Building a Successful Scalable Parallel Numerical Library: Lessons From the PETSc Library

William D. Gropp
www.cs.uiuc.edu/homes/wgropp

What is PETSc?

- PETSc is a numerical library that is organized around mathematical concepts appropriate for the solution of linear and nonlinear systems of equations that arise from discretizations of Partial Differential Equations
- PETSc began as a tool to aid in research into domain decomposition methods for elliptic and hyperbolic (with implicit time stepping) partial differential equations. A new library was needed because
  - Numerical libraries of the time were organized around particular algorithms, rather than mathematical problems, making experimentation with different algorithms difficult
  - Most libraries were not re-entrant, making recursive use impossible
- PETSc addressed these limitations and clearly filled a need; PETSc is now used by both applications scientists and researchers (100's of users including DOE and NSF leadership computing platforms)

The PETSc Team (Past and Present)


Satish Balay


Matt Knepley


Lisandro Dalcin


Kris Buschelman


Lois Curfman McInnes


Victor Eijkhout


Bill Gropp


Barry Smith

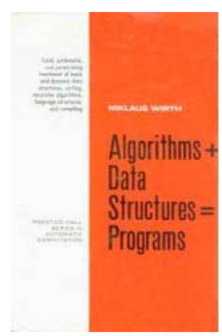

Dmitry Karpeev


Dinesh Kaushik


Hong Zhang

Everyone contributed to the results described in this talk

How Should Numerical Libraries Be Designed?



- It is common to choose an algorithms and a data structure
  - "Algorithms + Data Structures = Programs"
- Rarely unique choices
- Better is to say: I'm solving Ax=b
  - PETSc really looks at F(x,t) = 0, for which Ax=b is a key component
- Lets look at the Ax=b part…

The Constraints - Parallel Computing Issues

- Distributed Memory Model
  - Programmer must participate in handling decomposition of objects across processes
- Shared Memory Model
  - Poor integration with language (race detection, volatile, lack of write/read barriers)
  - Difficult to achieve scalability (hardware costly, complicated)
- Consequences
  - Choose MPI distributed memory model
  - Would do the same today
    - But maybe PGAS language will be appropriate soon
  - Scalability requires careful attention to message latency

5

Distributed Objects

- PETSc must provide a mechanism to work with objects that are distributed across a collection of processors
- Common patterns:
  - Err = <THING>Create(parallel-context,<INFO>, <SIZE>, &object)
  - Err = <THING>Destroy(object)
    Err = <THING><Operation>(object, <other-parms>)
- For example,
  - VecCreate(MPI_COMM_WORLD, PETSC_DECIDE, n, &x)
  - MatMult(A, x, y)
- Operations use the same name when possible:
  - <THING>SetFromOptions(object)
    - Use command line, environment variables, or defaults file to set basic properties

6

Vectors in PETSc

- Mathematical Objects
  - *Not* a contiguous section of memory
- Distributed across a set of processes
  - May be a subset of all processes in the parallel job
  - First decision:
    - How general a distribution is allowed for the representation of data?
    - For example, should ghost cells be allowed? Non-contiguous sections?
  - PETSc uses a very simple decomposition
    - A single, contiguous segment, ordered with the rank of the processes
    - Chosen for performance
  - Lesson 1: Permit the best performance

7

Does PETSc Need More General Vectors?

- So, how do you handle more general decompositions? PETSc provides several alternatives, depending on the type of generality
  - Non-contiguous in process: copy
    - Not as bad as it seems, as the copy may provide better cache locality and not be that costly
  - Non-contiguous across processes: permutations
    - Often better to apply permutations, then use
  - Plus, PETSc allows the use of arbitrary representations for vectors
    - But then the user is responsible for implementing all operations between vectors, and operations on vectors by other objects, such as matrix-vector product
  - Lesson 2: Provide an escape for customization

8

Accessing Elements of a Vector

- The element-wise approach seems simple:
  - V[k] = 3
- But what is involved with this in a parallel computer?
  - One possibility is:
    - Retrieve cache line containing v[k] from the current owner of that cache line if any, assert ownership (flush from owner's cache)
    - Update the bytes corresponding to V[k].
    - Write out the data to memory
    - When the original owner needs to access (even to read), figure out if the ownership of the cache line should move
    - !!!!
  - There are other options, but they all must handle where data is cached and how it is updated.
- Is this a good operation to support?
  - Rarely a natural mathematical operation
  - E.g., usually define entire vector, as in v = f(x,y)
  - Setting a single element in a vector is both costly and rarely necessary

9

Improving Performance of Vector Element Update

- Lesson 3: Define/Update objects as a single operation
  - Defer the "synchronization"; v[k] tells the language that after the assignment, v[k] is visible anywhere v is defined. This may force the system to wait until the data is available or to implement complex caching strategies
  - The alternative is relaxed consistency models, which may lead the programmer to refer to data before it is available (because of a mismatch between the computer language and these memory consistency models)
- Instead, define an operation that allows the user to define when the object must be ready for use. This is a simple generalization of the notion of matrix assembly

