

# PETSc Overview

Design and Implementation strategy for  
Extensibility, Portability and Performance  
(The Grand Tour)

Satish Balay, Bill Gropp, Dinesh Kaushik, Lois C. McInnes,  
Barry Smith

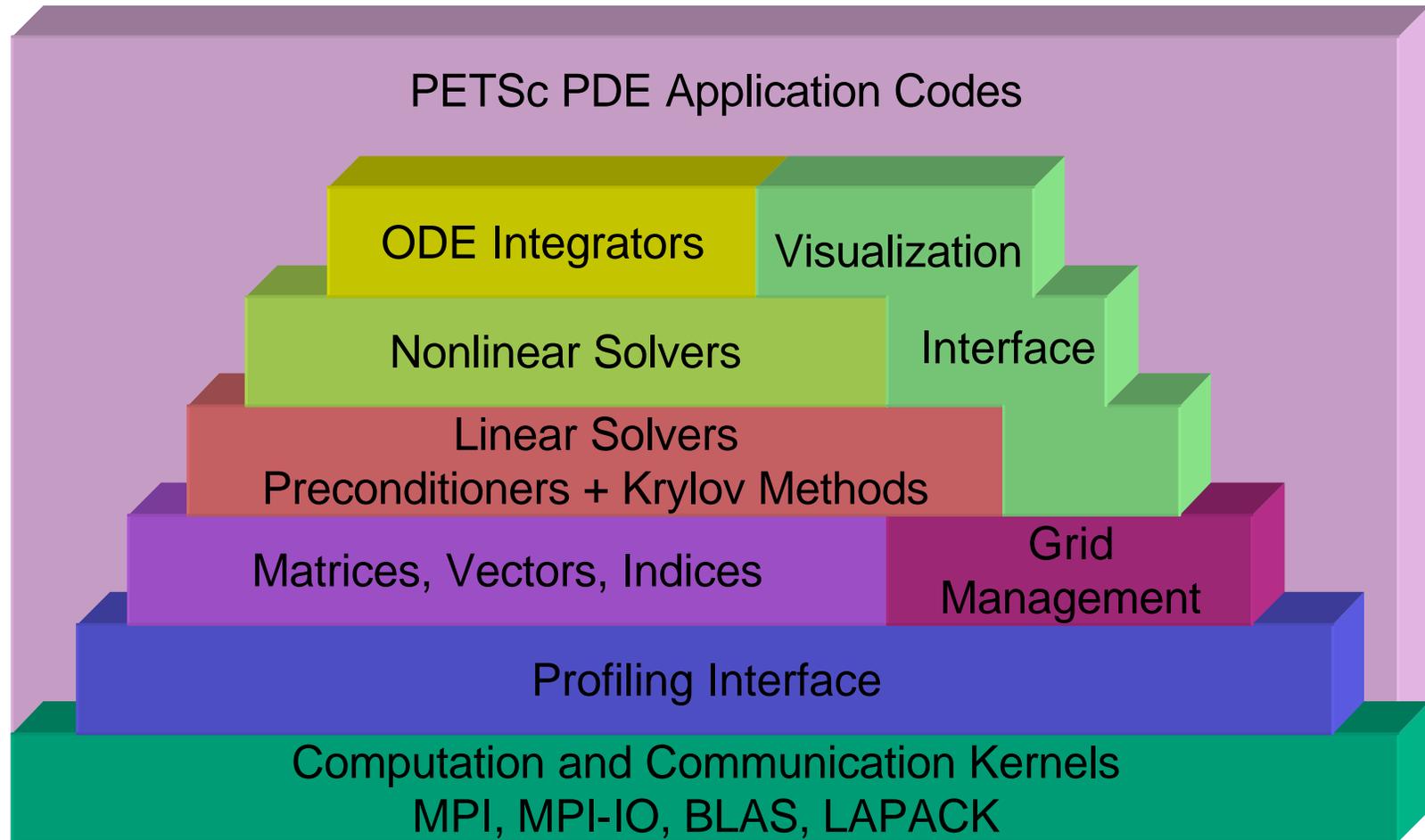
Mathematics and Computer Science Division  
Argonne National Laboratory

# PETSc Philosophy

- Writing hand-parallelized application codes from scratch is extremely difficult and time consuming.
- Scalable parallelizing compilers for real application codes are very far in the future.
- We can ease the development of parallel application codes by developing general-purpose, parallel numerical PDE libraries.
- Caveats
  - Developing parallel, non-trivial PDE solvers that deliver high performance is still difficult, and requires months of concentrated effort.
  - PETSc is a toolkit that can reduce the development time, but it is not a black-box PDE solver nor a silver bullet.



