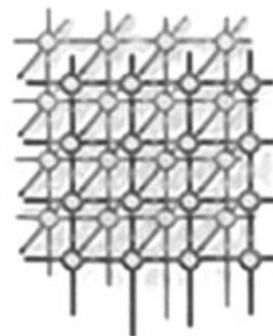


# Usability levels for sparse linear algebra components<sup>†‡</sup>

M. Sosonkina<sup>1,\*,\*†</sup>, F. Liu<sup>2</sup> and R. Bramley<sup>2</sup>

<sup>1</sup>*Ames Laboratory, Iowa State University, Ames, IA 50011, U.S.A.*

<sup>2</sup>*Indiana University, Bloomington, IN 47405, U.S.A.*



## SUMMARY

Sparse matrix computations are ubiquitous in high-performance computing applications and often are their most computationally intensive part. In particular, efficient solution of large-scale linear systems may drastically improve the overall application performance. Thus, the choice and implementation of the linear system solver are of paramount importance. It is difficult, however, to navigate through a multitude of available solver packages and to tune their performance to the problem at hand, mainly because of the plethora of interfaces, each requiring application adaptations to match the specifics of solver packages. For example, different ways of setting parameters and a variety of sparse matrix formats hinder smooth interactions of sparse matrix computations with user applications. In this paper, interfaces designed for components that encapsulate sparse matrix computations are discussed in the light of their matching with application usability requirements. Consequently, we distinguish three levels of interfaces, high, medium, and low, corresponding to the degree of user involvement in the linear system solution process and in sparse matrix manipulations. We demonstrate when each interface design choice is applicable and how it may be used to further users' scientific goals. Component computational overheads caused by various design choices are also examined, ranging from low level, for matrix manipulation components, to high level, in which a single component contains the entire linear system solver. Published in 2007 by John Wiley & Sons, Ltd.

*Received 5 April 2007; Revised 17 August 2007; Accepted 17 September 2007*

KEY WORDS: component architecture and interfaces; sparse matrix computations; usability

\*Correspondence to: M. Sosonkina, Ames Laboratory, Iowa State University, Ames, IA 50011, U.S.A.

<sup>†</sup>E-mail: masha@scl.ameslab.gov

<sup>‡</sup>This article is a U.S. Government work and is in the public domain in the U.S.A.

Contract/grant sponsor: Iowa State University of Science and Technology; contract/grant number: DE-AC02-07CH11358 with the U.S. Department of Energy

Contract/grant sponsor: Minnesota Supercomputing Institute and University of Minnesota Duluth

Contract/grant sponsor: Indiana University; contract/grant number: National Science Foundation Grants EIA-0202048, MRI CDA-0116050

Contract/grant sponsor: The DOE Office of Science Center for Technology for Advanced Scientific Component Software



## 1. INTRODUCTION

Sparse linear system solution is an integral part of many engineering and scientific applications. It also tends to be the most computationally intensive part, so gaining efficiency in solving linear systems makes a considerable improvement in the entire application performance. Many efficient algorithms have been proposed to solve sparse linear systems (see e.g. [1] for a list of freely available implementations). As a rule, the solver choice depends heavily on linear system properties, such as symmetry, spectral characteristics, number of non-zero entries, or sparsity pattern. Even the matrix storage format affects the selection. The size of the solver search space makes it infeasible to try all the available solution options, each of which may have its own calling sequence and format for input and output data. Thus, there is a need for common (standard) interfaces to different solver packages to facilitate their easy utilization in modern high-performance computing (HPC) applications. Different linear system solvers may be even chosen dynamically, depending on runtime conditions, if application and solvers are *componentized*, i.e. encapsulated into components using some standard interfaces and provided with a means to interconnect according to the rules of a *component model* [2]. A virtue of a component model is to combine components into a coherent application in a ‘loosely coupled’ manner, i.e. only via *exposed* (public) interfaces without modifications to the component internals.

The common component architecture (CCA) [3] is the component model that specifically targets high-performance scientific applications. The CCA specification consists of a set of abstract interfaces written in the scientific interface definition language (SIDL) [4]. Featuring application independence, SIDL enables the design of common interfaces across multiple libraries and multiple parties. SIDL is designed to address the needs of parallel scientific computing, specifically complex numbers, dynamic multi-dimensional arrays, and parallel communication directives. Babel [5], developed at the Lawrence Livermore National Laboratory, is a set of tools to interface scientific software packages for which SIDL interfaces are defined. CCA enables the assembly of HPC applications from software building blocks (components) using common SIDL interfaces and ensures their interoperability via a component *framework*, a software tool that supports CCA specifications. In particular, the Ccaffeine framework [6] has been developed under the SciDAC project [7] funded by the U.S. Department of Energy.

