Argonne
NATIONAL
LABORATORY

... for a brighter future

U.S. Department of Energy

UChicago ▸ Argonne LLC

Office of Science
U.S. DEPARTMENT OF ENERGY

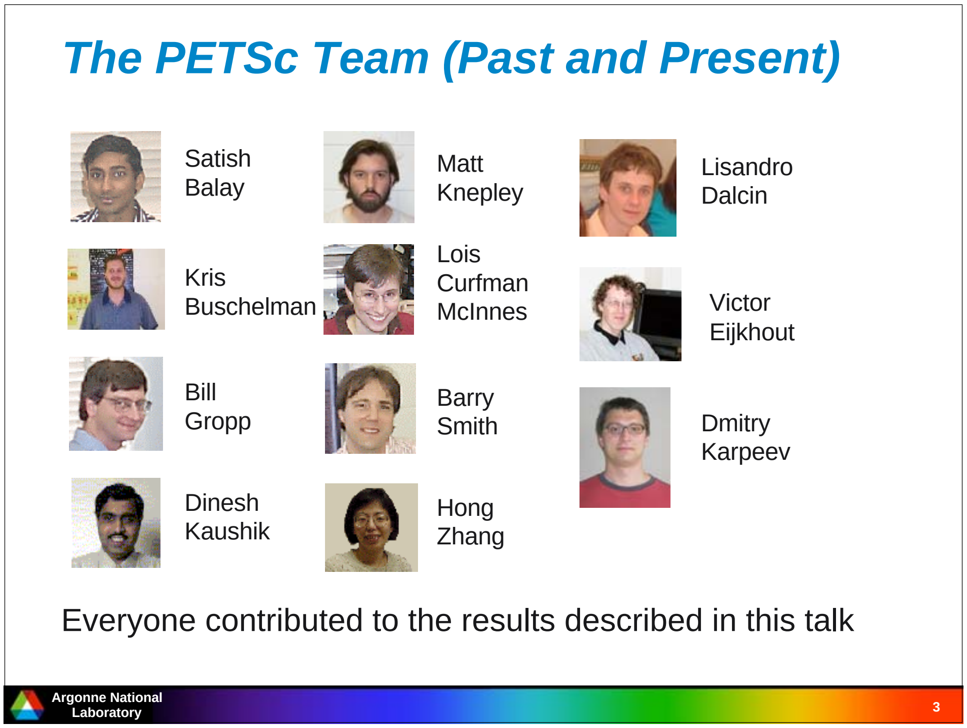
A U.S. Department of Energy laboratory managed by UChicago Argonne, LLC

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

William D. Gropp
Mathematics and Computer Science
www.mcs.anl.gov/~gropp

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



The PETSc Team (Past and Present)

Satish Balay

Matt Knepley

Lisandro Dalcin

Kris Buschelman

Lois Curfman McInnes

Victor Eijkhout

Bill Gropp

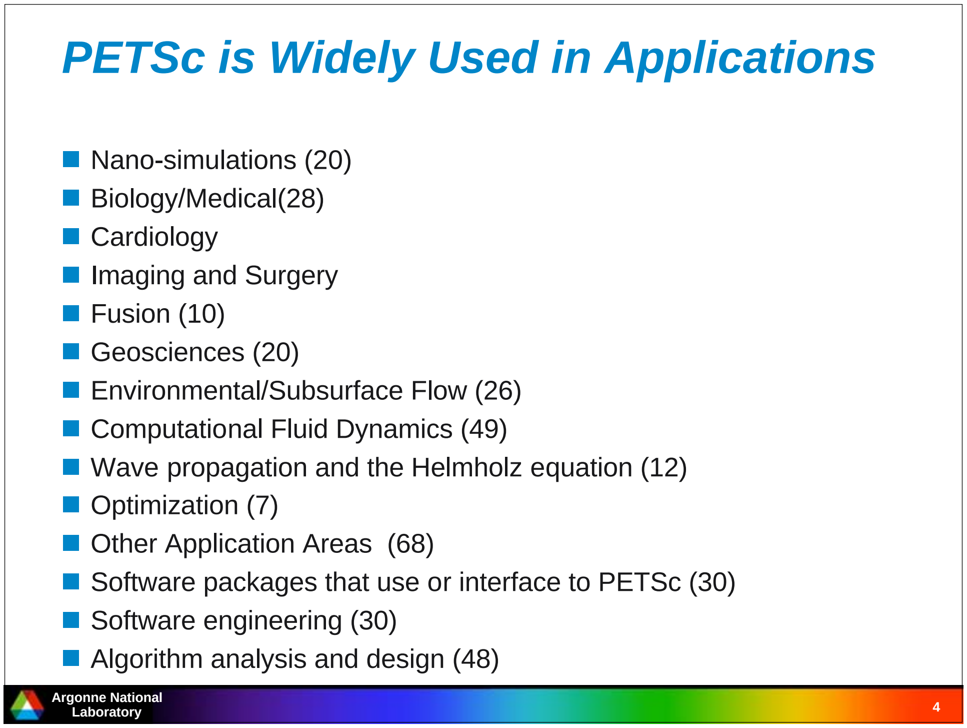
Barry Smith

Dmitry Karpeev

Dinesh Kaushik

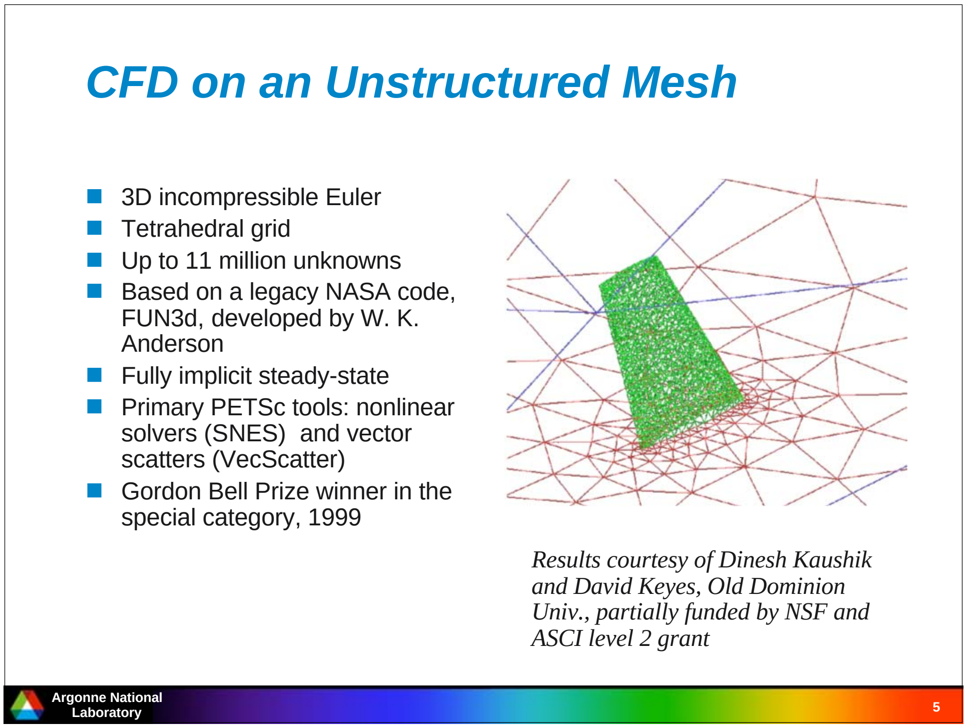
Hong Zhang

Everyone contributed to the results described in this talk



PETSc is Widely Used in Applications

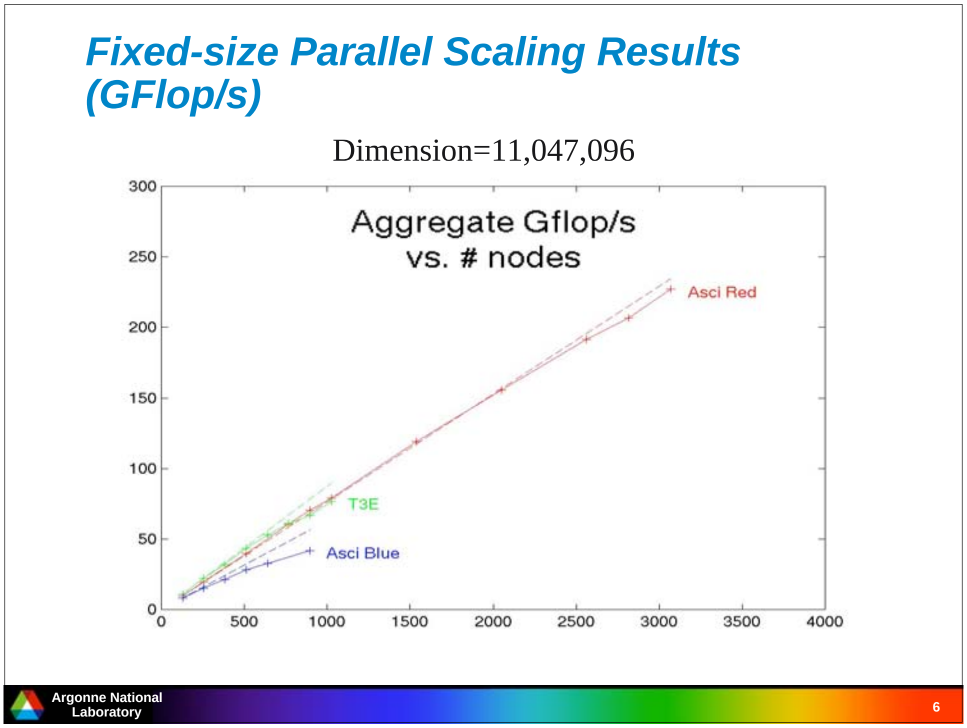
- Nano-simulations (20)
- Biology/Medical(28)
- Cardiology
- Imaging and Surgery
- Fusion (10)
- Geosciences (20)
- Environmental/Subsurface Flow (26)
- Computational Fluid Dynamics (49)
- Wave propagation and the Helmholz equation (12)
- Optimization (7)
- Other Application Areas (68)
- Software packages that use or interface to PETSc (30)
- Software engineering (30)
- Algorithm analysis and design (48)



CFD on an Unstructured Mesh

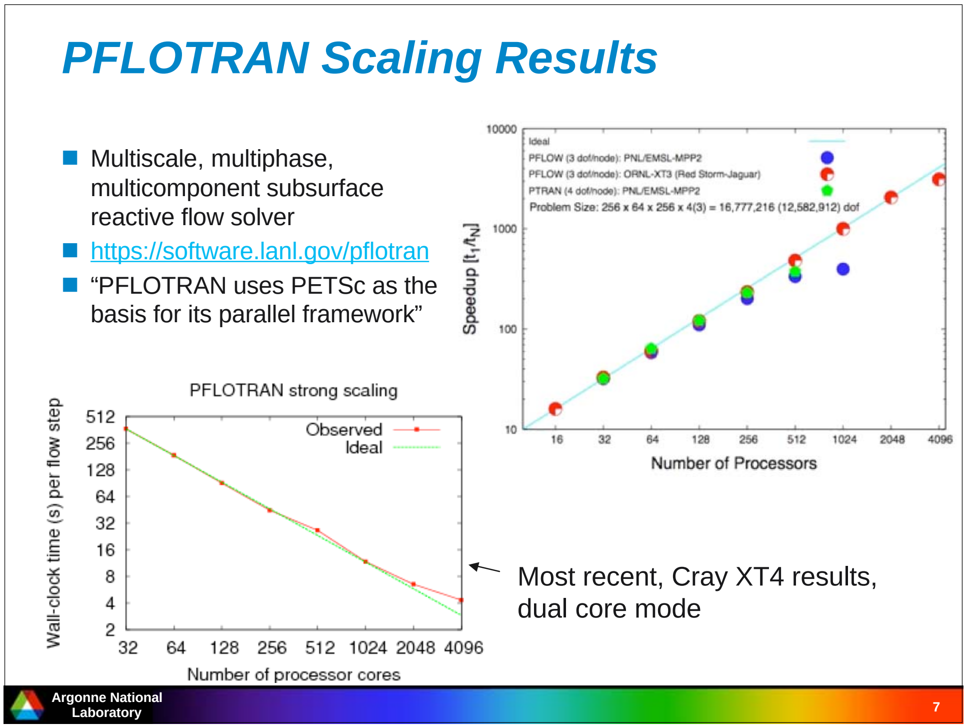
- 3D incompressible Euler
- Tetrahedral grid
- Up to 11 million unknowns
- Based on a legacy NASA code, FUN3d, developed by W. K. Anderson
- Fully implicit steady-state
- Primary PETSc tools: nonlinear solvers (SNES) and vector scatters (VecScatter)
- Gordon Bell Prize winner in the special category, 1999

Results courtesy of Dinesh Kaushik and David Keyes, Old Dominion Univ., partially funded by NSF and ASCI level 2 grant



Fixed-size Parallel Scaling Results (GFlop/s)

Dimension=11,047,096



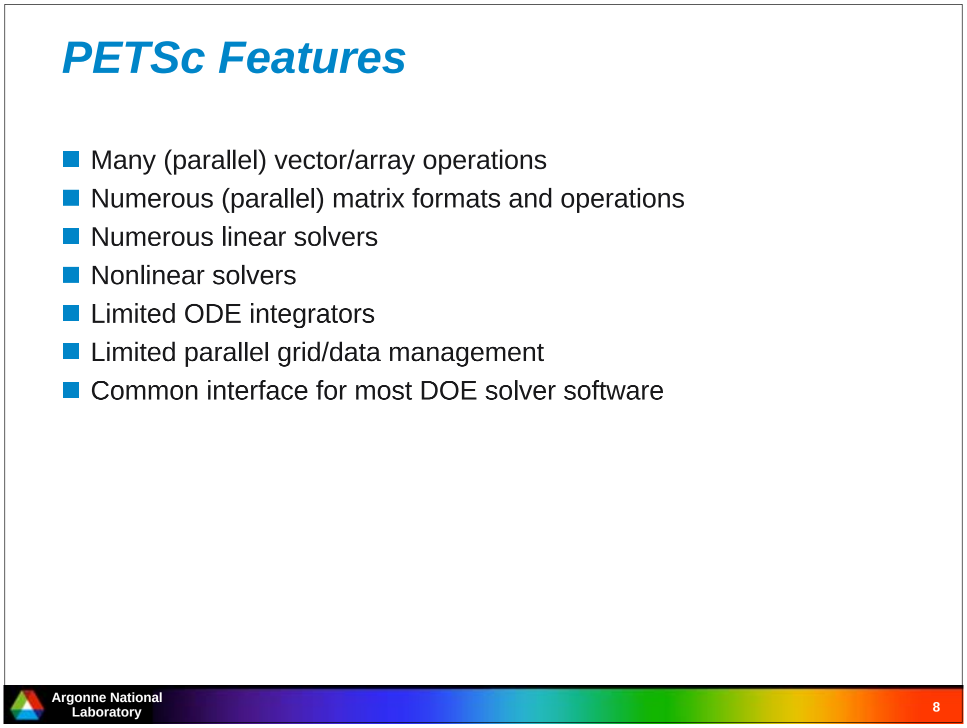
PFLOTRAN Scaling Results

- Multiscale, multiphase, multicomponent subsurface reactive flow solver
- https://software.lanl.gov/pflotran
- “PFLOTRAN uses PETSc as the basis for its parallel framework”