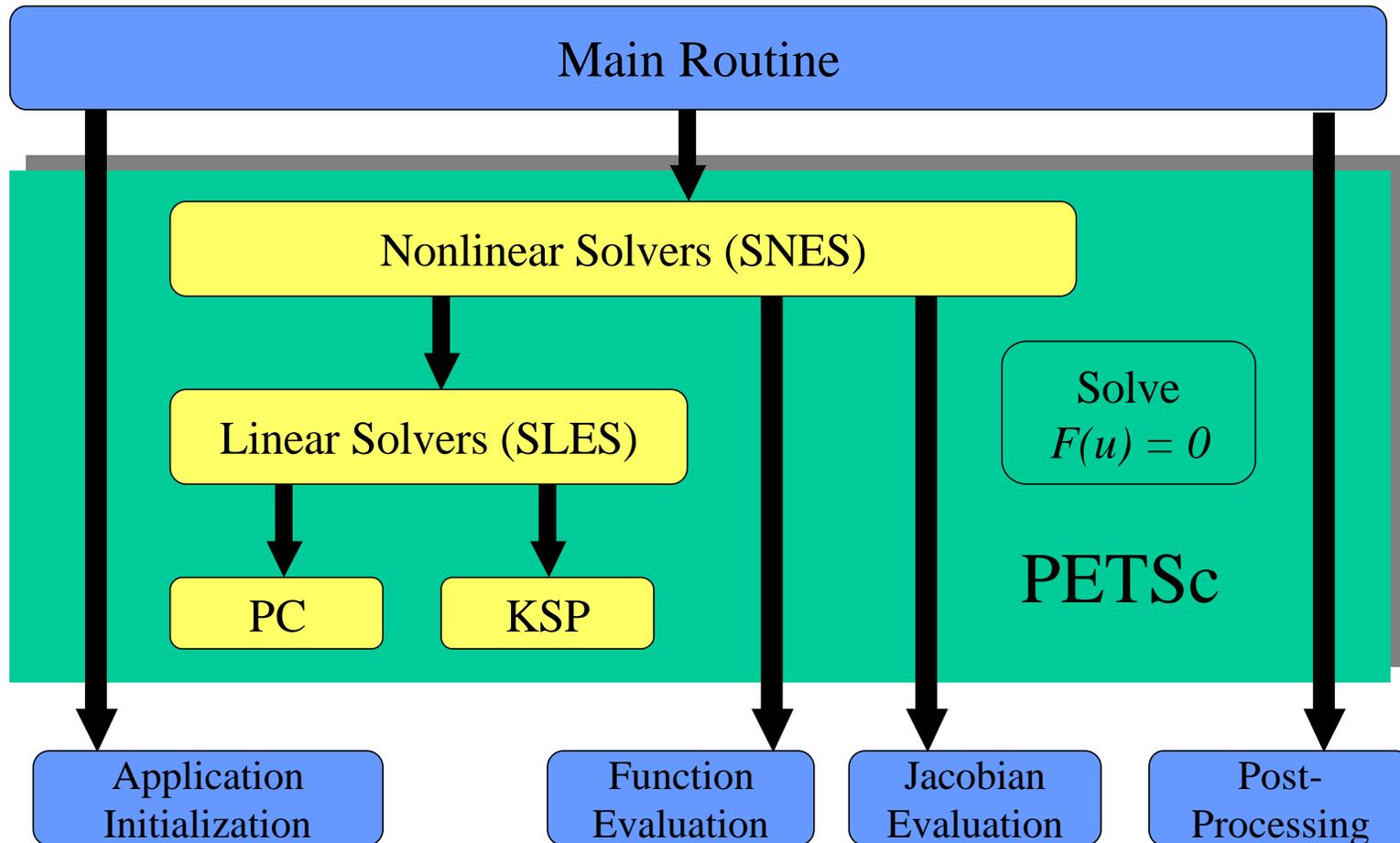
# PETSc Layering



# Sample PETSc Numerical Components

Nonlinear Solvers			Time Steppers				
Newton-based Methods		Other	Euler	Backward Euler	Pseudo Time Stepping	Other	
Line Search	Trust Region						
Krylov Subspace Methods							
GMRES	CG	CGS	Bi-CG-STAB	TFQMR	Richardson	Chebyshev	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	Other		
Vectors		Index Sets					
		Indices	Block Indices	Stride	Other		