Standardizing interfaces for sparse linear systems is a formidable task, much unlike the interfaces for the dense linear algebra routines. A major obstacle lies in the multitude of sparse matrix formats and solution methods that are based on these formats and on the sparsity structure of the system matrices. For example, matrices with a very irregular sparsity pattern may be easily solved by a sparse version of Gaussian elimination, using software packages such as MUMPS [8] or SuperLU [9]. On the other hand, large-scale matrices arising from certain three-dimensional partial differential equations (PDEs) may be intractable for the *direct-factorization* (exact) methods, whereas some techniques approximating the solution with a given accuracy show good performance. In realistic scientific applications, the use of approximate (called *iterative*) techniques is justified since the solution is desired typically with a certain accuracy, which may range from a few digits in mantissa to machine precision depending on the nature of application. The more the accuracy that is desired, the more the iterations that may be performed before *convergence* with a predefined tolerance. For direct methods, the amount of memory needed to *factorize* the matrix may be a major bottleneck. Once the factorization is completed, only one triangular solve is required to obtain the solution (with



a possible refinement to follow). For iterative methods, depending on the linear system properties, a transformation (called *preconditioning*) may be required in the pre-solution phase to make the system easier to solve. It is desirable to construct a preconditioner such that it is inexpensive, in terms of both the memory used and the overall solution time. Thus, preconditioning may balance the convergence rate with the cost to perform a single iteration.

Sparse linear system solver packages, such as PETSc [10], Trilinos [11], and *hypre* [12], contain a wide array of methods, from which the user may choose depending on the system matrix properties and the HPC application at hand. Recently, we have designed a single Linear Solver Interface [13] using CCA, which spans a broad range of solver packages and facilitates their access in both sequential and parallel environments. Since the CCA model is currently parallelism transparent (e.g. instances of the Ccaffeine framework are multiplexed in each processor and do not provide parallel communication intrinsically [6]), the interprocess communications are handled within each solver package and are not exposed via component interfaces. LISI assumes distributed data storage, such that each processing element owns a part of the matrix distributed in a block-row manner. Although this assumption is typical for existing solver packages, it is desirable to have a means of addressing the distributed matrix layout in the interface. In the future, LISI may be able to leverage the ongoing distributed array descriptor (DAD) [7] effort within CCA, when DAD extends its specifications from dense to sparse matrices. In our recent work [14], the legacy SPARSKIT package [15], developed in the 1990s by Yousef Saad at the University of Minnesota, has been redesigned into a set of CCA components. The new design enables easy interfacing of SPARSKIT with application codes and its extensibility with novel iterative solution methods (called *accelerators*) and preconditioning techniques. In particular, a component encapsulating the state-of-the-art preconditioner algebraic recursive multilevel solver (ARMS) [16] has been added to SPARSKIT.

The design and selection of Sparse Linear Algebra (SpLA) interfaces has to be driven by user requirements. For instance, coupling a linear solver in a simulation involving sophisticated multi-code interactions, such as in multi-scale problems, may require a ‘high-level’ view of the solver and hide the details of all its building blocks into a single component as *black box* [13]. When optimizing the linear system solution, however, one may want to focus on its separate stages. Thus, a finer grain (or *gray box*) view-point is desired, which distinguishes such solution functionalities as preconditioner construction and the iterative approximation method. At the most detailed level of control, such low-level operations as matrix–vector or matrix–matrix multiplication may be represented as components. This could be beneficial for handling very large matrices when conversions to adhere to the solver’s internal sparse matrix format become expensive or when the application code needs access to efficient (tuned) implementations of sparse matrix computational kernels for a given hardware architecture. In this paper, we first explore how these three user requirement levels may be reflected in a choice of interface. Then we provide interface examples for each level and consider usability and computational overheads of the respective interfaces.

This paper is organized as follows. In Section 2, we outline three design choices that we have considered when creating SpLA components. Section 3 describes how each interface level may be used. In Section 4, we examine the computational overheads incurred and provide examples of usability testing for different component interfaces. Section 5 presents a discussion of related work, while Section 6 concludes.



## 2. DESIGN CHOICES

To facilitate the interaction of SpLA software with a scientific application, component interfaces that are sufficiently general to encompass a large variety of SpLA codes and their usages may be developed. This entails a high-level approach (call it HI) that hides the details of SpLA algorithm implementations and handles enough information to distinguish the solution method groups, each of which may have its own parameter types and preprocessing stages. Given a particular SpLA solution method group (e.g. direct or iterative) or particular SpLA package, a user may want to tune its parts to a particular problem at hand. This may be achieved using a medium-level approach (call it MI) to interface design. The MI approach enables swapping and runtime tuning of major building blocks for a given solution method. However, the general features of MI interfaces may be determined and re-used for a class of MI components, such as preconditioners or accelerators. Finally, the basic (kernel) operations of SpLA, such as matrix–matrix and matrix–vector multiplication, may need to have component interfaces (call them low-level interfaces or LI). SpLA packages or user code optimizations may rely on LI components to gain transparency in handling many existing sparse matrix formats and underlying algorithms. The three design choices may be further characterized as follows.

### 2.1. LI level

The low-level design choice refers to operating on objects, such as matrices, directly. Many modern numerical software packages that embrace the object-oriented approach already implement this design choice in the conventional library format (see, e.g. [17]). Thus, it is interesting to consider the LI level implemented as *components* to serve in the componentized applications and to investigate its amount of overhead. In particular, we have encapsulated as the low-level component, called BLASSM, matrix–matrix and matrix–vector operations from SPARSKIT. BLASSM stands for Basic Linear Algebra Subroutines with Sparse Matrices. Its component form is a part of the SPARSKIT suite of components (SPARSKIT-CCA) and extends the SPARSKIT functionality by not requiring the user to know which variation of a function to call for a particular matrix format, and thus enables the application to freely choose its sparse matrix format. Similar to the conventional library approach, this feature is accomplished through the overloading of functions in object-oriented languages, such as C++ and Java, and in some non-object-oriented languages, such as FORTRAN90. Function overloading allows specifying more than one function of the same name in the same scope. As implemented in the BLASSM component, for example, it allows the user to call a *generic* function `amub` to multiply matrix  $A$  by matrix  $B$ , and the function itself will then determine which form of `amub` needs to be called: When the matrices input to `amub` are in the compressed sparse row (CSR) storage format, the implementation of `amub` for the CSR format is used. Such a design allows more flexibility in terms of type-agnostic code. However, it also may have the disadvantage of incurring overhead from function overloading in a programming language (C++ in the case of the BLASSM implementation).

