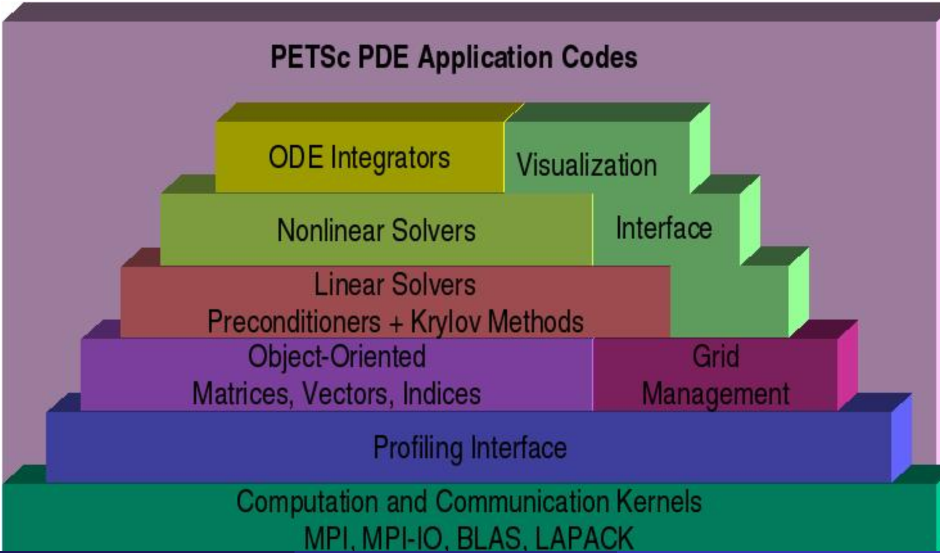
PETSc Structure


PETSc PDE Application Codes
ODE Integrators
Visualization
Nonlinear Solvers
Interface
Linear Solvers
Preconditioners + Krylov Methods
Object-Oriented
Matrices, Vectors, Indices
Grid
Management
Profiling Interface
Computation and Communication Kernels
MPI, MPI-IO, BLAS, LAPACK

The PETSc Programming Model

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

Library Design

Numerical libraries should interact at a higher level than MPI

- MPI coordinates data movement and synchronization for data parallel applications
- Numerical libraries should coordinate access to a given data structure
    - MPI can handle data parallelism and something else (runtime engine) handle task parallelism (van de Geijn, Strout, Demmel)
    - Algorithm should be data structure neutral, but its main operation is still to structure access

Collectivity

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

Initialization

- Call `PetscInitialize()`
  - Setup static data and services
  - Setup MPI if it is not already
  - Can set `PETSC_COMM_WORLD` to use your communicator (can always use subcommunicators for each object)
- Call `PetscFinalize()`
  - Calculates logging summary
  - Can check for leaks/unused options
  - Shutdown and release resources
- Can only initialize PETSc once

Vector Algebra

A PETSc Vec

- Supports all vector space operations
  - `VecDot(), VecNorm(), VecScale()`
- Has a direct interface to the values
  - `VecGetArray(), VecGetArrayF90()`
- Has unusual operations
  - `VecSqrtAbs(), VecStrideGather()`
- Communicates automatically during assembly
- Has customizable communication (**VecScatter**)

Object-Oriented Design

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

Vector Algebra

What are PETSc vectors?

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

How do I create vectors?

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

Vector Algebra

A PETSc Vec

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

Parallel Assembly
Vectors and Matrices

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

Vector Assembly

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

One Way to Set the Elements of a Vector

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

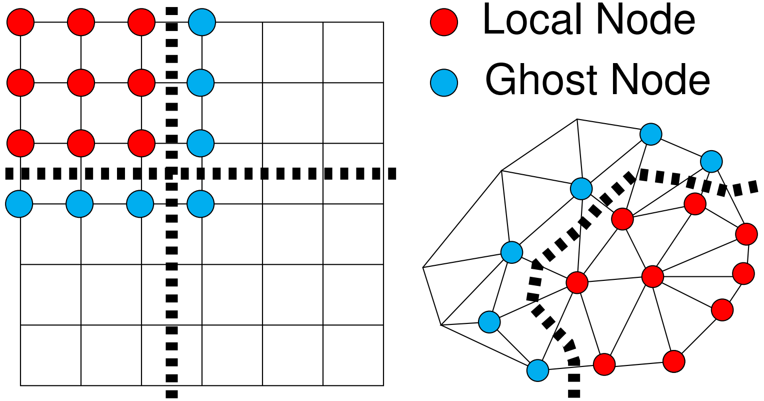
A Better Way to Set the Elements of a Vector

