

Granularity of Component Interfaces for Iterative Linear Algebra

J. Jones L. Liu M. Sosonkina

Abstract—Large-scale systems of linear equations often use parallel iterative methods to approximate solutions. They also transform (precondition) the given system, making it easier to solve. SPARSKIT contains a wide array of preconditioning and accelerating techniques to choose from depending on system matrix properties. In our earlier work, SPARSKIT, originally written in FORTRAN77, was redesigned into a set of components based on the Common Component Architecture (CCA) standards, which enable usability and extensibility of the package with novel preconditioners.

Although a linear solver is often a computationally-intensive part of a high-performance application, separate components of a linear system solution may be relatively light-weight. The component architecture, however, has the potential to add overhead.

In this paper, we examine the computational overheads caused by various design choices for iterative linear solver components, ranging from low-level, such as components for matrix manipulation, to high-level, such as single component containing the entire linear solver.

Index Terms—Common Component Architecture, SPARSKIT, Linear Algebra

I. INTRODUCTION

To solve large-scale systems of linear equations, parallel iterative methods are often used. For difficult systems, they approximate the solution while transforming (preconditioning) the given system to make it easier to solve. In particular, the SPARSKIT [9] package contains a wide array of preconditioning and accelerating techniques, which the user may mix and match depending on system matrix properties. In our earlier work, SPARSKIT, originally written in FORTRAN77 and developed by Yousef Saad at the University of Minnesota, has been redesigned into a set of components based on the Common Component Architecture (CCA) [6] standards, which enable usability and extensibility of the package with novel solution methods. The CCA group is comprised of researchers from different national laboratories and academic institutions whose goal is to define a standard component architecture for high performance computing. Babel [8], developed at Lawrence Livermore National Laboratory, is used to enable language interoperability in the CCA Tools. Babel uses the Scientific Interface Definition Language (SIDL) to define component interfaces. SIDL is designed to address the needs of parallel scientific computing, specifically complex numbers, dynamic multi-dimensional arrays, and parallel communication directives.

Although a linear solver is often a major computationally-intensive task of a high-performance application, separate parts

of a linear system solution, due to their iterative nature, may be relatively light-weight. However, the component architecture overhead – if incurred at every iteration, for example, – may significantly undermine their efficiency. Therefore, it is desirable to minimize component overhead while preserving flexibility and extensibility of the SPARSKIT package.

This paper is organized as follows: Section II provides a discussion of a few projects incorporating sparse linear algebra component designs. In Section III, we outline three design choices that we have considered when creating sparse linear algebra components. We examine the computational overheads incurred while solving a system of partial differential equations (PDE) with iterative linear solver components (Section IV).

II. RELATED WORK

The Matrix Template Library [11] 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.

Argonne National Laboratory has developed PETSc [3], a suite of data structures and routines for scalable (parallel) solution of scientific applications modeled by partial differential equations. PETSc has implemented a broad range of parallel and sequential algorithms as well as support for linear and non-linear solvers, distributed arrays, and parallel matrix and vector operations. The current version of PETSc is not component-oriented, but there are plans for the next major version to deal with components.

Many packages exist that use a “high-level”-like design. One such project is the Terascale Optimal PDE Simulations (TOPS) solver interfaces [1]. TOPS is an integrated software infrastructure focused on developing, implementing, and supporting optimal or near optimal schemes for partial differential equation-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 the PETSc solvers was to use the TOPS SIDL interfaces and extend them.

This work was supported in part by the U.S. Department of Energy under Contract W-7405-ENG-82

Ames Laboratory, Iowa State University, Ames, IA 50011, {jonesj,lexinliu, masha}@scl.ameslab.gov

The Scalable Linear Solvers project at Lawrence Livermore National Laboratory developed `hypre` [7], 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.

One current effort at the University of Indiana is the development of the Linear Solver Interfaces [5]. The development of the LSI was started in May 2005 and there currently exists a draft of the SIDL interfaces released. The goal of LSI is to provide a high abstraction on top of the current linear solver libraries and allow applications to be loosely tied to the library they use to make switching the library easier. The current SIDL interfaces support iterative and direct solvers and have an implementation for PETSc, Trilinos, and SuperLU in C++ using the CCA.

Trilinos is an effort from Sandia National Laboratory to develop and implement robust parallel algorithms using a modern object-oriented design and to leverage the value of established numerical libraries like PETSc, Aztec, BLAS, and LAPACK [2]. It emphasizes abstract interfaces for maximum flexibility of component interchanging, and provides a full-features set of concrete classes that implement all abstract interfaces. Trilinos currently contains serial direct solvers, parallel direct solvers, non-linear solvers, complex linear solvers, Krylov linear solvers, and numerous other numerical and linear algebra capabilities. One of the newer features is support for the Python scripting language and allowing for users to develop their own matrix modules in Python that can be used by Trilinos solvers.

