The Linear System Analyzer

R. Bramley, D. Gannon, T. Stuckey, J. Villacis, E. Akman,
J. Balasubramanian, F. Breg, S. Diwan, M. Govindaraju*
Department of Computer Science
Indiana University
Bloomington, IN

Abstract

The Linear System Analyzer (LSA) is a component-based problem-solving environment (PSE) for the manipulation and solution of large sparse linear systems of equations. Both the underlying component infrastructure and its application to the problem area of linear systems are introduced. Initial results indicate the utility of clearly separating the PSE infrastructure from the particular problem domain(s) which use it, leading to a resuable PSE builder. The component approach provides coarse-grain parallelism between components, allowing larger problems to be solved by simultaneously using the computational resources of multiple machines. Furthermore, the LSA system allows side-by-side comparisons of methods and rapid experimentation to develop practical solution strategies.

1 Introduction

An important new paradigm in software engineering has been the emergence of *distributed component architectures*. In this context, *components* are reusable building blocks for the construction of software systems [19, 15]. Modern systems for composing components include Microsoft's ActiveX/DCOM and Sun's JavaBeans and JavaStudio. Conceptually, a user has a palette of components from which to choose, and can *compose* or wire them together to create complete applications. Mechanisms are provided for defining new components which follow standards specifying *interfaces*, methods by which external codes can interact with the component. A useful model is that of a software integrated circuit; as a hardware IC has a specified set of pins that allow it to be connected with other IC's without requiring details of internal representations or methods, software IC's rely on published interfaces.

Components differ significantly from standard software pieces such as subroutines, libraries, or objects in at least three ways. First, component composition involves modifying and linking binaries, rather than source code which must then be re-compiled. Secondly, they interact on a peer-to-peer basis; no component is designated as a "main" program which

*Work supported in part by NSF grants CDA-9601632, ASC-9502292, and CCR-9527130

controls the others. More generally, component systems do not have a hierarchy of control, although typically a user can group several components to create a new, larger component. Finally, component systems are a natural environment for multi-language and heterogeneous computing systems, unlike object-oriented systems built using C++ or Smalltalk.

Although component architectures have revolutionized the desktop business application computing environment, they have made few inroads in problem-solving environments for computational science and engineering. As part of the PSEware project [17], we have built a scientific computing component system and implemented a problem-solving environment within it. The Linear System Analyzer (LSA) is an environment for examining and developing solution strategies for large-scale sparse linear systems of equations. It includes an extensible palette of many standard codes for manipulating and solving the linear systems, and a graphical user control system which presents a user with a "canvas" on which to compose components. Each component can be started on any networked computer. This article presents the purpose and design ideas behind the LSA and its architecture, which is designed to be reusable for problem domains beyond the solution of linear systems.

The LSA has revealed some important design issues for problem solving environments in scientific computing. Among these are the need for high-performance communications, the importance of providing access to both procedural and object-oriented programmers to add components, and an information subsystem which collects summary results from the distributed components and presents it to a user in a practical form. This information subsystem must also provide information about the state and performance of the PSE itself.

In the rest of this article, the LSA problem domain is described, some basic usage ideas are presented, its architecture is outlined, and examples of its use are given. Finally, future research using the LSA and the underlying PSE-building tools are described.

2 The LSA Problem Domain

Solving large sparse linear systems $Ax = b$ is an important computational subproblem in science and engineering which has generated significant research activity in the recent past. Much of this research has been incorporated in sophisticated libraries and codes which are freely available through Netlib and the National High-Performance Software Exchange [8], as well as individual researchers' Web sites. Software such as Sparskit [18] provides tools for manipulating sparse systems and converting them between standard data structure representations.

2.1 Strategies for Sparse Linear System

This profusion of software has brought its own problem; an applications user needs to connect together packages and navigate a combinatorially large parameter space to form an effective solution strategy. No current mathematical theory provides a practical guide to choosing a solution strategy, and even expert numerical linear algebraists require experimentation and testing to develop one. Incorporating a sophisticated solver within an applications code can require weeks of work - at the end of which the user may find the strategy fails on his linear systems.