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

Ghost Values

To evaluate a local function $f(x)$, each process requires

- its local portion of the vector x
- its ghost values, bordering portions of x owned by neighboring processes



Working With Local Vectors

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

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

VecGetArray in C

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

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

VecGetArray in F77

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

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

VecGetArray in F90

```
#include "finclude/petsc.h"
#include "finclude/petscvec.h"
#include "finclude/petscvec.h90"
Vec v;
PetscScalar pointer :: array(:)
PetscInt n, i
PetscErrorCode ierr

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

Selected Vector Operations

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

What is a DM?

- Interface for linear algebra to talk to grids
- Defines (topological part of) a finite-dimensional function space
    - Get an element from this space: `DMCreateGlobalVector()`
- Provides parallel layout
- Refinement and coarsening
    - `DMRefine()`, `DMCoarsen()`
- Ghost value coherence
    - `DMGlobalToLocalBegin()`
- Matrix preallocation:
    - `DMCreateMatrix()` (formerly `DMGetMatrix()`)

Topology Abstractions

- `DMDA`
  - Abstracts Cartesian grids in 1, 2, or 3 dimension
  - Supports stencils, communication, reordering
  - Nice for simple finite differences
- `DMPLEX`
  - Abstracts general topology in any dimension
  - Also supports partitioning, distribution, and global orders
  - Allows aribtrary element shapes and discretizations
- `DMCOMPOSITE`
  - Composition of two or more DMs
- `DMNetwork` - for discrete networks like power grids and circuits
- `DMMoab` - interface to the MOAB unstructured mesh library

DM Vectors

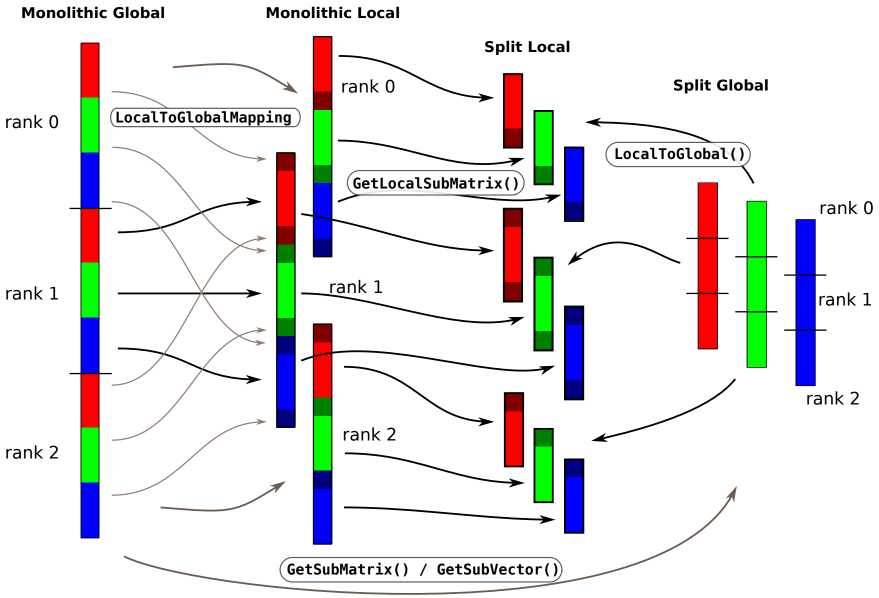
- The DM object contains only layout (topology) information
  - All field data is contained in PETSc **Vecs**
- Global vectors are parallel
  - Each process stores a unique local portion
  - `DMCreateGlobalVector(DM da, Vec *gvec)`
- Local vectors are sequential (and usually temporary)
  - Each process stores its local portion plus ghost values
  - `DMCreateLocalVector(DM da, Vec *lvec)`
  - includes ghost values!

Updating Ghosts

Two-step process enables overlapping computation and communication

- `DMGlobalToLocalBegin(dm, gvec, mode, lvec)`
  - `gvec` provides the data
  - `mode` is either `INSERT_VALUES` or `ADD_VALUES`
  - `lvec` holds the local and ghost values
- `DMGlobalToLocalEnd(dm, gvec, mode, lvec)`
  - Finishes the communication

The process can be reversed with `DMLocalToGlobalBegin()` and `DMLocalToGlobalEnd()`.



Work in Split Local space, matrix data structures reside in any space.

What is a DMDA?

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

- Handles local and global indices
  - `DMDAGetGlobalIndices()` and `DMDAGetAO()`
- Provides local and global vectors
  - `DMGetGlobalVector()` and `DMGetLocalVector()`
- Handles ghost values coherence
  - `DMGetGlobalToLocal()` and `DMGetLocalToGlobal()`

DMDA Global vs. Local Numbering

- **Global**: Each vertex has a unique id belongs on a unique process
- **Local**: Numbering includes vertices from neighboring processes
  - These are called ghost vertices

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

Local numbering

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

Global numbering

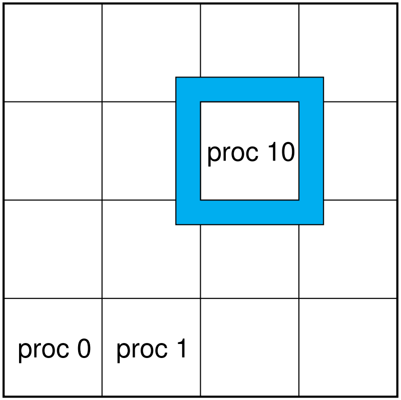
Creating a DADM

