Portable, Extensible Toolkit for Scientific Computation (PETSc)

Hong Zhang

Computer Science, Illinois Institute of Technology
Mathematics and Computer Science, Argonne National Laboratory

April, 2007 at the Center for Computation & Technology, LSU
Outline

• Overview of PETSc
• Linear solver interface: KSP
• Nonlinear solver interface: SNES
• Profiling, tracing and viewing of computational objects
• Ongoing research and developments
Original Goals of PETSc

• Provide software for the **scalable** (parallel) solution of **algebraic systems** arising from **partial differential equation simulations**.
  – Leverage inherited structure from the grid and the PDEs.
  – Eliminate the MPI from MPI programming!
  – Provide **wrappers** for other decent solver software.
Successfully transitioned from basic research to common community tool

- Applications of PETSc
  - 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 Helmholtz 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)
Who Uses PETSc?

• **Computational Scientists**
  – PyLith (TECTON), Underworld, Columbia group

• **Algorithm Developers**
  – Iterative methods and Preconditioning researchers

• **Package Developers**
  – SIPs, SLEPc, TAO, MagPar, StGermain, Dealll
The Role of PETSc

Developing parallel, nontrivial PDE solvers that deliver high performance is still difficult and requires months (or even years) of concentrated effort.

PETSc is a tool that can ease these difficulties and reduce the development time, but it is not a black-box PDE solver, nor a silver bullet.
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
Interfaced Packages

1. LU (Sequential)
   - SuperLU (Demmel and Li, LBNL)
   - ESSL (IBM)
   - Matlab
   - LUSOL (from MINOS - Michael Saunders, Stanford)
   - LAPACK
   - PLAPACK (van de Geijn, UT Austin)
   - UMFPACK (Timothy A. Davis)

2. Parallel LU
   - SuperLU_DIST (Demmel and Li, LBNL)
   - SPOOLES (Ashcroft, Boeing, funded by ARPA)
   - MUMPS (European)
   - PLAPACK (van de Geijn, UT Austin)

3. Parallel Cholesky
   - DSCPACK (Raghavan, Penn. State)
   - SPOOLES (Ashcroft, Boeing, funded by ARPA)
   - PLAPACK (van de Geijn, UT Austin)
Interfaced Packages

4. XYTlib – parallel direct solver (Fischer and Tufo, ANL)
5. SPAI – Sparse approximate inverse (parallel)
   • Parasails (Chow, part of Hypre, LLNL)
   • SPAI 3.0 (Grote/Barnard)
6. Algebraic multigrid
   • Parallel BoomerAMG (part of Hypre, LLNL)
   • ML (part of Trilinos, SNL)
7. Parallel ICC(0) – BlockSolve95 (Jones and Plassman, ANL)
8. Parallel ILU
   • BlockSolve95 (Jones and Plassman, ANL)
   • PILUT (part of Hypre, LLNL)
   • EUCLID (Hysom – also part of Hypre, ODU/LLNL)
9. Sequential ILUDT (SPARSEKIT2- Y. Saad, U of MN)
Interfaced Packages

10. Parititioning
   - Parmetis
   - Chaco
   - Jostle
   - Party
   - Scotch

11. ODE integrators
    - Sundials (LLNL)

12. Eigenvalue solvers
    - BLOPEX (developed by Andrew Knyazev)
Child Packages of PETSc

- **SIPs** - Shift-and-Invert Parallel Spectral Transformations
- **SLEPc** - scalable eigenvalue/eigenvector solver packages.
- **TAO** - scalable optimization algorithms
- **veltisto** ("optimum") - for problems with constraints which are time-independent pdes.

All have PETSc’s style of programming
What Can We Handle?

