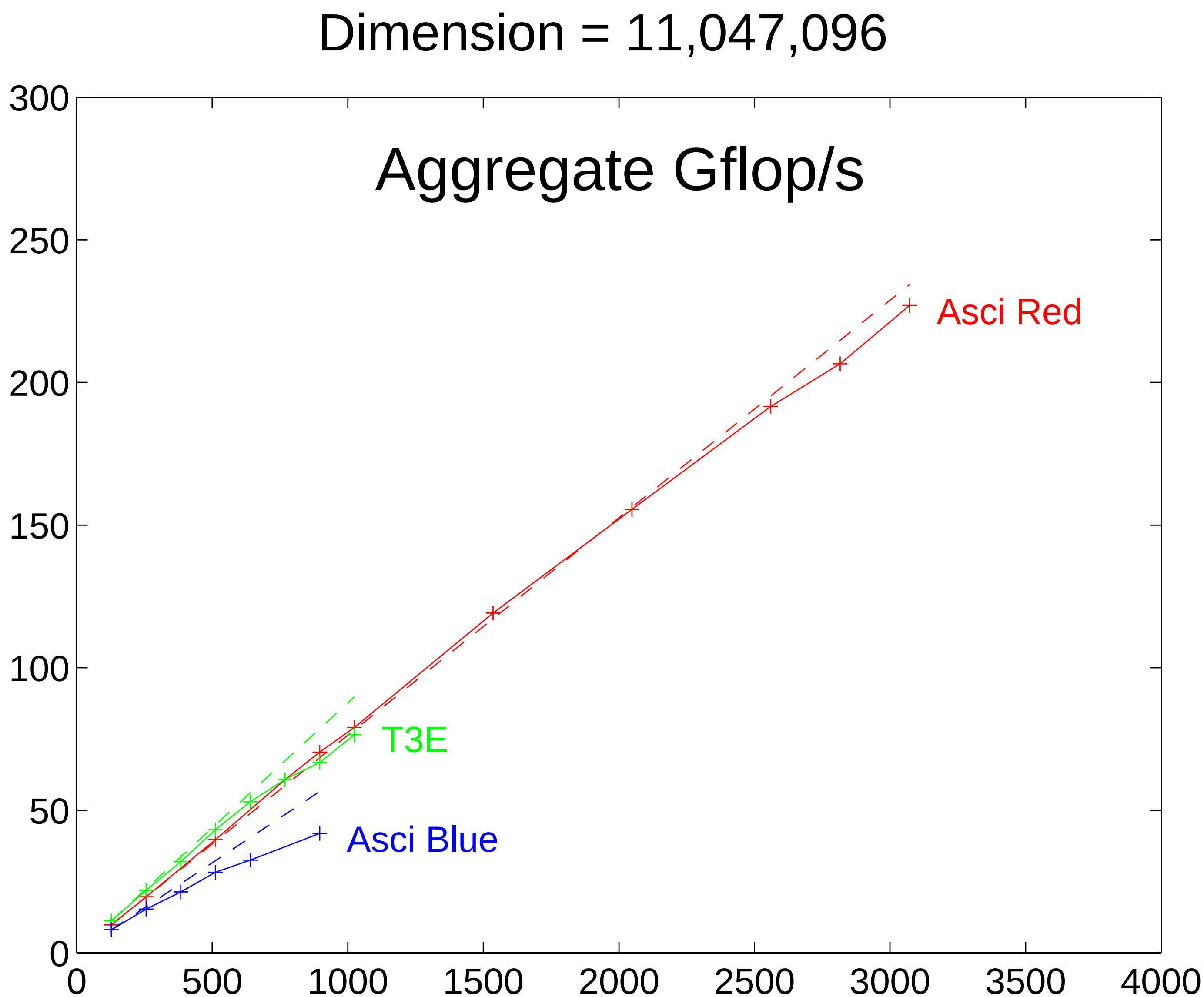
PETSc at Scale

- FUN3d, a legacy Fortran application, was parallelized using PETSc
  - 3D incompressible Euler
  - Tetrahedral grid
  - Up to 11 million unknowns
  - Based on a legacy NASA code, FUN3d, developed by W. K. Anderson



  - Fully implicit steady-state
  - Primary PETSc tools: nonlinear solvers (SNES) and vector scatters (VecScatter)

Performance of Fun3D/PETSc



Tutorial Overview

- Introduction to PETSc—Hello World
- Building a Poisson Solver in PETSc
  - Using distributed arrays to describe data parallelism
  - Using domain decomposition methods in PETSc
- Solving Nonlinear problems
  - Algorithms for nonlinear problems
  - Bratu example
  - More on distributed arrays in PETSc
- Time dependent problems
- Applications
  - Driven cavity example
- Wrapup

A Few Comments Before We Start

- PETSc is a very large library
  - This tutorial is designed to introduce PETSc without overwhelming you with information
  - Many features will not be covered. PETSc comes with extensive examples and documentation
- PETSc is a freely available and supported research code
  - Available via `http://www.mcs.anl.gov/petsc`
  - Free for everyone, including industrial users
  - Hyperlinked documentation and manual pages for all routines
  - Many tutorial-style examples
  - Support via email: `petsc-maint@mcs.anl.gov`
  - Usable from Fortran 77/90, C, and C++

- Portable to any parallel system supporting MPI, including
  - Tightly coupled systems
    Cray T3E, SGI Origin, IBM SP, HP 9000, Sun Enterprise
  - Loosely coupled systems, e.g., networks of workstations
    HP (including Compaq/DEC), IBM, SGI, Sun and PCs running
    Linux or Windows
- What is not in PETSc
  - Discretizations
  - Unstructured mesh generation or refinement
  - Load balancing tools
  - Sophisticated visualization support
  - (But PETSc provides ways to interface to other tools)

Prerequisites

This tutorial assumes that you have at least a basic background in

- Finite difference methods for PDEs
- Iterative methods for solving linear systems

In addition

- Familiarity with MPI (the Message Passing Interface) is helpful but not required.

A First PETSc Program

- What do PETSc programs look like?
- What do PETSc parallel programs look like?
- How to compile, link, and run PETSc programs?

Hello World

```
#include "petsc.h"

int main( int argc, char *argv[] )
{
    PetscInitialize( &argc, &argv, 0, 0 );

    PetscPrintf( PETSC_COMM_WORLD, "Hello World\n" );
    PetscFinalize( );
    return 0;
}
```

Understanding the Code

PetscInitialize Initialize PETSc. The arguments allow PETSc to initialize MPI if necessary

PetscFinalize Finalize PETSc. Causes PETSc to call `MPI_Finalize` if necessary and also to generate summary reports.

PetscPrintf Ensures that only one process prints the data (Try it!)

Hello World in Fortran

```
      integer ierr, rank
#include "include/finclude/petsc.h"
      call PetscInitialize( PETSC_NULL_CHARACTER, ierr )
      call MPI_Comm_rank( PETSC_COMM_WORLD, rank, ierr )
      if (rank .eq. 0) then
          print *, 'Hello World'
      endif
      call PetscFinalize(ierr)
      end
```

Understanding the Code

- Like the C code, except
  - PetscInitialize has fewer arguments because Fortran has no `argc` or `argv`
  - Must use `MPI_Comm_rank` and `print` because Fortran I/O uses a interface unavailable to libraries
- PETSc 2.1.6 adds a routine that can be used with a single character string (Fortran can't implement its own I/O operations, so PETSc can't provide parallel replacements)

How To Compile, Link, and Run

- PETSc make use of three environment variables. Two specify the location of PETSc and the particular machine architecture:

  PETSC_DIR The location of PETSc

  PETSC_ARCH The name of the machine architecture. In some cases, the script `$PETSC_DIR/bin/petscarch` can be used to get the value that should be used for this environment variable

- The third specifies the level of optimization to use.

  BOPT One of g, O, or Opg; these indicate the level of optimization and debugging support within the PETSc library. Usually set on make line:

  ```
  make BOPT=g hello
  ```

- Use PETSc makefiles to ensure that all of the necessary libraries and compiler options are used. The makefiles in the various example directories are good starting points
  - Alternately, just include the PETSc variables and write your own Makefile

A Sample Makefile

```
SHELL = /bin/bash
PETSC_DIR = c:/programs/petsc-2.1.5
PETSC_ARCH = win32_gnu
BOPT ?= g
NP   ?= 4
PGM  ?= hello
include $(PETSC_DIR)/bmake/common/base
EXECS = hello
all-redirect: $(EXECS) $(OBJS)

hello: hello.o chkopts
        $(CLINKER) -o hello hello.o $(PETSC_LIB)

run:
        $(MPIRUN) -np $(NP) $(PGM) $(ARGS)

clean-local:
        -rm -f $(EXECS) *.o
```

Using PETSc at This Tutorial

Use

PETSC_DIR = /usr/bin/petsc

PETSC_ARCH = linux

To run programs, make sure that your `PATH` includes mpirun.

Use `mpirun` to run programs:

```
mpirun -np 4 ./hello
```

Single process runs do not need `mpirun`:

```
./hello
```

A Parallel Program

- PETSc uses the distributed memory, shared-nothing model
- Parallel PETSc programs consist of separate communicating processes
- PETSc uses MPI for parallelism
  - You can always access MPI routines
  - You will rarely need to use MPI while using PETSc
  - Many PETSc routines are *collective* in the MPI sense (all processes must call); others are local.
  - Common uses of MPI in PETSc are the routines for communicator size and rank and for processor name.
  - This is illustrated in a revised (and obviously parallel) hello world program.

Hello World Revisited

```
#include "petsc.h"

int main( int argc, char *argv[] )
{
    int rank;
    PetscInitialize( &argc, &argv, 0, 0 );

    MPI_Comm_rank( PETSC_COMM_WORLD, &rank );
    PetscSynchronizedPrintf( PETSC_COMM_WORLD,
                             "Hello World from rank %d\n", rank );
    PetscSynchronizedFlush( PETSC_COMM_WORLD );
    PetscFinalize( );
    return 0;
}
```

Understanding the Program

PetscSynchronizedPrintf Like `PetscPrintf`, except output comes from all processes in rank order.

PetscSynchronizedFlush Indicates that the calling process is done printing.

- Allows the use of multiple PetscSynchronizedPrintf calls

PETSC_COMM_WORLD The PETSc version of MPI_COMM_WORLD, they are usually the same set of processes. PetscSetCommWorld, used *before* PetscInitialize, may be used to give PETSc a subset of processes

PETSc and PDEs

- PETSc is designed around the mathematics of the problem
  - Specify the data in terms of vectors
  - Specify the problem as linear (using matrices) or nonlinear (using vector-valued functions) equations to be solved
  - Support parallel computing by automatically distributing these objects across all processes
- We'll see a sequence of increasingly sophisticated PDE examples. . .

Poisson Problem

Lets solve a simple linear elliptic PDE

$$\begin{aligned} \nabla^2 u &= f \text{ in } [0,1] \times [0,1] \\ u &= 0 \text{ on the boundary} \end{aligned}$$

using a simple discretization ($u_{i,j} = u(x_i, y_j)$, $x_i = ih$)

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} = f(x_i, y_j).$$

(We use finite differences for simplicity; finite elements can be used as well.) For simplicity, consider $f = sin(\pi x) sin(\pi y)$.
We will discretize the interior of the mesh only for this example.

Schematic for Example

In PETSc, your main program remains in control:

main program
  PetscInitialize()
  A = create the matrix
  b = create a vector
  Use SLES to solve A x = b
  print solution
  PetscFinalize()

SLES is the “simplifi ed linear equation solver” component of PETSc

Creating the Matrix

```
#include "petscsles.h"

/* Form the matrix for the 5-point finite difference 2d Laplacian
   on the unit square. n is the number of interior points along a side */
Mat FormLaplacian2d( int n )
{
    Mat    A;
    int    r, rowStart, rowEnd, i, j;
    double h, oneByh2;

    h = 1.0 / (n + 1);      oneByh2 = 1.0 / (h * h);
    MatCreate( PETSC_COMM_WORLD, PETSC_DECIDE, PETSC_DECIDE,
               n*n, n*n, &A );
    MatSetFromOptions( A );
    MatGetOwnershipRange( A, &rowStart, &rowEnd );
```

Creating the Matrix II

```
    /* This is a simple but inefficient way to set the matrix */
    for (r=rowStart; r<rowEnd; r++) {
        i = r % n;  j = r / n;
        if (j - 1 > 0) {
            MatSetValue( A, r, r - n, oneByh2, INSERT_VALUES ); }
        if (i - 1 > 0) {
            MatSetValue( A, r, r - 1, oneByh2, INSERT_VALUES ); }
        MatSetValue( A, r, r, -4*oneByh2, INSERT_VALUES );
        if (i + 1 < n - 1) {
            MatSetValue( A, r, r + 1, oneByh2, INSERT_VALUES ); }
        if (j + 1 < n - 1) {
            MatSetValue( A, r, r + n, oneByh2, INSERT_VALUES ); }
    }
    MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
    return A;
}
```

Understanding the Code I

MatCreate Create a matrix object.

- n^2 equations, so matrix is of size **n*n**$\times$**n*n**
- PETSC_DECIDE tells PETSc to choose the distribution of the matrix across the processes

MatSetFromOptions Set basic matrix properties (such as data structure) from command line

MatGetOwnershipRange Get the rows of the matrix that PETSc assigned to this process

- PETSc uses a simple assignment of consecutive rows to a process. This simplifies much of the internal structure of PETSc, and, as we shall see, does not reduce the generality
- It is not necessary to set values on the “owning” process
- Returns first row to one $+$ last row on process.
  - Matches common C idiom (`for (i=start; i<end; i++)`)
  - Number of rows is `end-start`

Understanding the Code II

MatSetValue Insert (or optionally add with ADD_VALUES) a value to a matrix (*Warning:* This is a macro and needs braces)

MatAssemblyBegin and MatAssemblyEnd Complete the creation of matrix. The matrix may not be used for any operation (other than MatSetValue) until after MatAssemblyEnd.