### 2.2. MI level

A medium-level design choice represents a higher level of abstraction than the LI level. Specifically, the major steps of an iterative solution process are encapsulated as components. They may



include separately iterative accelerators, preconditioners, and matrix generation routines, whereas matrix–vector multiplication may not need to be a separate component. For a medium-level design, there is no function overloading, but rather a component implementation for each possible choice of the functionality parameterized by a specific matrix format. This can be more easily seen when considering two different preconditioners. In a low-level design, the preconditioner component would depend on some information to determine which overloaded preconditioner method to call. In a medium-level design, however, a particular preconditioner component interface implements only one preconditioner and that will be the one that is called. In SPARSKIT-CCA, the three solver functionalities—accelerator, preconditioner, and matrix generation—are made into components. The component interface for each functionality is designed such that it has a standard argument list to facilitate component reuse and make easier the interchanging of different component implementations. Two types of preconditioner components, `Base` and `Generic`, have been designed in SPARSKIT-CCA. `Generic` reflects complex data structures and is geared for multi-level formulations used in novel preconditioning techniques, while the `Base` type leverages the FORTRAN77-style argument lists, which are widely used in existing application codes. `Base` preconditioner interface facilitates easy conversion from procedural to component-style programming and is beneficial for existing large-scale simulations relying on the old SPARSKIT functionality. On the other hand, the `Generic` preconditioner interface makes it easier to add more preconditioner component implementations in the future, expanding the preconditioner suite with those existing in other SpLA solver packages. Here, an excerpt from the SPARSKIT-CCA preconditioner interface (Figure 1) is provided along with the interface for accelerator application (Figure 2). For a detailed interface specification, the reader is referred to [14].

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

Figure 1. Component interface for `Generic` preconditioner.

```
package sparskit version 1.0 {
  interface accelerator extends gov.cca.Port {
    void apply(in   rarray<double> rhs(n),
              inout rarray<double> sol(n),
              inout rarray<int> ipar(y),
              inout rarray<double> fpar(y),
              inout rarray<double> w(x),
              in int n,
              in int x,
              in int y); } }
```

Figure 2. Component interface for application of accelerator.



### 2.3. HI level

The high-level design choice provides a more abstract level compared with the other two interface levels. It describes the interaction between scientific applications and SpLA solvers. In particular, a SpLA solver is treated as a *black box* by the scientific application. By using HI, application developers and users do not have to worry about the details of the SpLA solver but simply invoke HI by passing a few required solver parameters. The parameter passing method is generic and allows for a wide variety of data types and values. In particular, both accelerator and preconditioner are encapsulated into one `solver` component, which is a fundamental HI unit for SpLA packages. Thus, the choice of accelerator and preconditioner may be declared in the parameter list passed to the `solver` component. HI aims to facilitate simple and easy access to a multitude of SpLA solver packages, such as Trilinos and PETSc. For example, Figure 3 presents a part of the HI `solver` component interface (*SparseSolver* interface) describing the invocation of the linear system solution method (`solve`). The matrix setup method is overloaded with different sets of pass-in parameters, so that different sparse matrix formats and Fortran-style indexing can be properly handled. Although a generic matrix object could be used to hide the linear system data and provide good data encapsulation, it introduces another level of complexity for the interface users that forces them to construct a matrix object first and only then use the solver. This is unnecessary if there is no other application functionality involving the constructed matrix object. We have also observed that, in many HPC applications, a sparse linear system is represented as three arrays; hence we have opted for this common case as a trade-off between a less general but potentially more light-weight and the ‘all-encompassing’ matrix interface designs. In our HI interface, `setXXX` methods provide the generic way to set up the internal solver parameters, and `key` is the parameter name. The agreement on the key’s name should be associated with the HI, and currently the list of keys contains such items as `solver`, `preconditioner`, `tolerances`, and the maximum number of iterations. The `solve` invokes a solution method from a desired SpLA package and returns the solution

```
package lisi version 0.1 {
  ...
  interface SparseSolver extends gov.cca.Port{
    ...
    int setupMatrix[few_args](
      in rarray<double,1> Values(NNZ),
      in rarray<int,1> Rows(NNZ),
      in rarray<int,1> Columns(NNZ),
      in int NNZ);
    ...
    int solve(
      inout rarray<double,1> Solution(NumLocalRow),
      inout rarray<double,1> Status(StatusLength),
      in int NumLocalRow, in int StatusLength);
    int set(in string key, in string value);
    int setInt(in string key, in int value);
    int setBool(in string key, in bool value);
    int setDouble(in string key, in double value);
    string get_all(); } }
```

Figure 3. High-level component interface for the linear system solver invocation (`solve` method).



vector. Built on top of the contemporary SpLA packages, such as Trilinos and PETSc, HI abstracts the minimal common set of interfaces. Thus, HI allows for easy package switching and seamless interacting of SpLA packages and application during the runtime, which is difficult to achieve if the interfaces of any particular package are adopted at the HI level. For a detailed description and the design rationale of HI interfaces, see [13].

