Introduction to PETSc
(2)

Linear Algebra I

- Vectors
  - Has a direct interface to the values
  - Supports all vector space operations
    - VecDot(), VecNorm(), VecScale()
  - Also unusual ops, e.g. VecSqrt()
  - Automatic communication during assembly
  - Customizable communication (scatters)

Integration

PETSc Numerical 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 Schwartz	Block Jacobi	Jacobi	ILU	ICC	LU (Sequential only)	Others

Matrices

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

Distributed Arrays

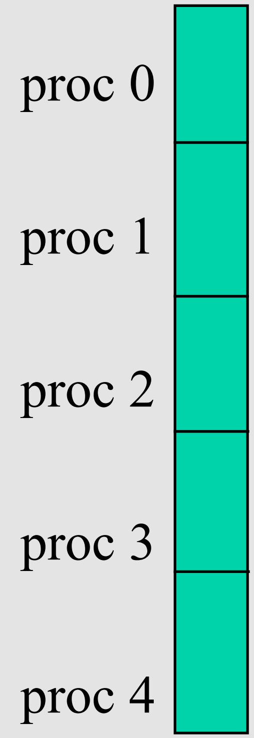
Index Sets

Indices	Block Indices	Stride	Other

Vectors

Vectors

- What are PETSc vectors?
  - Fundamental objects for storing field solutions, right-hand sides, etc.
  - Each process locally owns a subvector of contiguously numbered global indices
- Create vectors via
  - VecCreate(MPI_Comm comm,Vec *x)
    - comm - processes that share the vector
  - VecSetSizes(Vec x, int n, int N)
    - n: number of elements local to this process
    - N: total number of elements
  - VecSetType(Vec x,VecType type)
    - type: where VecType is: VEC_SEQ, VEC_MPI, or VEC_SHARED
  - VecSetFromOptions(Vec x)
    - lets you set the type at *runtime*



data objects: vectors

Creating a vector

```
Vec x;
int N;
…
PetscInitialize(&argc,&argv,(char*)0,help);
PetscOptionsGetInt(PETSC_NULL,"-n",&N,PETSC_NULL);
…
VecCreate(PETSC_COMM_WORLD,&x);
VecSetSizes(x,PETSC_DECIDE,N);
VecSetType(x,VEC_MPI);
VecSetFromOptions(x);
```