# Flow of Control for a Non-Linear Solve



◆ User code

◆ PETSc code

# The PETSc Programming Model

- **Goals**

- Portable, runs everywhere
- Performance
- Scalable parallelism

- **Approach**

- Distributed memory, “shared-nothing”
  - Requires only a compiler (single node or processor)
  - Access to data on remote machines through MPI
- Can still exploit “compiler discovered” parallelism on each node (e.g., OpenMP)
- Hide within parallel objects the details of the communication
- User orchestrates communication at a higher abstract level than message passing

# A Freely Available and Supported Research Code

- Available via <http://www.mcs.anl.gov/petsc>
- Usable in C, C++, and Fortran77/90 (with minor limitations in Fortran 77/90 due to their syntax)
- Users manual
- Hyperlinked manual pages for all routines
- Many tutorial-style examples
- Support via email: [petsc-maint@mcs.anl.gov](mailto:petsc-maint@mcs.anl.gov)

# True Portability

- Tightly coupled systems
  - Cray T3D/T3E
  - SGI/Origin
  - IBM SP
  - Convex Exemplar
- Loosely coupled systems, e.g., networks of workstations
  - Sun OS, Solaris
  - IBM AIX
  - DEC Alpha
  - HP
  - Linux
  - FreeBSD
  - Windows 98/2000
  - Mac OS X
  - BeOS

# Collectivity

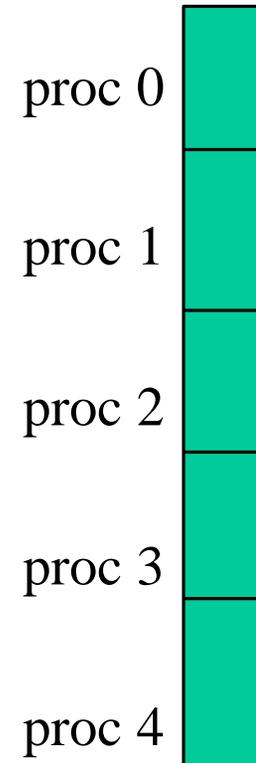
- MPI communicators (MPI\_Comm) specify collectivity (processors involved in a computation)
- All PETSc creation routines for solver and data objects are collective with respect to a communicator, e.g.,  
`VecCreate(MPI_Comm comm, int m, int M, Vec *x)`
- Some operations are collective, while others are not, e.g.,
  - collective: `VecNorm( )`
  - not collective: `VecGetLocalSize()`
- If a sequence of collective routines is used, they **must** be called in the same order on each processor

# Overview

- How to specify the mathematics of the problem
  - Data objects
    - vectors, matrices
- How to solve the problem
  - Solvers
    - Nonlinear solvers
- Design
  - Extensibility
- Parallel computing complications
  - Parallel data layout/ Ghost values
    - Scatter routines, index sets
- Other Issues
  - Portability
  - Performance
- Fluid application using PETSc

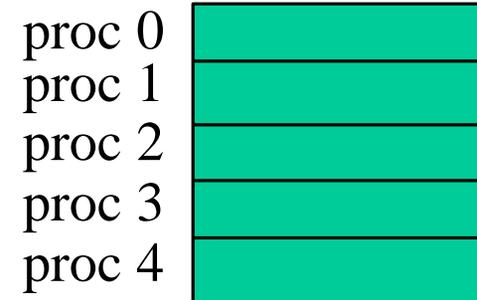
# Vectors

- Fundamental objects for storing field solutions, right-hand sides, etc.
- `VecCreateMPI(..., Vec *)`
  - `MPI_Comm` - processors that share the vector
  - number of elements local to this processor
  - total number of elements
- Each process locally owns a sub-vector of contiguously numbered global indices



# Sparse Matrices

- Fundamental objects for storing linear operators (e.g., Jacobians)
- `MatCreateMPIAIJ(...,Mat *)`
  - `MPI_Comm` - processors that share the matrix
  - number of local rows and columns
  - number of global rows and columns
  - optional storage pre-allocation information
- Each process locally owns a sub-matrix of contiguously numbered global rows.



