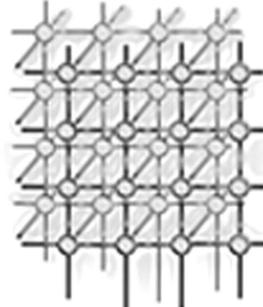

Component-based iterative methods for sparse linear systems *



J. Jones^{1,*}, M. Sosonkina¹ and Y. Saad²

¹ Ames Laboratory, Iowa State University, Ames, IA 50011

² Department of Computer Science and Engineering, University of Minnesota, 200 Union Street S.E., Minneapolis, MN 55455

SUMMARY

Iterative methods play an important role in solving large-scale systems of linear equations that arise in real-world applications. Due to numerous linear system properties that may affect solution, it is rather difficult for a user to develop a good sparse linear system solver from scratch. Thus various collections of solution methods are made available to the user. One such a software package is SPARSKIT, which is well-known in scientific community. Written in FORTRAN77 and provided with cumbersome interface, it is considered, however, a legacy code. Our objective is to enable its wider usage in modern applications and to facilitate further SPARSKIT enhancements. Applying a “peer-component” design, we have created a set of SPARSKIT components that (a) incorporate both original and new iterative methods, (b) are readily extensible with more methods, (c) may be connected to applications in a component framework, and (d) provide access from a variety of programming languages. Tools available from the Common Component Architecture (CCA) Forum enabled our component design of SPARSKIT.

KEY WORDS: Component architecture; Iterative methods; Sparse matrices

1. Introduction

Sparse linear system solution is one of the most computationally-intensive tasks of many high-performance applications. Sophisticated solution techniques may be required for

*Correspondence to: Ames Laboratory, Iowa State University, Ames, IA 50011

Contract/grant sponsor: U.S. Department of Energy; contract/grant number: W-7405-ENG-82

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



efficient execution and, for difficult and large linear systems, to obtain a solution at all. Thus, an application user is encouraged to look for an appropriate solution method within existing software collections rather than write his/her own implementation. See, for example, Netlib [10] for a list of such packages. Besides the desired functionality, the solution method has to handle application-specific representation for the sparse matrix. The multitude of sparse matrix formats and solution methods based on these formats calls for finding a general common ground - a well-defined interface - between sparse matrix packages and user applications. Such an interface would allow an application to connect an appropriate solution method, chosen from a variety of packages regardless of their native programming language. In particular, Babel [5], developed at Lawrence Livermore National Laboratory, provides language interoperability using Scientific Interface Definition Language (SIDL). SIDL is an extension of the OMG Interface Definition Language (IDL) that adds multi-dimensional array support and complex value support to the original IDL. Babel generates “glue code” that enables the inter-operation of the supported collection of programming languages: C, C++, FORTRAN77, FORTRAN90, Java, and Python. One way to provide a common interface is to use the specifications defined by the Common Component Architecture (CCA) Forum [3] for high-performance computing applications. As part of CCA, the Ccfe [2] component framework has been developed as a means to connect components in the “plug-and-play” fashion, i.e., dynamically at run time. The “plug-and-play” connectivity gives the user more options in the choice of solution methods/components and allows greater experimentation without the need for recompiling the application.

In this paper, we emphasize the component design for sparse linear system solution methods by transforming a FORTRAN77 legacy suite of codes SPARSKIT [11] into a set of components. This package, is a basic tool-kit for sequential sparse matrix computations and is widely used in scientific community.

1.1. Iterative methods for sparse linear systems

Systems of linear equations arise in the numerical solution of many real-world applications such as computational fluid dynamics, structural mechanics, and circuit simulations. Systems of linear equations are typically of the following form:

$$Ax = b, \quad (1)$$

where A is often referred to as the system matrix, b is called the right-hand side vector, and vector x contains the unknowns. If linear systems arise from the standard discretization techniques, e.g., finite elements, finite differences, and finite volumes, the system matrix A is sparse, meaning that the number of non-zeros per row is small. Discretization on a very fine computational grid will result in a linear system that has a very large number of unknowns. Traditional direct solution methods for (1), such as Gaussian elimination, will become inapplicable due to high operation counts, large storage demand, and high sensitivity to computer round-off errors. Large-scale linear systems are therefore better handled by iterative methods, which find the solution *approximately* with a certain accuracy.