Use PETSc to get value from command line

Global size

PETSc determines local size

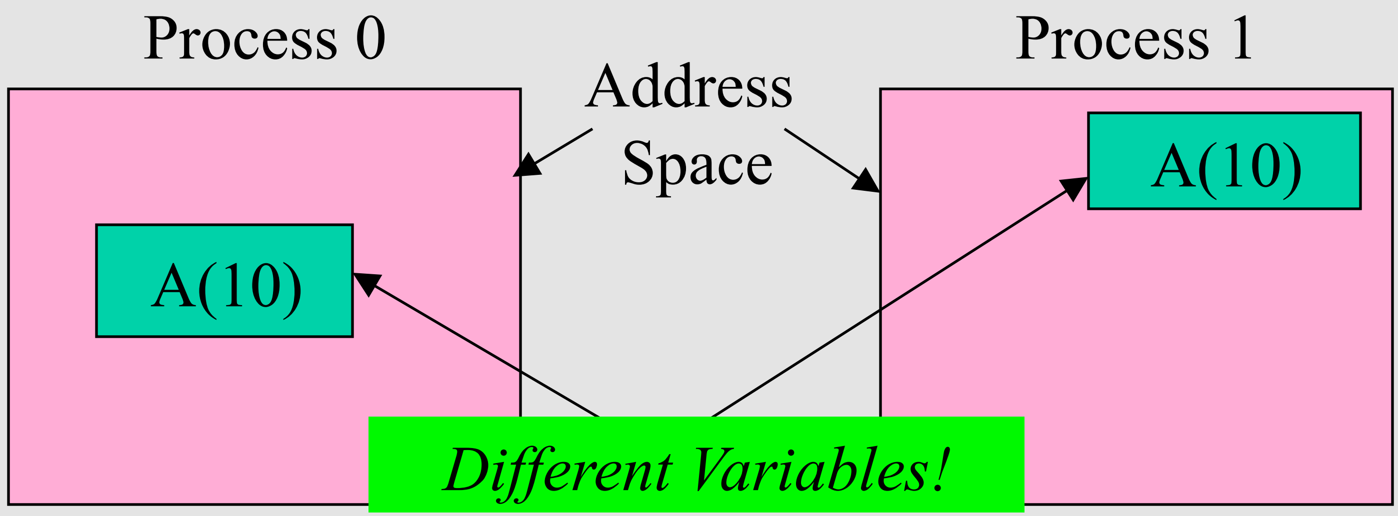
data objects: vectors

How Can We Use a PETSc Vector

- PETSc supports "data structure-neutral" objects
  - distributed memory "shared nothing" model
  - single processors and shared memory systems
- 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 *require* that the programmer work only with the "local" part of the vector; we permit operations, such as setting an element of a vector, to be performed globally
- Recall how data is stored in the distributed memory programming model…

Distributed Memory Model



- Integer A(10)

  …

  print *, A

- Integer A(10)
  do i=1,10
    A(i) = i
  enddo
  ...

This A is completely different from this one

Vector Assembly

- A three step process
  1) Each process tells PETSc what values to insert/add to a vector component.

     VecSetValues(x, n, indices[], values[], mode);

     - n: number of entries to insert/add
     - indices[]: indices of entries
     - values[]: values to add
     - mode: [INSERT_VALUES, ADD_VALUES]

  Once *all* values provided

  2) Begin communication between processes to ensure that values end up where needed

     VecAssemblyBegin(x);

     - allow other operations, such as some computation, to proceed

  3) Complete the communication

     VecAssemblyEnd(x);

data objects: vectors

Parallel Matrix and Vector Assembly

- Processes may generate any entries in vectors and matrices
- Entries need not be generated on the process on which they ultimately will be stored
- **PETSc automatically moves data during the assembly process if necessary**
  - e.g., ~petsc/src/vec/vec/examples/tutorials/ex2.c

data objects:
vectors

One Way to Set the Elements of A Vector

```
VecGetSize(x,&N);  /* Global size */
MPI_Comm_rank(PETSC_COMM_WORLD, &rank);

if (rank == 0) {
   for (i=0; i<N; i++)
      VecSetValues(x,1,&i,&i,INSERT_VALUES);
}
```

Vector index (the first &i)

Vector value (the second &i)

```
/* These two routines ensure that the data is distributed to the
other processes */
VecAssemblyBegin(x);
VecAssemblyEnd(x);
```

data objects: vectors

A Parallel Way to Set the Elements of a Distributed Vector

```
VecGetOwnershipRange(x,&low,&high);
for (i=low; i<high; i++)
    VecSetValues(x,1,&i,&i,INSERT_VALUES);

/* These two routines must be called (in case some other process
contributed a value owned by another process) */
VecAssemblyBegin(x);
VecAssemblyEnd(x);
```

data objects: vectors

Selected Vector Operations

Function Name	Operation
VecAXPY(Scalar *a, Vec x, Vec y)	$y = y + a*x$
VecAYPX(Scalar *a, Vec x, Vec y)	$y = x + a*y$
VecWAXPY(Scalar *a, Vec x, Vec y, Vec w)	$w = a*x + y$
VecScale(Scalar *a, Vec x)	$x = a*x$
VecCopy(Vec x, Vec y)	$y = x$
VecPointwiseMult(Vec x, Vec y, Vec w)	$w\_i = x\_i *y\_i$
VecMax(Vec x, int *idx, double *r)	$r = max\ x\_i$
VecShift(Scalar *s, Vec x)	$x\_i = s+x\_i$
VecAbs(Vec x)	$x\_i = \|x\_i\|$
VecNorm(Vec x, NormType type , double *r)	$r = \|\|x\|\|$

data objects:
vectors

A Complete PETSc Program

```
#include petscvec.h
int main(int argc,char **argv)
{
  PetscErrorCode ierr;
  Vec            x;
  PetscInt       n = 20;
  PetscTruth     flg;
  PetscScalar    one = 1.0, dot;

  PetscInitialize(&argc,&argv,0,0);
  PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);
  VecCreate(PETSC_COMM_WORLD,&x);
  VecSetSizes(x,PETSC_DECIDE,n);
  VecSetFromOptions(x);
  VecSet(&one,x);
  VecDot(x,x,&dot);
  PetscPrintf(PETSC_COMM_WORLD,"Vector length %dn",(int)dot);
  VecDestroy(x);
  PetscFinalize();
  return 0;
}
```

data objects: vectors

Working With Local Vectors

- It is sometimes more efficient to directly access the storage for the local part of a PETSc Vec.
  - E.g., for finite difference computations involving elements of the vector
- 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
  - For most common uses, these routines are inexpensive and do *not* involve a copy of the vector.