# Parallel Matrix and Vector Assembly

- Processes may generate any entries in vectors and matrices
- Entries need not be generated by the process on which they ultimately will be stored
- PETSc automatically moves data during assembly if necessary
- Vector example:
  - `VecSetValues(Vec,...)`
    - number of entries to insert/add
    - indices of entries
    - values to add
    - mode:  
[INSERT\_VALUES,ADD\_VALUES]
  - `VecAssemblyBegin(Vec)`
  - `VecAssemblyEnd(Vec)`

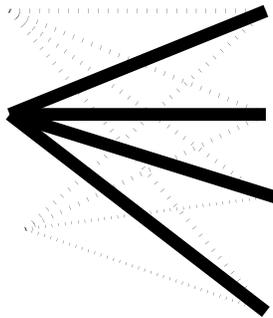


Agregation of Communication

# Solvers: Usage Concepts

## Solver Classes

- Linear (SLES)
- Nonlinear (SNES)
- Timestepping (TS)



## Usage Concepts

- Context variables
- Solver options
- Callback routines
- Customization/Extensibility

# Basic Nonlinear Solver Code (Fortran)

```
SNES  snes  
Mat   J  
Vec   x, F  
int   n, its
```

Context Variable



```
...  
call MatCreate(MPI_COMM_WORLD,n,n,J,ierr)  
call VecCreate(MPI_COMM_WORLD,n,x,ierr)  
call VecDuplicate(x,F,ierr)
```

```
call SNESCreate(MPI_COMM_WORLD,SNES_NONLINEAR_EQUATIONS  
call SNESSetFunction(snes,F,EvaluateFunction,PETSC_NULL,ierr)  
call SNESSetJacobian(snes,J,EvaluateJacobian,PETSC_NULL,ierr)  
call SNESSetFromOptions(snes,ierr)  
call SNESsolve(snes,x,its,ierr)
```

Callback Functions



Set Solver Options



# Setting Solver Options

- Standard function call interface:  
`SNESSetType(SNES snes,SNESType type)`
- Using options database from command line  
`-snes_type [ ls, ts, .. ]`

# Design and Extensibility

- Illustrate using Preconditioner object PC

PCCreate(..... &pc)

PCSetType(pc, PC\_TYPE)

PC\_TYPE (defaults)

- Jacobi, Bjacobi, SOR, ILU, ICC, LU, ASM

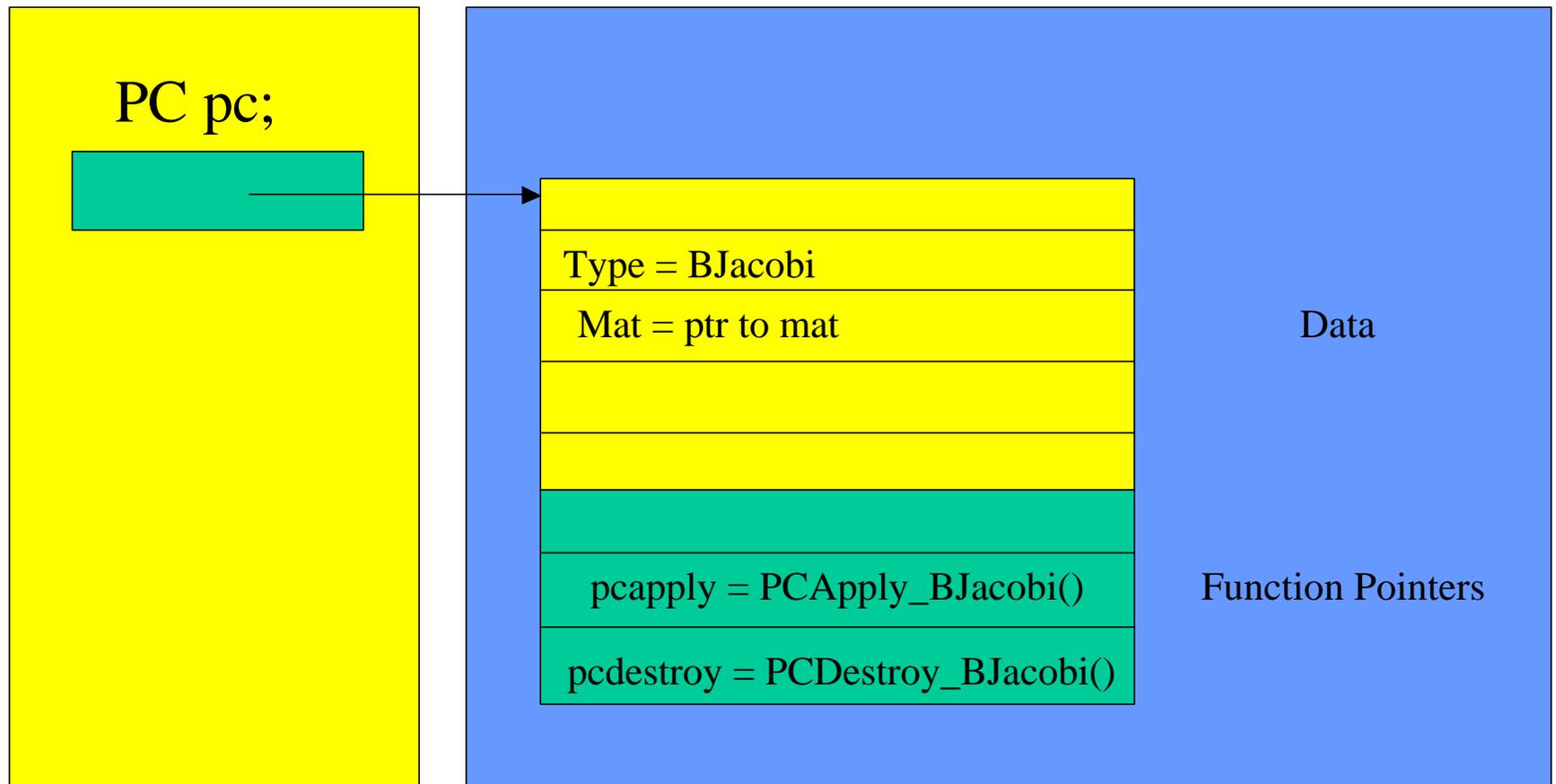
# PCApply() Implementation

```
PCApply(PC pc, Vec x, Vec y)
{
    (*pc->ops->apply)(pc,x,y);
}
```