The approach of separating setting values from assembly has several benefits

- Any process may set a value to *any* element of the matrix, even ones not “owned” by the calling process.
- PETSc manages all data communication between processes
- PETSc can optimize the insertion of matrix elements

Data Structure Neutral Design

- PETSc matrices are *objects for storing linear operators*
- They allow many types of data structures:
  - Default sparse format MATMPIAIJ and MATSEQAIJ
  - Block sparse MATMPIBAIJ and MATSEQBAIJ
  - Symmetric block sparse MATMPISBAIJ and MATSEQSBAIJ
  - Block diagonal MATMPIBDIAG and MATSEQBDIAG
  - Dense MATMPIDENSE and MATSEQDENSE
  - Many others (see $PETSC_DIR/include/petscmat.h)
- Choice of format is made from command line (with MatSetFromOptions) or program (with MatSetType)
- The *same* routines are used for all choices of data structure
- User-defined data-structures supported with “Shell” objects

Data Decomposition in PETSc

- How are objects distributed among processes in PETSc?
  - Continguous rows of a vector or matrix are assigned to processes, starting from the process with rank zero
- The matrix and vector for a 3×3 mesh, with two processes, has the following decomposition

$$
\begin{array}{c} \text{P0} \\ \hline \text{P1} \end{array}
\quad
\left(\begin{array}{c} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ \hline x_5 \\ x_6 \\ x_7 \\ x_8 \end{array}\right)
=
\left(\begin{array}{ccccccccc}
4 & -1 & & -1 & & & & & \\
-1 & 4 & -1 & & -1 & & & & \\
 & -1 & 4 & & & -1 & & & \\
-1 & & & 4 & -1 & & -1 & & \\
 & -1 & & -1 & 4 & -1 & & -1 & \\
\hline
 & & -1 & & -1 & 4 & & & -1 \\
 & & & -1 & & & 4 & -1 & \\
 & & & & -1 & & -1 & 4 & -1 \\
 & & & & & -1 & & -1 & 4
\end{array}\right)
$$

Why Are PETSc Matrices The Way They Are?

- No one data structure is appropriate for all problems
  - Blocked and diagonal formats provide significant performance benefits
  - PETSc provides a large selection of formats and makes it (relatively) easy to extend PETSc by adding new data structures
- Matrix assembly is difficult enough without being forced to worry about data partitioning
  - PETSc provide parallel assembly routines
  - Achieving high performance still requires making most operations local to a process, but this approach allows incremental development of programs
- Matrix decomposition by consecutive rows across processes is simple and makes it easier to work with other codes
  - For applications with other ordering needs, PETSc provides "Application Orderings" (AO)

Vectors In PETSc

- In order to support the distributed memory “shared nothing” model, as well as single processors and shared memory systems, a PETSc vector is a “handle” to the real vector
  - Allows the vector to be distributed across many processes
  - To access the elements of the vector, we cannot simply do

    ```
    for (i=0; i<n; i++) v[i] = i;
    ```
  - We do not want to require that the programmer work *only* with the “local” part of the vector; we want to permit operations, such as setting an element of a vector, to be performed by any process.
- The solution is to make vectors an object, just like a parallel matrix

Creating the Vectors I

```
#include "petscvec.h"

/* Form a vector based on a function for a 2-d regular mesh on the
   unit square */
Vec FormVecFromFunction2d( int n, double (*f)( double, double ) )
{
    Vec    V;
    int    r, rowStart, rowEnd, i, j;
    double h;

    h = 1.0 / (n + 1);
    VecCreate( PETSC_COMM_WORLD, &V );
    VecSetSizes( V, PETSC_DECIDE, n*n );
    VecSetFromOptions( V );
```

Creating the Vectors II

```
    VecGetOwnershipRange( V, &rowStart, &rowEnd );
    /* This is a simple but inefficient way to set the vector */
    for (r=rowStart; r<rowEnd; r++) {
        i = (r % n) + 1;
        j = (r / n) + 1;
        VecSetValue( V, r, (*f)( i * h, j * h ), INSERT_VALUES );
    }
    VecAssemblyBegin(V);
    VecAssemblyEnd(V);

    return V;
}

```

Understanding the Code

VecCreate Creates the vector. Unlike MatCreate, the size must be set separately

VecSetSizes Sets the global and local size of the vector. Use PETSC_DECIDE to have PETSc choose the distribution across processes

VecSetFromOptions Like the matrix counterpart. VecSetType may be used instead.

VecGetOwnershipRange Like the matrix counterpart

VecSetValue Sets the value for a vector element. Use ADD_VALUES to add to a vector element. Like the matrix routines, elements can be inserted or added by any process.

VecAssemblyBegin and VecAssemblyEnd Like the Matrix counterparts

Solving a Poisson Problem I

```
#include <math.h>
#include "petscsles.h"
extern Mat FormLaplacian2d( int );
extern Vec FormVecFromFunction2d( int, double (*)(double,double) );
/* This function is used to define the right-hand side of the
   Poisson equation to be solved */
double func( double x, double y ) {
    return sin(x*M_PI)*sin(y*M_PI); }

int main( int argc, char *argv[] )
{
    SLES        sles;
    Mat         A;
    Vec         b, x;
    int         its, n;

    PetscInitialize( &argc, &argv, 0, 0 );
```

Solving a Poisson Problem II

```
    n = 10;     /* Get the mesh size.  Use 10 by default */
    PetscOptionsGetInt( PETSC_NULL, "-n", &n, 0 );

    A = FormLaplacian2d( n );
    b = FormVecFromFunction2d( n, func );
    VecDuplicate( b, &x );
    SLESCreate( PETSC_COMM_WORLD, &sles );
    SLESSetOperators( sles, A, A, DIFFERENT_NONZERO_PATTERN );
    SLESSetFromOptions( sles );
    SLESSolve( sles, b, x, &its );

    PetscPrintf( PETSC_COMM_WORLD, "Solution in %d iterations is:\n", its
    VecView( x, PETSC_VIEWER_STDOUT_WORLD );

    MatDestroy( A ); VecDestroy( b ); VecDestroy( x );
    SLESDestroy( sles );
    PetscFinalize( );
    return 0;
```

Understanding the Code

`SLESCreate` Create a *context* used to to solve a linear system. This routine is used for *all* solvers, independent of the choice of algorithm or data structure

`SLESSetOperators` Define the problem.

- The third argument allows the use of a different matrix for preconditioning
- `DIFFERENT_NONZERO_PATTERN` indicates whether the preconditioner has the same nonzero pattern each time a system is solved. This default works with all preconditioners. Other values (e.g., `SAME_NONZERO_PATTERN`) can be used for particular preconditioners. Ignored when solving only one system

`SLESSetFromOptions` Set the algorithm, preconditioner, and the associated parameters, using the command-line

`SLESSolve` Actually solve the system of linear equations. The number of iterations is returned (a reflection of the bias towards iterative methods). If a direct method is used, one is returned in `its`

`SLESDestroy` Free the SLES context and all storage associated with it

Objects in PETSc

- How should a matrix be described in a program?
  - Old way:
    - Dense matrix

      ```
      double precision A(10,10)
      ```
    - Sparse matrix

      ```
      integer ia(11), ja(max_nz)
      double precision a(max_nz)
      ```
  - New way:

    ```
    Mat M
    ```
- Hides the choice of data structure
  - Of course, the library still needs to represent the matrix with some choice of data structure, but this is an *implementation detail*
- Benefit
  - Programs become independent of any particular choice of data structure, making it easier to modify and adapt programs.

Operations in PETSc

- How should operations like “solve linear system” be described in a program?
  - Old way

```
mpiaijgmres( ia, ja, a, comm, x, b, nlocal, nglobal,
             ndir, orthomethod, convtol, &its )
```

  - New way

```
SLESSolve( sles, b, x, &its )
```

- Hides the choice of algorithm
  - Algorithms are to operations as data structures are to objects
- Benefit
  - Programs become independent of a particular choice of algorithm, making it easier to explore algorithmic choices and to adapt to new methods
- In PETSc, operations have their own “handle”, called a “*context variable*”

Context Variables in PETSc

- Context variables are the key to solver organization
- They contain the complete state of an algorithm, including
  - parameters (e.g., convergence tolerance)
  - functions run by the algorithm (e.g., convergence monitoring routine)
  - information about the current state (e.g., iteration number)

SLES Structure

- Each SLES object contains two other objects:

KSP Krylov Space Method
  - The iterative method
  - The KSP context contains information on the method parameters, e.g. GMRES restart and search directions)

PC Preconditioners
  - Knows how to apply the preconditioner
  - The context contains information on the preconditioner, such as ILU fi ll level

Available Methods

KSP		PC	
Name	PETSc option	Name	PETSc option
Conjugate Gradient	cg	Block Jacobi	bjacobi
GMRES	gmres	Overlapping Additive Schwarz	asm
Bi-CG-stab	bicg	ILU	ilu
Transpose-free QMR	tfqmr	SOR	sor
Richardson	richardson	LU (direct solve)	lu
CG-Squared	cgs	Multigrid	mg
SYMMLQ	symmlq	Arbitrary matrix	mat
others		others	

Using the Command Line Interface

- PETSc makes it each to try different algorithms

```
mpiexec -n 4 poisson -ksp_type cg
mpiexec -n 4 poisson -ksp_type gmres
mpiexec -n 4 poisson -pc_type bjacobi -sub_pc_type ilu \
             -ksp_type bcgs
```

- PETSc make experimentation with different algorithms *easy*
  - Many are already built-in
  - You can add new algorithms and data structures to PETSc; these are then used just like the built-in ones (e.g., a new preconditioner can be used with an existing source code without *any* changes. (However, this is not a one-day project.)
- Many other options available. Use

```
poisson -help | more
```

to get a list of available options

Monitoring Convergence

- PETSc provides routines to check for and monitor convergence
- The choice of monitor and the output from that monitor can be controlled from the command line

  -ksp_monitor Print the preconditioned residual norm

  -ksp_xmonitor Plot the preconditioned residual norm

  -ksp_truemonitor Print the true residual norm $\|Ax - b\|_2$

  -ksp_truexmonitor Plot the true residual norm

- Custom monitors can be defi ned by the user

Accessing the Solution

- *Viewers* are used in PETSc to access and display the contents of an object
- A simple viewer prints data out standard output:

```
VecView( V, PETSC_VIEWER_STDOUT_WORLD );
```

- PETSc provides a wide range of viewers for all major objects
  - Viewers make it easy to send vectors and matrices to Matlab
  - Graphical viewers make it easy to display data
  - Binary viewers make it easy to save and load data

PETSc Viewers

- PETSc has many viewers

  PETSC_VIEWER_STDOUT_SELF Sequential, prints to stdout

  PETSC_VIEWER_STDOUT_WORLD Parallel, prints to stdout

  PETSC_VIEWER_DRAW_WORLD Parallel, draws using X-Windows

- Viewers exist for matrices, vectors, and other objects
  - Matrix viewers provide information and graphical display of matrix sparsity structure and assembly (try `-mat_view_draw`, `-mat_view_info`, or `-mat_view`
  - Viewers on other objects can print out information about the object

Working With Vectors

- It is sometimes helpful to have direct access to the storage for the local elements of a vector
- The routines VecGetArray and VecRestoreArray may be used to get and return the local elements
- The routine VecGetLocalSize returns the number of elements in the local part of the vector
- VecGetArray returns a pointer to an array that contains the locally-owned values in the vector. Normally, this is just a pointer into the storage that PETSc uses, but for special vector implementations, it may be different storage used just for VecGetArray
- VecRestoreArray gives the array back to PETSc. Normally, this has no work to do, but if PETSc had to allocate storage for VecGetArray, this routine will free that storage

Example: Computing $\|x - y\|$

- Often need to compute $\|x - y\|$, for example, for convergence tests. Also useful in checking a solution
- PETSc does provide routines to compute $x + \alpha y$ and $\|x\|$, but no single routine to compute the norm of the difference of two vectors
- As an example of accessing local elements of a vector, we will implement “mVecNormXPAY” which computes $\|x + \alpha y\|$
  - Accepts all PETSc norm types: NORM_1, NORM_2, and NORM_INFINITY.
- A single routine avoids creating an unneeded temporary vector and avoids extra memory motion needed when using multiple routines

Computing $\|x - y\|$ I

```
#include "petscvec.h"

/* This is a new vector routine for PETSc, illustating the use
   of several PETSc functions for accessing vector elements */

int mVecNormXPAY( Vec x, Vec y, const PetscScalar a, NormType ntype,
                  PetscReal *norm )
{
    const double * restrict xvals, * restrict yvals;
    int   nlocal, i, ierr = 0;
    MPI_Op normop;
    double sum = 0.0, totsum;

    /* Get the local arrays and the size */
    VecGetArray( x, (PetscScalar **)&xvals );
    VecGetArray( y, (PetscScalar **)&yvals );
    VecGetLocalSize( x, &nlocal );
```

Computing $\|x - y\|$ II

```

    if (a == -1) {
        /* Special case for difference of two vectors */
        switch (ntype) {
        case NORM_1:
            for (i=0; i<nlocal; i++) {
                sum += fabs(xvals[i] - yvals[i]);
            }
            normop = MPI_SUM;
            break;
        case NORM_2:
            for (i=0; i<nlocal; i++) {
                register PetscScalar tmp;
                tmp = xvals[i] - yvals[i];
                sum += tmp*tmp;
            }
            normop = MPI_SUM;
            break;
```

Computing $\|x - y\|$ III

```
        case NORM_INFINITY:
            for (i=0; i<nlocal; i++) {
                register PetscScalar tmp;
                tmp = fabs(xvals[i] - yvals[i]);
                if (tmp > sum) sum = tmp;
            }
            normop = MPI_MAX;
            break;
        default:
            ierr = 1;
            break;
        }
    }
    else {
        /* Unimplemented */
        ierr = 1;
    }
    if (!ierr) {
```

Computing $\|x - y\|$ IV

```
        MPI_Comm comm;
        PetscObjectGetComm( (PetscObject)x, &comm );
        MPI_Allreduce( &sum, &totsum, 1, MPI_DOUBLE, comm, normop );
        if (ntype == NORM_2) {
            totsum = sqrt( totsum );
        }
        *norm = totsum;
    }

    VecRestoreArray( x, (PetscScalar **)&xvals );
    VecRestoreArray( y, (PetscScalar **)&xvals );

    return ierr;
}

```

PetscScalar is just a name for `double`; using this name allows the PETSc to be rebuilt for `float` or `Complex` scalars.