10

Assembly

- PETSc uses the notion of object assembly
  - First, describe update
  - Initiate assembly
    - Allow other work (Communication/Computation Overlap)
    - At least, that was the theory:
      - Communication systems require extra hardware support to effectively overlap communication and computation
      - There may not be natural work to insert in this slot
  - Wait for the assembly to complete
    - Note: still introduces more synchronization than strictly required (there are possibilities here for improvement)
    - This model selected for its simplicity
- This applies to all objects that must be assembled
- Not a new idea
  - Same idea used to vectorize sparse matrix assembly
    - Same problem - single element updates do not fit vector model

Setting Elements of a Vector

- While changing the vector so that a single element is updated is inefficient, it is simple
- PETSc provides a way to "set/add to an element that will be visible after the assembly completes":
  - VecSetValue( )
- Since many multicomponent PDEs naturally compute/update all the values at a grid point
  - VecSetValues()
- Key features:
  - Values set are in the mathematical vector object
    - User need not know/understand decomposition of vector representation across processes
  - Values are not available until after Assembly step completes
  - PETSc efficiently manages the implementation of assembly
    - Caches data, aggregates values destined to the same process
    - Transparent to the user
- Lesson 4: Provide ease-of-use features, even if they are not high-performance. Note that this is not inconsistent with Lesson 1 (permit high performance).

Matrices

- What are PETSc matrices?
  - Fundamental objects for storing linear operators (e.g., Jacobians)
- Create matrices via
  - MatCreate(...,Mat *)
    - MPI_Comm - processes that share the matrix
    - 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.
    - MatSetFromOptions(Mat) lets you set the MatType at runtime.

Matrices and Polymorphism

- Single user interface independent of the underlying sparse data structure, 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.

Matrices in PETSc

- A matrix is defined by its properties and the operations that you can perform with it.
  - Not by its data structures
  - (Some operations require efficient access to matrix elements; that only means that some operations, such as incomplete factor, may not be available if the matrix uses a matrix-free representation)
- The ability to associate different code for the same abstract operation, depending on the circumstances (such as the data structure) is called polymorphism. It is a critical part of the PETSc implementation approach
  - Typically implemented with a function pointer

15

What Advantage Does This Approach Give You?

- Example: A Poisson Solver in PETSc
  - The following 7 slides show a complete 2-d Poisson solver in PETSc. Features of this solver:
    - Fully parallel
    - 2-d decomposition of the 2-d mesh
    - Linear system described as a sparse matrix; user can select many different sparse data structures
    - Linear system solved with any user-selected Krylov iterative method and preconditioner provided by PETSc, including GMRES with ILU, BiCGstab with Additive Schwarz, etc.
    - Complete performance analysis built-in
  - Only 7 slides of code!

16

Solve a Poisson Problem with Preconditioned GMRES

```
/* -*- Mode: C; c-basic-offset:4 ; -*- */
#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;

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

PETSC "objects" hide details of distributed data structures and function parameters

```
    /* 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 );
```

PETSc provides routines to access parameters and defaults

PETSc provides routines to create, allocate, and manage distributed data structures

```
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;
}
```

PETSc provides routines that solve systems of sparse linear (and nonlinear) equations

PETSc provides coordinated I/O (behavior is as-if a single process), including the output of the distributed "vec" object

```
/* -*- Mode: C; c-basic-offset:4 ; -*- */
#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 );

    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;
}
```

Almost the uniprocess code

Creating a **Sparse** Matrix, Distributed Across All Processes

```
/* -*- Mode: C; c-basic-offset:4 ; -*- */
#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 */
    DAGetCorners( grid, &is, &js, 0, &in, &jn, 0 );
    ie = is + in - 1;
    je = js + jn - 1;
```

Creates a parallel distributed matrix using compressed sparse row format

```
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;
}
```

Just the usual code for setting the elements of the sparse matrix (the complexity comes, as it often does, from the boundary conditions)

Computer Science Lessons

- Organize around user-centric concepts
  - PETSc used the mathematics
  - Provide all that is necessary to manage the objects, not just the "key" functions
- Exploit Computer Science techniques to provide that interface
  - Data Encapsulation and Data Hiding
  - Polymorphism
  - Inheritance
- Pay attention to performance

Numerical Analysis Lessons

- Algorithms!
  - ♦ Get the right ones
  - ♦ Get the scalable parallel ones
  - ♦ Note that there is (rarely) a unique best choice
    - Implies that the software must support many algorithms
    - This is why PETSc organized by problems-to-solve rather than algorithms
      - – This may be the most important lesson: Organize by mathematical problem

25

Final Comments

- The success of PETSc is due to:
  - ♦ Performance and Scalability
    - Performance is only weakly correlated with FLOPS
  - ♦ Consistent interface based on the mathematical problems
  - ♦ Completeness
    - Can overcome "ease of use"
  - ♦ Attention to portability and configuration issues
    - Particularly for libraries coming from research groups, this is often the critical factor
    - Portability requires care but isn't hard.
- A key advantage to the PETSc approach
  - ♦ Algorithm Independence
    - Until we know the best way, don't make the choice
    - Users can try new algorithms without giving up the ones with which they are comfortable

26