• PETSc has run problem with 500 million unknowns

• PETSc has run on over 6,000 processors efficiently

• PETSc applications have run at 2 Teraflops
  LANL PFLOTRAN code

• PETSc also runs on your laptop

• Only a handful of our users ever go over 64 processors
Structure of PETSc

PETSc Application Codes

- ODE Integrators
- Visualization
- Nonlinear Solvers
- Interface
- Linear Solvers
- Preconditioners + Krylov Methods
- Grid Management
- Matrices, Vectors, Indices

Profiling Interface

Computation and Communication Kernels
- MPI, MPI-IO, BLAS, LAPACK
The PETSc Programming Model

• Distributed memory, “shared-nothing”
  • Requires only a standard 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 than direct MPI calls
PETSc is only a Library

- PETSc is merely a set of library interfaces
  - You write main()
  - You control output
  - You control the basic flow of the program
  - We propagate the errors from underlying packages
  - We present (largely) the same interfaces in
    - C/C++
    - F77/F90

Flow of Control for PDE Solution

- **Main Routine**
- **PETSc**
  - **Timestepping Solvers (TS)**
  - **Nonlinear Solvers (SNES)**
  - **Linear Solvers (KSP)**
  - **PC**

- **Application Initialization**
- **Function Evaluation**
- **Jacobian Evaluation**
- **Post-Processing**

User code 🟢 | PETSc code 🟠
--- | ---

PETSc Structure
Getting Started

PetscInitialize();

ObjCreate(MPI_comm,&obj);

ObjSetType(obj,);

ObjSetFromOptions(obj,);

ObjSolve(obj,);

ObjGetXxx(obj,);

ObjDestroy(obj);

PetscFinalize()
# PETSc Numerical Components

<table>
<thead>
<tr>
<th>Nonlinear Solvers (SNES)</th>
<th>Time Steppers (TS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newton-based Methods</td>
<td>Euler</td>
</tr>
<tr>
<td>Line Search</td>
<td>Backward Euler</td>
</tr>
<tr>
<td>Trust Region</td>
<td>Pseudo Time Stepping</td>
</tr>
<tr>
<td>Other</td>
<td>Other</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Krylov Subspace Methods (KSP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMRES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preconditioners (PC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additive Schwartz</td>
</tr>
<tr>
<td>Jacobi</td>
</tr>
<tr>
<td>ICC</td>
</tr>
<tr>
<td>Others</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matrices (Mat)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compressed Sparse Row (AIJ)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distributed Arrays (DA)</th>
<th>Index Sets (IS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectors (Vec)</td>
<td>Indices</td>
</tr>
<tr>
<td></td>
<td>Block Indices</td>
</tr>
<tr>
<td></td>
<td>Stride</td>
</tr>
<tr>
<td></td>
<td>Other</td>
</tr>
</tbody>
</table>
Basic Linear Solver Code (C/C++)

```c
KSP  ksp;               /* linear solver context */
Mat   A;                  /* matrix */
Vec   x, b;              /* solution, RHS vectors */
int   n, its;            /* problem dimension, number of iterations */

MatCreate(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,n,n,&A);
MatSetFromOptions(A);
/* (code to assemble matrix not shown) */
VecCreate(PETSC_COMM_WORLD,&x);
VecSetSizes(x,PETSC_DECIDE, n);
VecSetFromOptions(x);
VecDuplicate(x,&b);
/* (code to assemble RHS vector not shown)*/

KSPCreate(PETSC_COMM_WORLD,&ksp);
KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN);
KSPSetFromOptions(ksp);
KSPSolve(ksp,b,x);
KSPDestroy(ksp);
```

Indicate whether the preconditioner has the same nonzero pattern as the matrix *each time a system is solved*. This default works with all preconditioners. Other values (e.g., SAME_NONZERO_PATTERN) can be used for particular preconditioners. Ignored when solving only one system.
Linear Solver Interface: **KSP**

- Main Routine
  - **PETSc**
    - Solve $Ax = b$
  - Linear Solvers (KSP)
    - **PC**
  - Application Initialization
  - Evaluation of $A$ and $b$
  - Post-Processing

**User code**

**PETSc code**

**beginner**

**solvers:** linear
Example

~petsc/src/ksp/ksp/examples/tutorials/ex10.c
# Linear Solvers in PETSc

## Krylov Methods (KSP)
- Conjugate Gradient
- GMRES
- CG-Squared
- Bi-CG-stab
- Transpose-free QMR
- etc.

## Preconditioners (PC)
- Block Jacobi
- Overlapping Additive Schwarz
- ICC, ILU via BlockSolve95
- ILU(k), LU (direct solve, sequential only)
- Arbitrary matrix
- etc.
Customization Options

• **Command Line Interface**
  – Applies same rule to all queries via a database
  – Enables the user to have complete control at runtime, with no extra coding

• **Procedural Interface**
  – Provides a great deal of control on a usage-by-usage basis inside a single code
  – Gives full flexibility inside an application
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 ...