A powerful class of such methods also called accelerators are Krylov subspace solvers [13]. The SPARSKIT package has a large collection of these methods, a good example of which is Generalized Minimum RESidual algorithm (GMRES) [13].

A drawback of iterative methods is that it is not easy to predict how fast a linear system can be solved to a certain accuracy and whether it can be solved at all by certain types of iterative solvers. This depends on the algebraic properties of the matrix. Linear system can be transformed into one that has the same solution but for which the iterative method achieves faster the desired solution accuracy. This transformation process is called *preconditioning*. With a good preconditioner, the total number of steps required for convergence can be reduced dramatically, at the cost of a slight increase in the number of operations per step, resulting in much more efficient algorithms. Incomplete LU factorization (ILU) is a technique for preconditioning. In essence, it factors the sparse system matrix *approximately*:

$$A = \tilde{L}\tilde{U}, \quad (2)$$

where \tilde{L} and \tilde{U} are lower- and upper-triangular matrices approximating the exact factor matrices L and U , respectively. Different ILU factorizations exist depending on how these approximating matrices are constructed. For example, the ILU featured in SPARSKIT originally is incomplete LU factorization with dual threshold ILUT(τ , p), which is usually more robust than the incomplete LU factorization with a pre-defined level of fill ILU(k). In the ILUT preconditioning to obtain \tilde{L} and \tilde{U} , a dropping strategy is applied in each row of L and U whenever the magnitude of the row entry is below (relative) tolerance τ . This is followed by the dropping strategy based on the row memory requirements. Only the p largest elements are left in the row.

1.2. Related work.

Three large projects, *Hypre* [4], PETSc [1], and Trilinos [8], contain sparse linear solvers, including parallel implementations, and provide object-oriented interoperable interfaces. These packages originated more recently than SPARSKIT and used more modern software engineering tools and practices. Thus, their designs yield to component architectures easier than a legacy code does so. Historically, the PETSc Equation Solver Interface Standards [6] was one of the first attempts to create a set of standards for equation-solver services and components. Fundamental questions – such as component structure, interaction, argument lists – require a general solution that would satisfy the needs of a majority of applications. For example, in the *Hypre* CCA interface, parameters are assigned using “set” methods that can be used to assign values to arguments. PETSc CCA interfaces use high-level classes and data structures from the TOPS CCA interfaces. The IFPACK package from the Trilinos project contains preconditioner C++ interfaces that may be called by other packages and software projects during the configuration at compile-time. There exist CCA interfaces to Eptera defining a common language for distributed linear algebra objects in Trilinos. Another sparse linear algebra software package is Optimized Sparse Kernel Interface



(OSKI) [15]. OSKI provides a collection of low-level C primitives for automatically tuning of computational sparse matrix kernels. Although OSKI is not implemented in Babel/CCA, it supplies SIDL interface definitions.

This paper is organized as follows. In Section 2, we outline new additions to the SPARSKIT package to be integrated into a single suite of SPARSKIT components. The component design and its relation to the old SPARSKIT are presented in Section 3. Implementation issues and test results are explained in Sections 4 and 5, respectively. Section 6 concludes.

2. SPARSKIT extensions

SPARSKIT is well-known for its state-of-the-art preconditioners and accelerators. For some time, however, the development of new solution techniques has been divorced from SPARSKIT. New diverse preconditioners, such as ILUC, VBILU, and ARMS (described later in this section), have been created recently with mature sparse matrix data structures and preconditioner-specific user interfaces. With the help of CCA tools we have integrated them into the SPARSKIT package, such that their efficiency is preserved and they all may be easily tested for a sparse linear system at hand.

ILUC: Crout version of incomplete LU factorization. This is a novel implementation of ILUT, which utilizes more efficiently the sparse matrix data structure and builds preconditioner by rows and columns *simultaneously*. It is faster than ILUT and enables more rigorous dropping strategies in the factorization.