10000
1000
100
10
Speedup [t_1/t_N]
Ideal
PFLOW (3 dof/node): PNL/EMSL-MPP2
PFLOW (3 dof/node): ORNL-XT3 (Red Storm-Jaguar)
PTRAN (4 dof/node): PNL/EMSL-MPP2
Problem Size: 256 x 64 x 256 x 4(3) = 16,777,216 (12,582,912) dof
16
32
64
128
256
512
1024
2048
4096
Number of Processors

PFLOTRAN strong scaling
Wall-clock time (s) per flow step
512
256
128
64
32
16
8
4
2
Observed
Ideal
32
64
128
256
512
1024
2048
4096
Number of processor cores

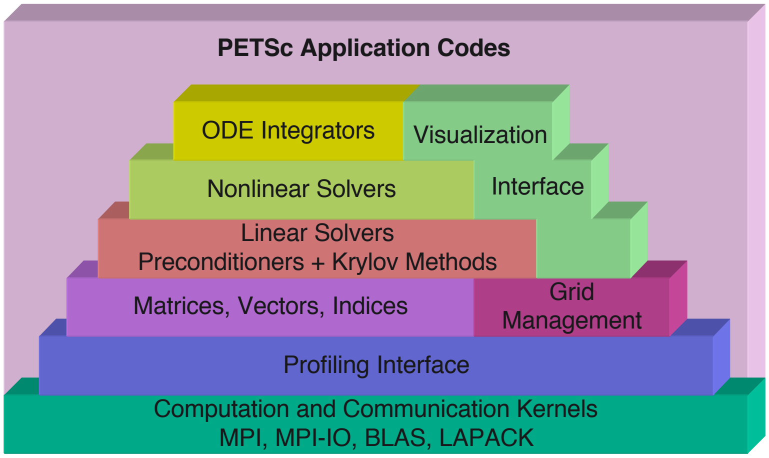
Most recent, Cray XT4 results, dual core mode



PETSc Features

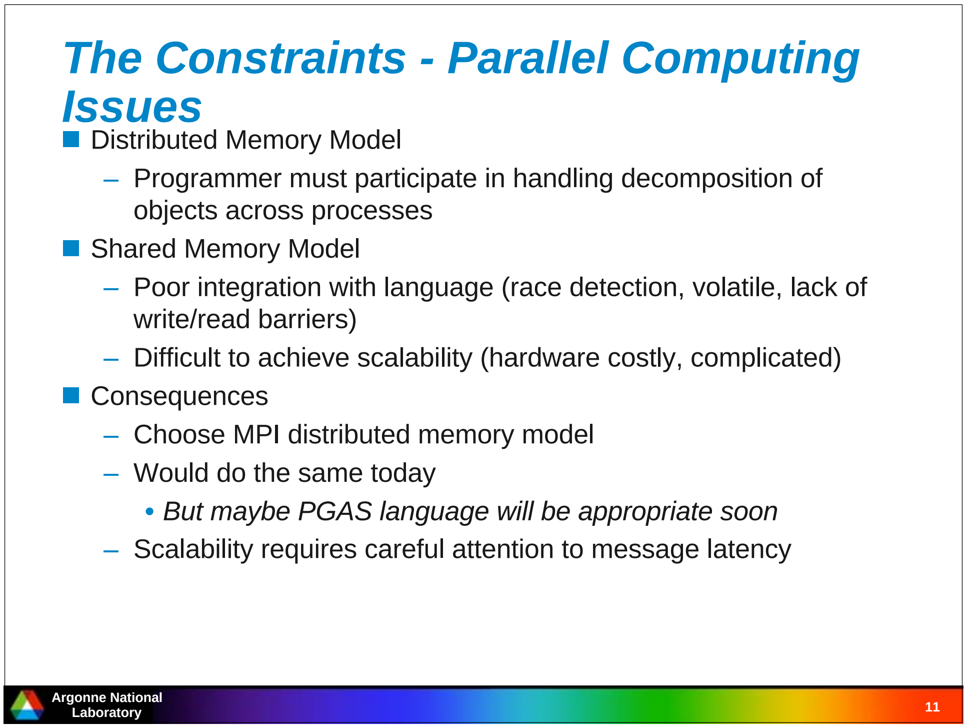
- Many (parallel) vector/array operations
- Numerous (parallel) matrix formats and operations
- Numerous linear solvers
- Nonlinear solvers
- Limited ODE integrators
- Limited parallel grid/data management
- Common interface for most DOE solver software

Structure of PETSc



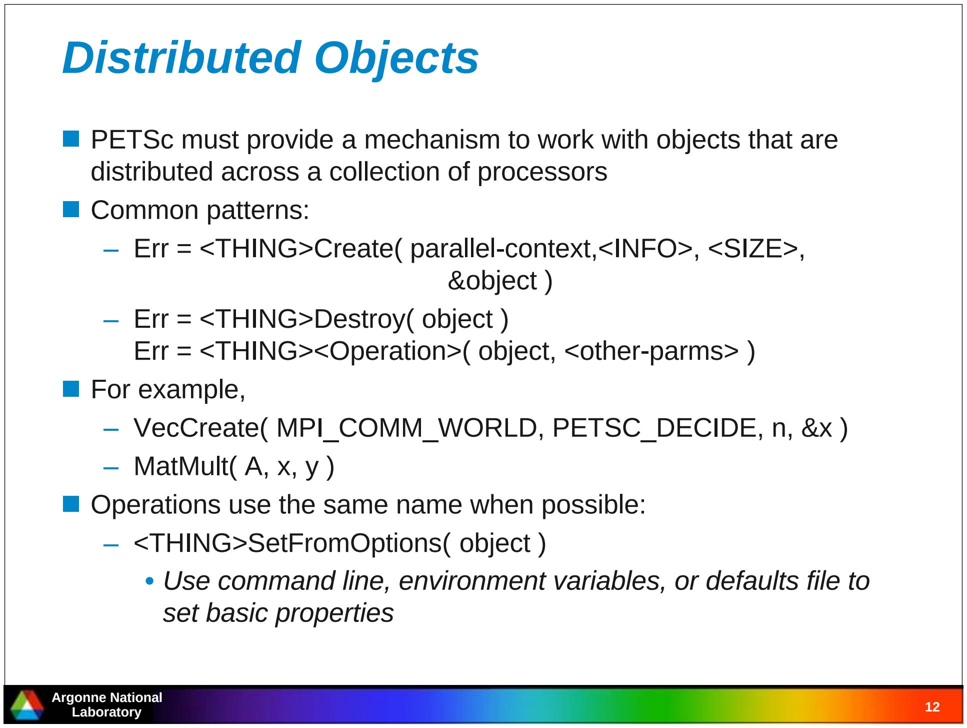
A Journey Through the PETSc Design

- Starting from the constraints on achieving effective parallel and single node performance, I will cover the basic design choices and rationales.
- Then I will show a complete parallel application, complete with performance instrumentation, a wide range of iterative and direct methods, and preconditioners



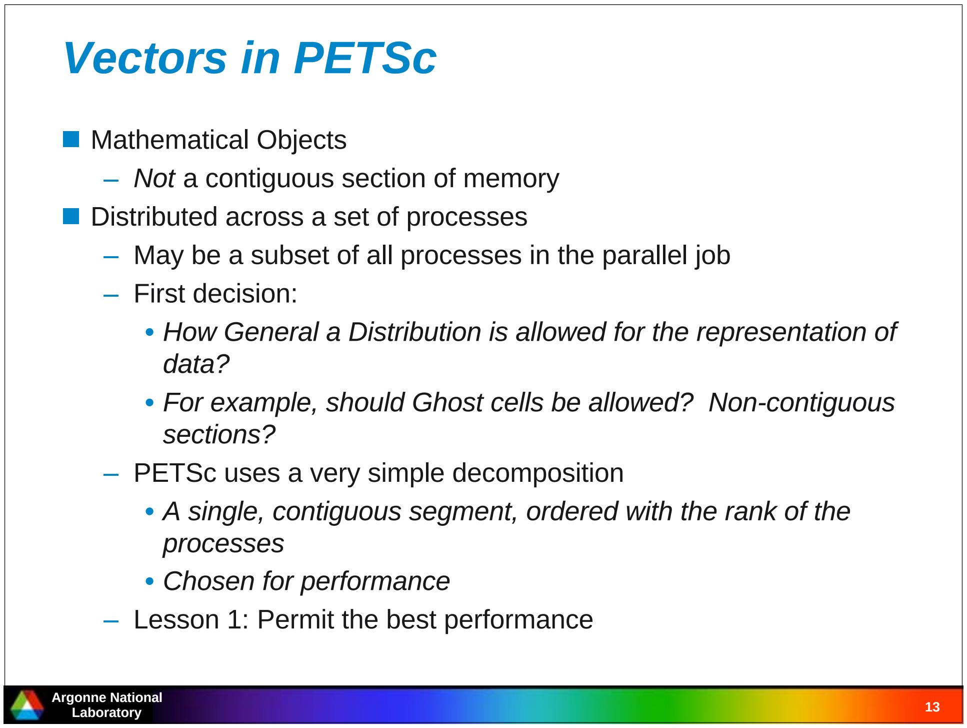
The Constraints - Parallel Computing Issues

- 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



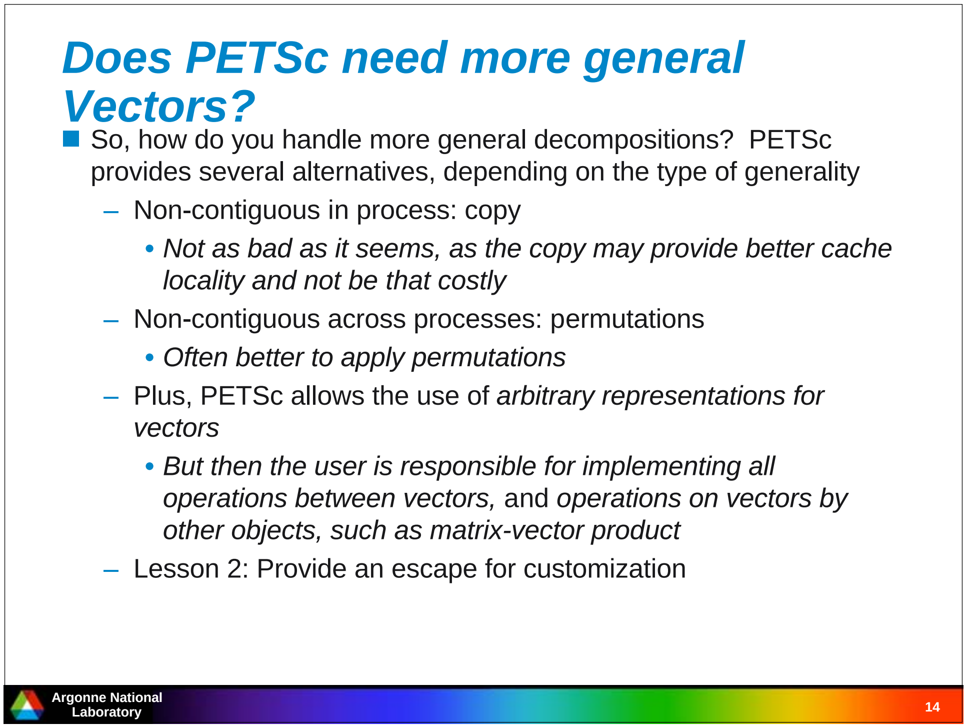
Distributed Objects

- 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*



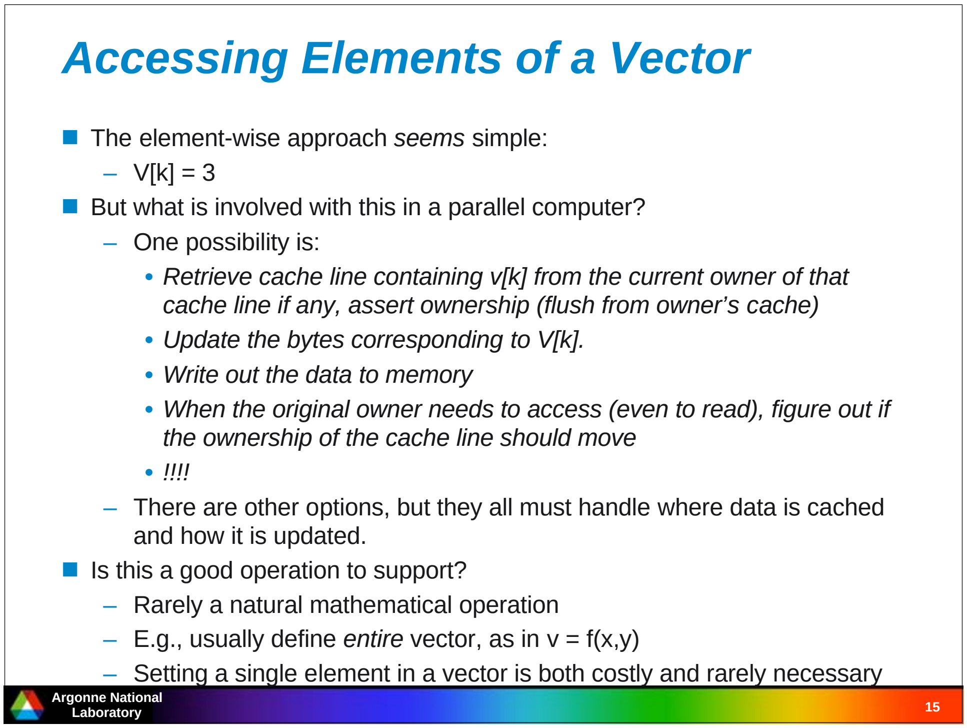
Vectors in PETSc

- 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



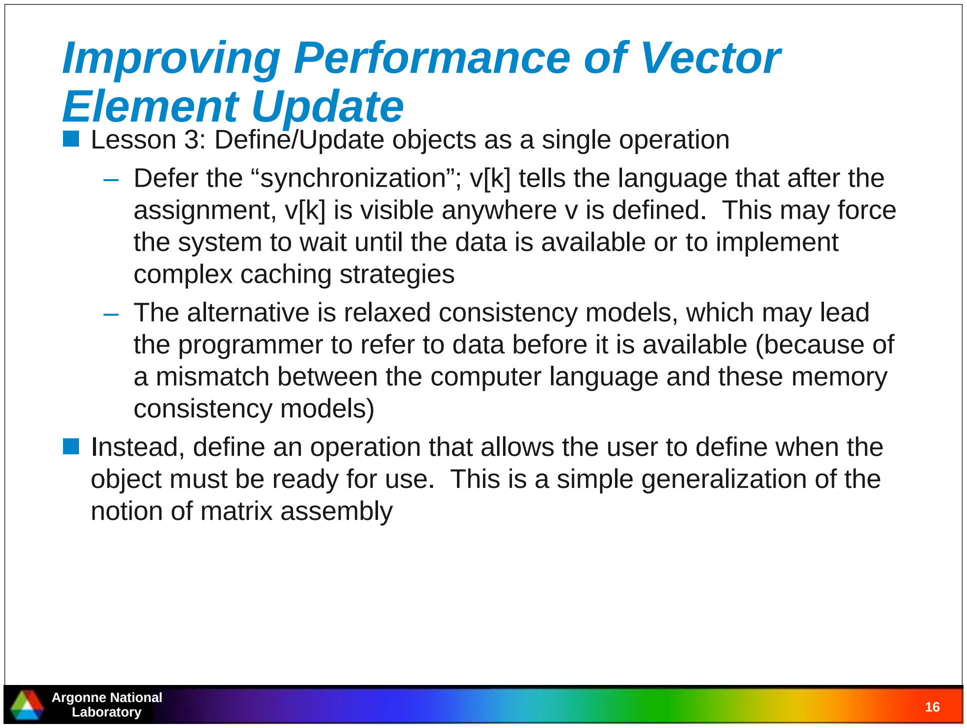
Does PETSc need more general Vectors?