```
DMDACreate2d(comm, bdX, bdY, type, M, N, m, n, dof, s, lm[], ln[], DMDA *c
```

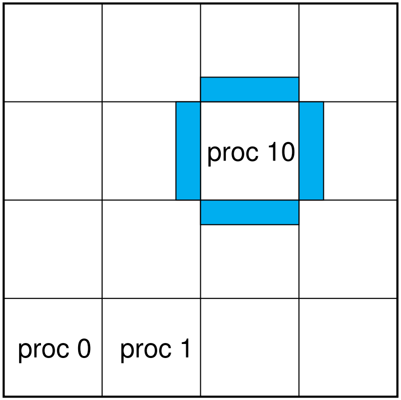
- `bd`: Specifies boundary behavior
  - `DMDA_BOUNDARY_NONE`, `DMDA_BOUNDARY_GHOSTED`, or `DMDA_BOUNDARY_PERIODIC`
- `type`: Specifies stencil
  - `DA_STENCIL_BOX` or `DA_STENCIL_STAR`
- `M/N`: Number of grid points in x/y-direction
- `m/n`: Number of processes in x/y-direction
- `dof`: Degrees of freedom per node
- `s`: The stencil width
- `lm/n`: Alternative array of local sizes
  - Use `PETSC_NULL` for the default

DMDA Stencils

Both the box stencil and star stencil are available.



Box Stencil



Star Stencil

Matrices

Definition (Matrix)

A matrix is a linear transformation between finite dimensional vector spaces.

Definition (Forming a matrix)

Forming or assembling a matrix means defining it's action in terms of entries (usually stored in a sparse format).

Matrices

Definition (Matrix)

A matrix is a linear transformation between finite dimensional vector spaces.

Definition (Forming a matrix)

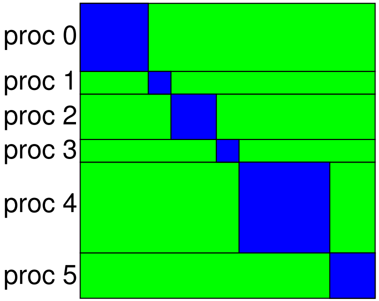
Forming or assembling a matrix means defining it’s action in terms of entries (usually stored in a sparse format).

How do I create matrices?

- `MatCreate(MPI_Comm, Mat *)`
- `MatSetSizes(Mat, int m, int n, int M, int N)`
- `MatSetType(Mat, MatType typeName)`
- `MatSetFromOptions(Mat)`
  - Can set the type at runtime
- `MatMPIBAIJSetPreallocation(Mat,...)`
  - important for assembly performance, more tomorrow
- `MatSetBlockSize(Mat, int bs)`
  - for vector problems
- `MatSetValues(Mat,...)`
  - **MUST** be used, but does automatic communication
  - `MatSetValuesLocal(),MatSetValuesStencil()`
  - `MatSetValuesBlocked()`

Matrix Storage Layout

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



🟦 diagonal blocks

🟩 offdiagonal blocks

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

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

Matrix Assembly

- A three step process
  - Each process sets or adds values
  - Begin communication to send values to the correct process
  - Complete the communication
- `MatSetValues(Mat A, m, rows[], n, cols[], values[], mode)`
  - `mode` is either INSERT_VALUES or ADD_VALUES
  - Logically dense block of values
- Two phase assembly allows overlap of communication and computation
  - `MatAssemblyBegin(Mat m, type)`
  - `MatAssemblyEnd(Mat m, type)`
  - `type` is either MAT_FLUSH_ASSEMBLY or MAT_FINAL_ASSEMBLY
- For vector problems