VBILU: Variable Block incomplete LU factorization. Blocking is a frequently used technique to increase the efficiency of numerical computations for modern architectures that feature several cache levels. For sparse matrices, however, the problem of blocking goes beyond finding an appropriate block size for a given computer. First, one has to *find* and *construct* blocks by possibly adding zero-valued sparse matrix entries in appropriate places. This is a dynamic run-time process that has been considered, for example, in SPARSITY [9] for basic sparse-matrix operations. For preconditioning, [12] proposes a technique that automatically discovers variable block structure of a sparse matrix. This technique, based on graph compression algorithms, is incorporated into a block incomplete LU factorization to yield a variable block preconditioner, called VBILU [12]. It is more efficient compared with the point versions of ILU and is more suitable for denser sparse matrices arising, for example, in multi-level solution techniques.

2.1. ARMS: Algebraic Recursive Multilevel Solver

The ARMS preconditioner [14] is based on multi-level ILU techniques which exploit *independent sets*. An independent set is a set of unknowns that are not coupled to each

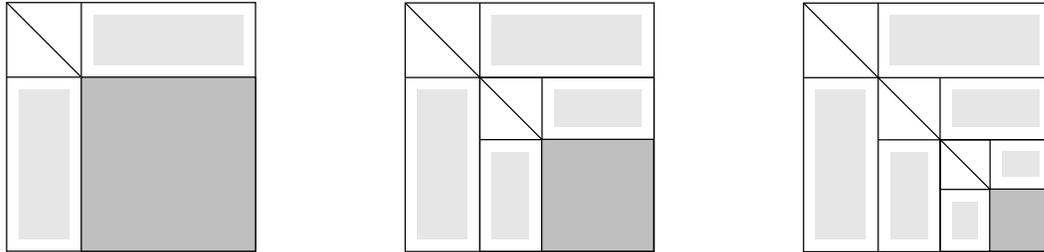


Figure 1. The ARMS procedure executed for three levels

other. Such orderings transform the original linear system into a system of the form

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \begin{pmatrix} u \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}, \tag{3}$$

where B is a diagonal matrix. A block-independent set is a generalization of the independent set to a collection of subsets (blocks) of unknowns, such that there is no coupling between unknowns of any two different blocks. Unknowns within the same block may be coupled. When the unknowns of the group independent sets are labeled first, the matrix A will have the block structure (3) in which the B block is block diagonal instead of diagonal.

In ARMS, the following block factorization is computed “approximately”

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \approx \begin{pmatrix} L & 0 \\ EU^{-1} & I \end{pmatrix} \times \begin{pmatrix} U & L^{-1}F \\ 0 & A_1 \end{pmatrix}, \tag{4}$$

where $LU \approx B$, and $A_1 \approx C - (EU^{-1})(L^{-1}F)$. In a nutshell, the ARMS procedure consists of two steps: first, obtain a block-independent set and reorder the matrix in the form (3); second, obtain an ILU factorization $B \approx LU$ for B and approximations to the matrices $L^{-1}F$, EU^{-1} , and A_1 . The process is repeated recursively on the matrix A_1 , which may now be renamed A , until a selected number of levels is reached. Figure 1 sketches the ARMS procedure, in which darker shaded areas represent level matrices A_i formed consecutively on levels $i = 1, 2, 3$. Recursive multilevel strategies of various types can be defined. In a preconditioner application step, it is not specified how the system $A_1 y = g_1$ is to be solved. This provides a source of many possible variations, and thus ARMS may be viewed as *preconditioner framework*. It may incorporate existing ILU factorizations with quite different properties based on the parameters, such as number of levels and size of block-independent set. For example, when the last level is reached, one can solve the system by an ILUT-preconditioned GMRES iteration [13] or a simple solve with the ILUT factors.