- 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*
  - 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



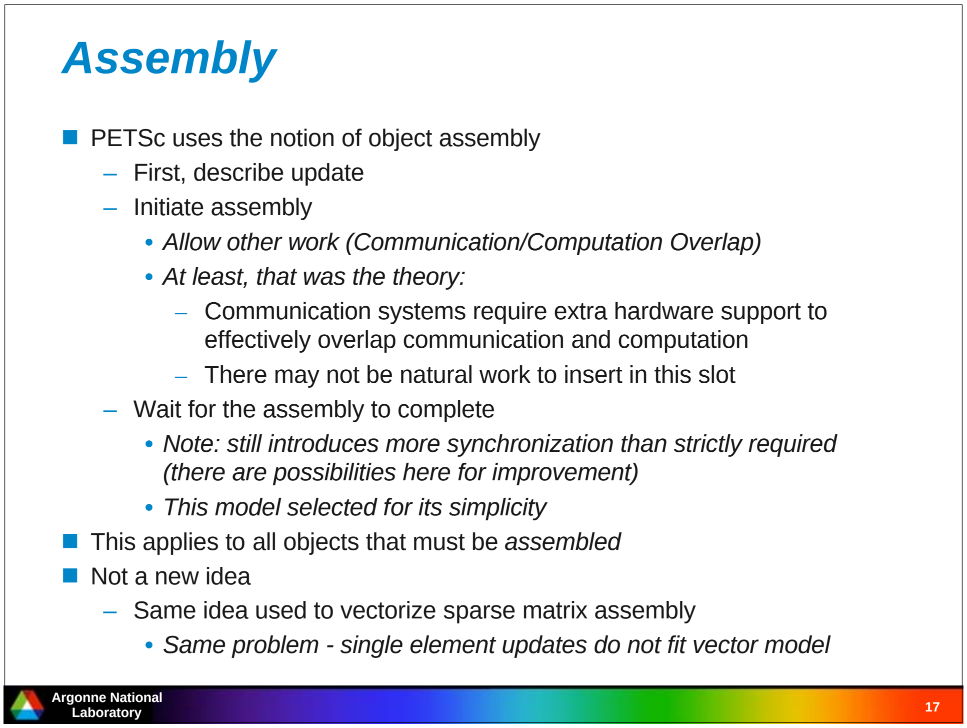
Accessing Elements of a Vector

- 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



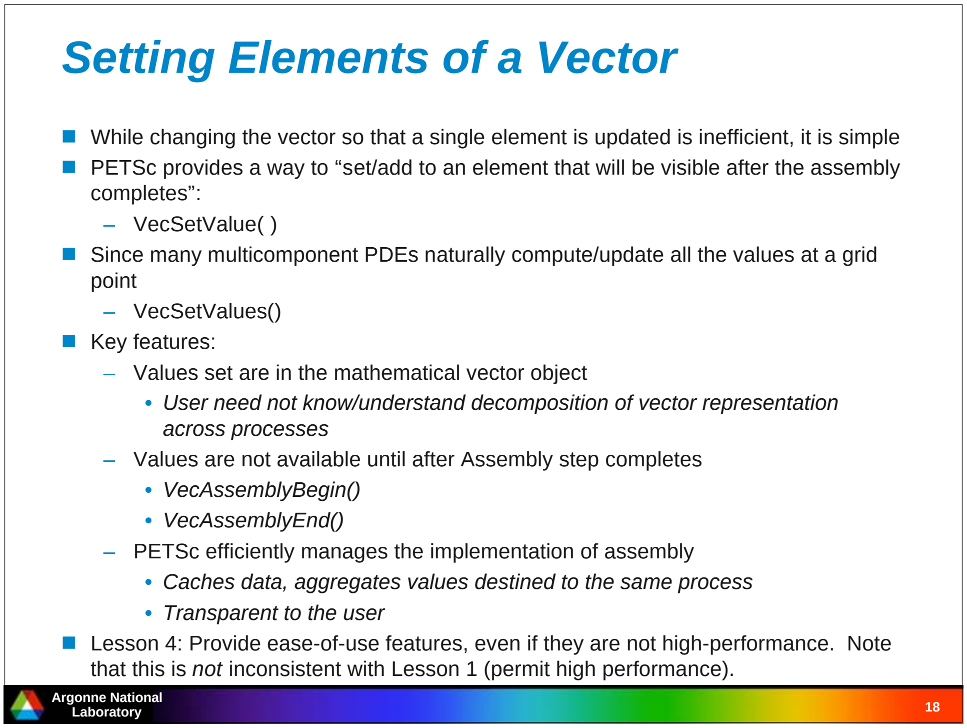
Improving Performance of Vector Element Update

- 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



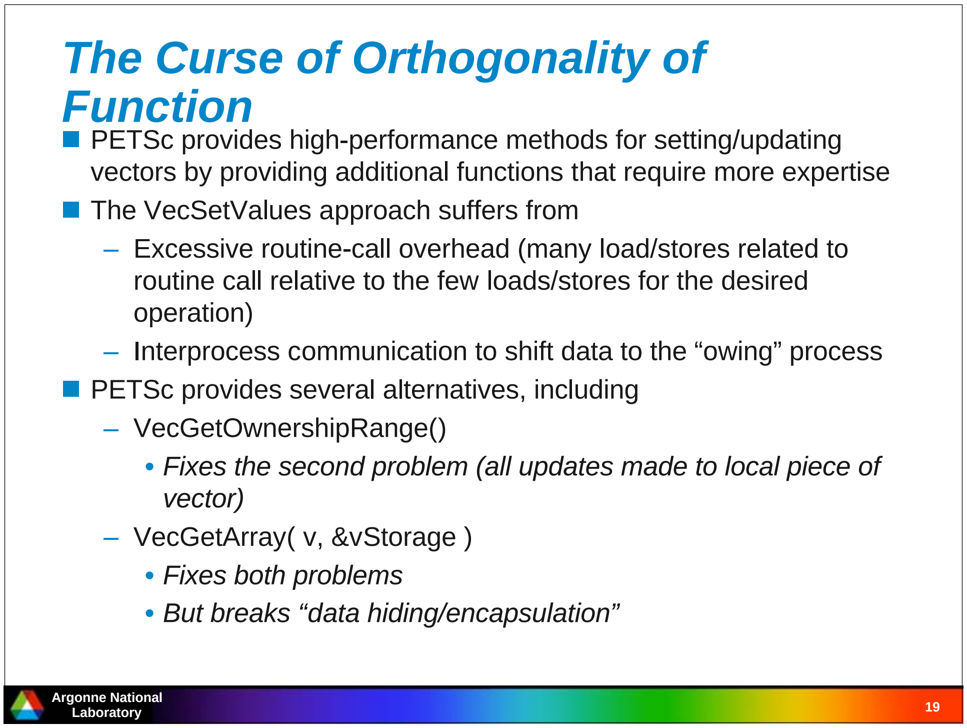
Assembly

- 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
    - *VecAssemblyBegin()*
    - *VecAssemblyEnd()*
  - 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).

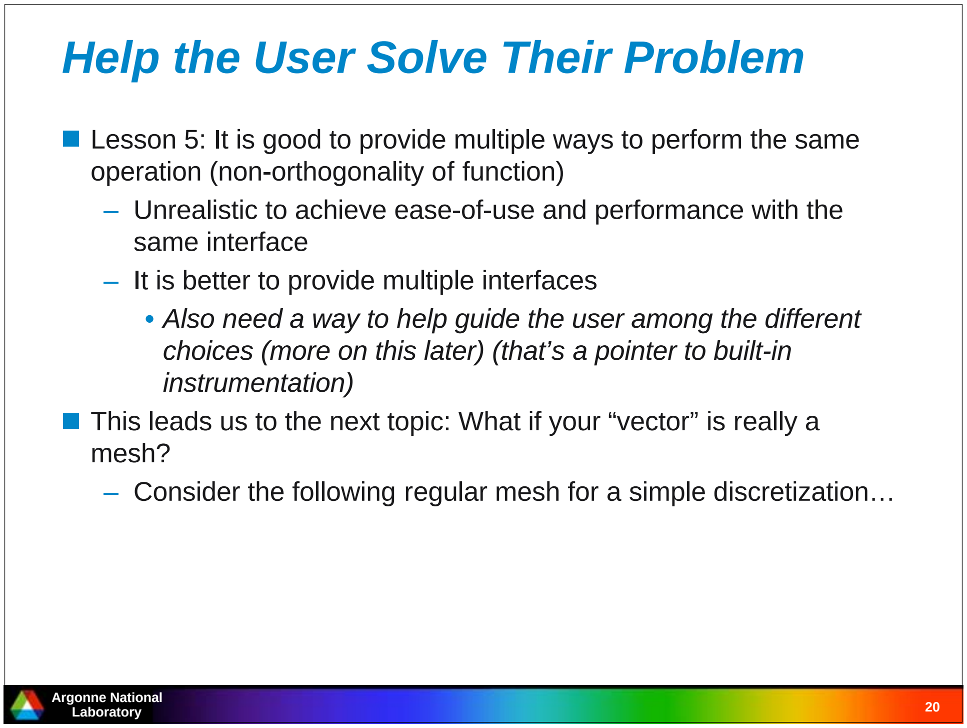


The Curse of Orthogonality of Function

- PETSc provides high-performance methods for setting/updating vectors by providing additional functions that require more expertise
- The VecSetValues approach suffers from
  - Excessive routine-call overhead (many load/stores related to routine call relative to the few loads/stores for the desired operation)
  - Interprocess communication to shift data to the “owing” process
- PETSc provides several alternatives, including
  - VecGetOwnershipRange()
    - *Fixes the second problem (all updates made to local piece of vector)*
  - VecGetArray(v, &vStorage)
    - *Fixes both problems*
    - *But breaks “data hiding/encapsulation”*

Argonne National Laboratory

19



Help the User Solve Their Problem

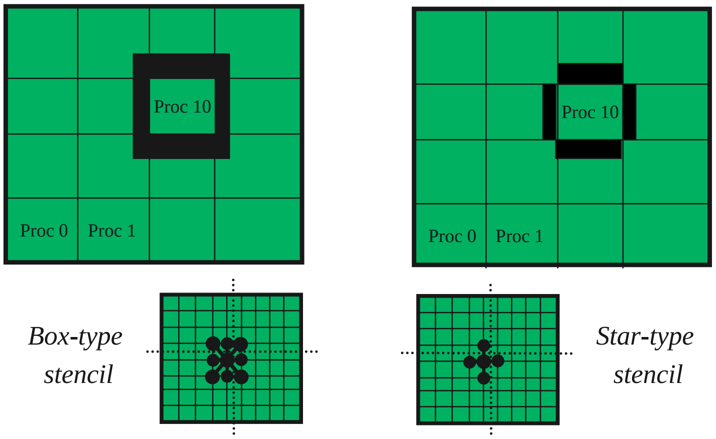
- Lesson 5: It is good to provide multiple ways to perform the same operation (non-orthogonality of function)
  - Unrealistic to achieve ease-of-use and performance with the same interface
  - It is better to provide multiple interfaces
    - *Also need a way to help guide the user among the different choices (more on this later) (that’s a pointer to built-in instrumentation)*
- This leads us to the next topic: What if your “vector” is really a mesh?
  - Consider the following regular mesh for a simple discretization…

Argonne National Laboratory

20

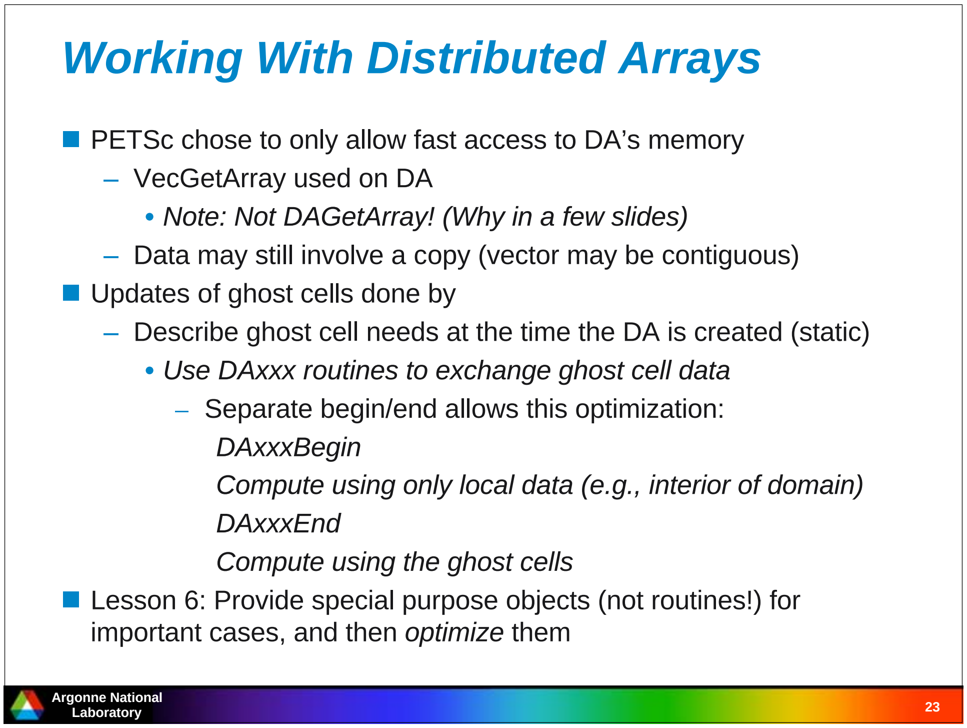
Distributed Arrays With Ghost Cells

Data layout and ghost values



Distributed Arrays as Extensions of Vectors

- PETSc defines a “Distributed Array” which is a
  - Multi-dimensional array
  - Optimized for stencil operations by providing “ghost cells”
- Same Issues as for vectors:
  - Element-wise operations are easy to describe
  - But an application (almost) never applies a stencil to a single point; always to entire (distributed) array
    - *May apply different stencils at different points, but that’s an aggregate operation if done properly*