1 beginner
2 intermediate

solvers: linear
# Linear Solvers: Monitoring Convergence

- **-ksp_monitor** - Prints preconditioned residual norm
- **-ksp_xmonitor** - Plots preconditioned residual norm
- **-ksp_truemonitor** - Prints true residual norm $\| b-Ax \|$
- **-ksp_xtruemonitor** - Plots true residual norm $\| b-Ax \|$
- User-defined monitors, using callbacks

---

**Beginner**

**Intermediate**

**Advanced**
**Recursion: Specifying Solvers for Schwarz Preconditioner Blocks**

- Specify KSP solvers and options with "-sub" prefix, e.g.,
  - Full or incomplete factorization
    - `sub_pc_type lu`
    - `sub_pc_type ilu -sub_pc_ilu_levels <levels>`
  - Can also use inner Krylov iterations, e.g.,
    - `sub_ksp_type gmres -sub_ksp_rtol <rtol>`
    - `sub_ksp_max_it <maxit>`
PETSc Programming Aids

• Correctness **Debugging**
  – Automatic generation of tracebacks
  – Detecting memory corruption and leaks
  – Optional user-defined error handlers

• Performance **Profiling**
  – Integrated profiling using `-log_summary`
  – Profiling by stages of an application
  – User-defined events
Debugging

Support for parallel debugging

- `start_in_debugger [gdb, dbx, noxterm]`
- `on_error_attach_debugger [gdb, dbx, noxterm]`
- `on_error_abort`
- `debugger_nodes 0,1`
- `display machinename:0.0`

When debugging, it is often useful to place a breakpoint in the function `PetscError()`.
Profiling

• Integrated monitoring of
  – time
  – floating-point performance
  – memory usage
  – communication

• Active if PETSc was configured with
  --with-debugging=1 (default)
  – Can also profile application code segments

• Print summary data with option: -log_summary

• Print redundant information from PETSc routines: -info [infofile]

• Print the trace of the functions called: -log_trace [logfile]
Nonlinear Solver Interface: **SNES**

**Goal:** For problems arising from PDEs, support the general solution of \( F(u) = 0 \)

User provides:
- Code to evaluate \( F(u) \)
- Code to evaluate Jacobian of \( F(u) \) (optional)
  - or use sparse finite difference approximation
  - or use automatic differentiation
    - AD support via collaboration with P. Hovland and B. Norris
    - Coming in next PETSc release via automated interface to ADIFOR and ADIC (see [http://www.mcs.anl.gov/autodiff](http://www.mcs.anl.gov/autodiff))
SNES: Review of Basic Usage

- SNESCreate() - Create SNES context
- SNESSetFunction() - Set function eval. routine
- SNESSetJacobian() - Set Jacobian eval. routine
- SNESSetFromOptions() - Set runtime solver options for [SNES, SLES, KSP, PC]
- SNESolve() - Run nonlinear solver
- SNESView() - View solver options actually used at runtime (alternative: -snes_view)
- SNESDestroy() - Destroy solver
Finite Difference Jacobian Computation

• Compute and explicitly store Jacobian via 1\textsuperscript{st}-order FD
  – Dense: -snes_fd, SNESDefaultComputeJacobian()
  – Sparse via colorings: MatFDColoringCreate(),
    SNESDefaultComputeJacobianColor()

• Matrix-free Newton-Krylov via 1\textsuperscript{st}-order FD, no
  preconditioning unless specifically set by user
  – -snes_mf

• Matrix-free Newton-Krylov via 1\textsuperscript{st}-order FD, user-
  defined preconditioning matrix
  – -snes_mf_operator
Uniform access to all linear and nonlinear solvers