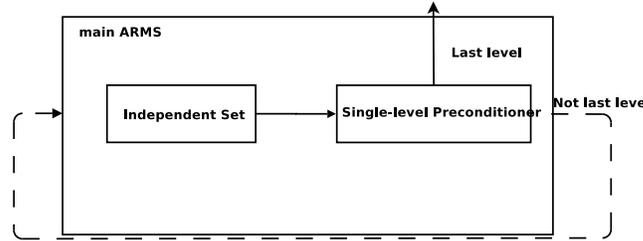


Figure 2. Component structure of ARMS preconditioner

3. Design of SPARSKIT components

We view ARMS as a preconditioner framework, which, due to its multi-level recursive structure, facilitates the construction of preconditioner specifically for the problem at hand. By setting appropriate input parameters, ARMS may be re-configured not only to produce a preconditioner of particular size but also to result in qualitatively different preconditioners, which use different incomplete LU factorizations. The versatility of ARMS gives rise to component design challenges. The general component structure is of primary importance because of the recursive nature of ARMS. Figure 2 displays this structure, as seen in one level of ARMS. Each box denotes a component in Figure 2. The component representation of ARMS is recursive and has a capability to interface with a variety of “base” preconditioners at each level. Recursivity of the main ARMS component is shown as a dashed line. Single-level preconditioners, such as ILUC or ILUT, are connected to ARMS via preconditioner interfaces, which reflect the argument list of single-level preconditioner. The rationale for such a design is to emphasize the independent nature of ARMS level structure, to provide general interfaces for single-level preconditioners, so that they may also be connected directly, not via ARMS; and to efficiently execute these preconditioners avoiding unnecessary copying and data conversions. Thus, we have implemented a preconditioner interface, `BasePreconditioner`, for the preconditioners with Fortran-style sparse matrix representations as arguments and a generic preconditioner interface `GenericPreconditioner`, for the preconditioners with more general sparse matrix representations. The following section describes our implementations.

4. Implementation issues

To adhere to our new SPARSKIT design as a set of components, which interoperates with both the original SPARSKIT and its extensions, written in FORTRAN77 and C,



respectively, we created a Java-like structure such that all the members of the same “class” perform a particular type of task in SPARSKIT. For example, preconditioning, iterative accelerating, or matrix formatting – each belongs to its own “class”, i.e., implements a particular interface. The following are the classes for SPARSKIT: Accelerator, Preconditioner, Matrix Generation, Matrix-Vector multiplication, and Matrix Input/Output. This implementation allows for more functionality, possibly developed in another language, to be supported by the CCA tools and be added transparently to the user by simply creating a connection to the new component in Ccafe. Figure 3 shows a solution of sparse linear systems using SPARSKIT components loaded into the Ccafe framework.

4.1. Preconditioner interface for original SPARSKIT

The user may be able to take advantage of the new functionality without changing the source code since we have developed for each interface a standard set of arguments and naming conventions. This issue is very important for easy and uniform access to SPARSKIT routines. Since the preconditioners in original SPARSKIT accept varying number and types of arguments, it was much harder to design a uniform interface for preconditioners than for accelerators, which typically accept the same arguments and all use reverse communication (see [11] for more details). Our implementation of SPARSKIT preconditioners also simplifies SPARSKIT usage by removing the need for temporary work arrays, which are often passed as subroutine arguments in FORTRAN77. In a nutshell, we propose the following argument list and names for a base SPARSKIT preconditioner `BasePreconditioner` interface. (Interface is also called `Port` in the CCA terminology.) Prefixes are the names as appear in original SPARSKIT and arguments consist of the input matrix represented as three arrays in a sparse storage format, an array of integer parameters, an array of double-valued parameters, and the output matrix, also represented as three arrays. We use sparse matrix representation as a set of arrays, similar to the one in FORTRAN77, to avoid the potential need for copying between different representations. The following example shows how the ILUT preconditioner can be accessed in C++ via the `BasePreconditioner` port:

```
skit::components::ilut ilut = skit::components::ilut::_create();
ilut.create(a, ja, ia, lfil, droptol, alu, jlu, ju);
```

The same example will look like this when accessed using Python:

```
import skit.components.ilut
...
ilut = skit.componenents.ilut.ilut();
ilut.create(a, ja, ia, lfil, droptol, alu, jlu, ju);
```

