Solving PDEs with PETSc

William Gropp

University of Illinois at Urbana-Champaign
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
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
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
  - 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 XTn, IBM BlueGene, SGI Altix, SiCortex, Sun, ... 
- Loosely coupled systems, e.g., networks of workstations
  HP, IBM, SGI, Sun, and PCs running Linux or Windows, and
  Apple Macs

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)
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?
#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

```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
```
• Like the C code, except
  – PetscInitialize has fewer arguments because Fortran has no argc or argv
  – Must use MPI_Commm_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)
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.
#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

Let's solve a simple linear elliptic PDE

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

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.
In PETSc, your main program remains in control:

```
main program
  PetscInitialize()
  A = create the matrix
  b = create a vector
  any other application code
  Use KSP to solve A x = b
  print solution
  PetscFinalize()
```

KSP is the “scalable linear equation solver” component of PETSc (the name is historical and was originally SLES, and the first “S” was for “simple”)

#include "petscksp.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, &A );
    MatSetSizes( A, PETSC_DECIDE, PETSC_DECIDE, n * n, n * n );
    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;
MatCreate  Create a matrix object.

MatSetSizes  Set the size of the matrix
  - $n^2$ equations, so matrix is of size $n \times 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
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, *automatically* moving data if necessary
- PETSc can optimize the insertion of matrix elements
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?
  - Contiguous rows of a vector or matrix are assigned to processes, starting from the process with rank zero.

- The matrix and vector for a $3 \times 3$ mesh, with two processes, has the following decomposition:

\[
\begin{pmatrix}
  x_0 \\
  x_1 \\
  x_2 \\
  x_3 \\
  x_4 \\
  x_5 \\
  x_6 \\
  x_7 \\
  x_8 \\
\end{pmatrix}
= \begin{pmatrix}
  4 & -1 & -1 \\
  -1 & 4 & -1 & -1 \\
  -1 & 4 & -1 & -1 \\
  -1 & 4 & -1 & -1 \\
  -1 & 4 & -1 & -1 \\
  -1 & 4 & -1 & -1 \\
  -1 & 4 & -1 & -1 \\
  -1 & 4 & -1 & -1 \\
\end{pmatrix}
\]
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)
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

  \[
  \text{for (i=0; i<n; i++) v[i] = i;}
  \]

- 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.
#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 );
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.

- **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.
#include <math.h>
#include "petscksp.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[] )
{
    KSP sles;
    Mat A;
    Vec b, x;
    int its, n;

    PetscInitialize( &argc, &argv, 0, 0 );
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 );
KSPCreate( PETSC_COMM_WORLD, &sles );
KSPSetOperators( sles, A, A, DIFFERENT_NONZERO_PATTERN );
KSPSetFromOptions( sles );
KSPSolve( sles, b, x );
KSPGetIterationNumber( sles, &its );
PetscPrintf( PETSC_COMM_WORLD, "Solution in %d iterations is:\n" 
VecView( x, PETSC_VIEWER_STDOUT_WORLD );

MatDestroy( A ); VecDestroy( b ); VecDestroy( x );
KSPDestroy( sles );
PetscFinalize( );
return 0;
Understanding the Code

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

**KSPSetOperators**  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.

**KSPSetFromOptions**  Set the algorithm, preconditioner, and the associated parameters, using the command-line.

**KSPSolve**  Actually solve the system of linear equations.

**KSPGetIterationNumber**  The number of iterations is returned. If a direct method is used, one is returned in its.

**KSPDestroy**  Free the KSP 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
    ```c
    mpiaijgmres( ia, ja, a, comm, x, b, nlocal, nglobal, ndir, orthomethod, convtol, &its )
    ```
  – New way
    ```c
    KSPSolve( ksp, b, x )
    KSPGetIterationNumber( ksp, &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)
Each KSP object contains two important objects:

**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 fill level
### Available Methods

<table>
<thead>
<tr>
<th>KSP Name</th>
<th>PETSc Option</th>
<th>PC Name</th>
<th>PETSc Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conjugate Gradient</td>
<td>cg</td>
<td>Block Jacobi</td>
<td>bjacobi</td>
</tr>
<tr>
<td>GMRES</td>
<td>gmres</td>
<td>Overlapping Additive Schwarz</td>
<td>asm</td>
</tr>
<tr>
<td>Bi-CG-stab</td>
<td>bicg</td>
<td>ILU</td>
<td>ilu</td>
</tr>
<tr>
<td>Transpose-free QMR</td>
<td>tfqmr</td>
<td>SOR</td>
<td>sor</td>
</tr>
<tr>
<td>Richardson</td>
<td>richardson</td>
<td>LU (direct solve)</td>
<td>lu</td>
</tr>
<tr>
<td>CG-Squared</td>
<td>cgs</td>
<td>Multigrid</td>
<td>mg</td>
</tr>
<tr>
<td>SYMMLQ</td>
<td>symmlq</td>
<td>Arbitrary matrix</td>
<td>mat</td>
</tr>
<tr>
<td>others</td>
<td></td>
<td>others</td>
<td></td>
</tr>
</tbody>
</table>
• 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_monitor_draw** Plot the preconditioned residual norm
  - **-ksp_monitor_true_residual_norm** Print the true residual norm $\|Ax - b\|_2$
  - **-ksp_monitor_draw_true_residual** Plot the true residual norm
- Custom monitors can be defined 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.
- We illustrate this with a routine to compute the norm of $\| x + ay \|$. 
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.
```c
#include "petscvec.h"

/* Comment out this next define if the compiler is not broken and supports C2000 */
#define restrict

/* This is a new vector routine for PETSc, illustrating 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 );
```
20 VecGetLocalSize( x, &nlocal );
21
22 if (a == -1) {
23   /* Special case for difference of two vectors */
24   switch (ntype) {
25     case NORM_1:
26       for (i=0; i<nlocal; i++) {
27         sum += fabs(xvals[i] - yvals[i]);
28       }
29     normop = MPI_SUM;
30     break;
31     case NORM_2:
32       for (i=0; i<nlocal; i++) {
33         register PetscScalar tmp;
34         tmp = xvals[i] - yvals[i];
35         sum += tmp*tmp;
36       }
37     normop = MPI_SUM;
38     break;
39  case NORM_INFINITY:
40      for (i=0; i<nlocal; i++) {
41         register PetscScalar tmp;
42         tmp = fabs(xvals[i] - yvals[i]);
43         if (tmp > sum) sum = tmp;
44      }
45      normop = MPI_MAX;
46     break;
47  default:
48      ierr = 1;
49     break;
50  }
51 }
52  else {
53     /* Unimplemented */
54     ierr = 1;
55  }
if (!ierr) {
    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.
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:

<table>
<thead>
<tr>
<th></th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>P1</td>
<td></td>
</tr>
</tbody>
</table>
On this \(2 \times 2\) process grid, the vector elements are numbered like this:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Natural numbering