Distributed Arrays in PETSc

How should a vector be distributed across processes? PETSc's default is a "one-dimensional decomposition"

How can you make use of different data decompositions in PETSc? PETSc provides "Distributed Arrays" (DAs) for this purpose.

For example, consider the layout of a mesh onto this processor mesh:

P2	P3
P0	P1

Layout Of Distributed Arrays

On this 2×2 process grid, the vector elements are numbered like this:

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

Natural numbering

$\longrightarrow$

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

PETSc's internal numbering

DAs provide a “logically Cartesian” decomposition. There are no physical coordinates associated with a DA.

Distributed Arrays

- PETSc distributed arrays (DAs) provide a way to describe a multidimensional arrays, distributed across a parallel processor
- DAs provide a way to use more complex data decompositions

```
DACreate2d( PETSC_COMM_WORLD, DA_NONPERIODIC,
            DA_STENCIL_STAR,
            nx, ny, px, py, 1, 1, 0, 0, &grid );
```

  creates a global $nx \times ny$ grid, with a $px \times py$ process decomposition
- The DA_STENCIL_STAR and the arguments after py have to do with the difference stencil that may be used with this array and will be discussed later.
- MPI_Dims_create may be used to determine good values for px and py.

Setting the Vector Values I

```
#include "petsc.h"
#include "petscvec.h"
#include "petscda.h"

/* Form a vector based on a function for a 2-d regular mesh on the
   unit square */
Vec FormVecFromFunctionDA2d( DA grid, int n,
                             double (*f)( double, double ) )
{
    Vec    V;
    int    is, ie, js, je, in, jn, i, j;
    double h;
    double **vval;

    h = 1.0 / (n + 1);
    DACreateGlobalVector( grid, &V );

```

Setting the Vector Values II

```
    DAVecGetArray( grid, V, (void **)&vval );
    /* Get global coordinates of this patch in the DA grid */
    DAGetCorners( grid, &is, &js, 0, &in, &jn, 0 );
    ie = is + in - 1;
    je = js + jn - 1;
    for (i=is ; i<=ie ; i++) {
        for (j=js ; j<=je ; j++){
            vval[j][i] = (*f)( (i + 1) * h, (j + 1) * h );
        }
    }
    DAVecRestoreArray( grid, V, (void **)&vval );

    return V;
}

```

Understanding the Code

DACreateGlobalVector Creates a PETSc vector that may be used with DAs

DAVecGetArray Get a multidimensional array that gives the illusion of a global array (PETSc uses tricks with the array indexing to provide access to the local elements of the vector). Otherwise, like VecGetArray.

DAVecRestoreArray Like VecRestoreArray, used to allow PETSc to free any storage allocated by DAVecGetArray

DAGetCorners Returns the indices of the lower-left corner of the local part of the distributed array relative to the global coordinates, along with the number of points in each direction.

Setting the Matrix Elements I

```
#include "petscsles.h"
#include "petscda.h"

/* Form the matrix for the 5-point finite difference 2d Laplacian
   on the unit square. n is the number of interior points along a
   side */
Mat FormLaplacianDA2d( DA grid, int n )
{
    Mat    A;
    int    r, i, j, is, ie, js, je, in, jn, nelm;
    MatStencil cols[5], row;
    double     h, oneByh2, vals[5];

    h = 1.0 / (n + 1); oneByh2 = 1.0 / (h*h);

    DAGetMatrix( grid, MATMPIAIJ, &A );
    /* Get global coordinates of this patch in the DA grid */
```

Setting the Matrix Elements II

```
DAGetCorners( grid, &is, &js, 0, &in, &jn, 0 );
ie = is + in - 1;
je = js + jn - 1;
/* This is a simple but inefficient way to set the matrix */
for (i=is; i<=ie; i++) {
    for (j=js; j<=je; j++){
        row.j = j; row.i = i; nelm = 0;
        if (j - 1 > 0) {
            vals[nelm]   = oneByh2;
            cols[nelm].j = j - 1; cols[nelm++].i = i;}
        if (i - 1 > 0) {
            vals[nelm]   = oneByh2;
            cols[nelm].j = j;     cols[nelm++].i = i - 1;}
        vals[nelm]   = - 4 * oneByh2;
        cols[nelm].j = j;         cols[nelm++].i = i;
        if (i + 1 < n - 1) {
            vals[nelm]   = oneByh2;
            cols[nelm].j = j;     cols[nelm++].i = i + 1;}
```

Setting the Matrix Elements III

```
            if (j + 1 < n - 1) {
                vals[nelm]   = oneByh2;
                cols[nelm].j = j + 1; cols[nelm++].i = i;}
            MatSetValuesStencil( A, 1, &row, nelm, cols, vals,
                                 INSERT_VALUES );
        }
    }

    MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);

    return A;
}
```

Understanding the Code

DAGetMatrix Returns a matrix whose elements can be accessed with the coordinates of the distributed array. The type of the matrix must be specifi ed; this choses a parallel matrix using AIJ format (MATMPIAIJ).

MatSetValuesStencil Sets elements of a matrix using mesh coordinates

MatStencil Data structure that contains the indices of a point in the DA, using the `i,j,k` members of the structure

Poisson Solver Revisited

```
#include <math.h>
#include "petscsles.h"
#include "petscda.h"
extern Mat FormLaplacianDA2d( DA, int );
extern Vec FormVecFromFunctionDA2d( DA, int, double (*)(double,double) );
/* This function is used to define the right-hand side of the
   Poisson equation to be solved */
double func( double x, double y ) {
    return sin(x*M_PI)*sin(y*M_PI); }

int main( int argc, char *argv[] )
{
    SLES        sles;
    Mat         A;
    Vec         b, x;
    DA          grid;
    int         its, n, px, py, worldSize;
```

Poisson Solver Revisited II

```
PetscInitialize( &argc, &argv, 0, 0 );

/* Get the mesh size.  Use 10 by default */
n = 10;
PetscOptionsGetInt( PETSC_NULL, "-n", &n, 0 );
/* Get the process decomposition.  Default it the same as without DAs */
px = 1;
PetscOptionsGetInt( PETSC_NULL, "-px", &px, 0 );
MPI_Comm_size( PETSC_COMM_WORLD, &worldSize );
py = worldSize / px;

/* Create a distributed array */
DACreate2d( PETSC_COMM_WORLD, DA_NONPERIODIC, DA_STENCIL_STAR,
            n, n, px, py, 1, 1, 0, 0, &grid );

/* Form the matrix and the vector corresponding to the DA */
```

Poisson Solver Revisited III

```
    A = FormLaplacianDA2d( grid, n );
    b = FormVecFromFunctionDA2d( grid, n, func );
    VecDuplicate( b, &x );
    SLESCreate( PETSC_COMM_WORLD, &sles );
    SLESSetOperators( sles, A, A, DIFFERENT_NONZERO_PATTERN );
    SLESSetFromOptions( sles );
    SLESSolve( sles, b, x, &its );

    PetscPrintf( PETSC_COMM_WORLD, "Solution is:\n" );
    VecView( x, PETSC_VIEWER_STDOUT_WORLD );
    PetscPrintf( PETSC_COMM_WORLD, "Required %d iterations\n", its );

    MatDestroy( A ); VecDestroy( b ); VecDestroy( x );
    SLESDestroy( sles ); DADestroy( grid );
    PetscFinalize( );
    return 0;
}
```

Scaling Studies

- Lab: Explore the scaling of the in terms of the iteration counts for solving Poisson problem using the default 1-d and the DA-based 2-d decomposition, as a function of the number of processes.

Incremental Application Improvement

- Get the application “up and walking”
- Experiment with options. Determine opportunities for improvement
- Extend algorithms and/or data structures as needed
- Consider interface and efficiency issues for integration and interoperability of multiple toolkits
- Full tutorials available at `http://www.mcs.anl.gov/petsc/docs/tutorials`

Examples of Linear Solves

ex1.c: Solves a tridiagonal linear system with SLES
ex2,3.c: Solves a linear system in parallel with SLES
ex4.c: Uses a different preconditioner matrix and linear system matrix in the SLES solvers
ex5.c: Solves two linear systems in parallel with SLES
ex7.c: Block Jacobi preconditioner for solving a linear system in parallel with SLES
ex8.c: Illustrates use of the preconditioner ASM
ex9.c: The solution of 2 different linear systems with different linear solvers
ex10.c: Reads a PETSc matrix and vector from a file and solves a linear system
ex11.c: Solves a linear system in parallel with SLES
ex12.c: Solves a linear system in parallel with SLES
ex13.c: Solves a variable Poisson problem with SLES
ex15.c: Solves a linear system in parallel with SLES
ex16.c: Solves a sequence of linear systems with different right-hand-side vectors
ex22.c: Solves 3D Laplacian using multigrid
ex23.c: Solves a tridiagonal linear system
ex25.c: Solves 1D variable coefficient Laplacian using multigrid
ex26.c: Solves a linear system in parallel with ESI
ex27.c: Reads a PETSc matrix and vector from a file and solves the normal equations

More Preconditioners

- PETSc provides a large collection of preconditioners, including domain decomposition preconditioners
  - Additive Schwarz

    ```
    mpiexec -n 4 poisson -pc_type asm
    ```

  - Control the subdomain solver with `-sub_pc_type`:

    ```
    mpiexec -n 4 poisson -pc_type asm -sub_pc_type ilu
    ```

    (In general, `-sub_pc_<pcparmname>` may be used to change the PC parameter `pcparmname` in the subdomain, and `-sub_ksp_<kspparmname>` for KSP in the subdomain.)

  - Control the subdomain overlap

    ```
    mpiexec -n 4 poisson -pc_type asm -pc_asm_overlap 2
    ```

- The tutorial example Makefile lets you run these with the “run” target:

```
make run PGM=poisson NP=4 ARGS="-pc_type asm -pc_asm_overlap 2"
```

PETSc's Automatic ASM

- PETSc automatically generates overlap by using the structure of the sparse matrix. Control with `-pc_asm_overlap`
- DAs allow you to control the local physical domain
- By using DAs, you can experiment with the effects of different decompositions

```
mpiexec -n 16 poisson  -n 64 -pc_type asm
mpiexec -n 16 poisson2 -n 64 -pc_type asm -mx 8 -my 2
mpiexec -n 16 poisson2 -n 64 -pc_type asm -mx 4 -my 4
```

- Other ASM types are available with `-pc_asm_type`

basic full interpolation and restriction

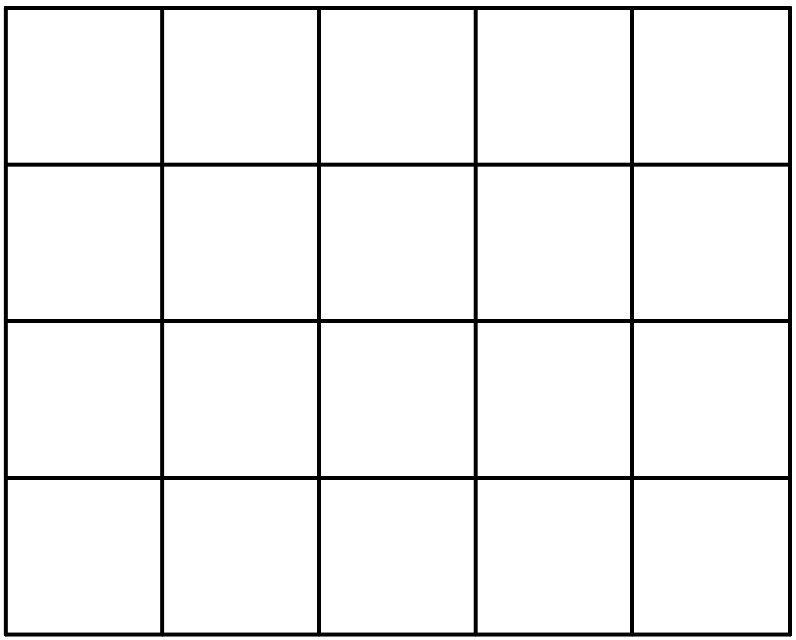
restrict full restriction, local process interpolation

interpolate full interpolation, local process restriction

none local process restriction and interpolation

Flow of Information

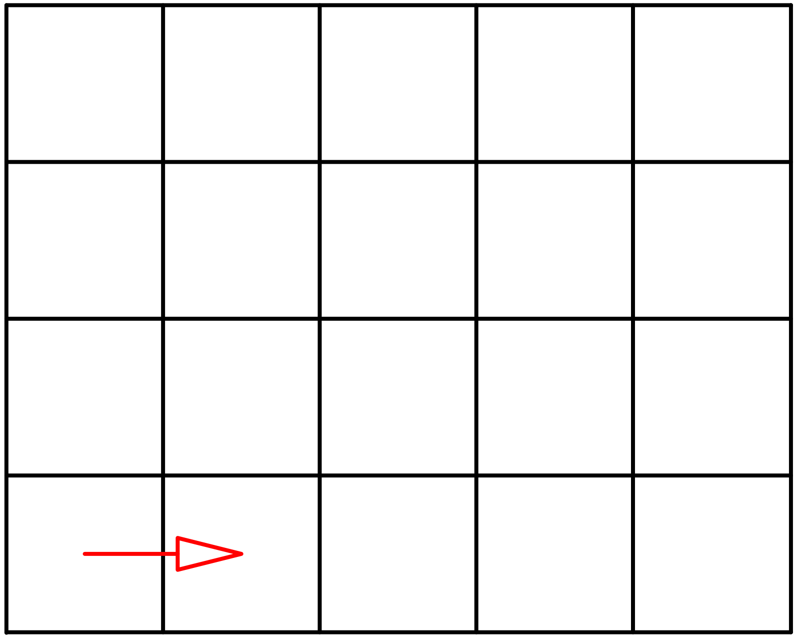
- The number and layout of domains sets a minimum for the number of iterations expected for convergence
- At the very least, data must travel from across the entire mesh:



- In general, solving with a $p_x \times p_y$ decomposition requires at least $(p_x - 1)(p_y - 1)$ steps, thus
- Square decompositions provide the best starting point

Flow of Information

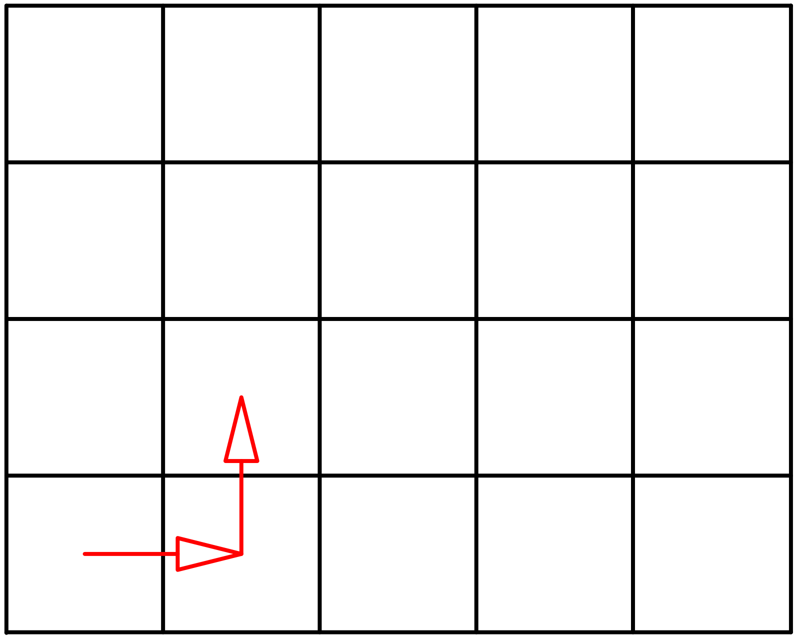
- The number and layout of domains sets a minimum for the number of iterations expected for convergence
- At the very least, data must travel from across the entire mesh:



- In general, solving with a $p_x \times p_y$ decomposition requires at least $(p_x - 1)(p_y - 1)$ steps, thus
- Square decompositions provide the best starting point

Flow of Information

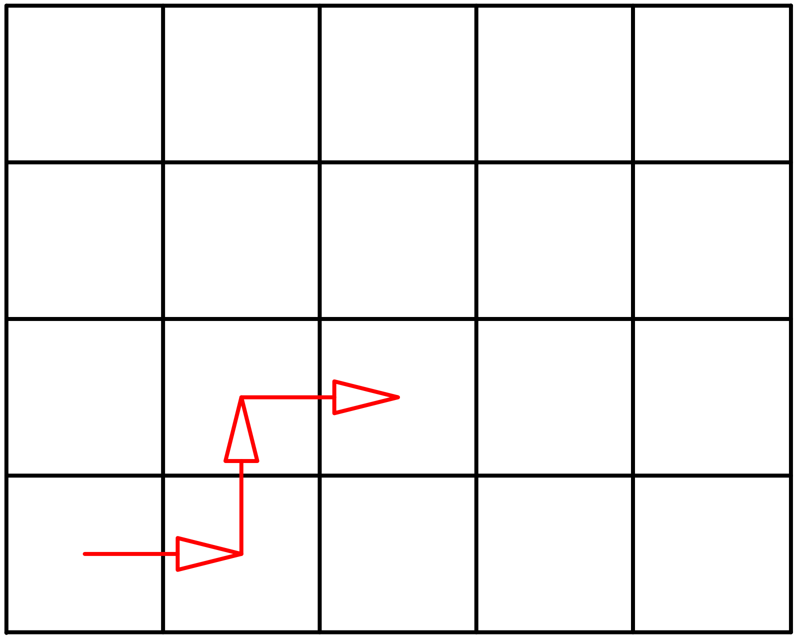
- The number and layout of domains sets a minimum for the number of iterations expected for convergence
- At the very least, data must travel from across the entire mesh:



- In general, solving with a $p_x \times p_y$ decomposition requires at least $(p_x - 1)(p_y - 1)$ steps, thus
- Square decompositions provide the best starting point

Flow of Information

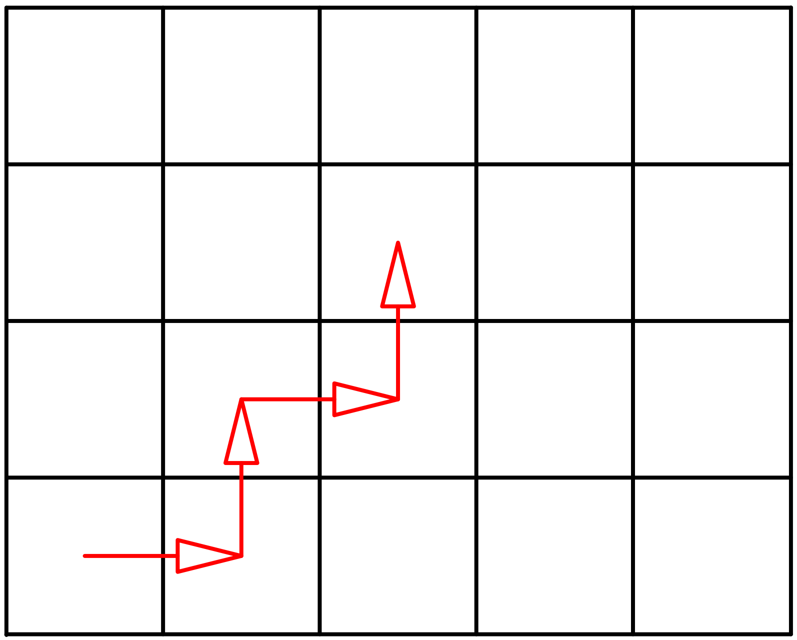
- The number and layout of domains sets a minimum for the number of iterations expected for convergence
- At the very least, data must travel from across the entire mesh:



- In general, solving with a $p_x \times p_y$ decomposition requires at least $(p_x - 1)(p_y - 1)$ steps, thus
- Square decompositions provide the best starting point

Flow of Information

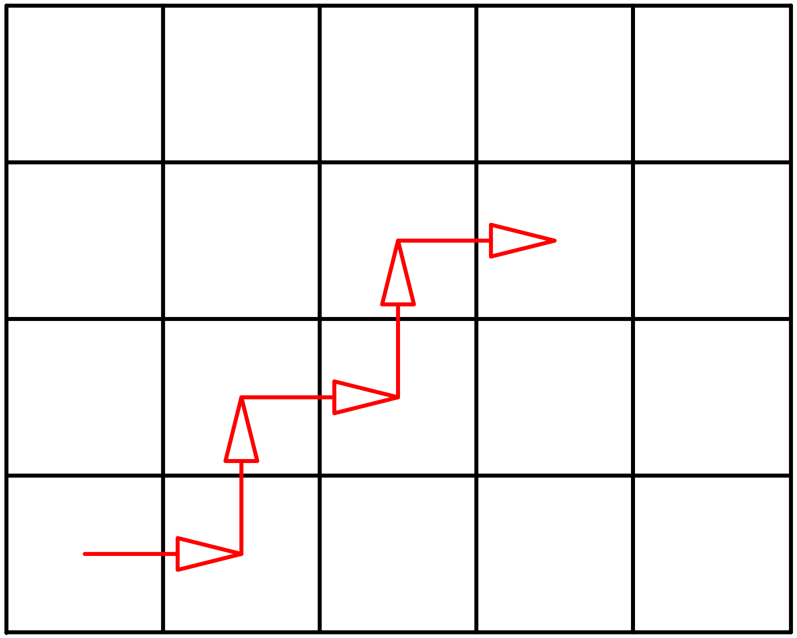
- The number and layout of domains sets a minimum for the number of iterations expected for convergence
- At the very least, data must travel from across the entire mesh:



- In general, solving with a $p_x \times p_y$ decomposition requires at least $(p_x - 1)(p_y - 1)$ steps, thus
- Square decompositions provide the best starting point

Flow of Information

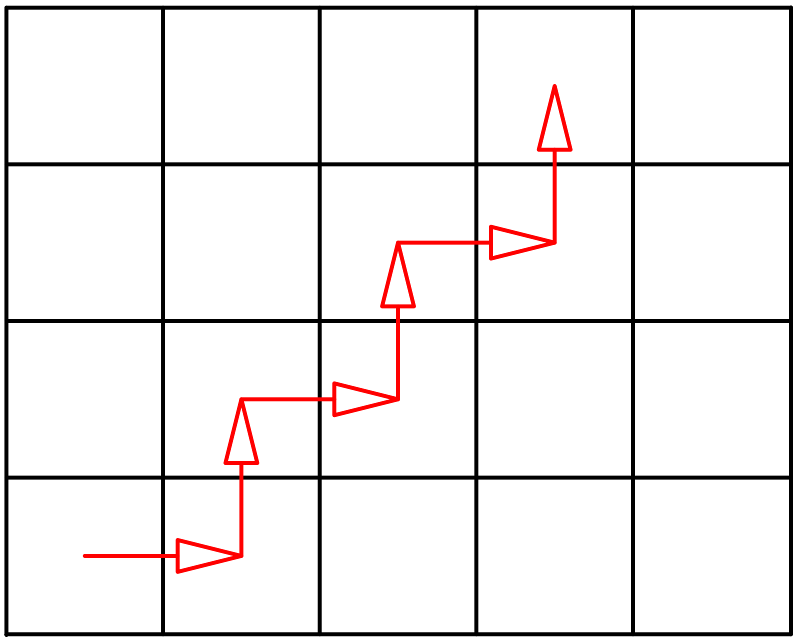
- The number and layout of domains sets a minimum for the number of iterations expected for convergence
- At the very least, data must travel from across the entire mesh:



- In general, solving with a $p_x \times p_y$ decomposition requires at least $(p_x - 1)(p_y - 1)$ steps, thus
- Square decompositions provide the best starting point

Flow of Information

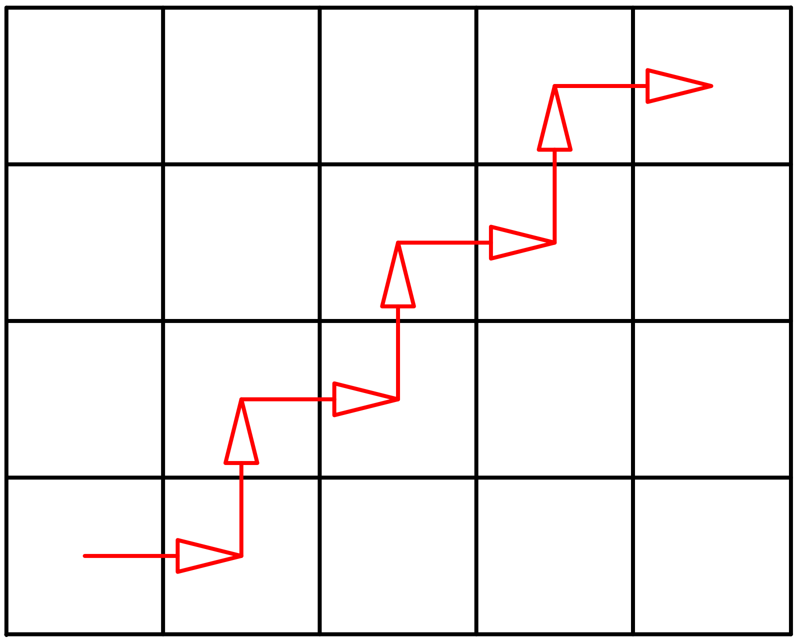
- The number and layout of domains sets a minimum for the number of iterations expected for convergence
- At the very least, data must travel from across the entire mesh:



- In general, solving with a $p_x \times p_y$ decomposition requires at least $(p_x - 1)(p_y - 1)$ steps, thus
- Square decompositions provide the best starting point

Flow of Information

- The number and layout of domains sets a minimum for the number of iterations expected for convergence
- At the very least, data must travel from across the entire mesh:



- In general, solving with a $p_x \times p_y$ decomposition requires at least $(p_x - 1)(p_y - 1)$ steps, thus
- Square decompositions provide the best starting point

Aside: Error Handling in PETSc

- All PETSc routines return an error value. This can be tested with CHKERRQ, as in

  ```
  ierr = SLESCreate( PETSC_COMM_WORLD, &sles ); CHKERRQ(ierr);
  ```

  Using CHKERRQ allows PETSc to provide clear and specific error messages
- An alternative is to set the error handler that PETSc calls when an error is first detected:

  ```
  PetscPushErrorHandler( PetscAbortErrorHandler, 0 );
  ```

  (only available in C in PETSc 2.1.5). Other handlers exist, including PetscAttachDebuggerErrorHandler .
- Command line options `-on_error_abort` and `-start_in_debugger` may also be used to change the default error handler

Solving Nonlinear Equations

We would like to solve

$$F(u) = 0$$

for u. A powerful method for this is *Newton's method*:

$$u^{k+1} = u^k - (F'(u^k))^{-1} F(u^k), \quad k = 0, 1, \ldots$$

where u^k is the approximation to u at the kth step. The term $F'(u^k)$ is a matrix, and this algorithm can be rewritten as

$$\begin{aligned} F'(u^k)\Delta u^k &= -F(u^k) \\ u^{k+1} &= u^k + \Delta u^k \end{aligned}$$

Newton-based Methods

In practice, various modifi cations are made to Newton's method. PETSc supports many of the most common:

- Line search strategies
- Trust region strategies
- Pseudo-transient continuation
- Matrix-free varients

PETSc provides a “Simplifi ed Nonlinear Equation Solver” (SNES) for nonlinear problems. SNES is the nonlinear analogue of SLES.