- \texttt{-ksp\_type [cg, gmres, bogs, tfqmr, …]}
- \texttt{-pc\_type [lu, ilu, Jacobi, sor, asm, …]}
- \texttt{-snes\_type [ls, …]}

1

- \texttt{-snes\_line\_search <line search method>}
- \texttt{-sles\_ls <parameters>}
- \texttt{-snes\_convergence <tolerance>}
- etc...

solvers: nonlinear
Parallel Data Layout and Ghost Values

Managing *field data layout* and required *ghost values* is the key to high performance of most PDE-based parallel programs.

**Mesh Types**
- Structured
  - DA objects
- Unstructured
  - VecScatter objects

**Usage Concepts**
- Geometric data
- Data structure creation
- Ghost point updates
- Local numerical computation

**Important Concepts**
- data layout
**Ghost Values**

- **Local node**
- **Ghost node**

**Ghost values**: To evaluate a local function $f(x)$, each process requires its local portion of the vector $x$ as well as its **ghost values** – or bordering portions of $x$ that are owned by neighboring processes.
Communication and Physical Discretization

**Communication**

- Geometric Data
- Data Structure Creation
- Ghost Point Data Structures
- Ghost Point Updates

**Local Numerical Computation**

Loops over I,J,K indices

---

**Structured meshes**

- stencil [implicit]
- DACreate( )
- DA AO
- DAGlobalToLocal( )

---

**Unstructured meshes**

- elements edges vertices
- VecScatterCreate( )
- VecScatter AO
- VecScatter()

Loops over entities

---

data layout
DA: Parallel Data Layout and Ghost Values for Structured Meshes

- Local and global indices
- Local and global vectors
- DA creation
- Ghost point updates
- Viewing
Global and Local Representations

**Global**: each process stores a unique local set of vertices (and each vertex is owned by exactly one process)

**Local**: each process stores a unique local set of vertices as well as ghost nodes from neighboring processes

Data layout: distributed arrays
Logically Regular Meshes

- **DA - Distributed Array**: object containing information about vector layout across the processes and communication of ghost values
- Form a DA
  - `DACreate1d(….,DA *)`
  - `DACreate2d(….,DA *)`
  - `DACreate3d(….,DA *)`
- Create the corresponding PETSc vectors
  - `DACreateGlobalVector( DA, Vec *)` or
  - `DACreateLocalVector( DA, Vec *)`
- Update ghostpoints (scatter global vector into local parts, including ghost points)
  - `DAGlobalToLocalBegin(DA, …)`
  - `DAGlobalToLocalEnd(DA,…)`
Distributed Arrays

Data layout and ghost values

Box-type stencil

Star-type stencil

data layout: distributed arrays
Sample Nonlinear Application: Driven Cavity Problem

- Velocity-vorticity formulation
- Flow driven by lid and/or bouyancy
- Logically regular grid, parallelized with DAs
- Finite difference discretization
- source code:

  petsc/src/snes/examples/tutorials/ex19.c

Solution Components

- velocity: $u$
- velocity: $v$
- vorticity: $\zeta$
- temperature: $T$

Application code author: D. E. Keyes
Ongoing Research and Developments

- Framework for **multi-model algebraic system**
  - petsc-dev/src/snes/examples/tutorials/ex31.c, ex32.c

- Framework for **unstructured meshes** and functions defined over them

- Bypassing the sparse matrix **memory bandwidth bottleneck**
  - Large number of processors (nproc =1k, 10k,…)
  - Peta-scale performance

- More TS methods
- …
Bypassing the sparse matrix memory bandwidth bottleneck:

- **Newton-multigrid** provides
  - good nonlinear solver
  - easy utilization of software libraries
  - low computational efficiency
- **Multigrid-Newton** provides
  - good nonlinear solver
  - lower memory usage
  - potential for high computational efficiency
  - requires “code generation/in-lining”
How will we solve numerical applications in 20 years?

• Not with the algorithms we use today?
• Not with the software (development) we use today?
References

• http://www.mcs.anl.gov/petsc