pc->ops->apply

- PCApply\_Jacobi()
- PCApply\_BJacobi()
- PCApply\_ILU()
- PCApply\_ASM()

# PETSc Objects



Object Reference

Library Code

# Extensibility of PC

```
PCRegister("newtype",0,"PCCreate_NewType",PCCreate_NewType);
```

```
PCCreate_NewType(PC pc)
```

```
{
```

```
.....
```

```
pc->ops->setup          = PCSetUp_NewType;
```

```
pc->ops->apply          = PCApply_NewType;
```

```
pc->ops->destroy        = PCDestroy_NewType;
```

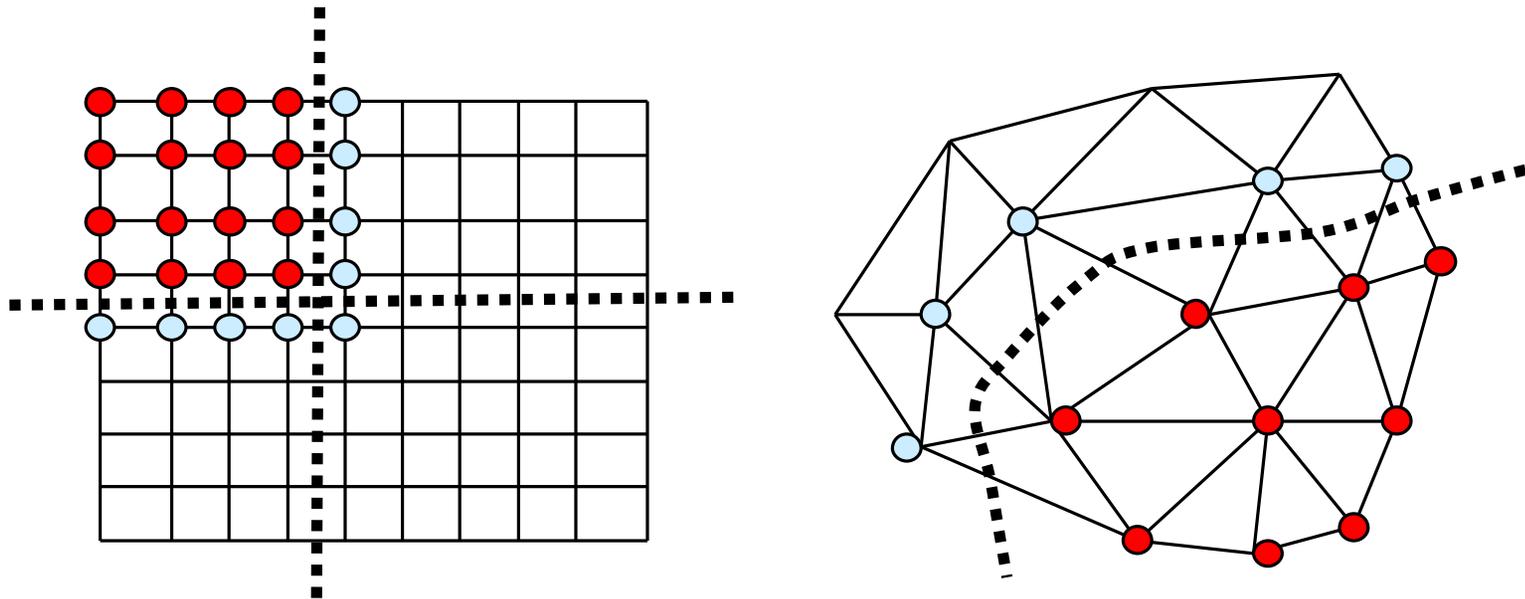
```
}
```

# Extensibility Issues

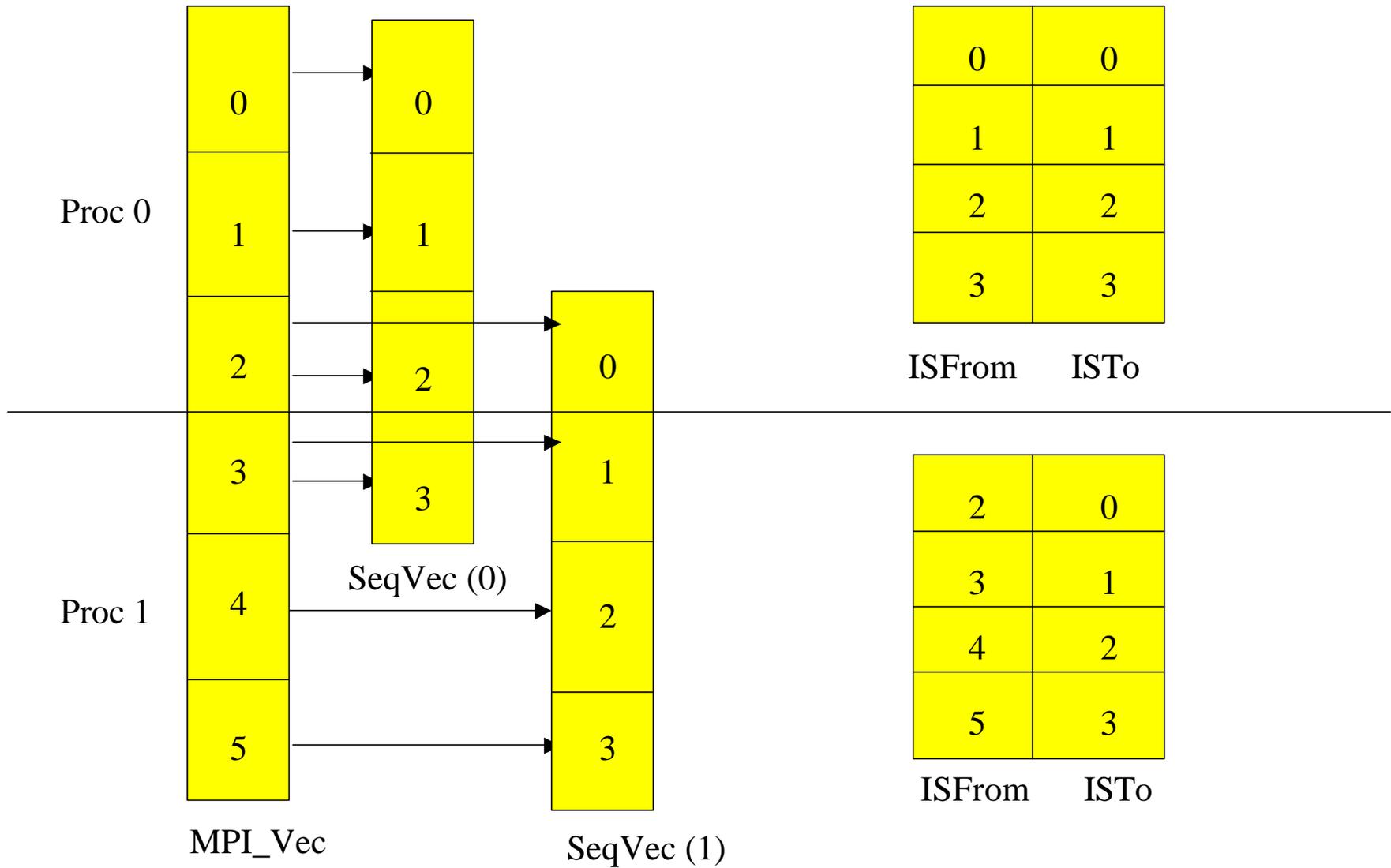
- Most PETSc objects are designed to allow one to “drop in” a new implementation with a new set of data structures (similar to implementing a new class in C++).
- Heavily commented example codes include
  - Krylov methods: [petsc/src/sles/ksp/impls/cg](#)
  - preconditioners: [petsc/src/sles/pc/impls/jacobi](#)