PDE Jacobian

The matrix $F'(u)$ is called the *Jacobian*.
For PDE problems, computing the Jacobian can be tricky. Three choices are:

1. Compute F' analytically, then discretize
2. Discretize F, then compute F' by fi nite difference approximation
3. Discretize F, then compute F' by analytically differentiating the discretization of F

PETSc provides additional support for 2, and by interfacing to ADIFOR and ADIC, support for 3

A Simple Nonlinear PDE

The *Bratu* problem is defined by

$$\begin{aligned} -\nabla^2 u - \lambda e^u &= 0 \text{ in } [0,1] \times [0,1] \\ u &= 0 \text{ on the boundary} \end{aligned}$$

We will use the same simple discretization for this problem as for the Poisson problem.

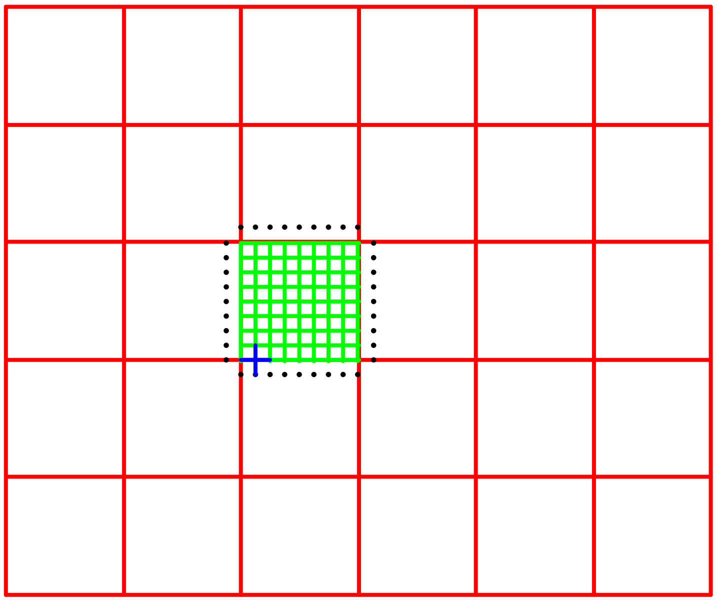
Evaluating the Function

- Evaluating the function $F(u) = -\nabla^2 u - \lambda e^u$ is somewhat diffi cult because it involves a differential operator. This requires information from the neighboring processes. We will use distributed arrays (DAs) to help with this, taking advantage of their support for different *stencils*.
- An alternate approach for *this* example is to use a matrix-vector multiply, using

  ```
  MatMult( A, x, y );
  ```

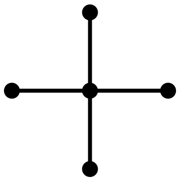
  to compute $y = Ax$. This routine handles all data motion required. However, it is suitable only for relatively simple $F(u)$. Thus, we will explore more general techniques

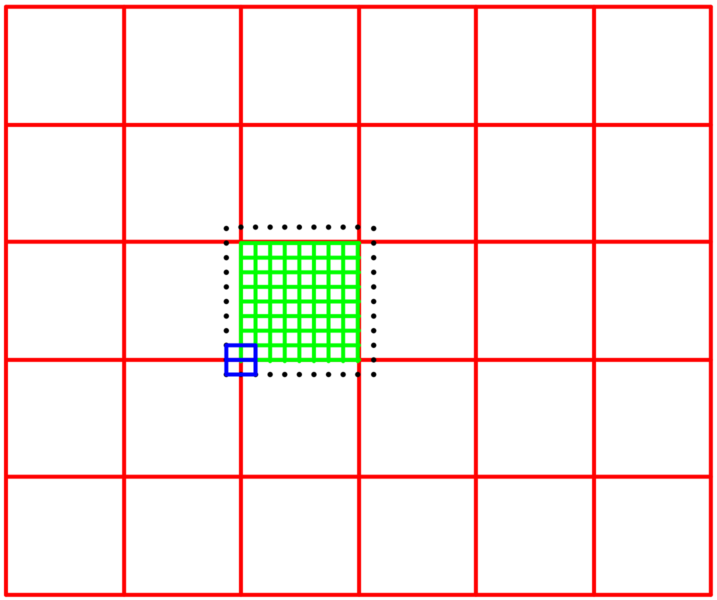
Stencils



Star Stencil

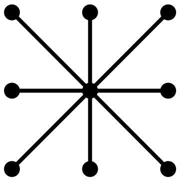
(DA_STENCIL_STAR)



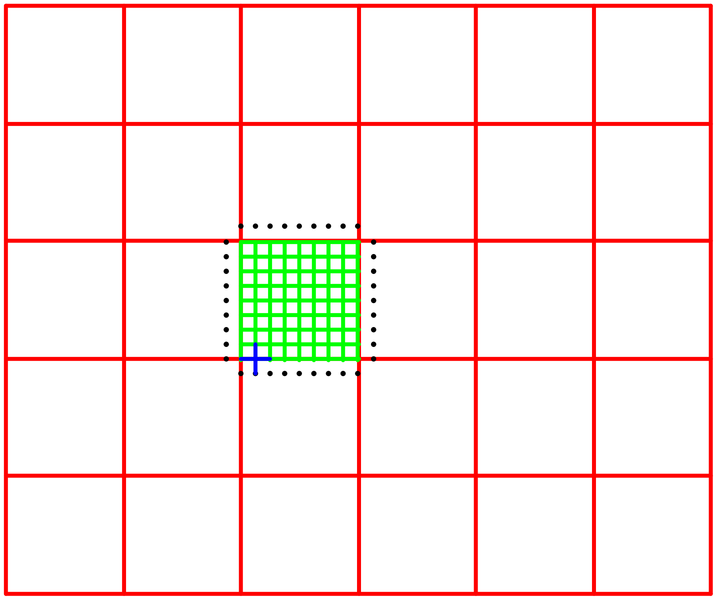


Box Stencil

(DA_STENCIL_BOX)

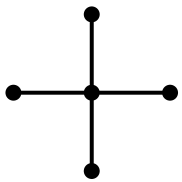


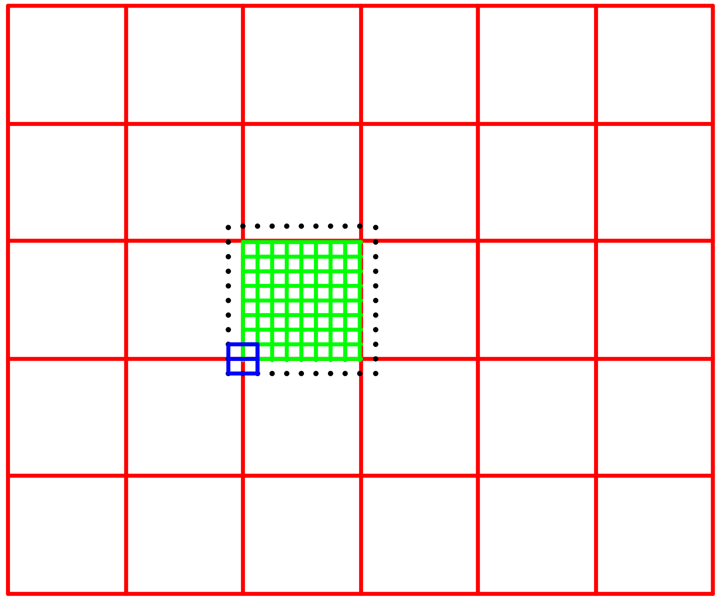
Stencils



Star Stencil

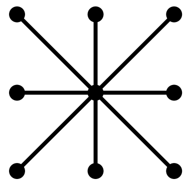
(DA_STENCIL_STAR)





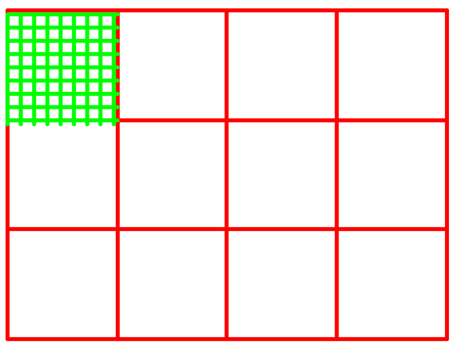
Box Stencil

(DA_STENCIL_BOX)

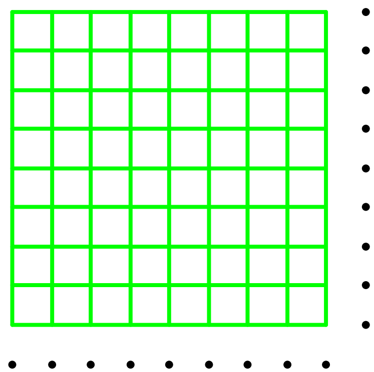


Global and Local Representations

- A vector associated with a DA has two representations: the *global* and the *local*
- The global representation is nothing more than the natural mesh, distributed across all processes
- The local representation is the local part of the global mesh, *plus* the ghost points



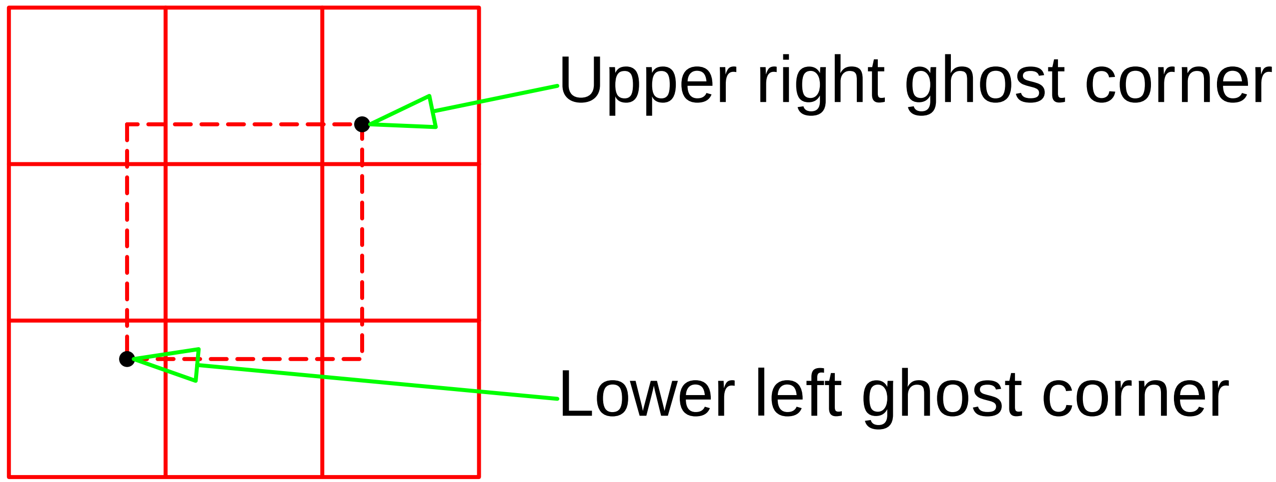
Global: each process stores a unique local set of vertices, and each vertex is owned by exactly one process



Local: each process stores a unique local set of vertices *as well as* ghost points from neighboring processes

Using Ghost Points with DAs

A ghost region is defined by the coordinates *in the global representation*:



The routine DAGetGhostCorners returns this information, similar to DAGetCorners

Moving Data Between the Global and Local Representations

DACreateLocalVector Creates a PETSc vector that can hold the local representation of a DA (the local mesh plus ghost points)

DAGlobalToLocalBegin and DAGlobalToLocalEnd Update the ghostpoint values. This involves communication with the neighboring processes

DALocalToGlobal Transfers values in the local representation back to the global representation. The ghost points are discarded.

Parallel Evaluation of the Function

In the Bratu example,

$$F(u) = -\nabla^2 u - \lambda e^u$$

so

$$F'(u)a = -\nabla^2 a - \lambda a e^u,$$

where ae^u is just $\{a_i \times e^{u_i}\}$. Thus the Jacobian $F'(u)$ is almost the same as the matrix for the Poisson problem, with a diagonal element that depends on u. Now that we know what these are, how do we provide them to PETSc?

Providing the Function and Jacobian

We now have functions that evaluate F and F'. How can these be used by the SNESSolve routine?

- The algorithm needs to evaluate both, under control of the algorithm
- The solution used in PETSc is to pass the functions themselves to the routine that defines the problem, much as the matrix defining a linear problem to solve is passed to SLESSetOperators.
- This is a “callback” method, because the user provides functions to the solver that are called back by the algorithm when their results are needed
- The *calling sequence* for the routine is specified by PETSc.

Specifying Callbacks

- User provides the routines to perform actions that the library requires. For example

```
SNESSetFunction(snes, f, userfunc, userctx )
```

snes SNES context

f Vector that will be used to store the function value

userfunc Name of (really, pointer to) the function

userctx Pointer to data passed that will be passed to the function

- The library can call this function whenever it needs to evaluate the function
- The userctx pointer allows the user to provide an “application context” object. By using this approach, the library need never know the details of data needed only by the application.