```
MatSetValuesBlocked(Mat A, m, rows[],
          n, cols[], values[], mode)
```

- The same assembly code can build matrices of different format

Matrix Assembly

- A three step process
  - Each process sets or adds values
  - Begin communication to send values to the correct process
  - Complete the communication
- `MatSetValues(Mat A, m, rows[], n, cols[], values[], mode)`
  - `mode` is either INSERT_VALUES or ADD_VALUES
  - Logically dense block of values
- Two phase assembly allows overlap of communication and computation
  - `MatAssemblyBegin(Mat m, type)`
  - `MatAssemblyEnd(Mat m, type)`
  - `type` is either MAT_FLUSH_ASSEMBLY or MAT_FINAL_ASSEMBLY
- For vector problems

```
MatSetValuesBlocked(Mat A, m, rows[],
          n, cols[], values[], mode)
```

- The same assembly code can build matrices of different format

One Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

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

A Better Way to Set the Elements of a Matrix

Simple 3-point stencil for 1D Laplacian

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

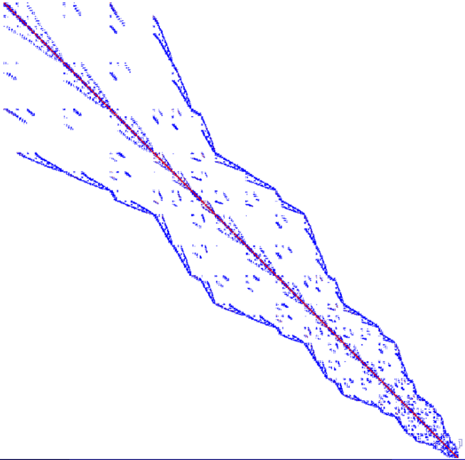
Matrix Memory Preallocation

- PETSc sparse matrices are dynamic data structures
  - can add additional nonzeros freely
- Dynamically adding many nonzeros
  - requires additional memory allocations
  - requires copies
  - can kill performance
- Memory preallocation provides
  - the freedom of dynamic data structures
  - good performance
- Easiest solution is to replicate the assembly code
  - Remove computation, but preserve the indexing code
  - Store set of columns for each row
- Call preallocation routines for all datatypes
  - `MatSeqAIJSetPreallocation()`
  - `MatMPIBAIJSetPreallocation()`
  - Only the relevant data will be used

Sequential Sparse Matrices

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

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



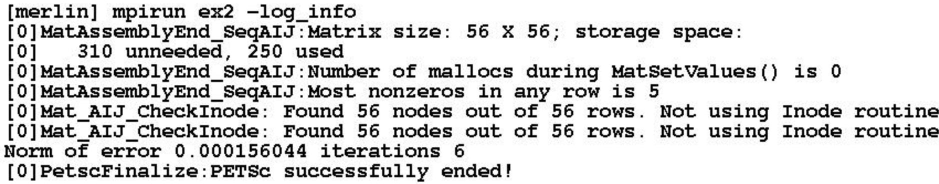
Parallel Sparse Matrices