### 3. USING THE THREE INTERFACE LEVELS

Since SpLA solver components are often the major computationally intensive part of an HPC application, their usability is of primary importance, along with their ability to solve a given problem. Usability affects the amount of execution overhead and programming effort that appear in coupling the solver with the application code. The SpLA interface should match application needs in terms of the degree to which the application wishes to control sparse matrix computations. Based on this requirement, three levels of application involvement may be identified.

The high-level model provides a single abstract solver component interface with a set of methods that the user may call on the solver component. In particular, the user needs to construct one `solver` object, and call the `set` methods associated with this object to pass the necessary information from the application to solvers.

```
solver.setupMatrix(<Matrix arguments>)  
solver.setupRHS(<RHS arguments>)  
solver.set("solver", "GMRES")  
solver.set("preconditioner", "ilu")  
solver.solve(<Solution arguments>)
```

To gain flexibility in the solver configuration, methods such as

```
solver.setInt("fillevel", <number>)  
solver.setInt("maxit", <number>)  
solver.setDouble("tol", <number>)
```

can be used to set typical solver parameters. For iterative methods, these parameters may include, among others, `fillevel`, `maxit`, and `tol` to specify, respectively, the preconditioner non-zero density, the maximum number of iterations, and stopping tolerance allowed for the accelerator.

In the medium-level model, the user needs to explicitly connect the components responsible for different solution stages. To recreate the example from the high-level model for the GMRES accelerator and the incomplete LU (ILU) preconditioner [18], which are both from the SPARSKIT suite of components, the user needs to connect individual accelerator and preconditioner components to his application.

```
accelerator.apply(<Matrix and RHS arguments>)  
preconditioner.create(<ilu arguments>)
```

The difference in usage between the medium-level and high-level interfaces is that application scientists need to know the individual parts/components of a linear solver and explicitly connect

---



them. They must also know the argument lists and matrix formats used by those components. Consequently, component developers are asked to create separate components for a given sparse matrix format, stage of the linear system solution, type of preconditioner, and specific input/output argument lists. For example, preconditioner components may have an interface consisting of `a`, `ja`, `ia`, `lfil`, `droptol`, `alu`, `jlu`, `ju`, and `nnz`, which comprise a typical set of input/output parameters for an ILU-type preconditioner: the triple `(a, ja, ia)` represents the sparse matrix (in the CSR format), `lfil` represents the maximum fill-in for each row of lower (L) and upper (U) triangular matrices, `droptol` is the tolerance for dropping small factors, and the triple `(alu, jlu, ju)` represents the output matrix of the ILU factorization stored in the modified sparse row format (see, e.g. [18]).

The low-level model is rather easy to use for the application scientist since details of the sparse matrix formats and corresponding specific implementations are hidden. The application scientist chooses the data format and passes the arguments to the required *generic* operation/method. The method essentially acts as a wrapper and is expected to determine by the type of the arguments which concrete version of the operation to substitute for the generic one. For example, to multiply two sparse matrices in the CSR format, the following sequence of operations may be issued:

```
matrix1->type = "CSR"
matrix2->type = "CSR"
blassm.amub(matrix1,matrix2)
```

The `amub` function is expected to call the version of the matrix–matrix multiplication that multiplies two matrices in the CSR format. In an HPC application, LI components may be used in a stand-alone fashion, i.e. without SpLA solver components. The advantage of using LI components becomes apparent for very large matrices when format conversions or data copying is prohibitively expensive.

#### 4. TESTING AND EVALUATION

SpLA solver CCA components with high-, medium-, and low-level interfaces have been tested in typical real-world application scenarios. Consider first the solution of a three-dimensional PDE, which arises in modeling of many physical phenomena:

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} = f \quad (1)$$

with Dirichlet boundary conditions, discretized with a seven-point centered finite-difference scheme on an  $n_x \times n_y \times n_z$  grid. The tests of MI and LI components have been performed at the Ames Laboratory on an Intel XEON 2.2 GHz processor with 768 MB RAM running Debian Sarge (3.1) and compiled with the Debian-shipped GCC 3.3.5. For the HI components, the tests were conducted at the Indiana University on an AMD Opteron 1.9 GHz processor with 4 GB RAM running Red Hat 3.4.5-2 and all the packages and software were compiled with GCC 3.4.6. The CCA Tools version 0.6.0 rc5 was used as the component framework for our tests.