Forming the Function I

```
#include "petscsnes.h"
#include "petscda.h"
#include "bratu.h"
#include <math.h>

/* Evaluate the function for the Bratu nonlinear problem on the local
   mesh points */
int FormBratuFunction( SNES snes, Vec v, Vec f, void *ctx )
{
    UserBratuCtx *bratu = (UserBratuCtx *)ctx;
    DA           da     = bratu->da;
    double       lambda = bratu->lambda;
    double       h      = bratu->h;
    Vec          lv;
    int          i, j;
    int          lli, llj, ni, nj; /* lower left i,j and size for local
                                      part of mesh */
```

Forming the Function II

```
const double **varr;
double       **fvarr;

/* Get the coordinates of our part of the global mesh */
DAGetCorners( da, &lli, &llj, 0, &ni, &nj, 0 );

DAGetLocalVector( da, &lv );

/* Scatter the ghost points to the other processes, using
   the values in the input vector v */
DAGlobalToLocalBegin( da, v, INSERT_VALUES, lv );
DAGlobalToLocalEnd( da, v, INSERT_VALUES, lv );

DAVecGetArray( da, lv, (void **)&varr );
DAVecGetArray( da, f,  (void **)&fvarr );

for (j=llj ; j<llj+nj ; j++)
    for (i=lli ; i<lli+ni ; i++) {
```

Forming the Function III

```
        if (i == 0 || j == 0 ||
            i == bratu->n + 1 || j == bratu->n + 1) {
            fvarr[j][i] = 0.0;
        }
        else {
            fvarr[j][i] = -( varr[j-1][i] + varr[j][i-1] +
                            varr[j+1][i] + varr[j][i+1] -
                            4 * varr[j][i] ) / (h*h) -
                lambda * exp(varr[j][i]);
        }
    }
    DAVecRestoreArray( da, f, (void **)&fvarr );
    DAVecRestoreArray( da, lv, (void **)&varr );
    DARestoreLocalVector( da, &lv );

    return 0;
}
```

Understanding the Code

- One key feature of this routine is the use of the fourth argument, “`ctx`”, to pass additional information to the Function. In this case, we use a user-defined structure define in bratu.h:

```
/* This typedef defines a struct that contains the
   data that we need to have when evaluating the
   function or the Jacobian for the Bratu problem  */
typedef struct {
  DA     da;         /* DA for grid */
  double h;          /* Mesh spacing */
  double lambda;     /* parameter in problem */
  int    n;          /* interior grid is n x n */
} UserBratuCtx;
```

- The rest of the code uses the DA to provide ghost values for the the evaluation of the finite difference scheme
  - Boundary conditions, as always, add complexity

Forming the Jacobian I

```
#include "petscsnes.h"
#include "petscda.h"
#include "bratu.h"
#include <math.h>

/* Form the matrix for the Jacobian of the Bratu problem, where the
   function uses a 5-point finite difference 2d Laplacian
   on the unit square. n is the number of interior points along a side */
Mat FormBratuJacobian( SNES snes, Vec u, Mat *A, Mat *B, MatStructure *flag,
                       void *ctx )
{
    Mat    jac = *A;
    UserBratuCtx *bratu = (UserBratuCtx *)ctx;
    DA     da = bratu->da;
    int    r, i, j, n = bratu->n;
    double oneByh2, **uvals;
    double h = bratu->h, lambda = bratu->lambda;
```

Forming the Jacobian II

```
int     lli, llj, ni, nj; /* lower left i,j and size for local
                             part of mesh */
MatStencil row, col[5];
double     v[5];

oneByh2 = 1.0 / (h*h);

DAGetCorners( da, &lli, &llj, 0, &ni, &nj, 0 );
DAVecGetArray( da, u, (void **)&uvals );

/* This is a simple but inefficient way to set the matrix */
for (j=llj; j<llj+nj; j++) {
    for (i=lli; i<lli+ni; i++) {
        row.i = i; row.j = j;
        if (i == 0 || j == 0 ||
            i == n + 1 || j == n + 1) {
            v[0] = 1.0;
            MatSetValuesStencil( jac, 1, &row, 1, &row, v, INSERT_VALUES
```

Forming the Jacobian III

```
        }
        else {
            col[0].i = i; col[0].j = j - 1; v[0] = - oneByh2;
            col[1].i = i; col[1].j = j + 1; v[1] = - oneByh2;
            col[2].i = i - 1; col[2].j = j; v[2] = - oneByh2;
            col[3].i = i + 1; col[3].j = j; v[3] = - oneByh2;
            col[4].i = i; col[4].j = j;
            v[4] = 4.0 * oneByh2 - lambda * exp( uvals[j][i] );
            MatSetValuesStencil( jac, 1, &row, 5, col, v, INSERT_VALUES
        }
    }
}
MatAssemblyBegin(jac, MAT_FINAL_ASSEMBLY);
DAVecRestoreArray( da, u, (void **)&uvals );

*flag = SAME_NONZERO_PATTERN;  /* preconditioner has same structure */
MatAssemblyEnd(jac, MAT_FINAL_ASSEMBLY);
```

Forming the Jacobian IV

```
    return 0;
}
```

Bratu Example I

```
#include "petscsnes.h"
#include "petscda.h"
#include "bratu.h"

extern int FormBratuJacobian( SNES, Vec, Mat *, Mat *, MatStructure *, void
extern int FormBratuFunction( SNES, Vec, Vec, void * );

int main( int argc, char *argv[] )
{
    UserBratuCtx bratu;
    SNES         snes;
    Vec          x, r;
    Mat          J;
    int          its;

    PetscInitialize( &argc, &argv, 0, 0 );
```

Bratu Example II

```
/* Get the problem parameters */
bratu.lambda = 6.0;
PetscOptionsGetReal( 0, "-lambda", &bratu.lambda, 0 );
if (bratu.lambda >= 6.81 || bratu.lambda < 0) {
    SETERRQ(1,"Lambda must be between 0 and 6.81");
}
bratu.n = 10; /* Get the mesh size.  Use 10 by default */
PetscOptionsGetInt( PETSC_NULL, "-n", &bratu.n, 0 );
bratu.h = 1.0 / (bratu.n + 1);

SNESCreate( PETSC_COMM_WORLD, &snes );

/* Create the mesh and decomposition */
DACreate2d( PETSC_COMM_WORLD, DA_NONPERIODIC, DA_STENCIL_STAR,
            bratu.n + 2, bratu.n + 2, PETSC_DECIDE, PETSC_DECIDE,
            1, 1, 0, 0, &bratu.da );

DACreateGlobalVector( bratu.da, &x );
```

Bratu Example III

```
    VecDuplicate( x, &r );  /* Use this as the vector to give SetFunction */
    SNESSetFunction( snes, r, FormBratuFunction, &bratu );

    DAGetMatrix( bratu.da, MATMPIAIJ, &J );
    SNESSetJacobian( snes, J, J, FormBratuJacobian, &bratu );

    SNESSetFromOptions( snes );

    FormBratuInitialGuess( &bratu, x );
    SNESSolve( snes, x, &its );

    PetscPrintf( PETSC_COMM_WORLD,
                 "Number of Newton iterations = %d\n", its );

    VecDestroy(r);
    SNESDestroy(snes);
    PetscFinalize( );
    return 0;
}
```

Understanding the Code

SNESCreate Creates the SNES context

SNESSetFunction Specify the function to be called to evaluate the function $F(u)$

SNESSetJacobian Specify the function to be called to create the Jacobian matrix.

SNESSetFromOptions Set SNES parameters from the commandline

VecSet Set all elements of a vector to the same value

SNESSolve Solve the system of nonlinear equations. Return the number of iteratoins in `its`

SNESDestroy Free the SNES context and recover space

Using the Command Line Interface

- Easy to control Newton features
  - -snes_type ls
  - -snes_type tr
  - -snes_rtol num (relative convergence tolerance)
- Complete control over solution of Jacobian problem—just use the same commandline parmeters
  - -ksp_type cgs
  - -pc_type asm

Convenience Functions

- PETSc's design makes it relatively easy to layer functionality
- One example is the support for function and Jacobian evaluation on DAs

  DASetLocalFunction Attach a function to a DA

  DASetLocalJacobian Attach a Jacobian to a DA

  SNESDAFormFunction Tell SNES that the function evaluation should use the function on a DA. to provide the function values

  SNESDAComputeJacobian Tell SNES that the Jacobian evaluation should use the Jacobian function on a DA
- The functions provide just the computation applied to the local vector (from the DA, which includes the ghost points)
- *Wrapper* functions provided by DASetLocalFunction and Jacobian handle all of the details of setting up the local vectors and arrays.
- The function passed to DASetLocalFunction has the calling sequence:

```
FormFunctionLocal(DALocalInfo *info,PetscScalar **x,
                  PetscScalar **f,AppCtx *user)
```

Example Local Function I

```
int FormFunctionLocal(DALocalInfo *info,PetscScalar **x,
                      PetscScalar **f,AppCtx *user)
{
  int         ierr,i,j;
  PetscReal   two = 2.0,lambda,hx,hy,hxdhy,hydhx,sc;
  PetscScalar u,uxx,uyy;

  PetscFunctionBegin;

  lambda = user->param;
  hx     = 1.0/(PetscReal)(info->mx-1);
  hy     = 1.0/(PetscReal)(info->my-1);
  sc     = hx*hy*lambda;
  hxdhy  = hx/hy;
  hydhx  = hy/hx;
```

Example Local Function II

```
  /*
     Compute function over the locally owned part of the grid
  */
  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];
        uxx     = (two*u - x[j][i-1] - x[j][i+1])*hydhx;
        uyy     = (two*u - x[j-1][i] - x[j+1][i])*hxdhy;
        f[j][i] = uxx + uyy - sc*PetscExpScalar(u);
      }
    }
  }

  ierr = PetscLogFlops(11*info->ym*info->xm);CHKERRQ(ierr);
  PetscFunctionReturn(0);
}
```

Time Stepping Solvers

PETSc can solve time-dependent equations of the form

$$\frac{\partial u}{\partial t} = F(U, t)$$

by making use of the TS (timestepping solvers). F may be linear in U (i.e., of the form AU or $A(t)U$) or nonlinear, and may involve derivatives Two classic examples are

$$U_t = \kappa \nabla^2 U \qquad \text{Heat equation}$$

$$U_t = UU_x + \epsilon U_{xx} \qquad \text{Burger's equation}$$

Features of Timestepping Solvers

PETSc's timestepping solvers are layered over the SLES and SNES solvers

- Full access to all parameters for the linear and nonlinear solvers
- Distributed arrays available for managing regular meshes

Following the other solvers, the TS solvers complete control of the solution process. Commandline options include

- -ts_max_steps, -ts_type beuler, -ts_view

Key Routines

See petsc-tut/frompetsc/heat-eqn.c or petsc-2.1.5/src/ts/examples/tutorials/ex4.c for some examples

TSCreate Create a Time Stepping context.

TSSetProblemType Set the problem type. Use

- **TS_LINEAR** for $U_t = AU$ or $U_t = A(t)U$
- **TC_NONLINEAR** for $U_t = F(t, U)$

TSSetRHSMatrix Defines the matrix A or $A(t)$ (TS_LINEAR only)

TSSetInitialTimeStep Set the initial time and timestep

TSSetSolution Set the initial solution (U at the initial time)

TSSetDuration Set the maximum time and number of time steps

TSSetFromOptions Like all other PETSc objects

TSSetType Specify the algorithm to use. May be one of TS_EULER, TS_BEULER, TS_PSEUDO, and (if installed) TS_PVODE

TSStep Step until the maximum time or time steps is reached

Extending PETSc

- KSP Convergence test
- Matrix-free Solvers (adding a matrix)
- Adding a custom precondioner
- Letting Petsc know about a custom preconditioner

Changing the Convergence Test

Most operations in PETSc are implemented by calling a function for that operation.

Most functions can be replaced, with a `<Object>Set<operation>`. For example, the convergence test for the Krylov method used in a SLES solve can be replaced:

```
MyConvData convdata;
SLESGetKSP( sles, &ksp );
KSPSetConvergenceTest( ksp, MyConvTest, &convdata )
```

The following example implements a test based on $\|W(Ax - b)\|_2$, where W is a diagonal matrix of weights.

Weighted Convergence Test I

```
#include <math.h>
#include "petscsles.h"

typedef struct {
    double ttol, rnorm0;
    Vec    weight;
} MyConvData;

int MyConvTest( KSP ksp, int it, PetscReal rnormUnweighted,
                KSPConvergedReason *reason, void *convdata )
{
    Vec       V, WV;
    PetscReal rtol, atol, dtol;
    double    rnorm;
    int       maxits;
    MyConvData *cdata = (MyConvData *)convdata;
```

Weighted Convergence Test II

```
*reason = KSP_CONVERGED_ITERATING; /* Continue iterating */

KSPBuildResidual( ksp, 0, 0, &V );
/* Scale the residual vector */
VecDuplicate( V, &WV );
VecPointwiseMult( cdata->weight, V, WV );
/* Compute the norm */
VecNorm( WV, NORM_2, &rnorm );
VecDestroy( V ); VecDestroy( WV );

KSPGetTolerances( ksp, &rtol, &atol, &dtol, &maxits );
if (it == 0) {
    /* save the initial values */
    cdata->ttol   = fmax( rtol*rnorm,atol );
    cdata->rnorm0 = rnorm;
}
```

Weighted Convergence Test III

```
    /* The following is essentially the code from the
       default test, KSPDefaultConverged */
    if (rnorm <= cdata->ttol) {
        if (rnorm < atol) {
            *reason = KSP_CONVERGED_ATOL;
        } else {
            *reason = KSP_CONVERGED_RTOL;
        }
    } else if (rnorm >= dtol*cdata->rnorm0) {
        *reason = KSP_DIVERGED_DTOL;
    } else if (rnorm != rnorm) { /* NaN */
        *reason = KSP_DIVERGED_DTOL;
    }

    return 0;
}
```

Matrix-Free Solvers

You can create your own PETSc matrix with

```
MatCreateShell( MPI_Comm comm, int localRows, int localCols,
                int globalRows, int globalCols, void *mctx,
                Mat *A );
```

followed by

```
MatShellSetOperation( Mat A, MatOperation op, void (*f)(void) );
```

For example

```
MatShellSetOperation( A, MATOP_MULT, MyMatV );
```

tells Petsc to call MyMatV when performing a matrix-vector product with A.

