Interoperability of sparse linear system solvers represented as components

Masha Sosonkina[1]

[1]Ames Laboratory and Iowa State University, USA

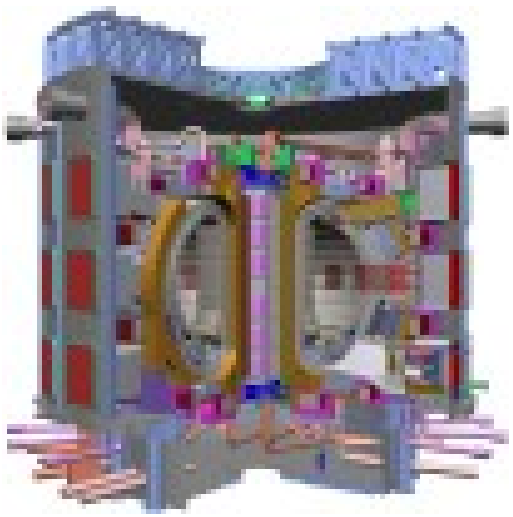
Parallel Processing and Applied Mathematics, 2007

Collaborators

Randall Bramley	Indiana University
Lois Curfman McInnes	Argonne National Laboratory
Li Li	Argonne National Laboratory
Fang Liu	Indiana University
Boyana Norris	Argonne National Laboratory

Outline

Growing complexity of simulations



- Address multi-scale phenomena arising in many scientific disciplines
  - Quantum chemistry: Quantum Mechanics/Molecular Mechanics interface.
  - Fusion: Magneto-hydrodynamics system resolution on very small scale combined with radiation transport.
- Transfer infornation from one scale to another.

Growing complexity of simulations

- Integrate existing and emerging codes.
  - Quantum chemistry: multiple codes on the same "theory-level" but one may be best for different input cases (e.g., integrals).
  - Fusion: couple adaptive mesh refinement (AMR) and plasma kinetics codes.
  - Needed an easy "on-the-fly" selection of these codes.
- Utilize efficiently massively-parallel platforms.
  - Couple with performance analysis tools.
  - Map various simulation tasks to different processor groups.

Sparse linear algebra in multi-scale simulations

- Ubiquitous: Sparse matrices arise from near-neighbor and long-range interactions.
  - Matrices have different characteristics and may be structured and unstructured, affect the numerical methods applied
- Sparse linear system solution (or eigen-value computation) is a significant cost factor.
  - Numerous implementations exist and depend on the problem and hardware architecture at hand.
  - Sequences of matrices are often to be solved during simulation cycles.

Switching of solution methods may be advantageous from one cycle to another in a simulation.

Universe of existing sparse linear system solvers

Dichotomy into sparse direct and iterative solvers

- Direct: solve liner system by performing Gaussian elimination directly while applying sparse matrix techniques.
- Iterative: find solution with a desired accuracy by improving solution one step at a time, say, by a projection method.
- Large number of implementations:
  - Example, a list of freely available solver software is at http://www.netlib.org/utk/people/JackDongarra/la-sw.html.
    - Contains 12 direct and 22 iterative solver entries.

Solver integration via Trilinos and PETSc

Trilinos from Sandia National Laboratories.

- Targets multi-physics complex simulations.
- Object-oriented framework for inclusion of packages:
  - Each package is self-contained software.
  - Minimal set of interfaces/add-ons to the package to add it to the Trilinos framework.
- http://trilinos.sandia.gov

PETSc from Argonne National Laboratory.

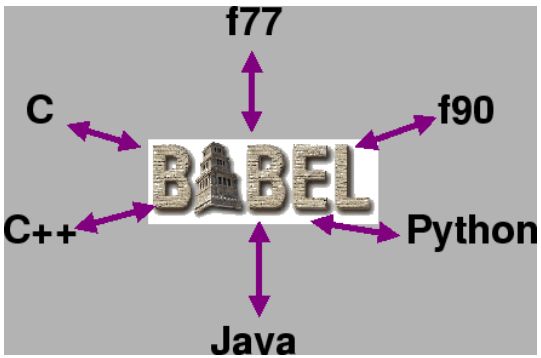
- Provides software for the scalable solution of systems of equations arising from PDEs (original goal).
- Has object-oriented programming style leveraging structured and unstructured matrices.
- Interfaces with many existing solvers as well as optimization techniques.
- http://www.mcs.anl.gov/petsc