#### 4.1. HI component

High-level interfaces are implemented in C++ for Trilinos [11] and PETSc [10], which provide widely used sparse linear solvers. Tests were conducted to compare the overheads introduced by component implementation during the linear system solution process. Table I shows a comparison of a non-component version of PETSc (column `PETSc`) and its component implementation (column `PETSc-CCA`), while Table II shows a comparison for the corresponding implementations of Trilinos (columns `Trilinos` and `Trilinos-CCA`, respectively). In both tables, column `its` shows the number of iterations taken to convergence, column `nmz` refers to the number of non-zero elements in the matrix obtained for  $n_x = n_y = 40, 50, \dots, 100$  and  $n_z = 10$ , and column `diff` shows the component overhead as the percentage of the non-component execution time. The execution times are averages of 10 runs for each problem size, and the system is solved using the Jacobi preconditioner and BICGSTAB accelerator (see, e.g. [18]), with a stopping tolerance of  $10^{-6}$  and other default input parameters as provided by PETSc and Trilinos, respectively. The testing involves three components: driver, PETSc, and Trilinos, which are wired as shown in Figure 4 (left). Either of the latter two may be easily connected to the driver at run-time. The solid and dotted lines show that the connection can be made for either one or another. In fact, this is how the results in Tables I and II were obtained for `PETSc-CCA` and `Trilinos-CCA`, respectively: No source code modification, re-compilation, or re-linking was used. Only single-processor executions of PETSc and Trilinos were performed to compare with the MI components of SPARSKIT-CCA, which is a sequential package. For the parallel experiments, see [13]. Test results show only a tiny component overhead due to a call to the light-weight C++ wrapper functions for the underlying

Table I. Timing comparisons (in seconds) of high-level PETSc component.

<code>nmz</code>	<code>its</code>	<code>PETSc</code>	<code>PETSc-CCA</code>	<code>diff (%)</code>
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

Table II. Timing comparisons (in seconds) of high-level Trilinos component.

<code>nmz</code>	<code>its</code>	<code>Trilinos</code>	<code>Trilinos-CCA</code>	<code>diff (%)</code>
76 760	40	0.179	0.180	0.41
122 880	38	0.289	0.289	0
179 800	44	0.474	0.481	1.44
247 520	44	0.652	0.658	1.03
326 040	44	0.922	0.929	0.73
415 360	44	1.175	1.176	0.07
515 480	42	1.382	1.394	0.84

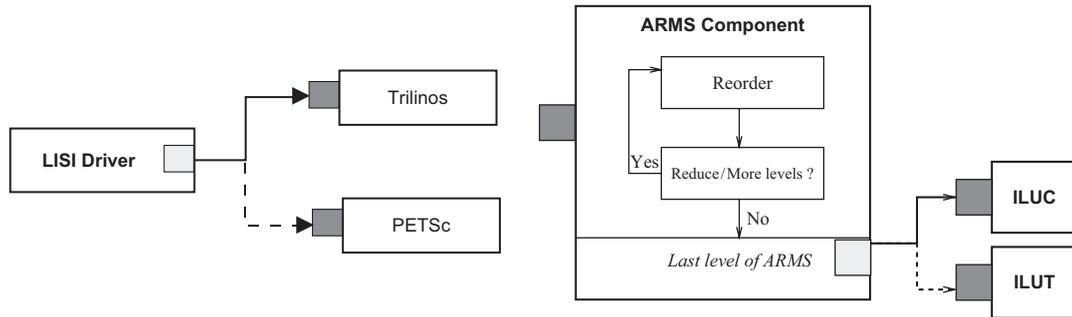


Figure 4. Switching of HI components (left) and MI components within the ARMS preconditioner framework (right). The *Uses* and *Provides* interfaces are depicted as light- and dark-shaded squares, respectively. The flowchart within the ARMS component indicates its multi-level structure that may be implemented recursively.

Table III. Timing comparisons (in seconds) of medium-level SPARSKIT components.

<i>nnz</i>	<i>its</i>	SKIT	SKIT-CCA	diff (%)
76 760	36	0.0792	0.08	1
122 880	36	0.14	0.14	0
179 800	36	0.208	0.215	3.36
247 520	36	0.334	0.345	3.29
326 040	36	0.443	0.448	1.13
415 360	36	0.570	0.588	3.12
515 480	36	0.7185	0.730	1.6

package functionality. A more substantial overhead, originating from the CCA framework initializations, also exists, but it may vary depending on the number of components instantiated in each application. This overhead should be constant for a typical HPC application in which the number of components is fixed. We have also observed that the overhead percentage decreases with increase in matrix size. The tables do not present the framework instantiation overhead since it is incurred only once and may be amortized over the course of large-scale simulations.

#### 4.2. MI components

Table III shows a comparison of the original SPARSKIT (column *SKIT*) and its component implementations (column *SKIT-CCA*) for the medium-level component design. The other columns in Table III represent the same type of data as in Table I. The total execution times (in seconds) are for an average of 10 runs on each problem, with standard deviations ranging from  $2.89 \times 10^{-4}$  to  $5.71 \times 10^{-3}$ . We have solved the linear systems with ILUT as the preconditioner and flexible GMRES [18] as the accelerator. The test was performed as a sequence of connections to the different components, an accelerator and a preconditioner, in each iteration of the solution process. These results show only a small overhead incurred by using the component implementations. The percentage of the incurred overhead appears to be relatively small and stable unlike the overhead



increase observed in the BLASSM component, which may also depend on the number of matrix operations performed.