III. DESIGN CHOICES

A. Low-Level

The low-level design choice refers to operating on objects such as matrices directly. One low-level component that we have developed is the BLASSM component that is for use with our SPARSKIT Components. BLASSM stands for Basic Linear Algebra Subroutines with Sparse Matrices. It includes computations for two matrices and also computations for a matrix and a vector. The component form of BLASSM extends the SPARSKIT functionality by not requiring the user to know which variation of a function to call. This feature is accomplished through the overloading of functions in C++. Function overloading refers to allowing for the specification for more than one function of the same name in the same scope. This is a useful programming language feature available in object-oriented languages such as C++ and Java as well as non-object-oriented languages such as FORTRAN90. For example, the user only needs to call a generic function AMUB to perform a general sparse matrix-matrix multiplication and the function will then determine which form of AMUB needs to be called: When the input arguments to AMUB are two matrices in the Compressed Sparse Row (CSR) storage format, the implementation of AMUB for the CSR format is called. Such a design allows more flexibility in terms type-agnostic code. However, it also may have the disadvantage of

potentially incurring overhead from the function overloading in C++.

B. Medium-Level

A medium-level design choice represents a slightly higher abstraction than the low-level design choice. In particular, major steps of an iterative solution process are encapsulated as components. They may include accelerators, preconditioners, and matrix generation routines, whereas matrix-vector multiply is not a separate component. For a medium-level design, there is no function overloading, but rather a component implementation for each possible functionality choice. This can be more easily seen when considering two different preconditioners. In a low-level design, a call to the preconditioner would depend on some information to determine which overloaded function to call, but in a medium-level design the preconditioner component interface that is connected implements only one preconditioner and that will be the one that is called. In the SPARSKIT suite of components, each of these function calls – accelerator, preconditioner, and matrix generation – are made into components. Each component interface is designed such that it has a standard argument list to make the interchanging of different implementations of the component easier on the user. To illustrate the standard argument list, we present an example of a call to a preconditioner port:

```
...
prec=Svc.getPort("BasePreconditioner");
prec.create(a, ja, ia, lf, dtol, alu, jlu, ju);
...
```

where `a, ja, ia` represent the matrix in the CSR format, `lf` represents the maximum fill-in for each row of lower (L) and upper (U) triangular matrices, `dtol` is the tolerance for dropping small factors, and `alu, jlu, ju` represent a matrix stored in the Modified Sparse Row (MSR), see, example e.g. [10], format containing both the L and U factors.

C. High-Level

The high-level design choice is the most abstracted of the three design choices we present. A high-level design has all of the functions needed (accelerators, preconditioners, etc.) integrated into one package. One feature of this design is the use of “set” and “get” methods. A user can call a function such as `setPreconditioner("ilut")` to have the component internally set the preconditioner it will use to `ilut`. The component may also have “get” methods that can either return an object representing the currently set value or alternatively can return a list of options for a “set” method or to get the currently set option. An advantage to designing a component in this way is that it allows for easier switching between different libraries that implement the same interface. This can be seen in packages such as LSI that aim to provide easy access to a number of libraries through their interfaces. There are drawbacks to this approach including difficulty of integrating with libraries written in procedural languages such

<i>nmz</i>	SKIT	SKIT-CCA	diff %
776,760	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

TABLE I

COMPARISON OF SPARSKIT USING LOW-LEVEL BLASSM LIBRARY

as FORTRAN77 and C and potential overhead incurred from object passing.

IV. TEST RESULTS

We have successfully created low- and medium-level components in the BLASSM library and our previous work on a suite of SPARSKIT components, respectively. Work on implementing and benchmarking a high-level component interfaces is currently underway. All necessary Ports are obtained at the beginning of the test runs and are not released until the runs are finished.

For testing the interfaces, 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. These tests are run on an Intel XEON 2.2 GHz processor with 768MB RAM running Debian Sarge (3.1) and all software used is compiled with the Debian-shipped GCC 3.3.5.

Tables I and II show comparisons of original SPARSKIT (column SKIT) and component implementations (column SKIT-CCA) for low- and medium-level component designs, respectively. 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 SKIT-CCA overhead as percentage of the SKIT execution time.