# Data Layout/Ghost Values

● Local node      ○ Ghost node

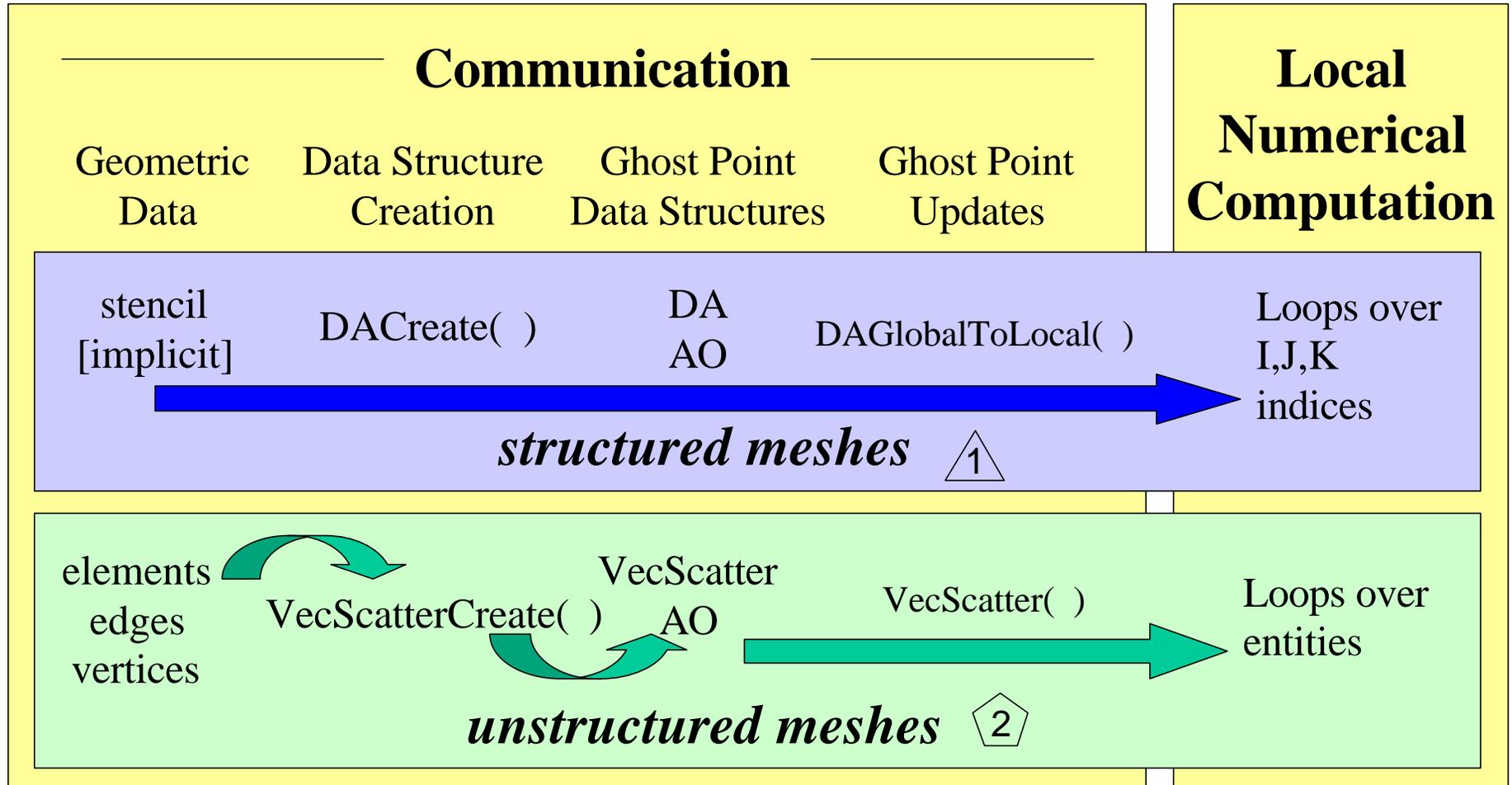


# VecScatters / Index Sets



```
VecScatterCreate(MPI_Vec,ISFrom,SeqVec,ISTo,&scatter)
VecScatterBegin(), VecScatterEnd()
```