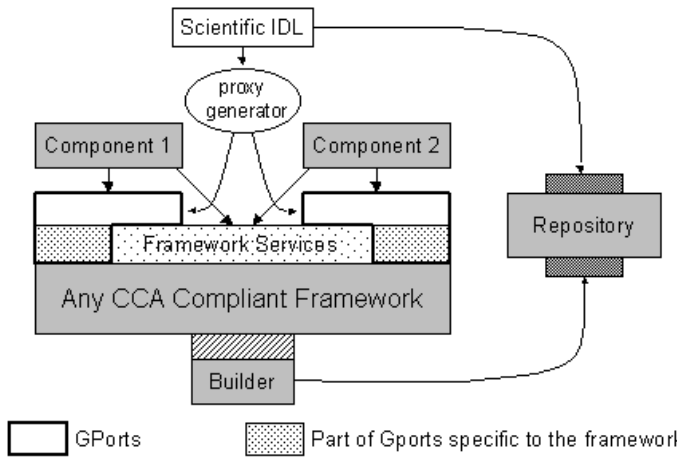
Overview of componentization for HPC

- Special requirements of HPC:
  - May need all available computational resources (problem scalability).
  - Perform mostly floating-point computations.
  - Exploit multiple levels of parallelism in applications.
  - Rely heavily on large legacy code base.
  - Adapt to novel HPC architectures.
- HPC components are to share, reuse, and redeploy codes:
  - Emphasis on performance.
  - Non-invasive encapsulation of functionality.
  - Assembly of applications from available components.
- Several component models exist.
  - In business world: DCOM, Corba Component Model.
  - In HPC community: Fractal, GCCM, CCA.

Common Component Architecture: Main features

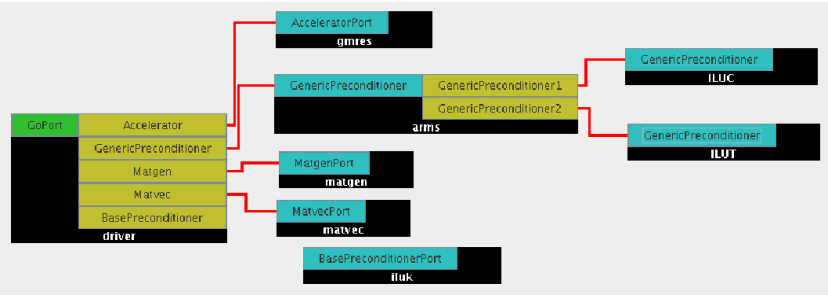
- U.S. Department of Energy project involving many national laboratories and academia. www.cca-forum.org.
- Based on Scientific Interface Definition Language (SIDL).
  - Babel [from LLNL] provides multi-lingual support for SIDL.
- Includes special interfaces (ports) to interact according to CCA standard.
- Relies on a *CCA-compliant framework* and *component builder* to ensure component interaction and application assembly.
- Provides API for component *repositories*.

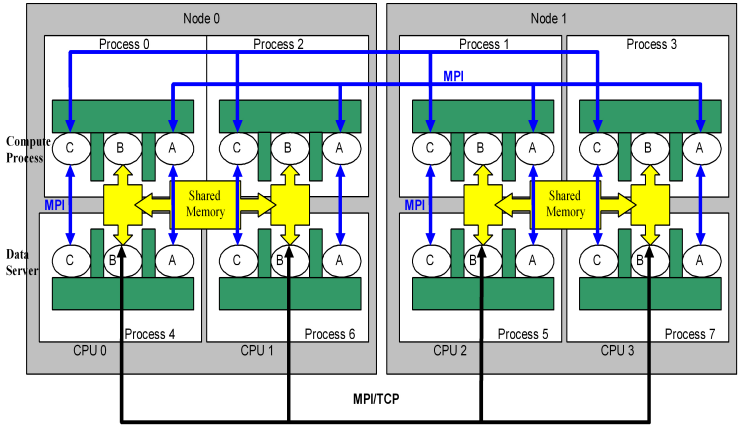




CCA application assembly

Component's *Uses* ports are connected to corresponding *Provides* ports of other components.





Parallelism in CCA

- CCA model is currently parallelism-transparent
- Framework instances are multiplexed in each processor.