Working With Distributed Arrays

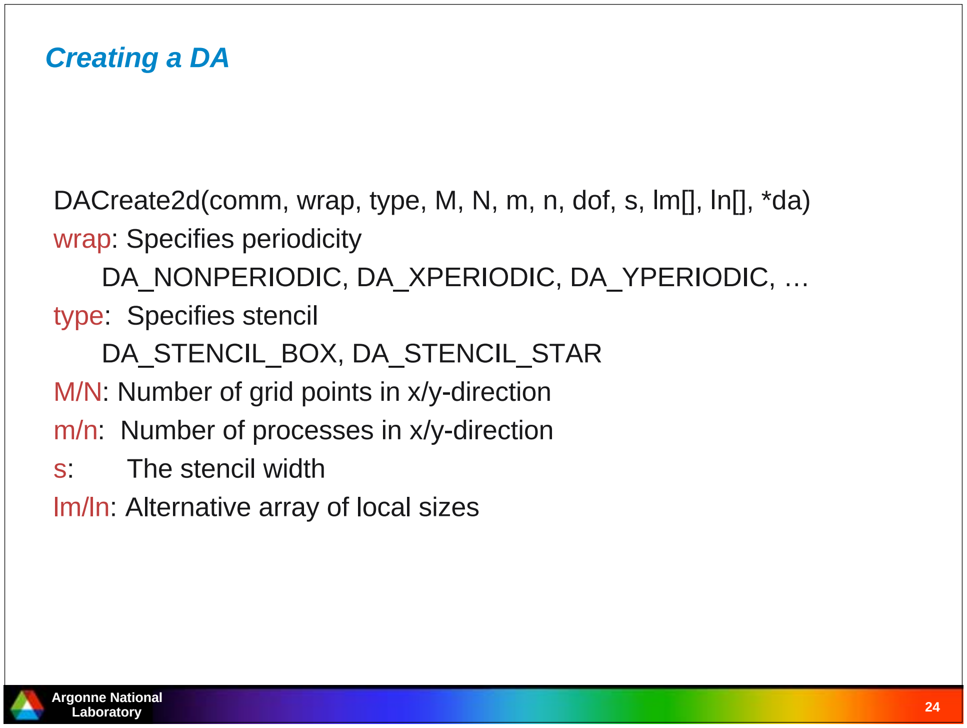
- PETSc chose to only allow fast access to DA's memory
  - VecGetArray used on DA
    - *Note: Not DAGetArray! (Why in a few slides)*
  - Data may still involve a copy (vector may be contiguous)
- Updates of ghost cells done by
  - Describe ghost cell needs at the time the DA is created (static)
    - *Use DAxxx routines to exchange ghost cell data*
      - Separate begin/end allows this optimization:

        DAxxxBegin

        Compute using only local data (e.g., interior of domain)

        DAxxxEnd

        Compute using the ghost cells
- Lesson 6: Provide special purpose objects (not routines!) for important cases, and then *optimize* them



Creating a DA

DACreate2d(comm, wrap, type, M, N, m, n, dof, s, lm[], ln[], *da)

wrap: Specifies periodicity

DA_NONPERIODIC, DA_XPERIODIC, DA_YPERIODIC, …

type: Specifies stencil

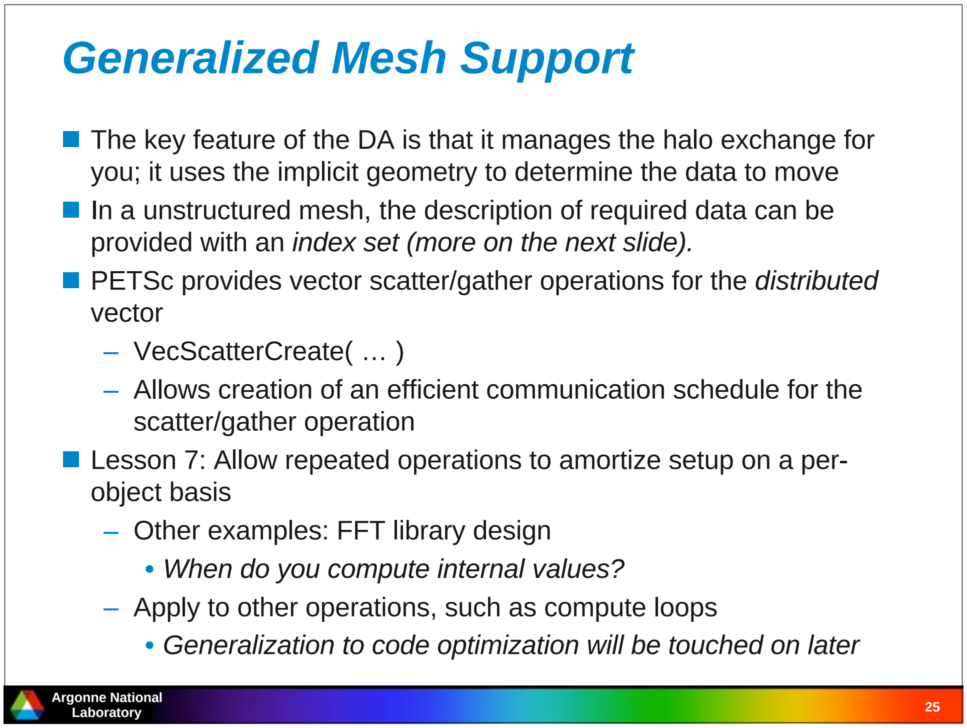
DA_STENCIL_BOX, DA_STENCIL_STAR

M/N: Number of grid points in x/y-direction

m/n: Number of processes in x/y-direction

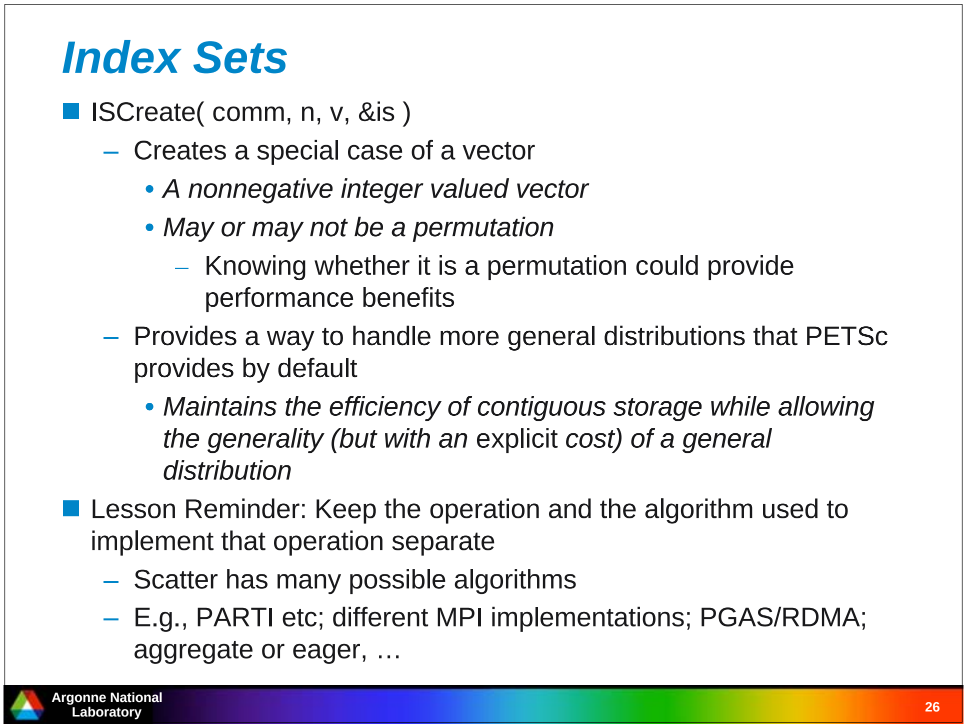
s: The stencil width

lm/ln: Alternative array of local sizes



Generalized Mesh Support

- The key feature of the DA is that it manages the halo exchange for you; it uses the implicit geometry to determine the data to move
- In a unstructured mesh, the description of required data can be provided with an *index set (more on the next slide).*
- PETSc provides vector scatter/gather operations for the *distributed* vector
  - VecScatterCreate(…)
  - Allows creation of an efficient communication schedule for the scatter/gather operation
- Lesson 7: Allow repeated operations to amortize setup on a per-object basis
  - Other examples: FFT library design
    - *When do you compute internal values?*
  - Apply to other operations, such as compute loops
    - *Generalization to code optimization will be touched on later*



Index Sets

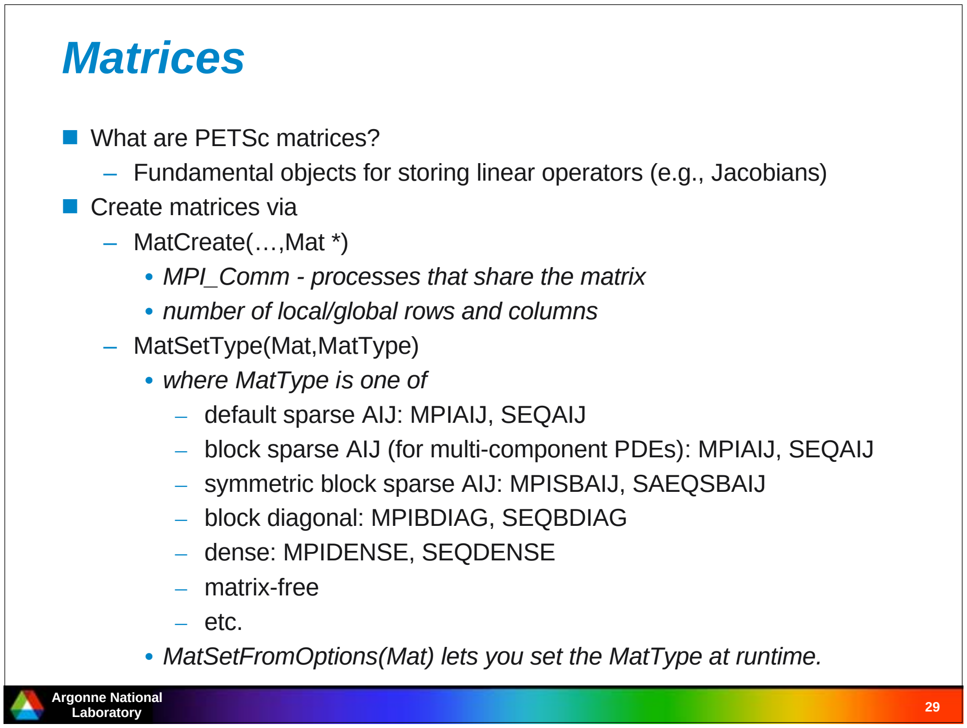
- ISCreate(comm, n, v, &is)
  - Creates a special case of a vector
    - *A nonnegative integer valued vector*
    - *May or may not be a permutation*
      - Knowing whether it is a permutation could provide performance benefits
  - Provides a way to handle more general distributions that PETSc provides by default
    - *Maintains the efficiency of contiguous storage while allowing the generality (but with an* explicit *cost) of a general distribution*
- Lesson Reminder: Keep the operation and the algorithm used to implement that operation separate
  - Scatter has many possible algorithms
  - E.g., PARTI etc; different MPI implementations; PGAS/RDMA; aggregate or eager, …

Object Oriented Design

- Same ideas continue into the Matrix, Linear Solver, Nonlinear Solver, …objects and operations
- This is an example of Object-Oriented Design (O-O)
  - Define the objects (e.g., Matrix) and the operations that act on them (methods)
  - Think of the objects as a class and consider the natural and necessary operations upon them
  - Internal information, such as the data structures used to represent the objects or to efficiently operate upon them, are not exposed to the user of the object (usually)
- O-O provides a implementation strategy for the ideas here
- O-O focuses on the objects, their relationships, and the operations on and between them. The particular code or function *may not be known until run time* (more later)
- O-O can be used in any computer language, though some are easier than others (PETSc is written almost entirely in C)

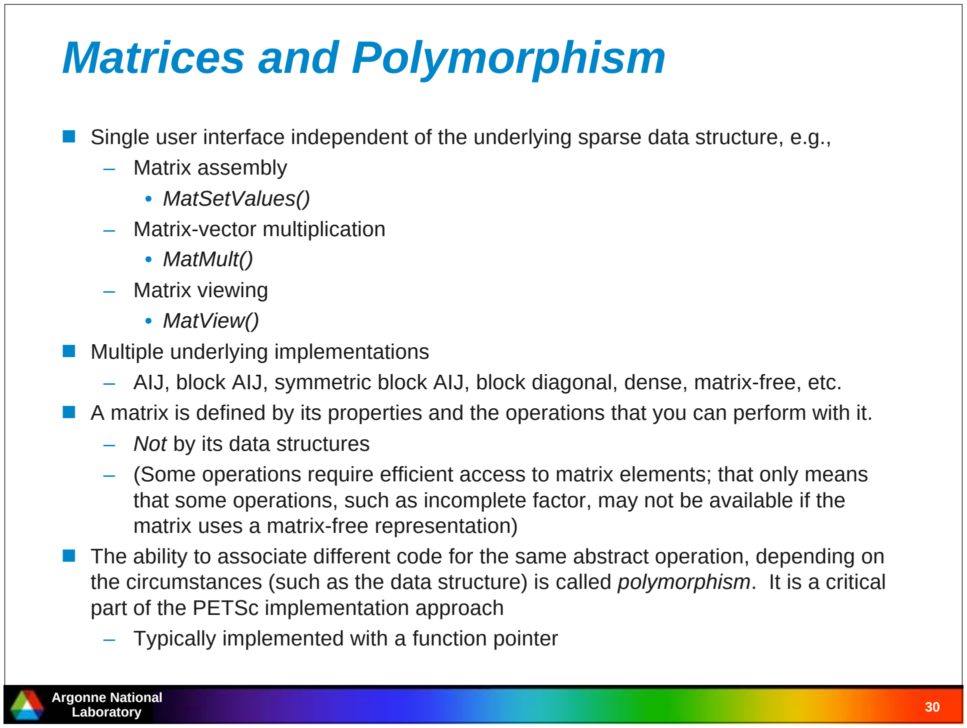
Inheritance

- It is typical that one object is an extension of another
- For example, a discretization mesh is a vector with additional properties, such as the geometric location of each element
- Some operations require this geometric knowledge, e.g., display, geometric-based preconditioners, while others, such as matrix-vector product, do not
- In Computer Science, an object (such as a mesh) that extends another object (such as a vector) is said to *inherit* from the base object.
- Inheritance does not require C++ or Java
- Inheritance helps organize the objects in your library by providing a well-defined taxonomy.
- Lesson 8: Use the principles of object-oriented design to help use hierarchy to structure a library --- use fewer basic concepts to simplify understanding, and use concepts such as inheritance to help (even if your computer language does not directly support it).