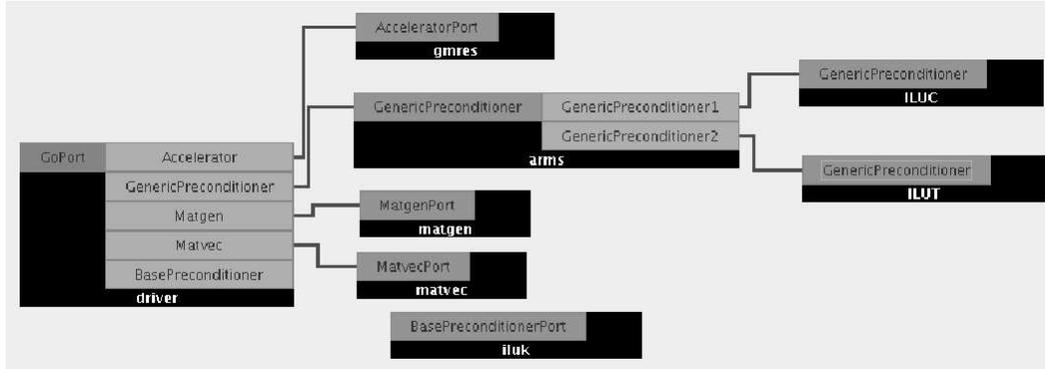


Figure 3. SPARSKIT Components in Ccafe

One feature of the CCA tools is that they use a standard type for arrays across all the supported languages. These arrays are provided by Babel and are called SIDL arrays and they allow for easy interoperability on arrays between the supported languages and a standard way of handling those arrays in each language. Babel also began to provide another mechanism for array access called *r*-arrays that allows for more traditional access in each supported language. For SPARSKIT CCA components, all the FORTRAN77 array indexes had to be shifted back by one. By default, the Fortran arrays in SPARSKIT started at one, but arrays in many other languages default to starting at zero, including the *r*-array definitions generated by Babel. We chose to shift the arrays in our components instead of requiring the users of our package to shift their arrays forward one place. These *r*-arrays also help us avoid the task of converting Fortran arrays to SIDL arrays.

4.2. Generic Preconditioner Interface for SPARSKIT Extensions

For interoperability of the Algebraic Recursive Multilevel Solver (ARMS), we have implemented the Generic Preconditioner CCA Port. ARMS makes use of complex C data structures that present a problem in argument passing situations under Babel/CCA. Babel does not have `struct` translation across its supported languages. To solve that issue, we created a group of methods, similar to the *Hypre* approach, that we term “set” methods. Each method takes two arguments: the name of the variable to be set and the value the user wishes to set that variable to. Each possible variable type that the component expects to be set has a method implemented. These methods are expected to check the name passed, to verify that the name is expected, and to correctly set the value of the local variable. To retrieve modified values, we created “get” methods.



Each method takes two arguments: the first is the name of the value to retrieve and the second is the variable to store the retrieved value in. Using this style of argument setting/passing, we can allow an arbitrary number of arguments to be set in any component that implements our interface so that we are not restricted by the number or types of arguments when integrating preconditioners from different languages or software packages. We call `GenericPreconditioner` an implementation of preconditioners that use the “set” and “get” methods.

Many of the data structures that ARMS uses can be stored inside the component and do not need to be exposed to the user, further alleviating usage complexity. A set of wrapper functions for each language will also be provided so that all a user has to do is to pass the necessary arguments to the wrapper function and the wrapper function does the actual calling of the set methods. In Java, for example, a wrapper function may look as follows:

```
set_Amat (ma, ja, nnz);

void set_Amat(double[][] ma, int[][] ja, int nnz) {
    arms.setDouble2DArray("Amat_ma", ma);
    arms.setInteger2DArray("Amat_ja", ja);
    arms.setInteger("Amat_nnz", nnz);
}
```

where `Amat` is the matrix whose arguments are to be set. This function would then make calls to `setDouble2DArray`, `setInteger2DArray`, and `setInteger`. Once that is done the user can call any of the main methods (e.g. `arms2`, `indset`) of the component. The potential overhead from having to copy-out values is averted because ARMS preconditioner is constructed once, before calling an accelerator, and remains unchanged. The only value that needs to be updated in the calling component is the vector to which the preconditioner is applied.