data objects: vectors

Example of VecGetArray

```
Vec          vec;
PetscScalar  *array;
…
VecCreate(PETSC_COMM_SELF,&vec);
VecSetSizes(vec,PETSC_DECIDE,N);
VecSetFromOptions(vec);

VecGetArray(vec,&array);

/* compute with array directly, e.g., */
PetscPrintf(PETSC_COMM_WORLD,
 “First element of local array of vec in each process is %f\n”, array[0] );

VecRestoreArray(vec,&array);
```

data objects:
vectors

Indexing

- Non-trivial in parallel
- PETSc IS object, generalization of
  - {0,3,56,9}
  - 1:4:55
  - Indexing by block

Linear Algebra II

- Matrices
  - Must use MatSetValues()
    - Automatic communication
  - Supports many data types
    - AIJ, Block AIJ, Symmetric AIJ, Block Diagonal, etc.
  - Supports structures for many packages
    - Spooles, MUMPS, SuperLU, UMFPack, DSCPack

Matrices

- What are PETSc matrices?
  - Fundamental objects for storing linear operators (e.g., Jacobians)
- Create matrices via
  - MatCreate(comm, &mat)
    - MPI_Comm - processes that share the matrix
  - MatSetSizes(mat,PETSC_DECIDE,PETSC_DECIDE,M,N)
    - number of local/global rows and columns
  - MatSetType(Mat, MatType)
    - where MatType is one of
      - default sparse AIJ: MPIAIJ, SEQAIJ
      - block sparse AIJ (for multi-component PDEs): MPIAIJ, SEQAIJ
      - symmetric block sparse AIJ: MPISBAIJ, SAEQSBAIJ
      - block diagonal: MPIBDIAG, SEQBDIAG
      - dense: MPIDENSE, SEQDENSE
      - matrix-free
      - etc (see ~petsc/src/mat/impls/)
  - MatSetFromOptions(Mat)
    - lets you set the MatType at *runtime*.

data objects: matrices

Matrices and Polymorphism

- Single user interface, e.g.,
  - Matrix assembly
    - MatSetValues()
  - Matrix-vector multiplication
    - MatMult()
  - Matrix viewing
    - MatView()
- Multiple underlying implementations
  - AIJ, block AIJ, symmetric block AIJ, block diagonal, dense, matrix-free, etc.
- A matrix is defined by its *interface*, the operations that you can perform with it.
  - Not by its data structure

data objects: matrices

Matrix Assembly

- Same form as for PETSc Vectors:

1) MatSetValues(mat, m, idxm[], n, idxn[], v[], mode)
   - m: number of rows to insert/add
   - idxm[]: indices of rows and columns
   - n: number of columns to insert/add
   - v[]: values to add
   - mode: [INSERT_VALUES,ADD_VALUES]

2) MatAssemblyBegin(mat, type)

3) MatAssemblyEnd(mat, type)