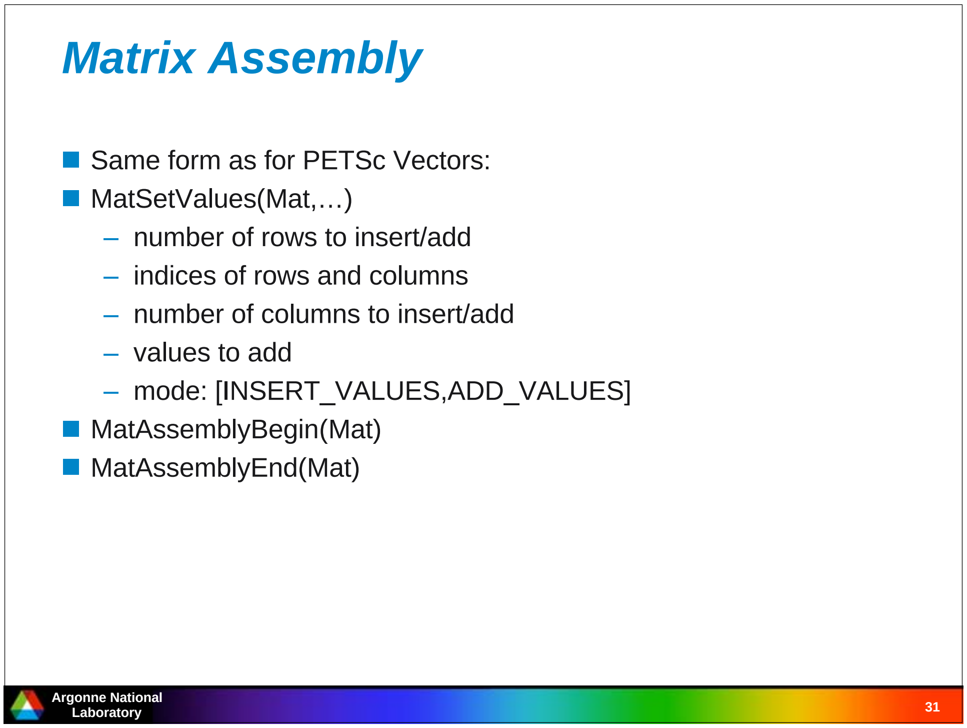
Matrices

- 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.*



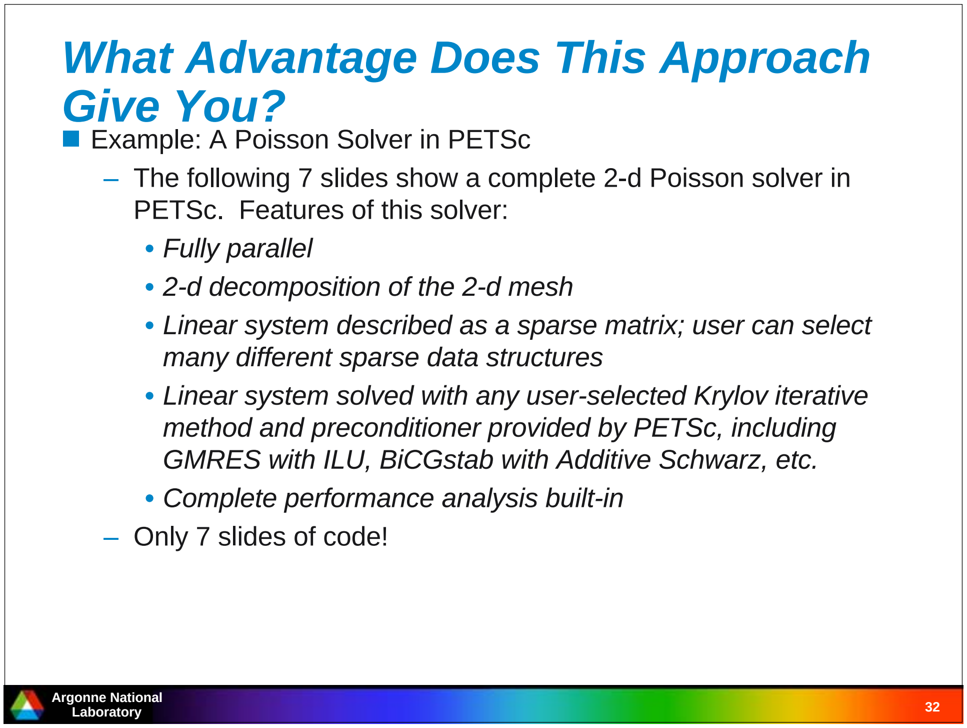
Matrices and Polymorphism

- 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.
- 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



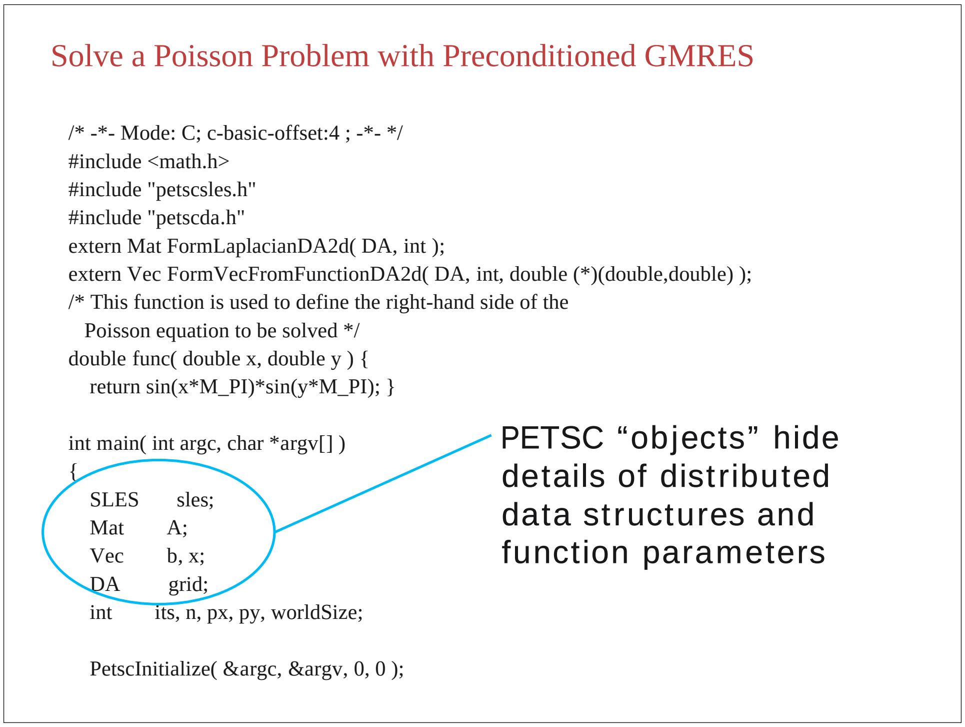
Matrix Assembly

- Same form as for PETSc Vectors:
- MatSetValues(Mat,…)
  - number of rows to insert/add
  - indices of rows and columns
  - number of columns to insert/add
  - values to add
  - mode: [INSERT_VALUES,ADD_VALUES]
- MatAssemblyBegin(Mat)
- MatAssemblyEnd(Mat)



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!



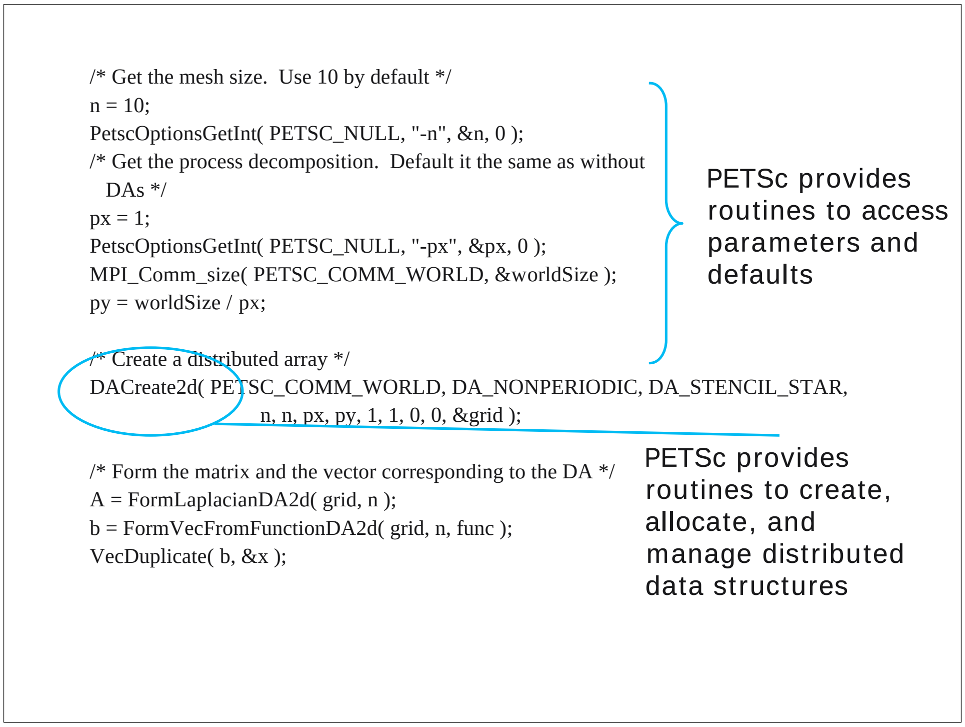
Solve a Poisson Problem with Preconditioned GMRES

```
/* -*- 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



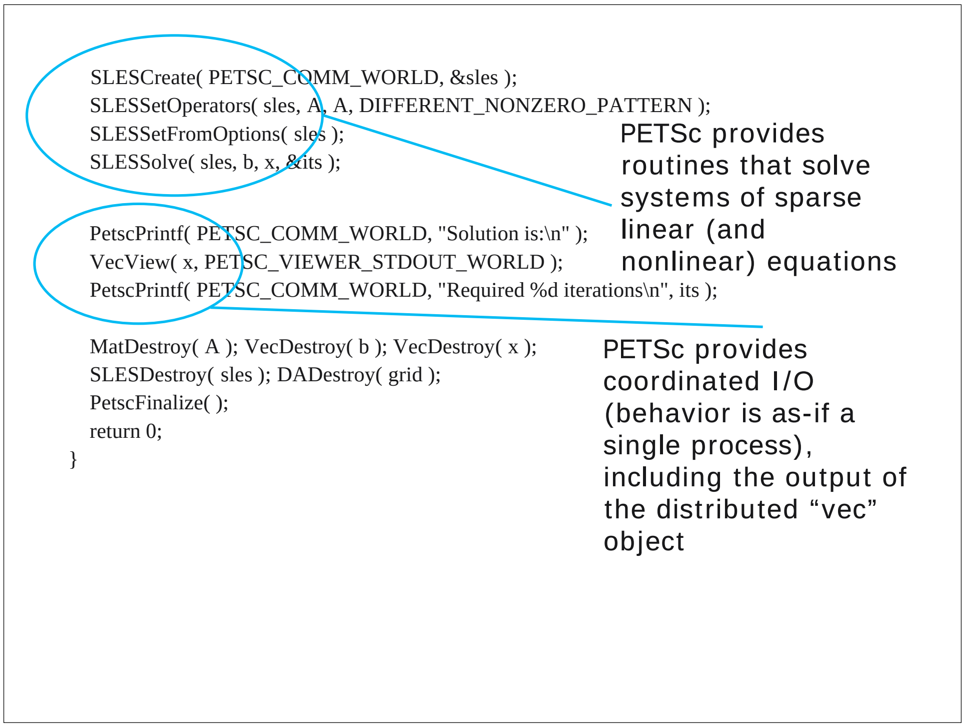
```
/* Get the mesh size.  Use 10 by default */
n = 10;
PetscOptionsGetInt( PETSC_NULL, "-n", &n, 0 );
/* Get the process decomposition.  Default it the same as without
   DAs */
px = 1;
PetscOptionsGetInt( PETSC_NULL, "-px", &px, 0 );
MPI_Comm_size( PETSC_COMM_WORLD, &worldSize );
py = worldSize / px;

/* 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



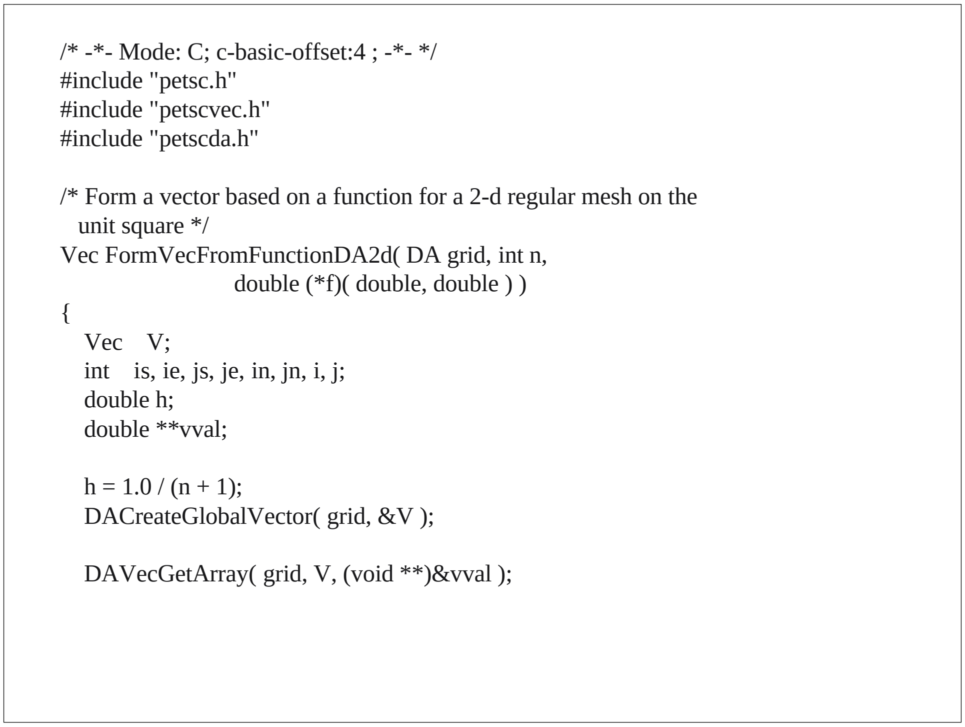
```
    SLESCreate( PETSC_COMM_WORLD, &sles );
    SLESSetOperators( sles, A, A, DIFFERENT_NONZERO_PATTERN );
    SLESSetFromOptions( sles );
    SLESSolve( sles, b, x, &its );

    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

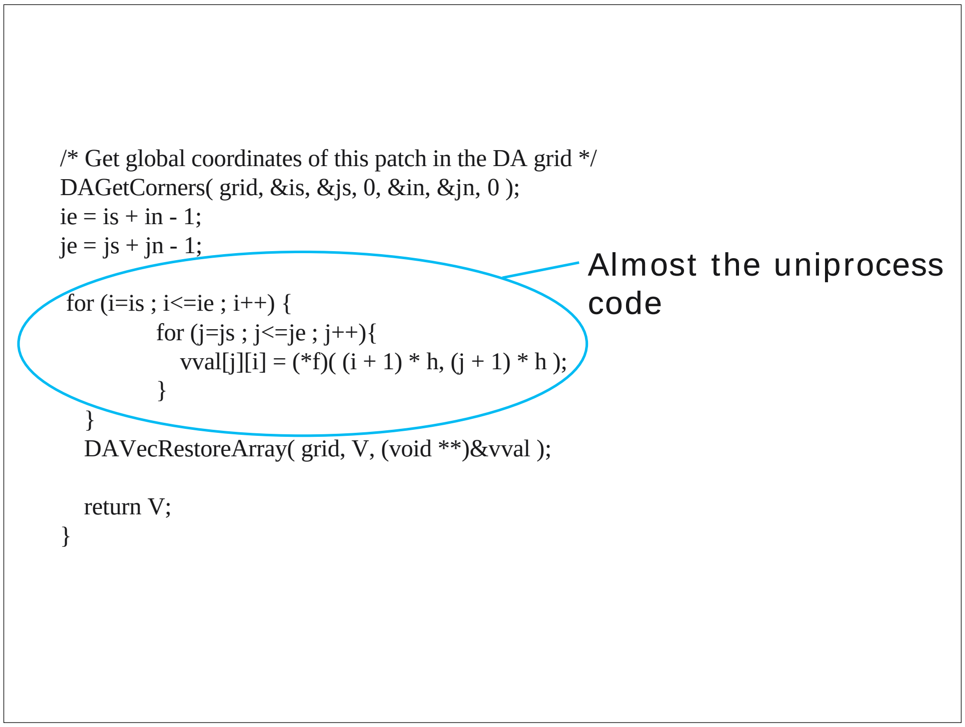