Design choices

- Low-level: User expresses all sparse matrix operations with components.
  - Benefitial for very large matrices when conversion is prohibitively expensive.
- Medium-level: Major solver parts are separate components.
  - Useful for expert tuning of solution;
  - User needs to know matrix representation.
- High-level: Entire linear system solution is encapsulated into a component.
  - Treats solver as “black box”;
  - Easy switching of solver packages;
  - Many solvers, such as PETSc and Trilinos, may be linked via high-level interfaces.

High-level design: CCA-LISI

CCA LInear system Solver Interfaces [Indiana University]

Design goals:

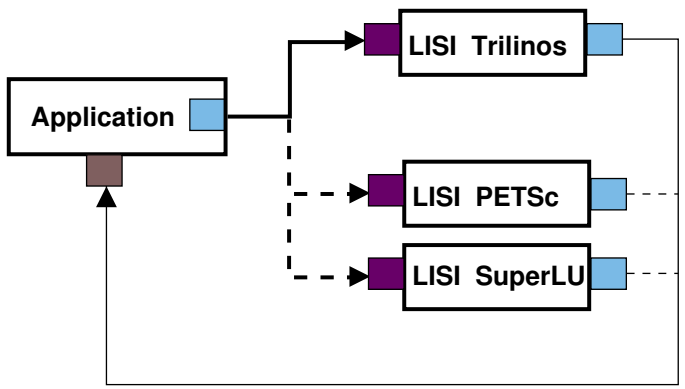
Hide the underlying implementation while preserving functionality and user flexibility.

- Encapsulate different sparse matrix formats into interface implementations.
- Handle parallelism assuming block-row matrix distribution and programming model of each underlying solver.
- Provide for user-defined matrix-vector operations similar in spirit to “reverse communication”.
- Ease and simplicity of use: Allow user to seamlessly switch linear system solver packages.

High-level design: CCA-LISI

LISI architecture

- `SparseSolver` interface has *provides* port for application.
- `MatrixFree` interface is to be implemented by application and is *used* by the solver.
- Matrix is passed to LISI solver as multiple-arrays to reduce complexity of matrix object construction on the application side.
- Solver parameters are set as (***key***, ***value***) pair, while explicit methods are proposed for matrix data input.



High-level design: TOPS

Interfaces for solvers developed in the Center "Towards Optimal Petascale Simulations" (TOPS)

Design Goals:

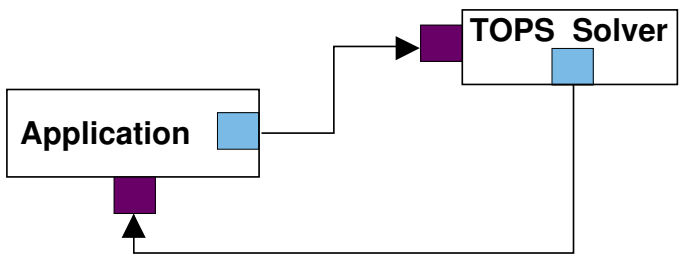
Provide scalable solution of linear and nonlinear systems arising from structured or unstructured meshes. Allow maximum flexibility to the application as to the data structures and solution choice.

- Enable experimentation with solvers without changing matrix data structures.
- Provide for structured and unstructured solvers.
- Construct the system on the application side and use it by the connected TOPS solver.

High-level design: TOPS

TOPS architecture

- `TOPS.System` interface is to be implemented in the application code and *used* by the TOPS.Solver component.
- `TOPS.Structured.Solver` and `TOPS.Unstructured.Solver` *provide* solution methods for the matrix objects implemented in `TOPS.System`.
- Separate methods exist for such functionality as residual and initial guess computations, construction of right-hand side.



Low- and Medium-level design: SPARSKIT-CCA

What is SPARSKIT?

- Well-known library of *serial* sparse matrix kernels by Yousef Saad (University of Minnesota).
- Written in Fortran77.
- Provides a range of functions for sparse matrix computations with a focus on iterative solution techniques.
  - Provides iterative procedure to obtain approximate solution (*accelerators*) and matrix transformations (*preconditioning*) used in pre-processing.
- BLASSM is a suite of BLAS-like operations on sparse matrices.

SPARSKIT is a suite of medium-level components

Component interfaces designed to have standard argument lists (e.g., for variations of the same preconditioner type).

- A component implements particular preconditioner creation process.
- A component implements particular accelerator.