We have used the tuning and analysis utilities (TAU) [19] performance component to do performance analysis on medium-level components. The TAU performance components are currently the best way to do timing analysis on CCA components. In the case of SPARSKIT-CCA, they give the user an ability to measure how much time is spent in each component during the iterative solution process. The results can be seen in Table IV, in which column `Func` states the function name that was called. The column `Norm` shows the difference between `SKIT-CCA` and `SKIT` execution times (before rounding) of the corresponding function relative to the `SKIT` execution time. For four different sizes of the problem in Equation (1), component function calls are presented in the descending order of their execution times. Variations in the relative order of the calls across different problems may be attributed to the varying problem difficulty, which may result in a denser preconditioner, and also to more memory allocations as the problem size increases. This fine-grain performance analysis indicates that no particular function call incurs excessive overhead. The majority of time is spent either in `lusol` or in `create` functions, which apply and construct the preconditioner, respectively. Note that the problems with  $nnz = 38\,880$  and  $nnz = 45\,847$  represent domains with  $n_x = n_y = n_z \equiv n_p$ , with  $n_p = 20$  and  $21$ , respectively. They also require more iterations to converge than the other cases in Table IV do.

#### 4.2.1. Fine-grain tuning of solution methods

ARMS [16] is a multi-level preconditioner framework designed to aid in solving difficult linear systems. In a nutshell, the construction of the ARMS preconditioner consists of two steps. First,

Table IV. Performance analysis using TAU.

<code>nnz</code>	<code>Func</code>	<code>Calls</code>	<code>SKIT-CCA</code>	<code>SKIT</code>	<code>Norm (%)</code>
38 880	<code>lusol</code>	46	20	16	25.0
	<code>create</code>	1	18	16	12.5
	<code>apply</code>	93	12	12	0.0
45 847	<code>create</code>	1	25	20	25.0
	<code>lusol</code>	46	20	20	0.0
	<code>apply</code>	93	13	12	8.3
76 760	<code>lusol</code>	36	35	34	2.4
	<code>create</code>	1	34	34	0.0
	<code>apply</code>	73	28	27	2.9
179 800	<code>create</code>	1	96	95	1.3
	<code>apply</code>	73	96	90	7.3
	<code>lusol</code>	36	91	81	13.1

Timings in columns `SKIT` and `SKIT-CCA` are in milliseconds, rounded up to the nearest integer.



reorder the matrix  $A$  into a  $2 \times 2$  block form:

$$A = \begin{pmatrix} B & F \\ E & C \end{pmatrix} \quad (2)$$

using the permutations of rows and columns defined according to some given criteria, such as making the sub-matrix  $B$  block-diagonal. Second, use an ILU technique to obtain an approximate factorization of  $B$  and approximations to the matrices  $L^{-1}F$ ,  $EU^{-1}$ , and  $A_1$ :

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \approx \begin{pmatrix} L & 0 \\ EU^{-1} & I \end{pmatrix} \times \begin{pmatrix} U & L^{-1}F \\ 0 & A_1 \end{pmatrix} \quad (3)$$

The process is repeated recursively on the matrix  $A_1$ , which may now be renamed  $A$ , until the selected number of levels is reached. At the last level, a simple (single-level) preconditioner is used on the entire (reduced) system. More detailed information about the ARMS preconditioner and its level structure may be found in [16,20].

The ARMS component allows application scientists to easily choose among available last-level system preconditioners and even add their own preconditioners. For testing, we have made a novel version of ILU preconditioner (ILUC [18]) available to the ARMS component as the last-level preconditioner. ILUC is a fast preconditioner and also has the ability to incorporate sophisticated drop tolerances, thus our desire to make it available for the connection to the ARMS component. Figure 4 (right) sketches the ARMS component and the switching between two different last-level preconditioner components, ILUC and ILUT. The last-level preconditioner choice is typically made at the stage of component assembly, e.g. in a configuration file. Different last-level preconditioners may require different arguments or use different data structures; hence, a method that allows a flexible way to set these arguments is needed. To solve this problem, the `GenericPreconditioner` interface was developed [14]. The method `getName` is also a part of the interface and is used by ARMS (or any other calling component) to determine which component implementation is connected. Using `getName`, ARMS can determine which arguments to set and in what format the arguments are expected to be for its last-level component.

To test the usability of the ARMS component and to observe the benefits of switching last-level preconditioners, we have used two linear systems arising in circuit simulation and device modeling. These fields are known to produce linear systems difficult to solve by iterative methods since the corresponding matrices are notorious for their irregular structure and poor conditioning. In particular, the circuit simulation matrix `scircuit` from the University of Florida collection [21] has 84 zeros on its diagonal, while its total dimension is 170 998 and the number of non-zero entries is 958 936. Figure 5 presents the matrix structure and indicates its entry magnitudes: the darker colors correspond to larger magnitudes. The second matrix `igbt3` has the dimension of 10 938 with 234 006 non-zeros, and no zero diagonal elements. This matrix is ill-conditioned: In [22], its condition number has been estimated to be  $4.74 \times 10^{19}$ .

Table V shows a comparison of using ARMS component and two available last-level preconditioners ILUC and ILUT (column `Precon`), as well as using only single-level ILUT and ILUC. (In this test, all the preconditioner components are connected via the `GenericPreconditioner` interface, and the ARMS version with non-symmetric permutations [20] has been used.) Column `nnz`

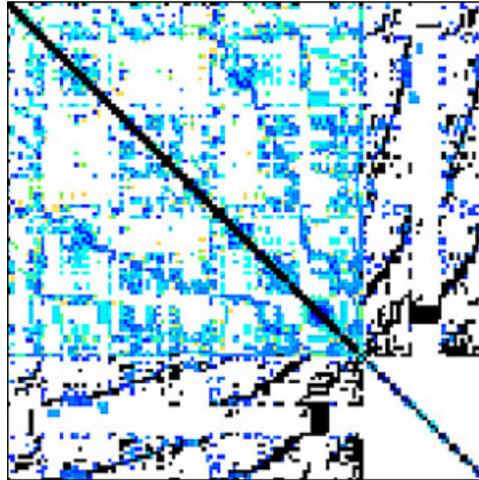


Figure 5. The structure and entry magnitudes for the `scircuit` matrix.