```
/* -*- Mode: C; c-basic-offset:4 ; -*- */
#include "petsc.h"
#include "petscvec.h"
#include "petscda.h"

/* 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 );
```

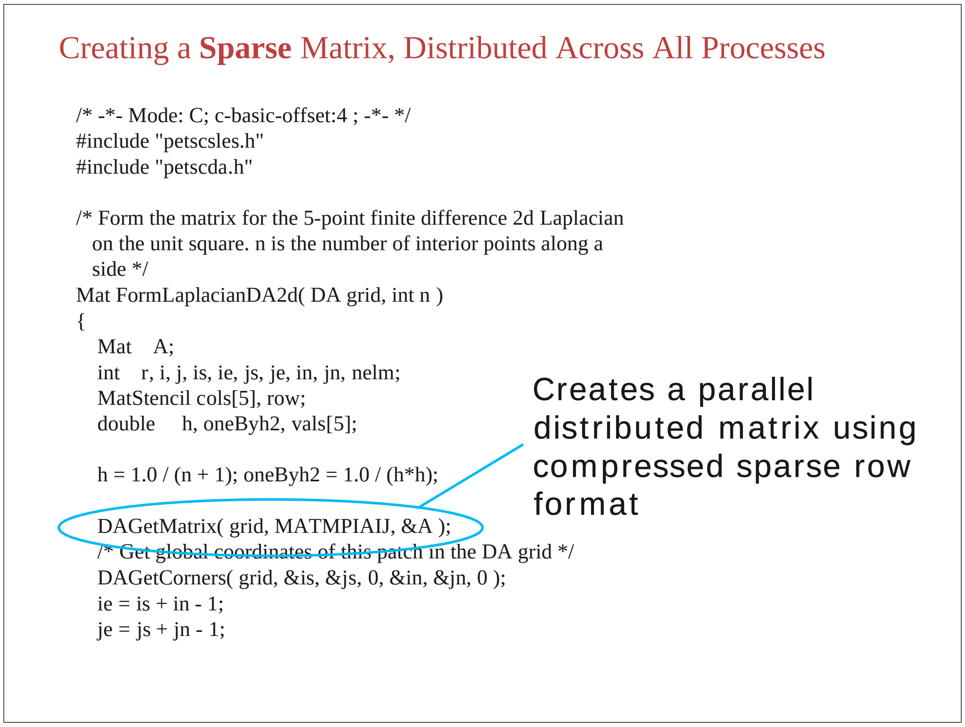


```
/* Get global coordinates of this patch in the DA grid */
DAGetCorners( grid, &is, &js, 0, &in, &jn, 0 );
ie = is + in - 1;
je = js + jn - 1;

 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

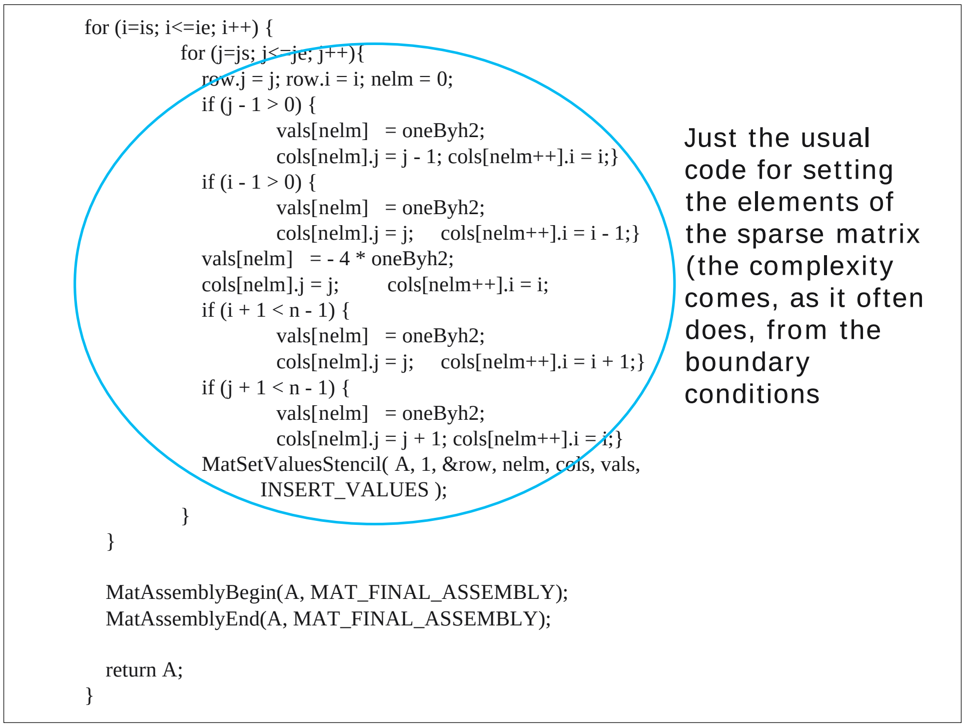
```
/* -*- Mode: C; c-basic-offset:4 ; -*- */
#include "petscsles.h"
#include "petscda.h"

/* 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

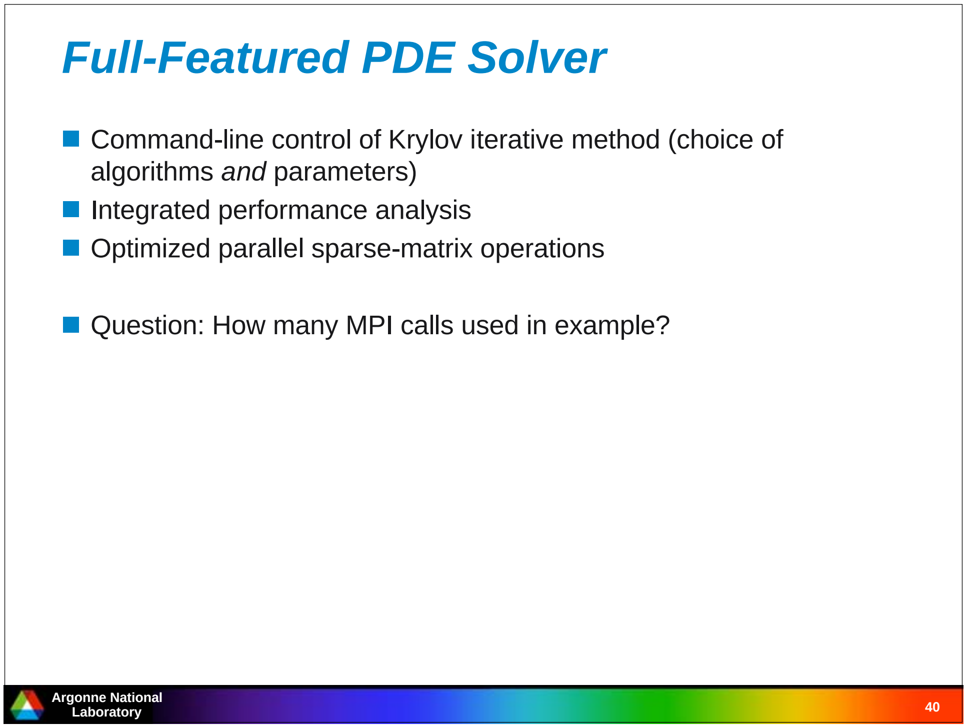