To handle this, a common method for developing a solution strategy is to "extract" the linear system and write it to a file in a standard format. Then the linear algebraist draws upon a collection of programs for applying transformations on the linear system, which read it in, perform the manipulations, and then write it out to another file. Other programs apply various solution methods, reading in the linear system and writing out a summary of the results of the computation such as error estimates, memory usage, and time required. Control parameters for these transformation and solver programs are typically from input files, command line arguments, or a GUI. The linear algebraist tries several combinations of these programs, comparing their results. If a program runs only on a certain machine, the user can either try to port it, or transfer a file with the linear system to the remote machine and transfer the results back. Applications now routinely generate and solve linear systems with $\mathcal{O}(10^5)$ unknowns and $\mathcal{O}(10^6 - 10^7)$ stored elements, requiring hundreds of Mbytes of internal memory to represent. Unless the linear algebraist is lucky and immediately finds a good combination of software for the problem, most of the time gets spent in file I/O and transfer.

For the applications user who tries to manage all of this without expert help the situation is worse; much of the time and effort is spent in recompiling code, trying to understand adjustable parameters in the solution methods, and trying to form a coherent picture of results from a variety of output from the codes. The LSA addresses these problems using a component architecture approach. Before describing that framework, first we list the actual modules currently in the LSA.

2.2 Computational Components in the LSA

The general categories in which the linear system components have been grouped are

- I/O components for getting systems into and solutions out of the LSA
- Filter components for manipulating the systems with re-orderings, scalings, or dropping of entries based on their relative sizes
- Solver components for actually solving the systems
- Information components for providing some analysis of the systems.

One goal of the LSA project has been to provide rapid encapsulation of existing codes, including both object-oriented and procedural languages. The components currently available are:

- NewSystem (I/O) uses some local and some Sparskit routines to read a system in a Harwell-Boeing [7] or Matrix Market [16] formatted file into the LSA. Later this will be extended to accept input from a running process.
- ExtractVector (I/O) outputs a vector to a file or another component. It is used to retrieve the solution vector, or some auxiliary information such as permutation or scaling vectors.
- BasicInfo (Information) uses primarily Sparskit routines to provide analysis of the system for symmetry, density, diagonal dominance, etc. It also creates a GIF image of the sparsity (nonzero) structure of the coefficient matrix.

- Reorder (Filter) applies some standard re-orderings: reverse Cuthill-McKee, nested dissection, and minimum degree. These are modifications of Sparspak [12] routines.
- Scale (Filter) applies row and/or column scaling, and equilibration.
- Squeeze (Filter) removes entries based on their relative absolute value.
- Banded (Solver) is based on Linpack [6] routines, and converts the system to banded data structure and then solves the system.
- Dense (Solver) uses Lapack [1] routines to solve the system after first converting the system to a dense 2D array data structure.
- SuperLU (Solver) is a sparse direct solver based on supernodal methods, and is a code developed at UC Berkeley [5].
- SPLIB (Solver) is a package of preconditioned iterative solvers developed at Indiana University [3]. SPLIB includes thirteen iterative methods and seven preconditioners.

These are not an exhaustive compendium of numerical software available for large sparse linear systems, but they span enough of the common operations to demonstrate the application of the LSA.

A key point is that many of the solvers have a large number of parameters to experiment with and choose from. SuperLU allows setting the Markovitz pivoting parameter, choosing the panel size for supernodes, and other parameters which can have large impact on solution accuracy, memory usage, and performance. SPLIB has a combinatorially large parameter space, varying settings for the preconditioners, stopping tests, and choosing maximum Krylov subspace sizes. In addition to the PSE component infrastructure software goals of the LSA project, we also have targeted providing a practical tool for users to quickly and easily explore the large parameter space associated with sparse linear systems.

3 LSA Usage

Using the LSA involves starting the LSA Manager and a user control system which currently is a graphical user interface (GUI). The GUI is used in a fashion similar to the Iris Explorer [13] or Paradise [2] systems. It presents a list of machines and selecting one causes a database query which returns the palette of components available on that machine. A user clicks on a button in the palette to choose that component, and the LSA system starts it on the selected machine. Typically the first component selected would be "NewSystem", which reads in a sparse matrix from a file. The user can then select further machines and components in the same way, starting up the corresponding processes.

Each component has a similar icon on the canvas, as shown in Figure 4. At the top is a status bar, and below that the name of the component (its type and an identifier number.) Below that are two side-by-side buttons. Selecting the left one brings up a sub-GUI, which allows input of any "control parameters". For example, with SPLIB the sub-GUI fields include choice of iterative method, preconditioner, stopping tests, etc. The right-hand "View Results" button retrieves summary results of the component's execution by accessing an information subsystem described later. At the bottom of the icon is the name of the component's host machine.