Because of its multi-level structure, ARMS may use different preconditioners in each level. Our goal is to make the preconditioning methods that ARMS uses into CCA ports such that they can be “plugged” into ARMS. The easiest way to accomplish this is to use the `GenericPreconditioner` port for these internal preconditioners instead of creating a new preconditioner port or trying to massage ARMS arguments. In doing so we assume that the user knows the type and the arguments to be set for the internal preconditioner to be connected to ARMS. This assumption is not too restrictive since typically a class of preconditioners has the same argument list for all its members. For example, ARMS may require two `GenericPreconditioners`, one for the intermediate levels and one for the last level. Both preconditioners are variations of incomplete ILU factorizations and have the same argument lists. For a different preconditioner type, a different wrapper function may have to be used.



Table I. Comparison of SPARSKIT implementations on 3D Laplacian equation

<i>nnz</i>	SKIT, <i>sec</i>	SKIT-CCA, <i>sec</i>	diff, %	<i>iters</i>
41440	0.027	0.035	29.6	32
50257	0.035	0.045	28.6	37
65241	0.044	0.057	29.5	39
95205	0.083	0.11	32.5	49
132660	0.14	0.18	28.6	57

5. Test Results

We have successfully created a CCA port of SPARSKIT components: accelerators, preconditioners, format conversion, and matrix-vector multiplication. Clients, using these functionalities, have also been written in C, C++, and Python to demonstrate the interoperability achieved by our SPARSKIT components. The CCA-Tools 0.5.8 “sumo” was used for the port development. For testing, we have considered the solution of the 3-dimensional Laplacian partial differential equation:

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

with Dirichlet boundary conditions, discretized with seven-point centered finite-difference scheme on $n_x \times n_y \times n_z$ grid. Column *nnz* in Table I refers to the number of non-zero elements in the matrix obtained for different values of n_x, n_y, n_z . We have solved these linear systems with ILUT as the preconditioner, and flexible GMRES [13] as the accelerator. In Table I, column *iters* shows the number of iterations required for convergence. The execution times are from an average of five runs on each problem solved with original SPARSKIT (column SKIT) and componentized SPARSKIT (column SKIT-CCA). The tests were performed on a 3.2GHz Intel Pentium4 Processor. The overhead incurred by the Component framework appears to remain almost constant as percentage of the execution time of original SPARSKIT with the problem size increase. With the exception of the fourth row, the overhead amount hovers between 28.6% and 29.6%. This can be seen in column *diff* in Table I.

We have also created a CCA port of ARMS as a `GenericPreconditioner` port that uses ILUT as preconditioner on the last level and have solved linear systems with matrices provided from the Harwell-Boeing sparse matrix collection [7]. Table II gives details about these linear systems. Flexible GMRES has been used as the accelerator to solve these systems with an artificial right-hand side and a zero initial guess. In Figure 4 we see only a small amount of overhead incurred on the largest matrix RAEFSKY2. On the two smallest matrices,



Table II. Harwell-Boeing Test Matrices

<i>Matrix Name</i>	<i>nnz</i>	<i>Source</i>
BCSSTK14	32,630	Stiffness Matrix - Roof of Omni Coliseum, Atlanta
BCSSTK16	147,631	Stiffness Matrix - Corp of Engineers Dam
BCSSTK26	16,129	Stiffness Matrix - Reactor Containment Floor
RAEFSKY1	294,276	Incompressible Flow in Pressure Driven Pipe, $T=5$
RAEFSKY2	294,276	Incompressible Flow in Pressure Driven Pipe, $T=25$
SHERMAN5	20,793	Fully Implicit Black Oil Simulator