```
for (i=is; i<=ie; i++) {
          for (j=js; j<=je; j++){
            row.j = j; row.i = i; nelm = 0;
            if (j - 1 > 0) {
                    vals[nelm]   = oneByh2;
                    cols[nelm].j = j - 1; cols[nelm++].i = i;}
            if (i - 1 > 0) {
                    vals[nelm]   = 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;
}
```

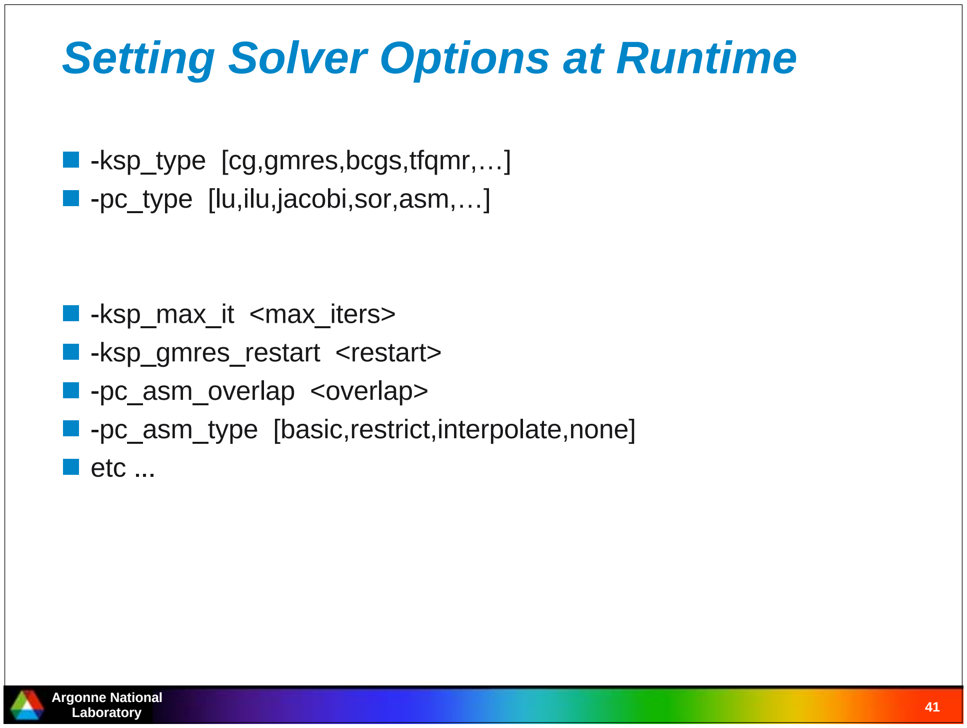
Just the usual code for setting the elements of the sparse matrix (the complexity comes, as it often does, from the boundary conditions



Full-Featured PDE Solver

- Command-line control of Krylov iterative method (choice of algorithms *and* parameters)
- Integrated performance analysis
- Optimized parallel sparse-matrix operations

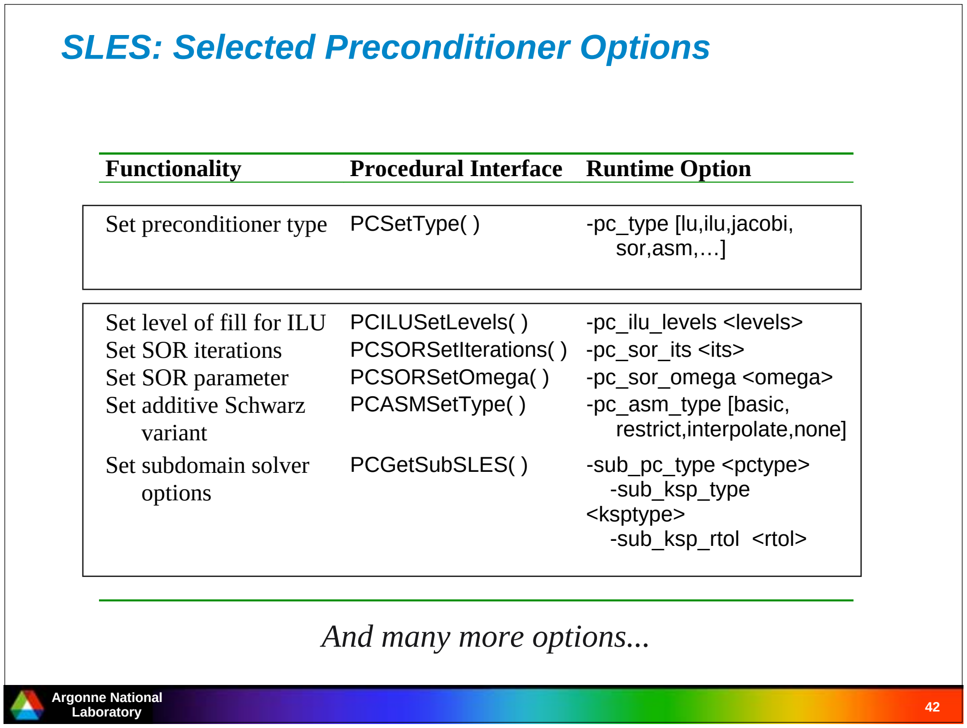
- Question: How many MPI calls used in example?



Setting Solver Options at Runtime

- -ksp_type [cg,gmres,bcgs,tfqmr,…]
- -pc_type [lu,ilu,jacobi,sor,asm,…]

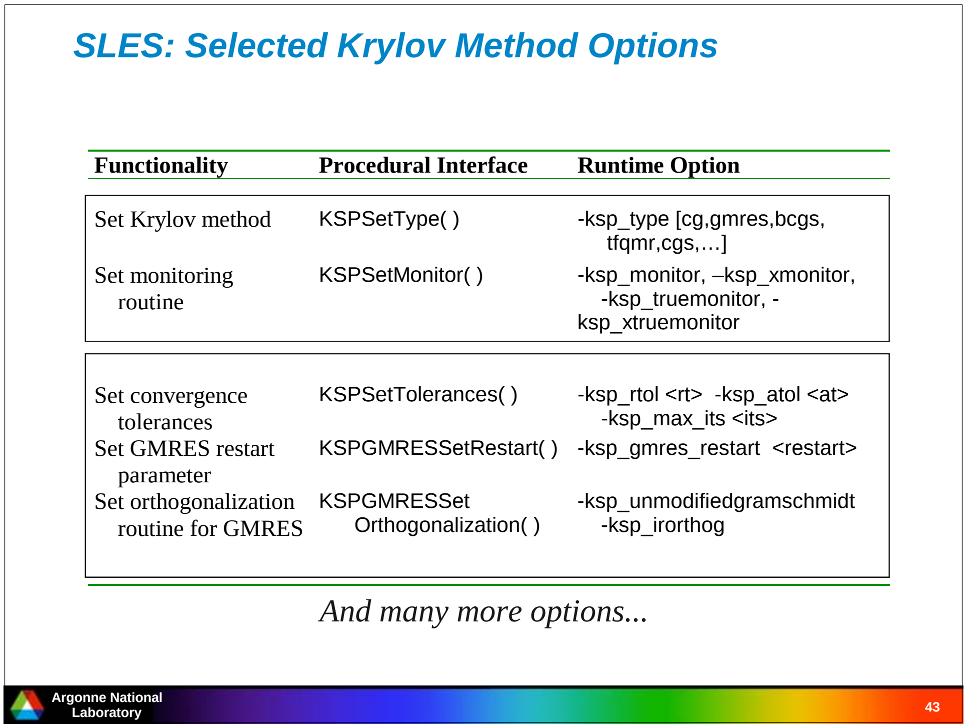
- -ksp_max_it <max_iters>
- -ksp_gmres_restart <restart>
- -pc_asm_overlap <overlap>
- -pc_asm_type [basic,restrict,interpolate,none]
- etc ...



SLES: Selected Preconditioner Options

Functionality	Procedural Interface	Runtime Option
Set preconditioner type	PCSetType( )	-pc_type [lu,ilu,jacobi, sor,asm,…]
Set level of fill for ILU	PCILUSetLevels( )	-pc_ilu_levels <levels>
Set SOR iterations	PCSORSetIterations( )	-pc_sor_its <its>
Set SOR parameter	PCSORSetOmega( )	-pc_sor_omega <omega>
Set additive Schwarz variant	PCASMSetType( )	-pc_asm_type [basic, restrict,interpolate,none]
Set subdomain solver options	PCGetSubSLES( )	-sub_pc_type <pctype> -sub_ksp_type <ksptype> -sub_ksp_rtol <rtol>

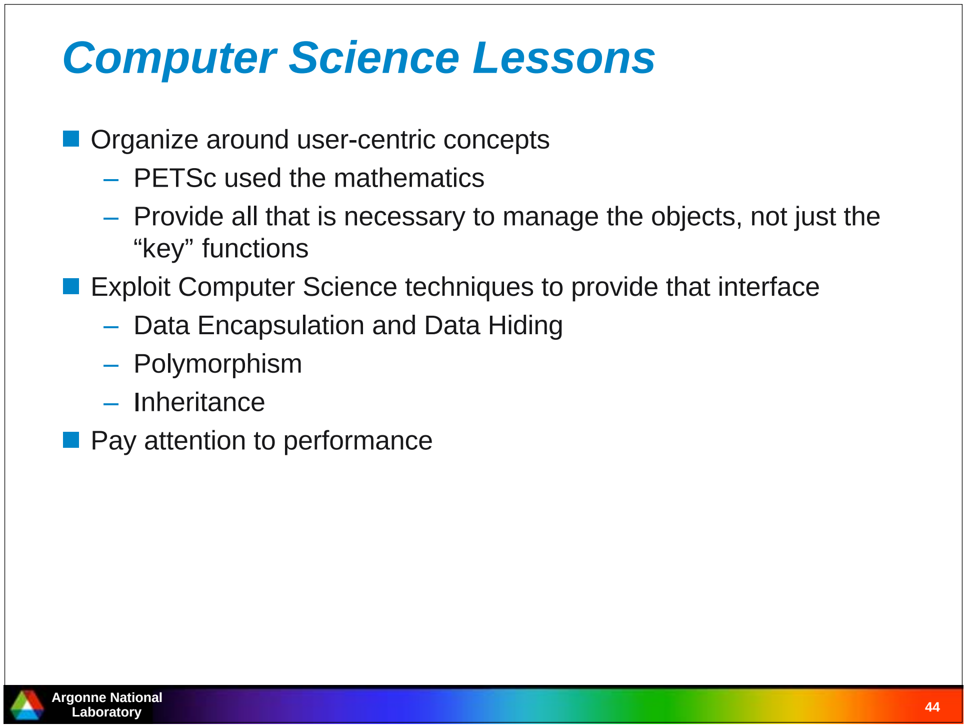
And many more options...



SLES: Selected Krylov Method Options

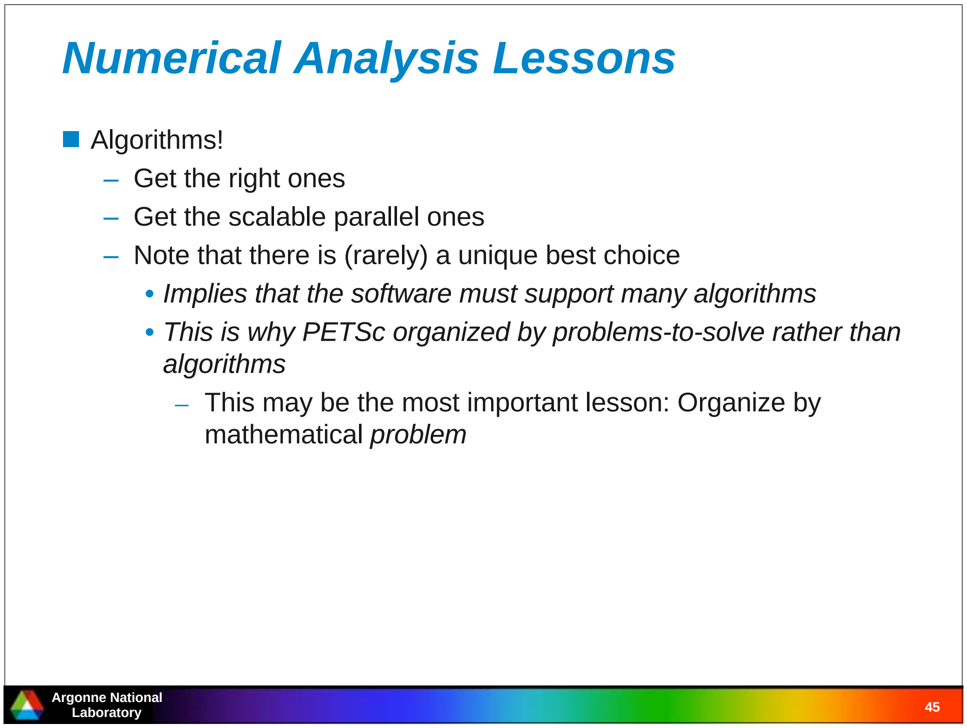
Functionality	Procedural Interface	Runtime Option
Set Krylov method	KSPSetType( )	-ksp_type [cg,gmres,bcgs, tfqmr,cgs,…]
Set monitoring routine	KSPSetMonitor( )	-ksp_monitor, –ksp_xmonitor, -ksp_truemonitor, -ksp_xtruemonitor
Set convergence tolerances	KSPSetTolerances( )	-ksp_rtol <rt> -ksp_atol <at> -ksp_max_its <its>
Set GMRES restart parameter	KSPGMRESSetRestart( )	-ksp_gmres_restart <restart>
Set orthogonalization routine for GMRES	KSPGMRESSet Orthogonalization( )	-ksp_unmodifiedgramschmidt -ksp_irorthog

And many more options...



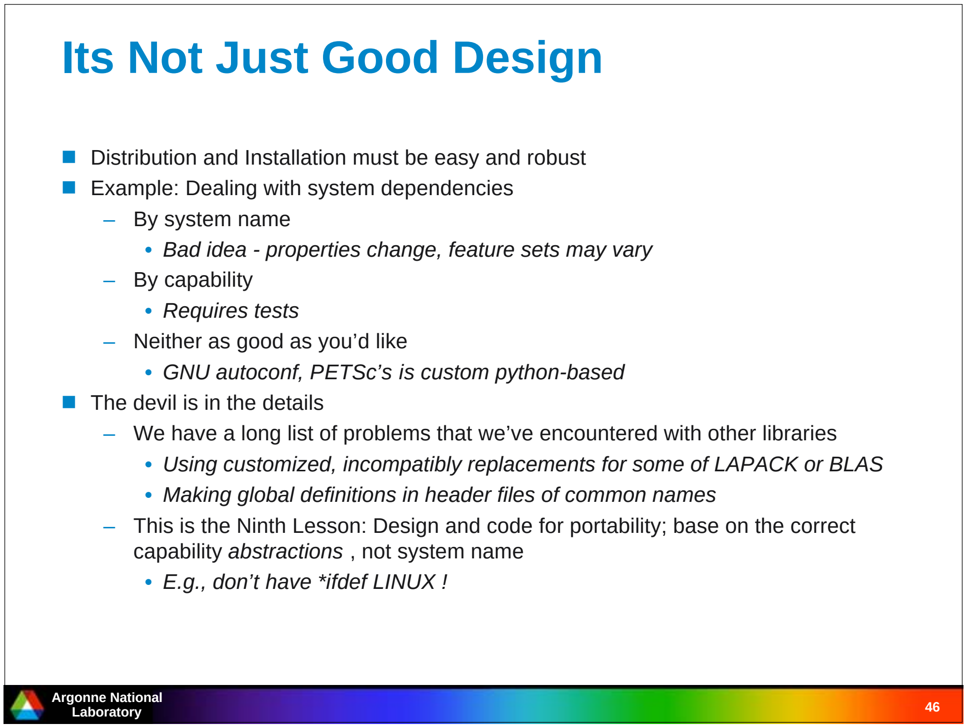
Computer Science Lessons

- 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



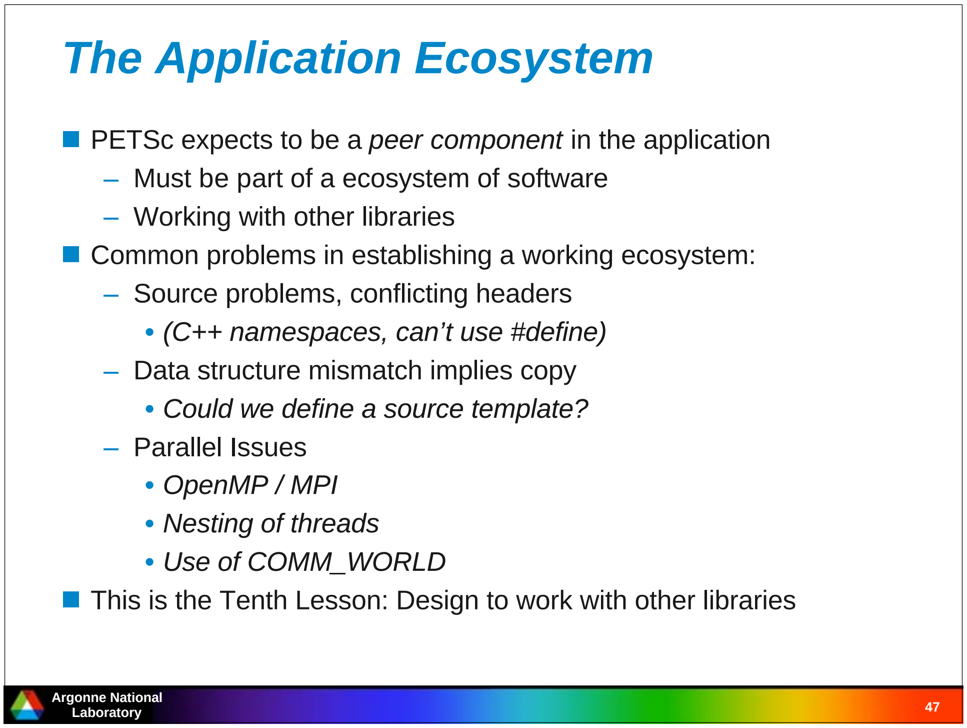
Numerical Analysis Lessons

- 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*



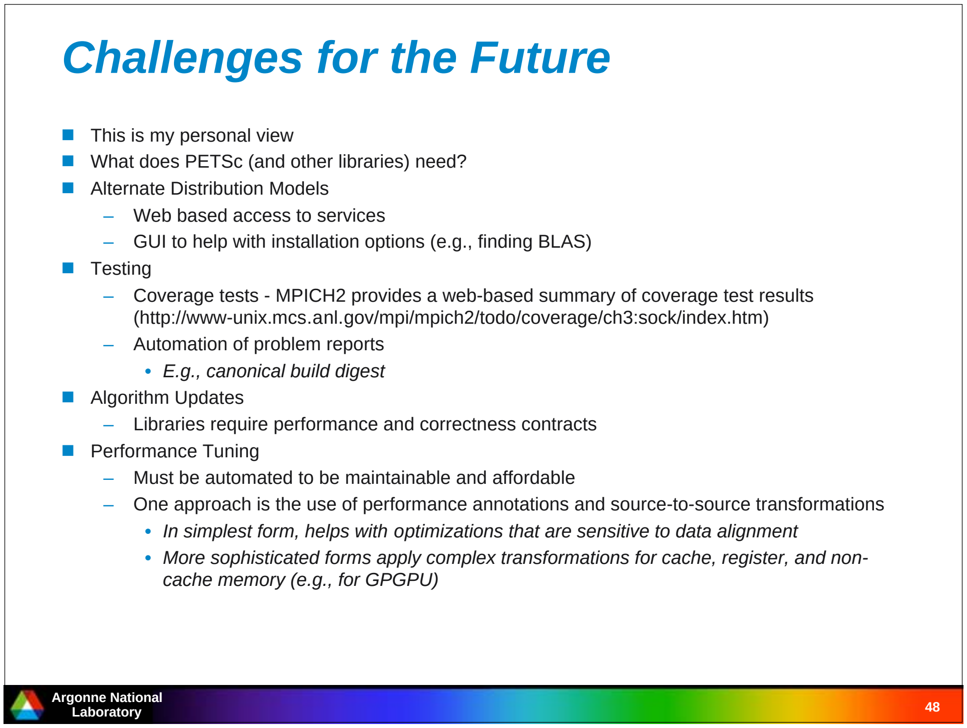
Its Not Just Good Design

- Distribution and Installation must be easy and robust
- Example: Dealing with system dependencies
  - By system name
    - *Bad idea - properties change, feature sets may vary*
  - By capability
    - *Requires tests*
  - Neither as good as you'd like
    - *GNU autoconf, PETSc's is custom python-based*
- The devil is in the details
  - We have a long list of problems that we've encountered with other libraries
    - *Using customized, incompatibly replacements for some of LAPACK or BLAS*
    - *Making global definitions in header files of common names*
  - This is the Ninth Lesson: Design and code for portability; base on the correct capability *abstractions* , not system name
    - *E.g., don't have *ifdef LINUX !*



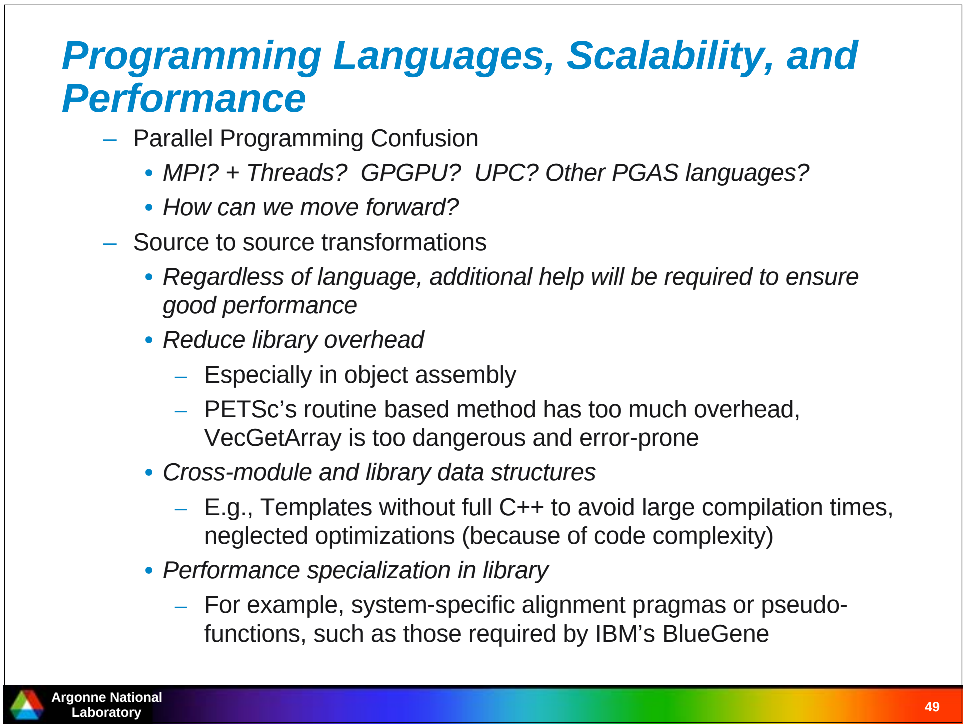
The Application Ecosystem

- PETSc expects to be a *peer component* in the application
  - Must be part of a ecosystem of software
  - Working with other libraries
- Common problems in establishing a working ecosystem:
  - Source problems, conflicting headers
    - *(C++ namespaces, can't use #define)*
  - Data structure mismatch implies copy
    - *Could we define a source template?*
  - Parallel Issues
    - *OpenMP / MPI*
    - *Nesting of threads*
    - *Use of COMM_WORLD*
- This is the Tenth Lesson: Design to work with other libraries



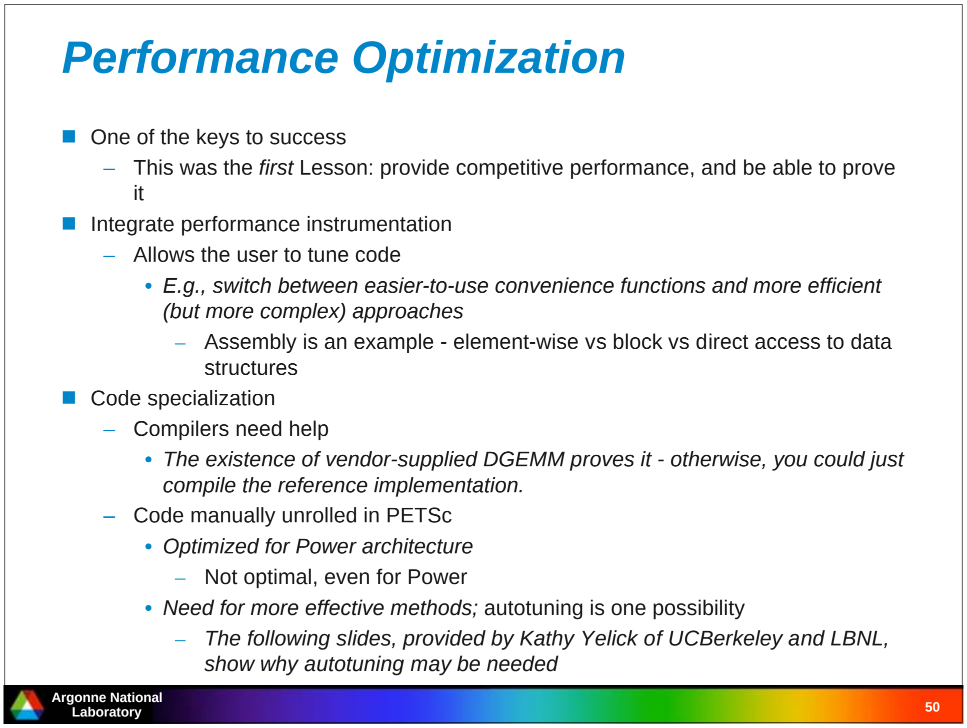
Challenges for the Future

- This is my personal view
- What does PETSc (and other libraries) need?
- Alternate Distribution Models
  - Web based access to services
  - GUI to help with installation options (e.g., finding BLAS)
- Testing