Table V. Comparison of a linear system solution process when `GenericPreconditioner` is used with easy switching of preconditioner components.

matrix	<i>mz</i>	Precon	<i>its</i>	<i>time</i>
<code>scircuit</code>	958 936	<i>ARMS + ILUC</i>	7	124.9
		<i>ARMS + ILUT</i>	7	125.1
		<i>ILUC</i>	55	11.2
		<i>ILUT</i>	*	*
<code>igbt3</code>	234 006	<i>ARMS + ILUC</i>	8	20.23
		<i>ARMS + ILUT</i>	6	19.5
		<i>ILUC</i>	*	*
		<i>ILUT</i>	19	0.55

refers to the number of non-zero entries in the matrix. Columns *its* and *time* refer to the number of iterations and time (in seconds) needed for preconditioner construction and iterative convergence, respectively. An asterisk denotes an inability to converge when the stopping tolerance of  $10^{-6}$  is used. The results for `scircuit` show that ARMS with ILUC performs slightly better than ARMS with ILUT and that the ILUC preconditioner has the best time, but takes many more iterations to converge than the ARMS preconditioner. Note that execution times with ARMS preconditioner are higher than those with single-level preconditioners in Table V because the ARMS preconditioner took longer to construct. The ILUT preconditioner implementation used for testing returns an error since it does not accept zeros on the diagonal. For the `igbt3`, the ILUC preconditioner is not able to converge, contrary to the solution with ILUT that takes 19 iterations and is fast (0.55 s). Although ARMS with ILUC converges in a small number of iterations, ARMS with ILUT performs better in terms of both iteration count and the total execution time.



Table VI. Comparison of SPARSKIT using low-level componentizing of BLASSM library.

<i>nnz</i>	BLASSM	BLASSM-CCA	diff (%)
76 760	0.0028	0.0036	28.57
122 880	0.00472	0.0062	31.35
179 800	0.00728	0.0088	20.88
247 520	0.00984	0.00122	23.98
326 040	0.01296	0.0152	17.28
415 360	0.01668	0.02	19.9
515 480	0.021	0.0254	17.59

### 4.3. LI components

Table VI shows a comparison of the original SPARSKIT (column BLASSM) and a component implementation (column BLASSM-CCA) for low-level component design, while the other column designations are the same as in Table III. The test results for *BLASSM-CCA* were obtained by calling the matrix–matrix multiply function `amub` with the matrices in the CSR format as input arguments and by taking the average over 10 calls, with standard deviations ranging from  $5 \times 10^{-5}$  to  $7.26 \times 10^{-4}$ . We observe that the incurred overhead varies from 17.28 to 31.35%. Only a small part of the overhead is incurred from the componentization, while most of the overhead comes from the function overloading within the BLASSM component.

## 5. RELATED WORK

The Matrix Template Library (MTL) [23] is a high-performance generic component library that provides comprehensive linear algebra functionality for a wide variety of matrix formats. The MTL uses a five-fold approach, consisting of generic functions, containers, iterators, adaptors, and function objects developed for high-performance numerical linear algebra. The containers, iterators, and adaptors are used to represent and manipulate linear algebra objects such as matrices. Using an optimizing compiler, the MTL has been able to produce performance equal to and sometimes better than vendor-tuned math libraries such as the Sun Performance Library.

Many packages exist that use a ‘high-level’-like design. One such project is the toward optimal petascale simulations (TOPS) solver interfaces [24]. TOPS is an integrated software infrastructure focused on developing, implementing, and supporting optimal or near-optimal schemes for PDE-based simulations and closely related tasks, including optimization of PDE-constrained systems, sensitivity analysis, eigenanalysis, adaptive time integration, and core implicit linear and nonlinear solvers. A common interface for the TOPS software infrastructure has been developed and is being integrated into CCA. The idea for the next generation of PETSc solvers is to use and extend the TOPS SIDL interfaces.

The Scalable Linear Solvers project at the Lawrence Livermore National Laboratory developed *hypre* [12], a library of high-performance preconditioners that features parallel multi-grid methods for both structured and unstructured grid problems. While not currently being implemented using



the CCA, *hypre* has developed some sample SIDL interfaces and currently has a development version that uses Babel.

## 6. CONCLUSIONS

In this paper, we have examined in detail three different design choices when creating sparse linear system solver components and discussed their usability issues. Depending on their component granularity, we have distinguished three interface levels, high (HI), medium (MI), and low (LI), which also correspond to three distinct usability requirements. HI enables an easy selection among the multiple solver packages with the ‘black box’ style of usage. MI gives more control to an application scientist to work directly on the preconditioner and accelerator tuning. LI provides easy access to a wide array of matrix handling routines without the tedious matrix format conversions.