data objects: matrices

Matrix Assembly Example

simple 3-point stencil for 1D discretization

```
Mat    A;
int    column[3], i;
double value[3];
...
MatCreate(PETSC_COMM_WORLD,
        PETSC_DECIDE,PETSC_DECIDE,  N,N,&A);
MatSetFromOptions(A);
/* mesh interior */
value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
if (rank == 0) {  /* Only one process creates matrix entries */
   for (i=1; i<N-2; i++) {
      column[0] = i-1; column[1] = i; column[2] = i+1;
      MatSetValues(A,1,&i,3,column,value,INSERT_VALUES);
   }
}
/* also must set boundary points  (code for global row 0 and N-1 omitted) */
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

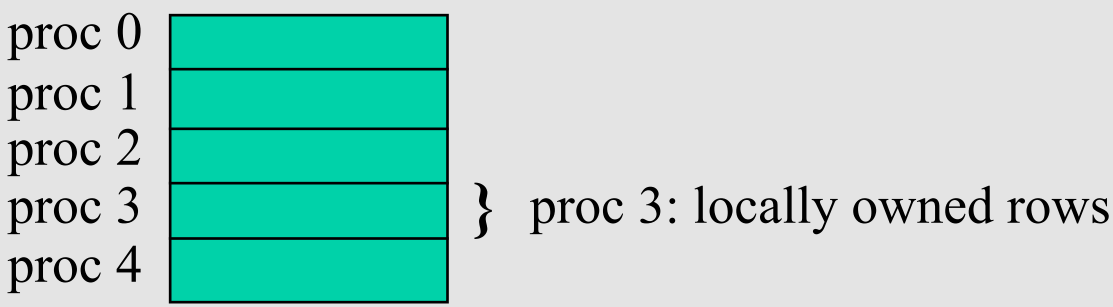
Choose the global size of the matrix

Let PETSc decide how to allocate matrix across processes

data objects: matrices

Parallel Matrix Distribution

Each process locally owns a submatrix of contiguously numbered global rows.



MatGetOwnershipRange(Mat A, int *rstart, int *rend)

- rstart: first locally owned row of global matrix
- rend -1: last locally owned row of global matrix

data objects: matrices

Matrix Assembly Example With Parallel Assembly

simple 3-point stencil for 1D discretization

```
Mat     A;
int     column[3], i, start, end,istart,iend;
double value[3];
…
MatCreate(PETSC_COMM_WORLD,
          PETSC_DECIDE,PETSC_DECIDE,n,n,&A);

MatSetFromOptions(A);
MatGetOwnershipRange(A,&start,&end);
/* mesh interior */
istart = start; if (start == 0) istart = 1;
iend = end; if (iend == n-1) iend = n-2;
value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
for (i=istart; i<iend; i++) { /* each processor generates some of the matrix values */
   column[0] = i-1; column[1] = i; column[2] = i+1;
   MatSetValues(A,1,&i,3,column,value,INSERT_VALUES);
}
/* also must set boundary points  (code for global row 0 and n-1 omitted) */
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

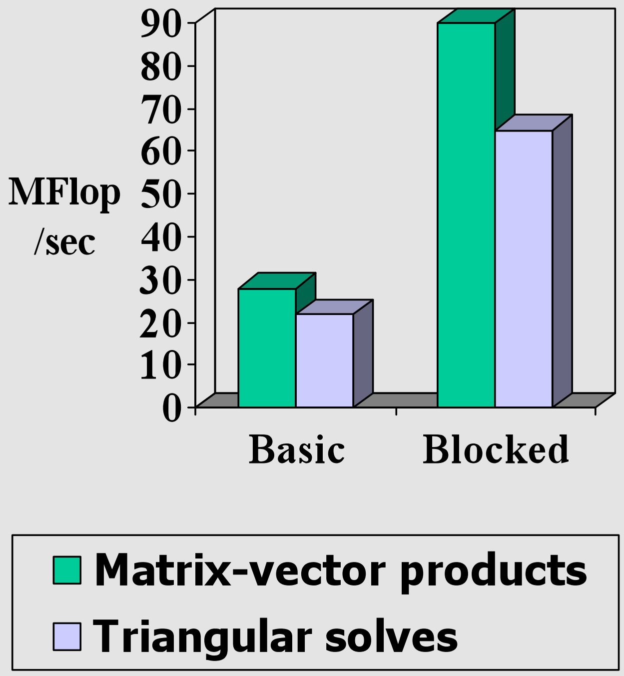
data objects: matrices

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 provides parallel assembly routines
  - Achieving high performance still requires making most operations local to a process but programs can be incrementally developed.
- Matrix decomposition by consecutive rows across processes, **for sparse matrices**, is simple and makes it easier to work with other codes.
  - For applications with other ordering needs, PETSc provides "Application Orderings" (AO).

Blocking: Performance Benefits

More issues discussed in full tutorials available via PETSc web site.



- 3D compressible Euler code
- Block size 5
- IBM Power2

data objects: matrices