  - Coverage tests - MPICH2 provides a web-based summary of coverage test results (http://www-unix.mcs.anl.gov/mpi/mpich2/todo/coverage/ch3:sock/index.htm)
  - Automation of problem reports
    - *E.g., canonical build digest*
- Algorithm Updates
  - Libraries require performance and correctness contracts
- Performance Tuning
  - Must be automated to be maintainable and affordable
  - One approach is the use of performance annotations and source-to-source transformations
    - *In simplest form, helps with optimizations that are sensitive to data alignment*
    - *More sophisticated forms apply complex transformations for cache, register, and non-cache memory (e.g., for GPGPU)*



Programming Languages, Scalability, and Performance

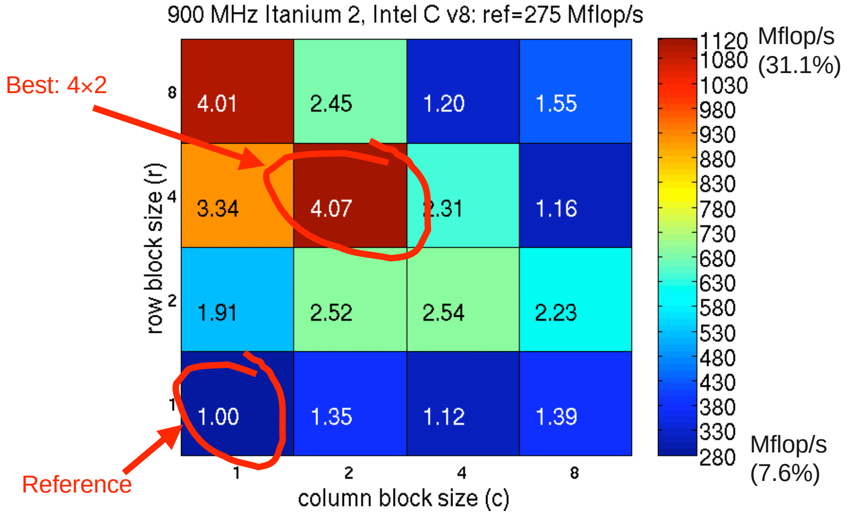
- Parallel Programming Confusion
  - *MPI? + Threads? GPGPU? UPC? Other PGAS languages?*
  - *How can we move forward?*
- Source to source transformations
  - *Regardless of language, additional help will be required to ensure good performance*
  - *Reduce library overhead*
    - Especially in object assembly
    - PETSc’s routine based method has too much overhead, VecGetArray is too dangerous and error-prone
  - *Cross-module and library data structures*
    - E.g., Templates without full C++ to avoid large compilation times, neglected optimizations (because of code complexity)
  - *Performance specialization in library*
    - For example, system-specific alignment pragmas or pseudo-functions, such as those required by IBM’s BlueGene



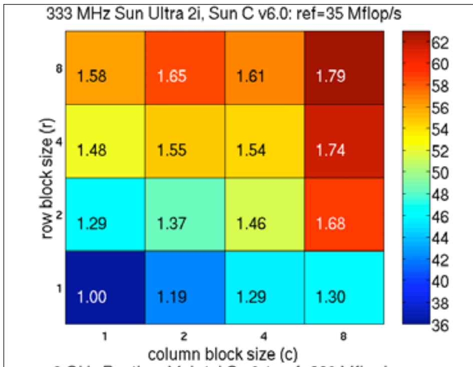
Performance Optimization

- One of the keys to success
  - This was the *first* Lesson: provide competitive performance, and be able to prove it
- Integrate performance instrumentation
  - Allows the user to tune code
    - *E.g., switch between easier-to-use convenience functions and more efficient (but more complex) approaches*
      - Assembly is an example - element-wise vs block vs direct access to data structures
- Code specialization
  - Compilers need help
    - *The existence of vendor-supplied DGEMM proves it - otherwise, you could just compile the reference implementation.*
  - Code manually unrolled in PETSc
    - *Optimized for Power architecture*
      - Not optimal, even for Power
    - *Need for more effective methods;* autotuning is one possibility
      - *The following slides, provided by Kathy Yelick of UCBerkeley and LBNL, show why autotuning may be needed*

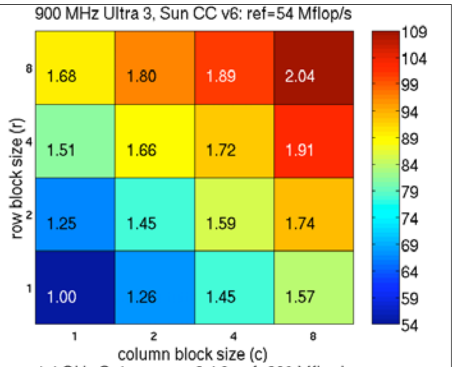
Speedups on Itanium 2: The need for search



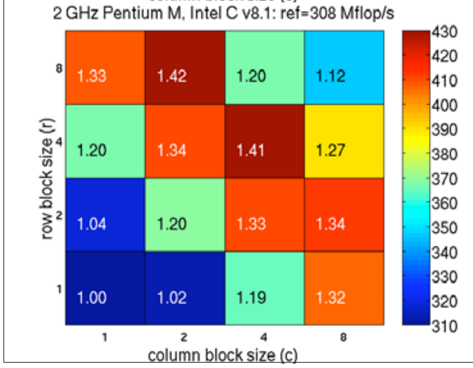
	1	2	4	8
8	4.01	2.45	1.20	1.55
4	3.34	4.07	2.31	1.16
2	1.91	2.52	2.54	2.23
1	1.00	1.35	1.12	1.39



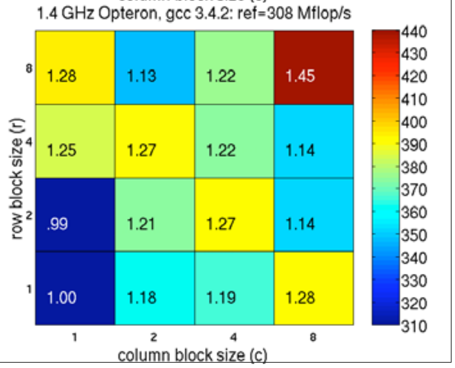
	1	2	4	8
8	1.58	1.65	1.61	1.79
4	1.48	1.55	1.54	1.74
2	1.29	1.37	1.46	1.68
1	1.00	1.19	1.29	1.30



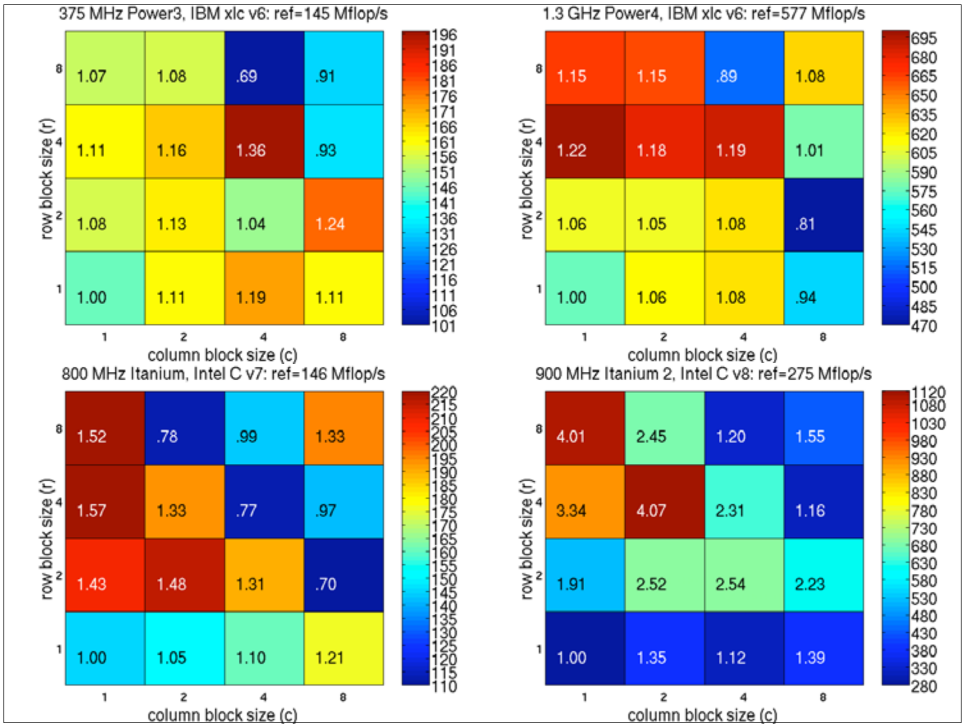
	1	2	4	8
8	1.68	1.80	1.89	2.04
4	1.51	1.66	1.72	1.91
2	1.25	1.45	1.59	1.74
1	1.00	1.26	1.45	1.57



	1	2	4	8
8	1.33	1.42	1.20	1.12
4	1.20	1.34	1.41	1.27
2	1.04	1.20	1.33	1.34
1	1.00	1.02	1.19	1.32

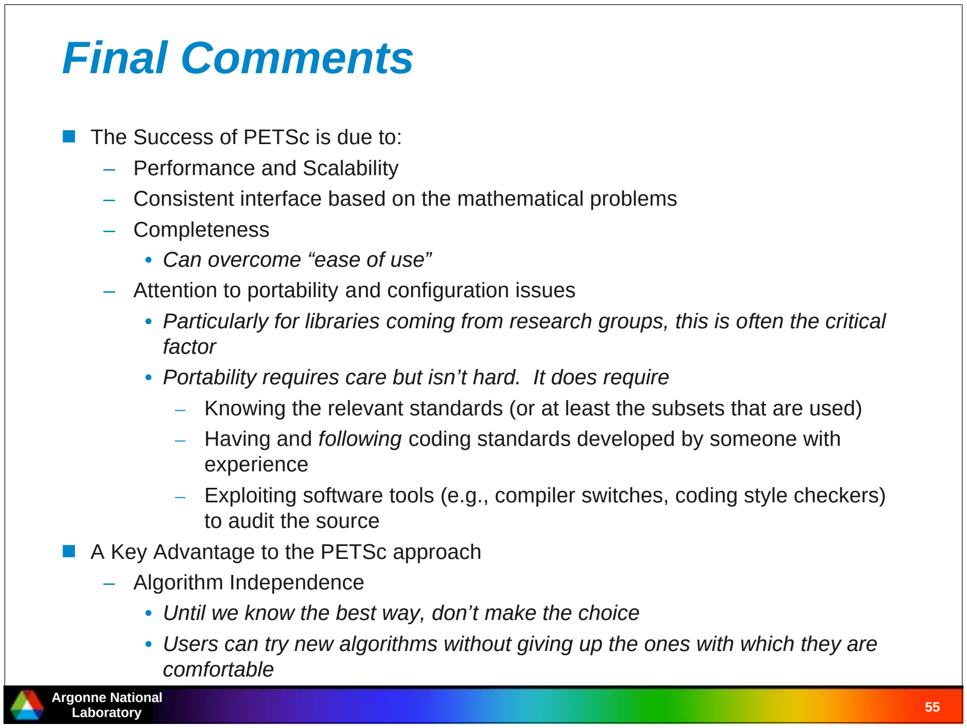


	1	2	4	8
8	1.28	1.13	1.22	1.45
4	1.25	1.27	1.22	1.14
2	.99	1.21	1.27	1.14
1	1.00	1.18	1.19	1.28



Lessons

1. Permit the best performance
2. Provide an escape for customization
3. Define/Update objects as a single operation
4. Provide ease-of-use features, even if they are not high-performance
5. It is good to provide multiple ways to perform the same operation (non-orthogonality of function)
6. Provide special purpose objects (not routines!) for important cases, and then optimize them
7. Allow repeated operations to amortize setup on a per-object basis
8. Use the principles of object oriented design to help use hierarchy to structure a library
9. Design and code for portability; base on the correct capability abstractions, not system name
10. Design to work with other libraries



Final Comments

- The Success of PETSc is due to:
  - Performance and Scalability
  - 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. It does require*
      - Knowing the relevant standards (or at least the subsets that are used)
      - Having and *following* coding standards developed by someone with experience
      - Exploiting software tools (e.g., compiler switches, coding style checkers) to audit the source
- 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*