The test results for BLASSM component timing (Table I) were obtained by calling the AMUB function with the matrices in the CSR format as input arguments and taking the average over ten calls, with standard deviations ranging from 5×10^{-5} to 7.26×10^{-4} . We observe that the average incurred overhead varies from 17.28% to 32.45% after zero overhead in the first two small problem sizes. It should be noted that only part of the overhead incurred is from the component framework (Ccafe [4]), while most of the overhead comes from the function overloading done in the BLASSM component implementation.

In Table II, the execution times are from an average of ten runs on each problem, with standard deviations ranging from 2.89×10^{-4} to 5.71×10^{-3} . We have solved a linear system with ILUT as the preconditioner, and flexible GMRES [10] as the accelerator. The test was performed with calls to methods provided by three different components - an accelerator, a preconditioner, and a matrix-vector multiplication component

<i>nmz</i>	iters	SKIT, sec	SKIT-CCA, sec	diff, %
76,760	59	0.0792	0.08	1
122,880	59	0.14	0.14	0
179,800	59	0.208	0.215	3.36
247,520	61	0.334	0.345	3.29
326,040	61	0.443	0.448	1.13
415,360	61	0.570	0.588	3.12
515,480	61	0.7185	0.730	1.6

TABLE II

COMPARISON OF MEDIUM-LEVEL SPARSKIT ON 3D LAPLACIAN EQUATION

<i>nmz</i>	Function	# Calls	Time (msec)	Norm %
38,880	lusol	46	20	100.0
	create	1	18	89.4
	amux	47	12	63.4
	apply	93	12	62.3
45,847	create	1	25	100.0
	lusol	46	20	80.7
	amux	47	13	52.8
	apply	93	13	50.9
53,600	lusol	46	25	100.0
	create	1	24	95.0
	amux	47	16	64.6
	apply	93	15	59.4
62,181	lusol	45	36	100.0
	apply	96	31	86.6
	create	1	27	75.6
	amux	51	24	66.0

TABLE III

PERFORMANCE ANALYSIS USING TAU

- in each step of the iterative solution process. Column *iters* shows the number of iterations required for convergence. These results only show a small overhead incurred by using the Ccaffeine component framework. The percent of 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) [12] Performance Component to do performance analysis on our medium-level component. The results can be seen in Table III. Column *nmz* refers to the number of non-zero elements and column *Function* refers to the function that was called. The column *Norm %* is a measure of the run-time of the function compared to the largest run-time from that iterative solution process. The performance analysis shows that we spend the majority of our runtime in the *lusol* function provided by the preconditioner component and the *create* function, also provided by the preconditioner component, also takes a significant amount of time in our tests.

V. CONCLUSIONS

In this paper we have described three different design choices to consider when creating sparse linear algebra components. Test results show that the medium-level maintains a relatively stable overhead in terms of percentage of the original runtime while the low-level implementation shows a large overhead. Based on these results, the medium-level

design choice appears to be a feasible choice when creating a sparse linear algebra component. In the future, we plan to have a high-level component implemented to compare against our current results to draw more concrete conclusions about the feasibility of using each design choice when creating sparse linear algebra components.

REFERENCES

- [1] "Terascale optimal pde simulation (TOPS)," <http://www-unix.mcs.anl.gov/scidac-tops/>.
- [2] "The trilinos project," <http://software.sandia.gov/trilinos>.
- [3] S. Balay, K. Buschelman, W. Gropp, D. Kaushik, M. . Knepley, L. McInnes, B. Smith, and H. Zhang., "PETSc web page," <http://www.mcs.anl.gov/petsc>.
- [4] 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.
- [5] R. Bramley and F. Liu, "Cca sparse linear solver interface," <http://www.cs.indiana.edu/~fangliu/lisi/intro.html>.
- [6] "The common component architecture forum," <http://www.cca-forum.org>.
- [7] E. Chow, A. Cleary, and R. Falgout, "Hypre User's manual, version 1.6.0," Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. UCRL-MA-137155, 1998.
- [8] T. Dahlgren, T. Epperly, G. Kumpf, , and L. Leek, "Babel User's guide," http://www.llnl.gov/CASC/components/docs/users_guide/index.html.
- [9] Y. Saad, "SPARSKIT: a basic tool kit for sparse matrix computations," <http://www-users.cs.umn.edu/~saad/software/SPARSKIT/paper.ps>.
- [10] —, *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.
- [11] J. G. Siek and A. Lumsdaine, "The matrix template library: A generic programming approach to high performance numerical linear algebra," in *ISCOPE*, 1998, pp. 59–70. [Online]. Available: citeseer.ist.psu.edu/article/siek98matrix.html
- [12] "Tuning and analysis utilities," <http://www.cs.uoregon.edu/research/tau/home.php>.