# Communication and Physical Discretization



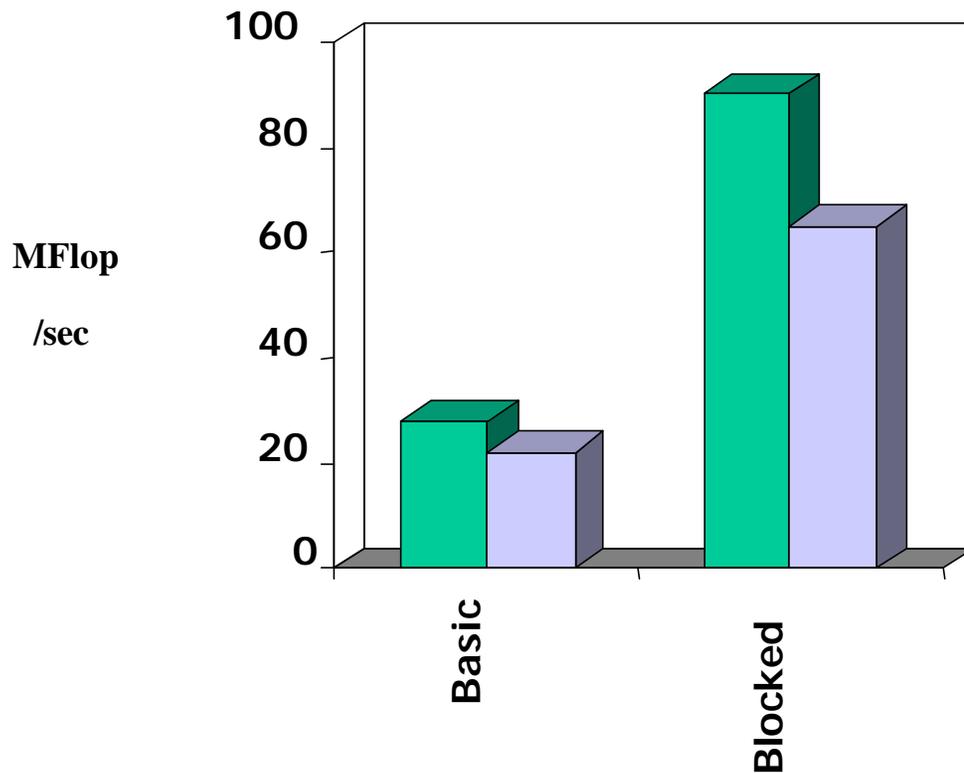
# Portability Issues

- Use makefiles and environment variable (PETS\_ARCH) to manage multiple architectures.
- Machine specific functionality.
- Run test-suite on various architectures every night.
- Provide support to avoid machine specific information in the user's code and makefiles.

# Performance Issues

- Flexible design to allow experimentation.
- Do certain optimizations after analyzing performance.
- Use `-log_summary` as a tool, but always use API, tuned for high performance.
- Modular design enables multiple implementations of the same component (AIJ,BAIJ etc..)
- Machine specific optimizations possible (using fortran kernels, for loops etc..)
- Create once and reuse – Scatters, factorizations etc..
- Pay attention to data layout/cache issues.

# Performance through Multiple Implementations (separate code for each block size)



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



# Other Issues

- Object header, creation, composition, dynamic methods etc.
- Extensive and consistent error handling.
- Profiling interface – application information, performance.
- Fortran interface/Fortran 90 support.
- Viewers – to debug/visualize PETSc objects.
- Interoperability with BlockSolve, PVM, Overture.
- Alice memory snooper(AMS), Toolkit for Advanced Optimization(TAO).

# Caveats Revisited

- Developing parallel, non-trivial PDE solvers that deliver high performance is still difficult, and requires months (or even years) of concentrated effort.
- PETSc is a toolkit that can ease these difficulties and reduce the development time, but it is not a black-box PDE solver nor a silver bullet.
- Users are invited to interact directly with us regarding correctness or performance issues by writing to [petsc-maint@mcs.anl.gov](mailto:petsc-maint@mcs.anl.gov).