PETSc’s internal numbering

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

- PETSc's distributed array (DA) provides a way to describe a multidimensional array, distributed across a parallel computer.
- DAs provide a way to use more complex data decompositions.

```c
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.
#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 );

18          DAVecGetArray( grid, V, (void **) &vval );
19          /* Get global coordinates of this patch in the DA grid */
20          DAGetCorners( grid, &is, &js, 0, &in, &jn, 0 );
21          ie = is + in - 1;
22          je = js + jn - 1;
23          for (i=is ; i<=ie ; i++) {
24              for (j=js ; j<=je ; j++){
25                  vval[j][i] = (*f)( (i + 1) * h, (j + 1) * h );
26              }
27          }
28          DAVecRestoreArray( grid, V, (void **) &vval );
29          return V;
30      }
31  }
32
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.
#include "petscksp.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 */
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;
        }
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 specified; this chooses a parallel matrix using AIJ format (MATMPIAIJ).

**MatSetValuesStencil** Sets elements of a matrix using grid coordinates

**MatStencil** Data structure that contains the indices of a point in the DA, using the $i, j, k$ members of the structure
#include <math.h>
#include "petscksp.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[] )
{
    KSP           ksp;
    Mat           A;
    Vec           b, x;
    DA            grid;
    int           its, n, px, py, worldSize;
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 */
A = FormLaplacianDA2d( grid, n );
b = FormVecFromFunctionDA2d( grid, n, func );
VecDuplicate( b, &x );
KSPCreate( PETSC_COMM_WORLD, &ksp );
KSPSetOperators( ksp, A, A, DIFFERENT_NONZERO_PATTERN );
KSPSetFromOptions( ksp );
KSPSolve( ksp, b, x );
KSPGetIterationNumber( ksp, &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 );
KSPDestroy( ksp ); DADestroy( grid );
PetscFinalize( );
return 0;
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`
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
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 $k$th step. The term $F'(u^k)$ is a matrix, and this algorithm can be rewritten as

$$F'(u^k)\Delta u^k = -F(u^k)$$

$$u^{k+1} = u^k + \Delta u^k$$
Newton-based Methods

In practice, various modifications are made to Newton’s method. PETSc supports many of the most common:

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

PETSc provides a “Simplified Nonlinear Equation Solver” (SNES) for nonlinear problems. SNES is the nonlinear analogue of KSP.
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 finite 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

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

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 difficult 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

  \[ \text{MatMult}(A, x, y); \]

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

The routine `DAGetGhostCorners` returns this information, similar to `DAGetCorners`
**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. The update may use INSERT_VALUES or ADD_VALUES.

**DALocalToGlobal** Transfers values in the local representation back to the global representation. The ghost points are discarded.
In the Bratu example,

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

so

\[ F'(u)a = -\nabla^2 a - \lambda ae^u, \]

where \( ae^u \) is just \( \{ a_i \times e^{ui} \} \). 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 SNESolve 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 KSPSetOperators.
- 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

  \[ \text{SNESSetFunction}(\text{snes}, f, \text{userfunc}, \text{userctx}) \]

  \textbf{snes}  SNES context
  \textbf{f}  Vector that will be used to store the function value
  \textbf{userfunc}  Name of (really, pointer to) the function
  \textbf{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

```c
#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 */
```
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++) {
        
    }
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;
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:

```c
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 evaluation of the finite difference scheme:

- Boundary conditions, as always, add complexity
#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  h = bratu->h, lambda = bratu->lambda;
  double oneByh2 = 1.0 / (h*h), **uvals, v[5];
int lli, llj, ni, nj; /* lower left i,j and size for local part of mesh */

MatStencil row, col[5];

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 );
        } 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[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);

return 0;
}
```c
#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 );
```
/* 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 );
SNESolve( snes, x, &its );

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

VecDestroy(r); DADestroy(bratu.da);
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

**SNESolve**  Solve the system of nonlinear equations. Return the number of iterations in its

**SNESDestroy**  Free the SNES context and recover space

**SETERRQ**  The counterpart to CHKERRQ, it sets the error and returns a message
• 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 parameters
  – -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
  - **SNESDADeformJacobian** 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:

```c
FormFunctionLocal(DALocalInfo *info, PetscScalar **x, PetscScalar **f, AppCtx *user)
```
Example Local Function I

```c
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;

    PetscFunctionReturn(ierr);
}
```
/*
  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);
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](http://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.

  - Research and publications that make use of PETSc

- MPI information [www.mpi-forum.org](http://www.mpi-forum.org)

- **Using MPI** (2nd Edition), by Gropp, Lusk, and Skjellum
- **Domain Decomposition**, by Smith, Björstad, and Gropp
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
  - Hypre  
    www.llnl.gov/casc/hypre
  - 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

- **ODE solvers**
  - PVODE  
    www.llnl.gov/CASC/PVODE

- **Mesh and discretization tools**
  - Overture  
    www.llnl.gov/CASC/Overture
  - SAMRAI  
    www.llnl.gov/CASC/SAMRAI
  - SUMAA3d  
    www.mcs.anl.gov/sumaa3d

- **Optimization software**
  - TAO  
    www.mcs.anl.gov/tao
  - Veltisto  
    www.cs.nyu.edu/~biros/veltisto

- **Others**
  - Matlab  
    www.mathworks.com
  - ParMETIS  
    www.cs.umn.edu/~karypis/metis/parmetis
  - SLEPc  
    www.grycap.upv.es/slepc