Example of a component SIDL interface for preconditioner

```
package sparskit version 1.0 {
 interface GenericPreconditioner extends gov.cca.Port {
  void setDoubleArgument(in string name, in double value);
  void getDoubleArgument(in string name, inout double value);
  void setIntArgument(in string name, in int value);
  void getIntArgument(in string name, inout int value);
  void getName(inout string name);
  void apply();
  void create(); } }
```

Low-level BLASSM components

- Low-level interface refers to operating on objects, such as matrices, directly.
  - Implemented in many modern numerical software packages in the conventional library format. (e.g., Diffpack, MTL)
- BLASSM components extend SPARSKIT functionality to many different matrix formats.
  - Accomplished as overloading.
  - Example: call a *generic* function `amub` to multiply matrix ***A*** by matrix ***B***; A specific version of `amub` is chosen at the run-time based on the matrix format.
- Format-agnostic code has its toll on component overhead.

Overhead for different-level components

High-level interfaces in CCA-LISI

3-D Poisson equation with Dirichlet boundary conditions

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} = f$$

- Solver parameters: BiCGSTAB, Jacobi preconditioning, stopping tolerance $\mathbf{10^{-6}}$.

High-level PETSc component

nnz	its	PETSc	PETSc-CCA	diff, %
76, 760	37	0.154	0.154	0
122,880	44	0.287	0.289	0.43
179,800	41	0.399	0.400	0.32
247,520	42	0.565	0.568	0.55
326,040	45	0.865	0.870	0.61
415,360	43	1.020	1.038	1.72
515,480	44	1.284	1.296	0.91

Overhead for different-level components

Medium- and low-level interfaces in SPARSKIT

- Solver parameters: Flexible GMRES(20), ILUT preconditioner.
- $\boldsymbol{A} \times \boldsymbol{B}$ computation (`amub`) in BLASSM.

SPARSKIT medium-level components BLASSM

nnz	its	SKIT	SKIT-CCA	diff	diff
76.760	36	0.0792	0.08	1	28.57
122,880	36	0.14	0.14	0	31.35
179,800	36	0.208	0.215	3.36	20.88
247,520	36	0.334	0.345	3.29	23.98
326,040	36	0.443	0.448	1.13	17.28
415,360	36	0.570	0.588	3.12	19.9
515,480	36	0.7185	0.730	1.6	17.59

Adaptivity features of preconditioners

Algebraic Recursive Multilevel Solver (ARMS) is an adaptive preconditioner [Saad 2002].

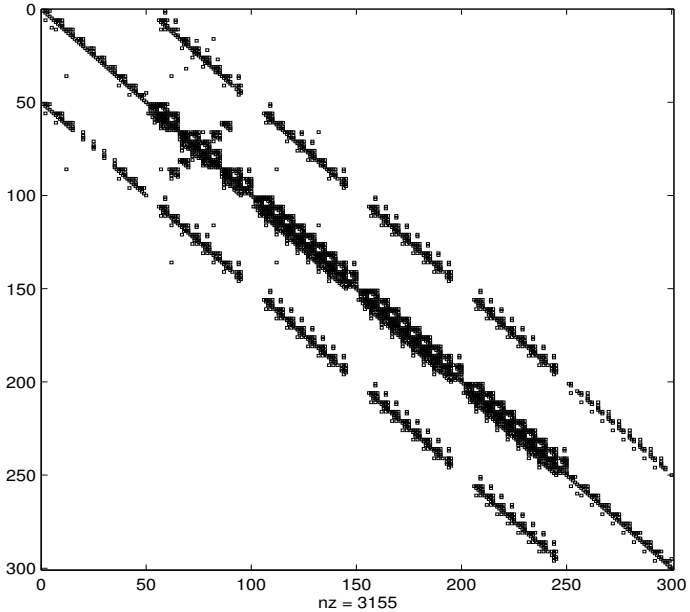
- Two-step construction of the ARMS preconditioner
  1. Reorder the matrix $\boldsymbol{A}$ into a $\mathbf{2} \times \mathbf{2}$ block form