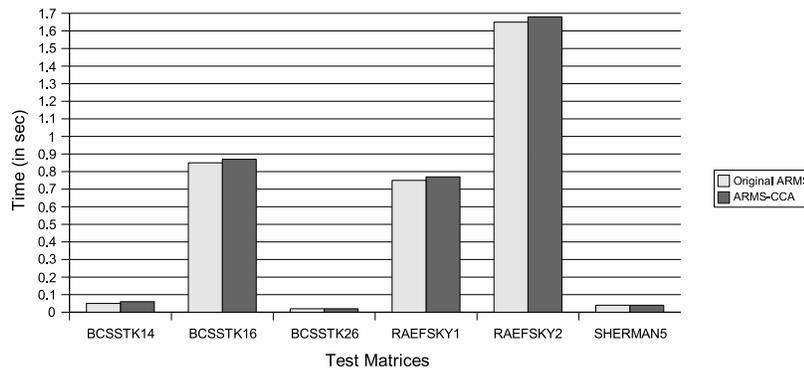


Figure 4. Comparison of evaluation times of ARMS implementations on six general sparse linear systems

BCSSTK26 and SHERMAN5, we observed no difference in the running time between CCA port implementation and the original implementation.

6. Conclusions

In this paper, we have described a component design and implementation of state-of-the-art iterative methods for solving sparse linear systems. Due to the proposed component structure and interfaces, we were able to assemble many diverse methods, both legacy and novel ones, into a single recursive framework defined mathematically by the Algebraic Recursive Multilevel Solver (ARMS). As the result, ARMS has become a part of the well-known iterative method collection SPARSKIT, which we have presented as a set of components



in the Common Component Architecture (CCA). Our tests have shown a constant amount of overhead when our component implementations were used, as opposed to the original FORTRAN77 codes, to solve a set of simple partial differential equation problems of increasing sizes. For general large sparse linear systems, only a negligible overhead has been observed when novel ARMS methods are employed. The developed SPARSKIT components may be easily interoperable with applications written in a variety of programming languages and having various sparse-matrix representation formats.

REFERENCES

1. S. Balay, K. Buschelman, W.D. Gropp, D. Kaushik, M.G. Knepley, L. McInnes, B. Smith, and H. Zhang. PETSc web page. <http://www.mcs.anl.gov/petsc>.
2. D. Bernholdt, W. Elwasif, J. Kohl, and T. Epperly. A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02)*, 2002.
3. The common component architecture forum. <http://www.cca-forum.org>.
4. E. Chow, A. Cleary, and R. Falgout. *Hypre* User's manual, version 1.6.0. Technical Report UCRL-MA-137155, Lawrence Livermore National Laboratory, Livermore, CA, 1998.
5. T. Dahlgren, T. Epperly, G. Kumfert, , and L. Leek. Babel User's guide. http://www.llnl.gov/CASC/components/docs/users_guide/index.html.
6. The equation solver interface standards forum. <http://z.ca.sandia.gov/esi/>.
7. Harwell-Boeing sparse matrix collection. <http://math.nist.gov/MatrixMarket/collections/hb.html>.
8. M. Heroux, D. Day, R. Hoekstra, R. Lehoucq, R. Tuminaro, and A. Williams. Trilinos project. Web Site, <http://www.cs.sandia.gov/mheroux>.
9. E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *In proceedings of the International conference on Computational Science*, volume 2073, pages 127–136, San Francisco, CA, May 2001. Springer-Verlag.
10. Netlib repository. <http://www.netlib.org/>.
11. Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations. <http://www-users.cs.umn.edu/~saad/software/SPARSKIT/paper.ps>.
12. Y. Saad. Finding exact and approximate block structures for ILU preconditioning. *SIAM Journal on Scientific Computing*, 24:1107–1123, 2003.
13. Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.
14. Y. Saad and B. Suchoamel. ARMS: an algebraic recursive multilevel solver for general sparse linear systems. *Numerical linear algebra with applications*, 9(5):359–378, July/August 2002.
15. R. Vuduc, J. Demmel, and K. Yelick. OSKI: Optimized sparse kernel interface. <http://bebop.cs.berkeley.edu/oski/>.