```
MatMPIAIJSetPreallocation(Mat A, int dnz, int
dnnz[],
      int onz, int onnz[])
```

dnz:	expected number of nonzeros in any row in the diagonal block
dnnz(i):	expected number of nonzeros in row i in the diagonal block
onz:	expected number of nonzeros in any row in the offdiagonal portio
onnz(i):	expected number of nonzeros in row i in the offdiagonal portion

Verifying Preallocation

- Use runtime options
  `-mat_new_nonzero_location_err`
  `-mat_new_nonzero_allocation_err`
- Use runtime option `-info`
- Output:
  ```
  [proc #] Matrix size:  %d X %d; storage space:
  %d unneeded, %d used
  [proc #] Number of mallocs during MatSetValues( )
  is %d
  ```



```
[merlin] mpirun ex2 -log_info
[0]MatAssemblyEnd_SeqAIJ:Matrix size: 56 X 56; storage space:
[0]   310 unneeded, 250 used
[0]MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues() is 0
[0]MatAssemblyEnd_SeqAIJ:Most nonzeros in any row is 5
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routine
[0]Mat_AIJ_CheckInode: Found 56 nodes out of 56 rows. Not using Inode routine
Norm of error 0.000156044 iterations 6
[0]PetscFinalize:PETSc successfully ended!
```

Matrix Polymorphism

The PETSc `Mat` has a single user interface,

- Matrix assembly
  - `MatSetValues()`
- Matrix-vector multiplication
  - `MatMult()`
- Matrix viewing
  - `MatView()`

but multiple underlying implementations.

- AIJ, Block AIJ, Symmetric Block AIJ,
- Dense, Elemental
- Matrix-Free
- etc.

A matrix is defined by its interface, not by its data structure.

Block and symmetric formats

- BAIJ
  - Like AIJ, but uses static block size
  - Preallocation is like AIJ, but just one index per block
- SBAIJ
  - Only stores upper triangular part
  - Preallocation needs number of nonzeros in upper triangular parts of on- and off-diagonal blocks
- `MatSetValuesBlocked()`
  - Better performance with blocked formats
  - Also works with scalar formats, if `MatSetBlockSize()` was called
  - Variants `MatSetValuesBlockedLocal()`, `MatSetValuesBlockedStencil()`
  - Change matrix format at runtime, don't need to touch assembly code

Performance of blocked matrix formats

Kernel \ Format	Core 2, 1 process			Opteron, 4 processes		
	AIJ	BAIJ	SBAIJ	AIJ	BAIJ	SBAIJ
`MatMult`	812	985	1507	2226	2918	3119
`MatSolve`	718	957	955	1573	2869	2858

Throughput (Mflop/s) for different matrix formats on Core 2 Duo (P8700) and Opteron 2356 (two sockets). `MatSolve` is a forward- and back-solve with incomplete Cholesky factors. The AIJ format is using "inodes" which unrolls across consecutive rows with identical nonzero pattern (pairs in this case).

Objects