Creating A New Preconditioner for SLES

To create a new preconditioner, follow these steps. The routines "myPCMult" and "myPCSetup" implement $y \leftarrow Mx$ and the initialization of the preconditioner M.

```
PC pc;
SLESGetPC( sles, &pc );
PCSetType( pc, PCSHELL );
PCShellSetName( pc, "MyPreconditioner" );
PCShellSetApply( pc, myPCMult, &pcdata );
PCShellSetSetUp( pc, myPCSetup ); /* Optional (e.g., for ILU
                                     factorization */
```

Example PC I

Compute $Mx - \frac{(Mx)^T w}{w^T w} w$ where $w = \{1, 1, 1, \ldots, 1\}^T$ (project off the component of all ones, e.g., for a problem where $Aw = 0$):

```
#include "petscpc.h"

typedef struct {
    Mat m;
} MyPCData;
int myPCMult( void *ctx, Vec xin, Vec xout )
{
    Vec ones;
    int size;
    double one = 1, r, scale;
```

Example PC II

```
    MyPCData *pcdata = (MyPCData *)ctx;

    MatMult( pcdata->m, xin, xout );
    VecDuplicate( xin, &ones );
    VecGetSize( xin, &size );
    VecSet( &one, ones );
    VecDot( ones, xout, &r );
    scale = r / size;
    VecAXPY( &scale, ones, xout );

    VecDestroy( ones );
}
```

Adding Your Precondioner to PETSc

```
PCRegister( "MyPreconditioner", 0, "PCMyPreconditioner", MyPCCreate );
```

where

```
typedef struct { ... } MyPCData;
int MyPCCreate( PC pc )
{
   MyPCData *pcdata;
   PetscNew( MyPCData, &pcdata );
   pc->data                = (void *)pcdata;
   pc->ops->apply          = myPCMult;
   pc->ops->setup          = 0;
   pc->ops->destroy        = myPCDestroy;
   pc->ops->setfromoptions = myPCFromOptions;
   ...
}
```

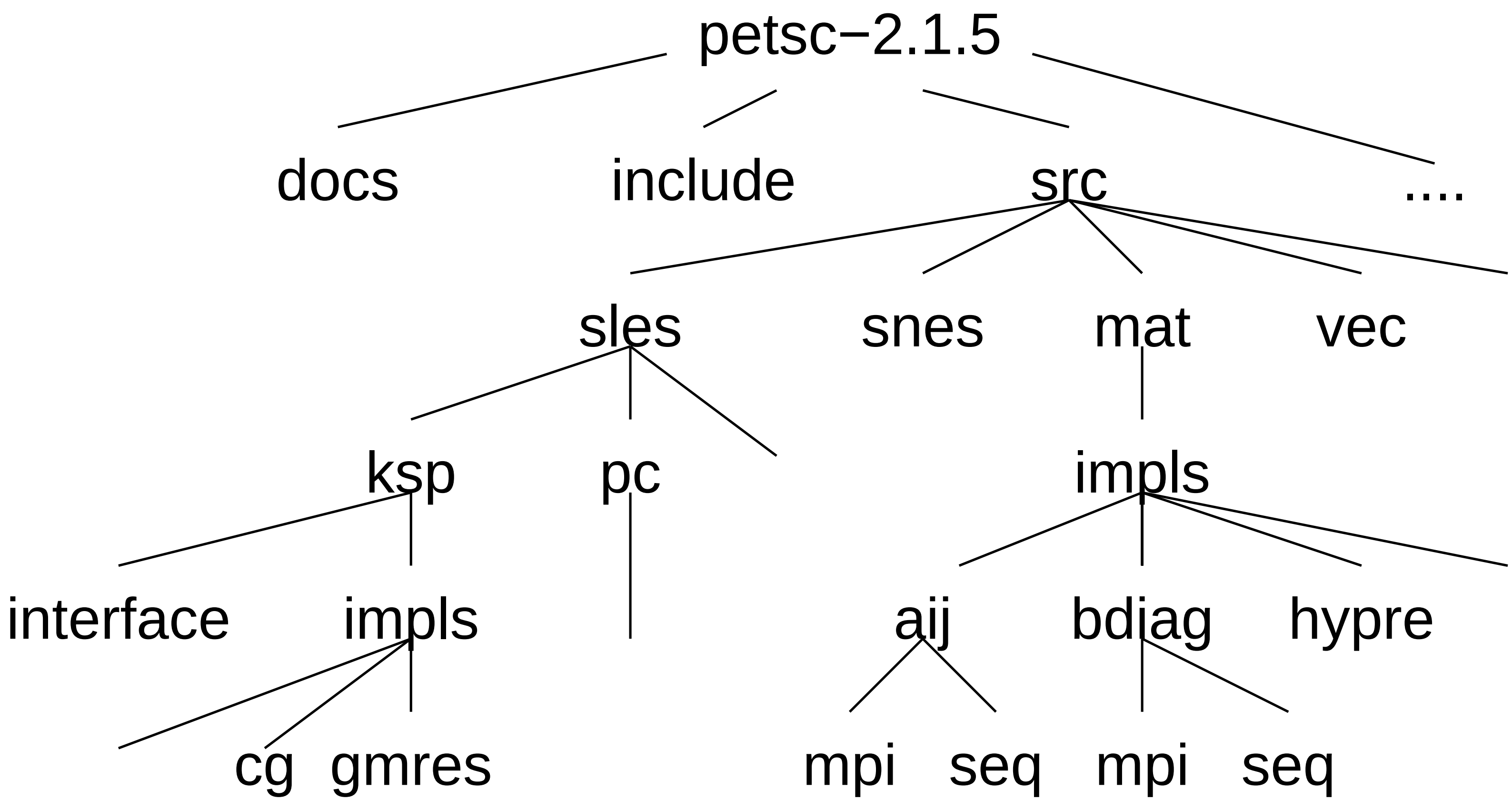
(To build this, look at an example such as petsc/src/sles/pc/impls/jacobi.c). Then

```
  poisson -pc_type MyPreconditioner
```

will use your new preconditioner!

Use the Source!

To find out more about PETSc, look at the implementation. PETSc has a regular directory structure:



Some Applications

PETSc includes examples of some applications:

- Driven cavity (snes/.../ex19.c)
- MHD (snes/.../ex29.c)
- Radiative transport (snes/.../ex18.c)

Nonlinear Solvers Examples

ex1.c: Newton's method to solve a two-variable system, sequentially

ex2.c: Newton method to solve $u_{xx} + u^2 = f$, sequentially

ex3.c: Newton methods to solve $u_{xx} + u^2 = f$ in parallel

ex5.c: Bratu nonlinear PDE in 2d

ex5s.c: 2d Bratu problem in shared memory parallel with SNES

ex6.c: $u_{xx} + u^2 = f$

ex14.c: Bratu nonlinear PDE in 3d

ex18.c: Nonlinear Radiative Transport PDE with multigrid in 2d

ex19.c: Nonlinear driven cavity with multigrid in 2d

ex20.c: Nonlinear Radiative Transport PDE with multigrid in 3d

ex21.c: Solves PDE optimization problem

ex22.c: Solves PDE optimization problem

ex23.c: Solves PDE problem from ex22

ex24.c: Solves PDE optimization problem of ex22

ex25.c: Minimum surface problem

ex26.c: Grad-Shafranov solver for one dimensional CHI equilibrium

Driven Cavity

The problem

$$
\begin{aligned}
-\nabla^2 u - \frac{\partial \omega}{\partial y} &= 0, \\
-\nabla^2 v + \frac{\partial \omega}{\partial x} &= 0, \\
-\nabla^2 \omega + u\frac{\partial \omega}{\partial x} + v\frac{\partial \omega}{\partial y} - \text{Gr}\frac{\partial T}{\partial x} &= 0, \\
-\nabla^2 T + \text{Pr}\left(u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y}\right) &= 0,
\end{aligned}
$$

with velocity $= (u, v)$, vorticity ω, and temperature T.

Boundary conditions

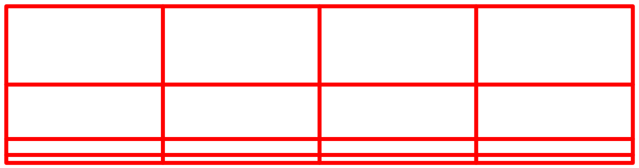
$$
\begin{aligned}
\text{bottom:} \quad u &= v = 0, \frac{\partial T}{\partial y} = 0, \\
\text{top:} \quad u &= V_{lid}, v = 0, \frac{\partial T}{\partial y} = 0, \\
\text{left:} \quad u &= v = 0, T = 0 \\
\text{right:} \quad u &= v = 0, T = 1 \text{ if Gr} > 0, \\
& T = 0 \text{ otherwise}
\end{aligned}
$$

with $\omega = -\partial u/\partial y + \partial y/\partial x$ along the boundary.

This is the velocity-vorticity formulation

Notes on the Discretization

- The examples use a very simple discretization to concentrate on the use of PETSc to solve the system of nonlinear equations
- Improving the discretization is relatively easy
  - Higher order discretizations can be used by increasing the stencil width
  - Because the DA is a *logical* mesh, it is easy to put more mesh points along the boundaries:



    (But you must manage the discretization.)
  - PETSc provides support for general sparse matrices:
    - Index sets (ISxxx routines); VecScatter and VecGather
    - Access to matrix partitioning for parallelism

Driven Cavity Example I

- Try matrix-free Jacobian approximation with no preconditioning (via `-snes_mf`)
  - 1 process: (thermally-driven flow)

    ```
    ex19 -snes_mf -snes_monitor -grashof 1000.0 -lidvelocity 0.0
    ```

  - 2 processes, view DA

    ```
    mpirun -np 2 ex19 -snes_mf -snes_monitor \
    -da_view_draw -draw_pause 1
    ```

  - View contour plots of converging iterates

    ```
    ex19 -snes_mf -snes_monitor -snes_vecmonitor
    ```

Driven Cavity Example II

- Use MatFDColoring for sparse finite difference Jacobian approximation; view SNES options used at runtime

```
ex19 -snes_view -mat_view_info
```

- Set trust region Newton method instead of default line search

```
ex19 -snes_type tr -snes_view -snes_monitor
```

- Set transpose-free QMR as the Krylov method and set relative KSP convergence tolerance to 0.01

```
ex19 -ksp_type tfqmr -ksp_rtol 0.01 -snes_monitor
```

PETSc Programming Aids

- Correctness Debugging
  - Automatic generation of tracebacks
  - Detecting memory corruption and leaks
  - Optional user-defined error handlers
  - Differential debugging
- Performance Debugging
  - Integrated profiling using `-log_summary`
  - Profiling by stages of an application
  - User-defined events

Debugging Tools

- Error handlers
- Many useful commandline options:

  -start_in_debugger

  -on_error_attach_debugger name

  -on_error_abort

  You may also need `-display $DISPLAY` or `-display ‘hostname‘:0.0` to get the separate debugger windows to appear. Also, placing a breakpoint in PetscError will often give you control when PETSc fi rst detects an error.

Performance Tuning

- Limits of performance
- Finding problems
  - Built-in timing information
  - Adding user-specifi ed states
  - Pitfalls
- Using PETSc features
  - Better data structures
  - Aggregate operations
- Making best use of C or Fortran

Limits of Performance

- Real systems have many levels of memory
  - Programming models try to hide memory hierarchy
- Simplest model: Two levels of memory
  - Divide at the largest (relative) gap
  - Processes have their own memory
    - Managing a processes memory is known (if unsolved) problem
  - Exactly matches the distributed memory model
- But even the single process job is often bound by memory performance

Sparse Matrix-Vector Product

- Common operation for optimal (in floating-point operations) solution of linear systems
- Sample code

```
for row=0,n-1
    m = i[row+1] - i[row];
    sum = 0;
    for k=0,m-1
        sum += *a++ * x[*j++];
    y[row] = sum;
```

- Data structures are a[nnz], j[nnz], i[n], x[n], y[n]

Simple Performance Analysis

- Memory motion:
  - nnz (sizeof(double) + sizeof(int)) +
    n (2*sizeof(double) + sizeof(int))
  - Perfect cache (never load same data twice)
- Computation:
  - nnz multiply-add (MA)
- Roughly 12 bytes per MA
- Typical workstation node can move $\frac{1}{2}$–4 bytes/MA
  - *Maximum* performance is 4–33% of peak

More Performance Analysis

- Instruction counts:
  - nnz (2 * load-double + load-int + mult-add) + n (load-int + store-double)
- Roughly 4 instructions per multiply-add
- Maximum performance is 25% of peak (33% if MA overlaps one load or store)
- Changing the matrix data structure (e.g., exploit small block structure) allows some reuse of data in register, eliminating some loads (of x and j)
- Implementation improvements (tricks) cannot improve on these limits

Why use BAIJ?

The BAIJ format can provide added performance:

Format	Mflops	
	Ideal	Achieved
AIJ	49	45
BAIJ	64	55

These results, from a 250 MHz R10000, are for matrices with a natural blocksize of four.

Multiple right-hand sides show much greater improvement, if you can take advantage of them.

See “Toward Realistic Performance Bounds for Implicit CFD Codes,” in the proceedings of Parallel CFD’99 (preprint also available at `www.mcs.anl.gov/~gropp/bib/papers/1999/pcfd99/gkks.ps`)

Finding Performance Problems

- PETSc provides built-in tools to measure and report on performance

  -log_summary Provides a breakdown by routine of each PETSc routine

  -log_info Provides information on object use

  -log_trace Trace the execution of each PETSc routine

- Make sure that you use an optimized version of PETSc (BOPT=O) and that you have avoided “cold start” problems.
  - PETSc provides PreLoadBegin, PreLoadStage, and PreLoadEnd to help. This make it easy to ensure that a test is run once to get memory “warmed up” and that timings are taking from a second test.