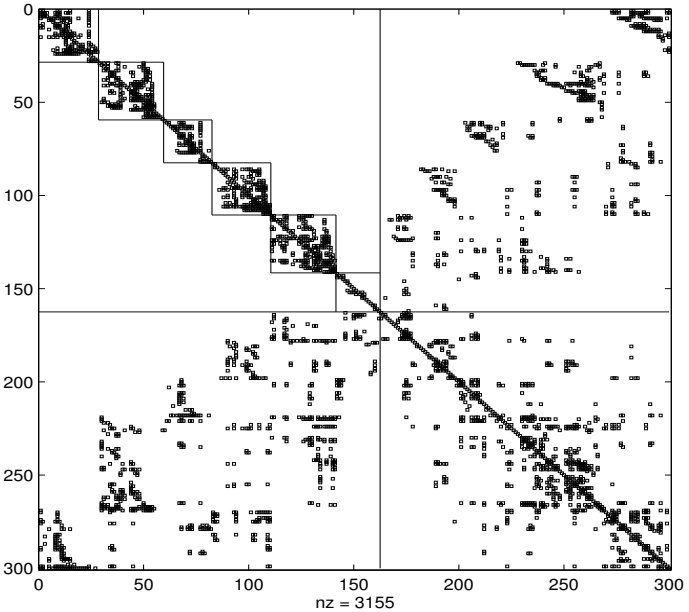
$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{B} & \boldsymbol{F} \\ \boldsymbol{E} & \boldsymbol{C} \end{pmatrix}.$$

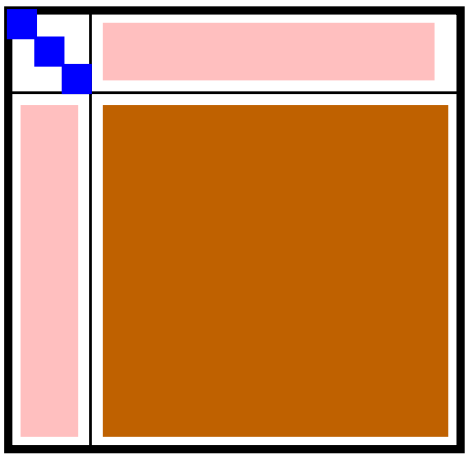
  2. Use an incomplete LU technique to obtain an approximate factorization of $\boldsymbol{B}$ and approximations to the matrices $\boldsymbol{L}^{-1}\boldsymbol{F}$, $\boldsymbol{E}\boldsymbol{U}^{-1}$, and $\boldsymbol{A}_1$:

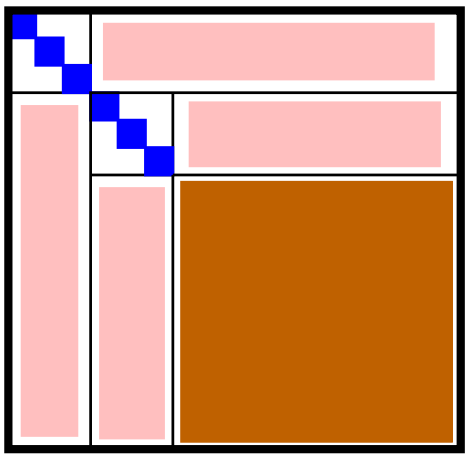
$$\begin{pmatrix} \boldsymbol{B} & \boldsymbol{F} \\ \boldsymbol{E} & \boldsymbol{C} \end{pmatrix} \approx \begin{pmatrix} \boldsymbol{L} & \mathbf{0} \\ \boldsymbol{E}\boldsymbol{U}^{-1} & \boldsymbol{I} \end{pmatrix} \times \begin{pmatrix} \boldsymbol{U} & \boldsymbol{L}^{-1}\boldsymbol{F} \\ \mathbf{0} & \boldsymbol{A}_1 \end{pmatrix}.$$

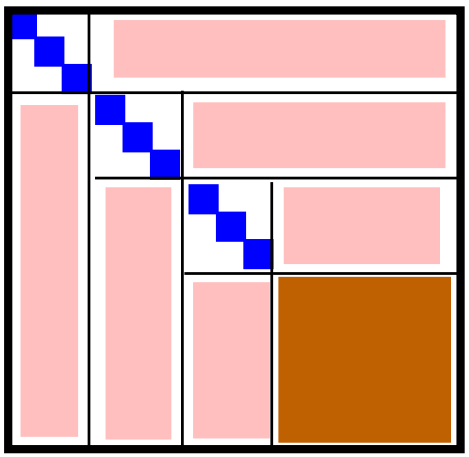
The process is repeated recursively on the matrix $\boldsymbol{A}_1$.