```
Mat A;
PetscInt m,n,M,N;
MatCreate(comm,&A);
MatSetSizes(A,m,n,M,N);      /* or PETSC_DECIDE */
MatSetOptionsPrefix(A,"foo_");
MatSetFromOptions(A);
/* Use A */
MatView(A,PETSC_VIEWER_DRAW_WORLD);
MatDestroy(A);
```

- `Mat` is an opaque object (pointer to incomplete type)
  - Assignment, comparison, etc, are cheap
- What's up with this "Options" stuff?
  - Allows the type to be determined at runtime: `-foo_mat_type sbaij`
  - Inversion of Control similar to "service locator", related to "dependency injection"
  - Other options (performance and semantics) can be changed at

Matrices, redux

What are PETSc matrices?

- Linear operators on finite dimensional vector spaces.
- Fundamental objects for storing stiffness matrices and Jacobians
- Each process locally owns a contiguous set of rows
- Supports many data types
  - AIJ, Block AIJ, Symmetric AIJ, Block Diagonal, etc.
- Supports structures for many packages
  - MUMPS, Spooles, SuperLU, UMFPack, DSCPack

Matrices, redux

What are PETSc matrices?

- Linear operators on finite dimensional vector spaces.
- Fundamental objects for storing stiffness matrices and Jacobians
- Each process locally owns a contiguous set of rows
- Supports many data types
  - AIJ, Block AIJ, Symmetric AIJ, Block Diagonal, etc.
- Supports structures for many packages
  - MUMPS, Spooles, SuperLU, UMFPack, DSCPack

Why Are PETSc Matrices That Way?

- No one data structure is appropriate for all problems
    - Blocked and diagonal formats provide significant performance benefits
    - PETSc has many formats and makes it easy to add new data structures
- Assembly is difficult enough without worrying about partitioning
    - PETSc provides parallel assembly routines
    - Achieving high performance still requires making most operations local
    - However, programs can be incrementally developed.
    - `MatPartitioning` and `MatOrdering` can help
- Matrix decomposition in contiguous chunks is simple
    - Makes interoperation with other codes easier
    - For other ordering, PETSc provides “Application Orderings” (AO)

`MatGetLocalSubMatrix()` spaces

- Newton method for $F(x) = 0$ solves

$$J(x)\delta x = -F(x)$$
$$J = \begin{pmatrix} J_{aa} & J_{ab} & J_{ac} \\ J_{ba} & J_{bb} & J_{bc} \\ J_{ca} & J_{cb} & J_{cc} \end{pmatrix}.$$

- Conceptually, there are three spaces in parallel

V	"monolithic" globally assembled space
V_i	"split" global space for a single physics i
$\overline{V}_i$	Local space (with ghosts) for a single physcs i
$\overline{V}$	$\prod_i \overline{V}_i$ Concatenation of all single-physics local spaces

- Different components need different relationships

$V_i \to V$	field-split
$\overline{V} \to V$	coupled Neumann domain decomposition methods
$\overline{V}_i$	natural language for modular residual evaluation and assembly

`MatGetLocalSubMatrix(Mat A,IS rows,IS cols,Mat *B);`

- Primarily for assembly
  - `B` is not guaranteed to implement `MatMult`
  - The communicator for `B` is not specified, only safe to use non-collective ops (unless you check)
- `IS` represents an index set, includes a block size and communicator
- `MatSetValuesBlockedLocal()` is implemented
- MatNest returns nested submatrix, no-copy
- No-copy for Neumann-Neumann formats (unassembled across procs, e.g. BDDC, FETI-DP)
- Most other matrices return a lightweight proxy `Mat`
  - `COMM_SELF`
  - Values not copied, does not implement `MatMult`
  - Translates indices to the language of the parent matrix
  - Multiple levels of nesting are flattened

`MatGetLocalSubMatrix()` spaces

Spaces

V	Globally assembled space
V_i	Global space for a single physics i
$\overline{V}_i$	Local space (with ghosts) for a single physcs i
$\overline{V}$	$\prod_i \overline{V}_i$ Concatenation of all single-physics local spaces

- Multiple physics $x = [x_a, x_b, x_c]$

I_i Map indices from V_i to V.

R_i Global physics restriction $R_i : V \to V_i$

$$R_i x = x[I_i] = x_i$$

$\bar{I}_i$ Map indices from $\overline{V}_i$ to V_i

$\overline{R}_i$ Extract local single-physics part from global single-physics

$$\overline{R}_i x_i = x_i[\bar{I}_i] = \overline{x}_i$$

$\tilde{I}_i$ Map indices from $\overline{V}_i$ to $\overline{V}$

`MatGetLocalSubMatrix()` spaces

- Globally assembled coupled matrix in terms of assembled single-physics blocks

$$J = \sum_{ij} R_i^T J_{ij} R_j$$

  - Language of Schwarz and fieldsplit
- Assembled single-physics blocks in terms of local single-physics matrices

$$J_{ij} = \overline{R}_i^T \overline{J}_{ij} \overline{R}_j$$

  - Language of assembly and Neumann/FETI domain decomposition
  - `MatSetValuesLocal()`

Setting Values on Regular Grids

PETSc provides

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

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

DMDA matrices

- DMCreateMatrix(DM da,Mat *A)
- Evaluate only the local portion
  - No nice local array form without copies
- Use `MatSetValuesStencil()` to convert `(i,j,k)` to indices

-
  ```
  make NP=2 EXTRA_ARGS="-run test -da_grid_x 10 -da_grid_y 10
  -mat_view_draw -draw_pause -1" runbratu
  ```

-
  ```
  make NP=2 EXTRA_ARGS="-run test -dim 3 -da_grid_x 5 -da_grid_y 5
  -da_grid_z 5 -mat_view_draw -draw_pause -1" runbratu
  ```