Example log_summary Output I

```
/home/gropp/projects/software/petsc-tut/src/sles/poisson2 on a win32_gnu nam
Using Petsc Version 2.1.5, Patch 0, Released Jan 27, 2002

                         Max       Max/Min        Avg      Total
Time (sec):           7.709e-02      1.19728   6.854e-02
Objects:              0.000e+00      0.00000   0.000e+00
Flops:                1.735e+04      1.56176   1.422e+04  5.687e+04
Flops/sec:            2.694e+05      1.59360   2.073e+05  8.293e+05
Memory:               8.410e+04      1.06257              3.264e+05
MPI Messages:         3.000e+01      1.87500   2.350e+01  9.400e+01
MPI Message Lengths:  2.442e+03      1.90484   8.183e+01  7.692e+03
MPI Reductions:       4.450e+01      1.00000
```

Example log_summary Output II

```
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %F %M %L %R  %T %F %M %L %R Mflop
------------------------------------------------------------------------------------------------------------------------

--- Event Stage 0: Main Stage

VecMDot               12 1.0 4.7554e-03 3.7 2.38e+06 3.7 0.0e+00 0.0e+00 1.2e+01  5 27  0  0  7   5 27  0  0  7     3
VecNorm               13 1.0 5.2183e-03 1.7 2.58e+05 2.6 0.0e+00 0.0e+00 1.3e+01  6  5  0  0  7   6  5  0  0  7     0
VecScale              13 1.0 4.4698e-05 1.3 1.13e+07 1.9 0.0e+00 0.0e+00 0.0e+00  0  2  0  0  0   0  2  0  0  0    29
VecSet                15 1.0 5.3079e-05 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAXPY                1 1.0 1.0895e-05 1.6 8.95e+06 2.4 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0    18
VecMAXPY              13 1.0 7.3752e-05 1.2 8.52e+07 1.7 0.0e+00 0.0e+00 0.0e+00  0 32  0  0  0   0 32  0  0  0   244
VecScatterBegin       13 1.0 1.4695e-04 1.3 0.00e+00 0.0 7.2e+01 8.0e+01 0.0e+00  0  0 77 75  0   0  0 77 75  0     0
VecScatterEnd         13 1.0 4.4182e-03 2.2 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  5  0  0  0  0   5  0  0  0  0     0
MatMult               12 1.0 4.7492e-03 2.0 1.23e+06 2.8 7.2e+01 8.0e+01 0.0e+00  5 17 77 75  0   5 17 77 75  0     2
MatSolve              13 1.0 1.1091e-04 1.3 2.56e+07 1.5 0.0e+00 0.0e+00 0.0e+00  0 16  0  0  0   0 16  0  0  0    82
MatLUFactorNum         1 1.0 3.5479e-05 1.3 5.46e+06 1.8 0.0e+00 0.0e+00 0.0e+00  0  1  0  0  0   0  1  0  0  0    15
MatILUFactorSym        1 1.0 1.9489e-03 1.9 0.00e+00 0.0 0.0e+00 0.0e+00 7.0e+00  2  0  0  0  4   2  0  0  0  4     0
MatAssemblyBegin       2 1.0 2.0382e-03 1.9 0.00e+00 0.0 0.0e+00 0.0e+00 4.0e+00  2  0  0  0  2   2  0  0  0  2     0
MatAssemblyEnd         2 1.0 4.9942e-03 1.3 0.00e+00 0.0 6.0e+00 4.0e+01 2.0e+01  6  0  6  3 11   6  0  6  3 11     0
MatGetOrdering         1 1.0 6.5651e-04 1.3 0.00e+00 0.0 0.0e+00 0.0e+00 4.0e+00  1  0  0  0  2   1  0  0  0  2     0
PCSetUp                2 1.0 5.7393e-03 1.2 3.24e+04 1.7 0.0e+00 0.0e+00 2.2e+01  8  1  0  0 12   8  1  0  0 12     0
PCSetUpOnBlocks        1 1.0 2.5428e-03 1.4 7.31e+04 1.6 0.0e+00 0.0e+00 1.1e+01  3  1  0  0  6   3  1  0  0  6     0
PCApply               13 1.0 6.9981e-04 1.1 4.51e+06 1.7 0.0e+00 0.0e+00 0.0e+00  1 16  0  0  0   1 16  0  0  0    13
KSPGMRESOrthog        12 1.0 4.9157e-03 3.4 4.21e+06 3.4 0.0e+00 0.0e+00 1.2e+01  5 54  0  0  7   5 54  0  0  7     6
SLESSetup              2 1.0 8.8296e-03 1.2 2.11e+04 1.7 0.0e+00 0.0e+00 3.2e+01 12  1  0  0 18  12  1  0  0 18     0
SLESSolve              1 1.0 1.8024e-02 1.0 9.83e+05 1.6 7.2e+01 8.0e+01 4.5e+01 26 99 77 75 25  26 99 77 75 25     3
```

Adding User Events

It is easy to add user defi ned events to PETSc

```
int USER_EVENT;
PetscLogEventRegister(&USER_EVENT,"User event");
PetscLogEventBegin(USER_EVENT,0,0,0,0);
   [code segment to monitor]
   PetscLogFlops(user_flops)
PetscLogEventEnd(USER_EVENT,0,0,0,0);
```

“USER_EVENT” is returned by PETSc (instead of allowing you to defi ne it) so that many routines can defi ne user events without any possibility of two routines unintentionall using the same event value.

Obtaining Higher Performance with PETSc

- Often, the most important step is to make use of “aggregate operations” wherever possible. That is, use one routine that performs multiple operations, instead of multiple calls to a single routine.
  - For setting the elements of a matrix or vector, use MatSetValues and VecSetValues instead of MatSetValue and VecSetValue
  - MatSetValuesBlocked inserts submatrices
  - Same technique uses in parallel programming (both message-passing and shared-memory)
- Consider other sparse data structures, particularly BAIJ and Bdiag
- Those mysterious parameters (like `DIFFERENT_NONZERO_PATTERN`) can be very important. PETSc tries to provide a *correct* solution first
  - As a result, PETSc is more cautious that other environments
  - Setting these parameters correctly can make a huge difference in performance

Setting Multiple Matrix Values

Petsc provides several routines to add multiple entries at a time to a matrix:

```
MatSetValues( Mat mat, int nrows, int rowidx[],
              int ncols, int colidx[], PetscScalar vals[],
              INSERT_VALUES or ADD_VALUES )
MatSetValuesBlocked( ... )  same, but for blocked matrices
```

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 pre-allocation provides the freedom of dynamic data structures plus good performance

Indicating Expected Nonzeros

- For parallel sparse matrices

```
MatCreateMPIAIJ(..., int d_nz,
                const int d_nnz[], int o_nz,
                const int o_nnz[], Mat *A)
```

where

d_nnz expected number of nonzeros per row in diagonal portion of local submatrix. The “diagonal portion” is the square diagonal block of the rows owned by this process.

o_nnz expected number of nonzeros per row in off-diagonal portion of local submatrix

Verifying Predictions

Use runtime option: -log_info

```
[0]MatSetUpPreallocation: Warning not preallocating matrix storage
[0]MatAssemblyBegin_MPIAIJ:Stash has 0 entries, uses 0 mallocs.
[0]MatAssemblyEnd_SeqAIJ:Matrix size: 50 X 50; storage space: 50 unneeded,20
[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 50 nodes out of 50 rows. Not using Inode routin
[1]MatAssemblyBegin_MPIAIJ:Stash has 0 entries, uses 0 mallocs.
[1]MatAssemblyEnd_SeqAIJ:Matrix size: 50 X 50; storage space: 50 unneeded,20
[1]MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues() is 0
[1]MatAssemblyEnd_SeqAIJ:Most nonzeros in any row is 5
[1]Mat_AIJ_CheckInode: Found 50 nodes out of 50 rows. Not using Inode routin
[1]MatAssemblyEnd_SeqAIJ:Matrix size: 50 X 10; storage space: 90 unneeded,10
[1]MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues() is 0
[1]MatAssemblyEnd_SeqAIJ:Most nonzeros in any row is 1
[1]Mat_AIJ_CheckInode: Found 18 nodes of 50. Limit used: 5. Using Inode rout
[0]MatAssemblyEnd_SeqAIJ:Matrix size: 50 X 10; storage space: 90 unneeded,10
[0]MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues() is 0
```

Making the Best Use of C

- C2000 has features to allow compiles to optimize memory use

  `const` Data is constant (cannot change because of a store through another pointer)

  `restrict` Data is accessed only through this pointer

- These allow Fortran-like argument semantics, allowing a sophisticated compiler to produce code as good as Fortran allows

```
int dadd( double * restrict a,
          const double * restrict b, int n )
```

- Benefi t depends on compiler and system. Small on most PC's; factor of ten (!) on one vector machine.

Making the Best Use of Fortran

- Order array elements so that related references are first

```
double precision vars(2,100,100)
```

not

```
double precision u(100,100), v(100,100)
```

Benefit of Reordering

For the Fun3d CFD code, changing the order of arrays provided a factor of seven (!) improvement

Time on an IBM SP with different orderings, starting with original (Basic) code.

Basic	Interlaced	Interlaced Blocking	Interlaced Reordered	All
103.8	45.9	32	26.9	14.9

Conclusion

- PETSc provides a powerful framework for
  - Developing applications
  - Experimenting with different algorithms
  - Using abstractions to simplify parallel programming
- PETSc continues to grow and develop
  - New routines added as needed *and understood*
  - PETSc 3 will provide a more powerful framework for combining tools written in different programming languages

References

- Documentation `www.mcs.anl.gov/petsc/docs`
  - PETSc Users Manual
  - Manual pages (the most up-to-date)
  - Many hyperlinked examples
  - FAQ, Troubleshooting info, installation info, etc.
- Publications `www.mcs.anl.gov/petsc/publications`
  - Research and publications that make use of PETSc
- MPI information `www.mpi-forum.org`
- ***Using MPI*** (2nd Edition), by Gropp, Lusk, and Skjellum
- ***Domain Decomposition***, by Smith, Björstad, and Gropp

Topics Not Covered

- PETSc contains many features, each introduced to provide a necessary feature for an application or researcher
  - Unstructured Meshes
  - Matrix free methods
  - Access to other packages
  - Using different preconditioner matrices
  - Others

Using PETSc with Other Packages

- Linear solvers
  - AMG
    `www.mgnet.org/mgnet-codes-gmd.html`
  - BlockSolve95
    `www.mcs.anl.gov/BlockSolve95`
  - ILUTP
    `www.cs.umn.edu/~saad`
  - LUSOL
    `www.sbsi-sol-optimize.com`
  - SPAI
    `www.sam.math.ethz.ch/~grote/spai`
  - SuperLU
    `www.nersc.gov/~xiaoye/SuperLU`
- Optimization software
  - TAO
    `www.mcs.anl.gov/tao`
  - Veltisto
- Mesh and discretization tools
  - Overture
    `www.llnl.gov/CASC/Overture`
  - SAMRAI
    `www.llnl.gov/CASC/SAMRAI`
  - SUMAA3d
    `www.mcs.anl.gov/sumaa3d`
- ODE solvers
  - PVODE
    `www.llnl.gov/CASC/PVODE`
- Others
  - Matlab `www.mathworks.com`
  - ParMETIS
    `www.cs.umn.edu/~karypis/metis/parm`

Changing the Behavior of Viewer

- Change the standard viewer to output in cannonical order (independent of the number of processes)
- Change the behavior of the standard viewer *Danger!*

```
PetscViewerSetFormat( PETSC_VIEWER_STDOUT_WORLD,
                      PETSC_VIEWER_ASCII_COMMON );
VecView( vec, PETSC_VIEWER_STDOUT_WORLD )
```

- Change temporarily the behavior of the standard viewer

```
PetscViewerPushFormat( PETSC_VIEWER_STDOUT_WORLD,
                       PETSC_VIEWER_ASCII_COMMON );
VecView( vec, PETSC_VIEWER_STDOUT_WORLD )
PetscViewerPopFormat( PETSC_VIEWER_STDOUT_WORLD );
```

Procedural Interface for Options

- All PETSc features that can be set with command-line options can be controlled from within a program.
- Routines to do so are often named `<Object>Set<feature>`, as in KSPSetMonitor or PCLUSetMatOrdering

Some Vector Operations

Function	Operation
VecAXPY(Scalar *a, Vec x, Vec y)	$y \leftarrow y + ax$
VecAYPX(Scalar *a, Vec x, Vec y)	$y \leftarrow x + ay$
VecWAXPY(Scalar *a, Vec x, Vec y, Vec w)	$w \leftarrow ax + y$
VecScale(Scalar *a, Vec x)	$x \leftarrow ax$
VecCopy(Vec x, Vec y)	$y \leftarrow x$
VecPointwiseMult(Vec x, Vec y, Vec w)	$w_i \leftarrow x_i y_i$
VecMax(Vec x, int *idx, Scalar *r)	$r \leftarrow \mathrm{max}_i(x_i)$
VecNorm(Vec x, NormType type, double *r)	$r \leftarrow \|x\|_{normtype}$
VecSet(Scalar *a, Vec x)	$x_i = a$

This is just a sample; there are more. Check the manual page index under “V”.

PETSc Components

Nonlinear Solvers		
Newton-based Methods		Other
Line Search	Trust Region	

Time Steppers			
Euler	Backward Euler	Pseudo Time Stepping	Other

Krylov Subspace Methods							
GMRES	CG	CGS	Bi-CG-STAB	TFQMR	Richardson	Chebychev	Other

Preconditioners						
Additive Schwarz	BlockJacobi	Jacobi	ILU	ICC	LU(Sequential only)	Others

Matrics					
Compressed Sparse Row (AIJ)	Blocked Compressed Sparse Row (BAIJ)	Block Diagonal (BDIAG)	Dense	Matrix-free	Other