It has been observed that, among the three user interface levels, HI incurs the smallest overhead, which is not higher than 1.72% in the experiments considered in this paper. The MI level overhead appears to stay below 3.5%, but may vary with problem size due to a varying number of component calls for each problem size. The usability of MI has been demonstrated by connecting ‘on-the-fly’ the multi-level ARMS preconditioner with different single-level preconditioners. We have shown how this coupling enables efficient solution of difficult linear systems with novel preconditioning techniques. The LI overhead appears high mainly because of the function overloading and, thus, LI may be beneficial only for large-scale data.

The HI, MI, and LI levels may be also viewed as *hierarchical* interfaces in the sense that they may be combined into a single set of interfaces so that MI components are connected to HI ones, and LI components may be used on both HI and MI levels. For example, high-level interfaces may be supplied to the driver component currently governing (via MI) the selection of preconditioners and accelerators in SPARSKIT. In this way, the entire linear system solver assembled from SPARSKIT components may be accessed via HI. A detailed design of the hierarchical interfaces, however, is left as a future work.

## REFERENCES

1. Dongarra J, Butari A. Freely available software for linear algebra on the web. <http://www.netlib.org/utk/people/JackDongarra/la-sw.html> [August 2006].
2. Szyperski C. *Component Software: Beyond Object-oriented Programming*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, U.S.A., 2002.
3. CCA-Forum. The DOE common component architecture project. <http://www.cca-forum.org/> [November 2006].
4. Kohn S, Kumfert G, Painter J, Ribbens C. Divorcing language dependencies from a scientific software library. *Tenth SIAM Conference on Parallel Processing*, Portsmouth, VA, 12–14 March 2001. LLNL document UCRL-JC-140349.
5. Babel Team. The DOE Babel Project. <http://www.llnl.gov/casc/components/babel.html> [November 2006].
6. Bernholdt DE, Allan BA, Armstrong R, Bertrand F, Chiu K, Dahlgren TL, Damevski K, Elwasif WR, Epperly TGW, Govindaraju M, Katz DS, Kohl JA, Krishnan M, Kumfert G, Larson JW, Lefantzi S, Lewis MJ, Malony AD, McInnes LC, Nieplocha J, Norris B, Parker SG, Ray J, Shende S, Windus TL, Zhou S. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications* 2006; **20**(2):163–202.
7. TASCs SciDAC center. <http://www.scidac.gov/compsci/TASCs.html> [August 2007].
8. Amestoy PR, Duff IS, L’Excellent J-Y, Koster J. MUMPS: A general purpose distributed memory sparse solver. *Proceedings of PARA2000, the Fifth International Workshop on Applied Parallel Computing (Lecture Notes in Computer Science, vol. 1947)*, Gebremedhin AH, Manne F, Moe R, Sorevik T (eds.), Bergen, 18–21 June 2000. Springer: Berlin, 2000; 121–131.



9. Li X, Demmel J. SuperLU\_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software* 2003; **29**:110–140.
10. Balay S, Gropp WD, McInnes LC, Smith BF. PETSc users manual. *Technical Report ANL-95/11-Revision 2.1.0*, Argonne National Laboratory, 2001.
11. Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, Lehoucq RB, Long KR, Pawlowski RP, Phipps ET, Salinger AG, Thornquist HK, Tuminaro RS, Willenbring JM, Williams A, Stanley KS. An overview of the Trilinos project. *ACM Transactions on Mathematical Software* 2005; **31**(3):397–423. ISSN: 0098–3500.
12. Chow E, Cleary A, Falgout R. HYPRE user's manual, version 1.6.0. *Technical Report UCRL-MA-137155*, Lawrence Livermore National Laboratory, Livermore, CA, 1998.
13. Liu F, Bramley R. CCA-LISI: On designing a CCA parallel sparse linear solver interface. *Proceedings of the 21th International Parallel & Distributed Processing Symposium (IPDPS)*. ACM/IEEE Computer Society: Long Beach, CA, 2007; 10.
14. Jones J, Sosonkina M, Saad Y. Component-based iterative methods for sparse linear systems. *Concurrency and Computation: Practice and Experience* 2007; **19**:625–635.
15. Saad Y. *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations, No. 90–20*. NASA Ames Research Center, Moffett Field, CA, 1990.
16. Saad Y, Suchomel B. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. *Numerical Linear Algebra with Applications* 2002; **9**(5):359–378.
17. Yang D. *C++ and Object-oriented Numeric Computing for Scientists and Engineers* (1st edn). Springer: Berlin, 2000.
18. Saad Y. *Iterative Methods for Sparse Linear Systems*. SIAM: Philadelphia, PA, 2003.
19. Shende SS, Malony AD. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications* 2006; **20**(2):287–311. ISSN: 1094–3420.
20. Saad Y. Multilevel ILU with reorderings for diagonal dominance. *SIAM Journal on Scientific Computing* 2005; **27**(3): 1032–1057.
21. Davis T. University of Florida sparse matrix collection. *NA Digest*, 1997. Available at: <http://www.cise.ufl.edu/research/sparse>.
22. Schenk O, Röllin S, Gupta A. The effects of unsymmetric matrix permutations and scalings in semiconductor device and circuit simulation. *IEEE Transactions on CAD of Integrated Circuits and Systems* 2004; **23**(3):400–411.
23. Siek JG, Lumsdaine A. The matrix template library: A generic programming approach to high performance numerical linear algebra. *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, 1998. Springer: London, U.K., 1998; 59–70. ISBN: 3-540-65387-2.
24. TOPS SciDAC center. <http://www.scidac.gov/math/TOPS.html> [August 2007].