0
50
100
150
200
250
300
nz = 3155


0
50
100
150
200
250
300
nz = 3155



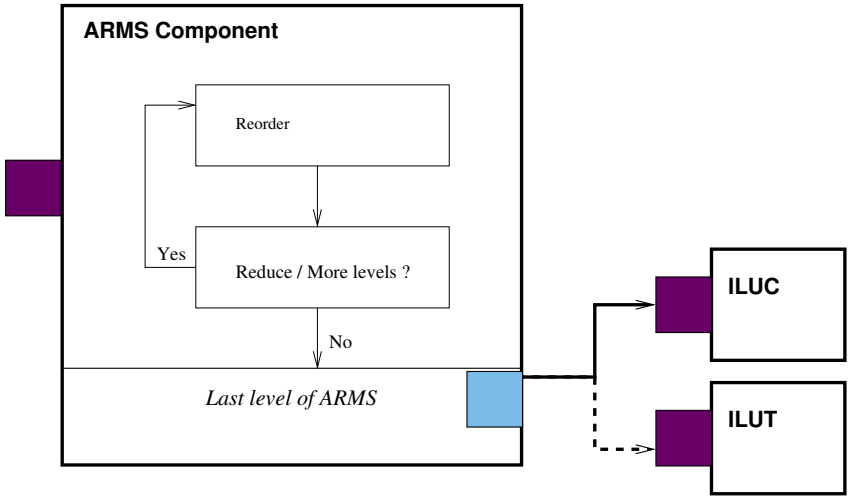




ARMS preconditioner

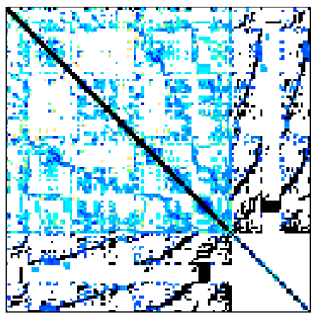
- At the last level, a simple (single-level) preconditioner is used on the entire reduced system.
- An ARMS component allows users to switch between last-reduced system preconditioners and even add their own preconditioners.

Example: Either ILUC or ILUT preconditioners at the last level.



Solution using ARMS components

- Consider two difficult to solve matrices, *scircuit* and *igbt3*.



matrix	*nnz*	Precon	its	time
scircuit	958,936	*ARMS+ILUC*	7	124.9
		ARMS+ILUT	7	125.1
		ILUC	55	11.2
		ILUT	*	*
igbt3	234,006	*ARMS+ILUC*	8	20.23
		ARMS+ILUT	6	19.5
		ILUC	*	*
		ILUT	19	0.55

Versatility of SPARSKIT components

1. Instantiate more than one “last-level” preconditioner.
2. Decide *at the run-time* on particular ARMS preconditioners.
   - Capture the convergence progress information.
   - Time individual components.

Provide computational Quality of Service to applications

- Need for infrastructure support.
- Components from Tuning and Analysis Utilities (TAU) [University of Oregon].
  - TAU is a portable profiling and tracing toolkit for performance analysis of parallel programs.
  - TAU Performance Component provides the `Measurement` interface.
  - Data collection is by *MasterMind* component.
  - Provides *Proxy Generator* tool.

Performance analysis using TAU

nnz	Func	Calls	SKIT-CCA	SKIT	Norm, %
38,880	lusol	46	20	16	25.0
	create	1	18	16	12.5
	amux	47	12	11	9.1
	apply	93	12	23	0.0
45,847	create	1	25	20	25.0
	lusol	46	20	20	0.0
	amux	47	13	13	0.0
	apply	93	13	12	8.3
76,760	lusol	36	34.8	34.0	2.4
	create	1	34	34	0.0
	apply	73	28	27.2	2.9
	amux	38	22	22	0.0
179,800	create	1	96.2	95	1.3
	apply	73	96	89.5	7.3
	lusol	36	91	80.5	13.1
	amux	38	65	65	0.0

Summary

- Linear system solver components enable HPC applications to immediately benefit from a vast knowledge and code bases in the field of numerical sparse linear algebra.
- Components is a viable programming model for developing sparse linear system solvers.
  - Incurs negligible overhead for coarse-grained componentization.
  - Provides an easy access to legacy codes.
  - Integrates existing and new solver packages.