When a component is first started, it has an additional "input" button that specifies where its input (if any) should come from. Pressing that button brings up a list of other active components whose output ports are compatible with the component's input port. Once the components are wired together, the "input" button disappears, and an arrowed line appears on the canvas indicating the connection. One component can send its output system to several other components, but each component gets its input from only one. This forest data structure for the component network is necessary in the LSA for retrieving the final solution vector since any reorderings and scalings must be undone in the reverse order in which they were applied. However, the underlying infrastructure allows constructing more general graph networks.

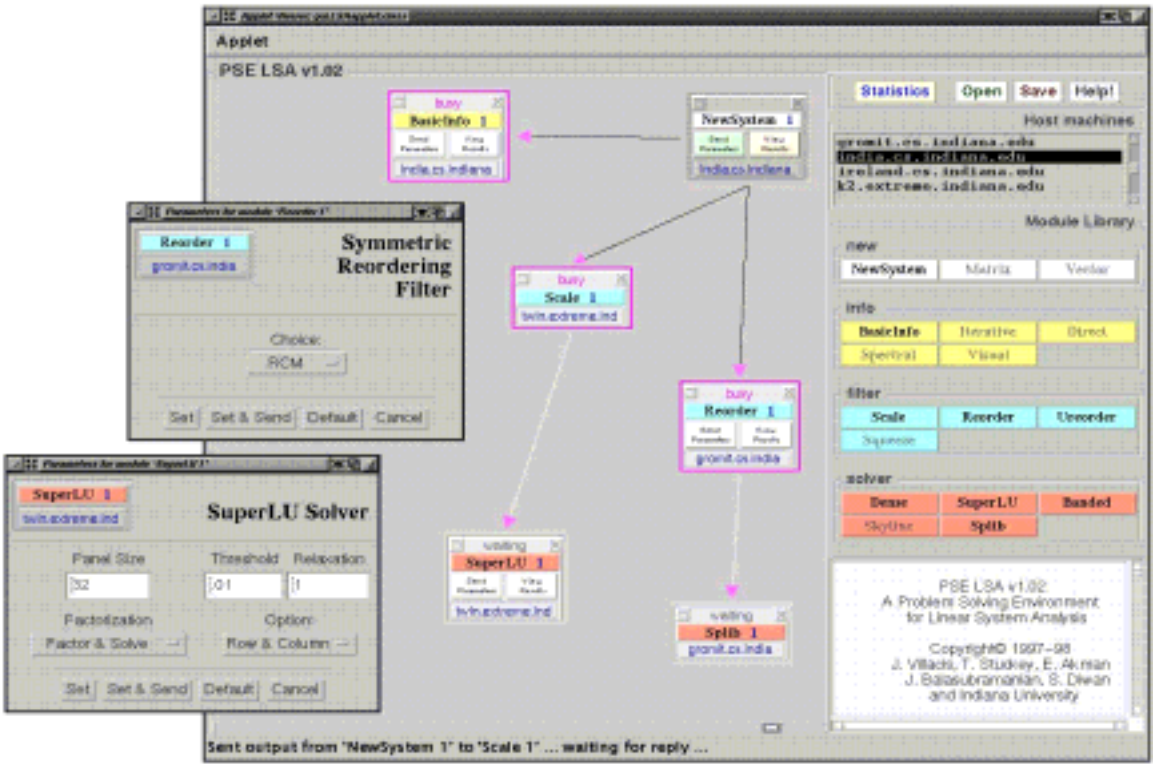

Figure 1: Snapshot of a Sample LSA Session

Figure 1 shows a sample LSA session. A NewSystem component feeds a system to a BasicInfo, Scale, and Reorder component - all three of which are running on different machines. The scaled system is sent to SuperLU, and the reordered one is sent to SPLIB. In this snapshot, the data flows in a tree-like fashion to each connected component starting from the NewSystem component. Components which are processing their input are marked as "busy", and those awaiting input are marked as "waiting". Two component subGUIs are shown overlaying the canvas, indicating each component's current control parameter settings.

With this framework, a user can test several solution strategies on a single system, comparing computational methods in a longitudinal study. By creating more NewSystem components, the same solution strategy can be tested on several systems for a latitudinal study.

4 LSA Architecture

Four modular parts comprise the LSA architecture: the user control, the manager, the communication subsystem, and the information subsystem. Figure 2 illustrates how these subsystems comprise the top-level LSA architecture.

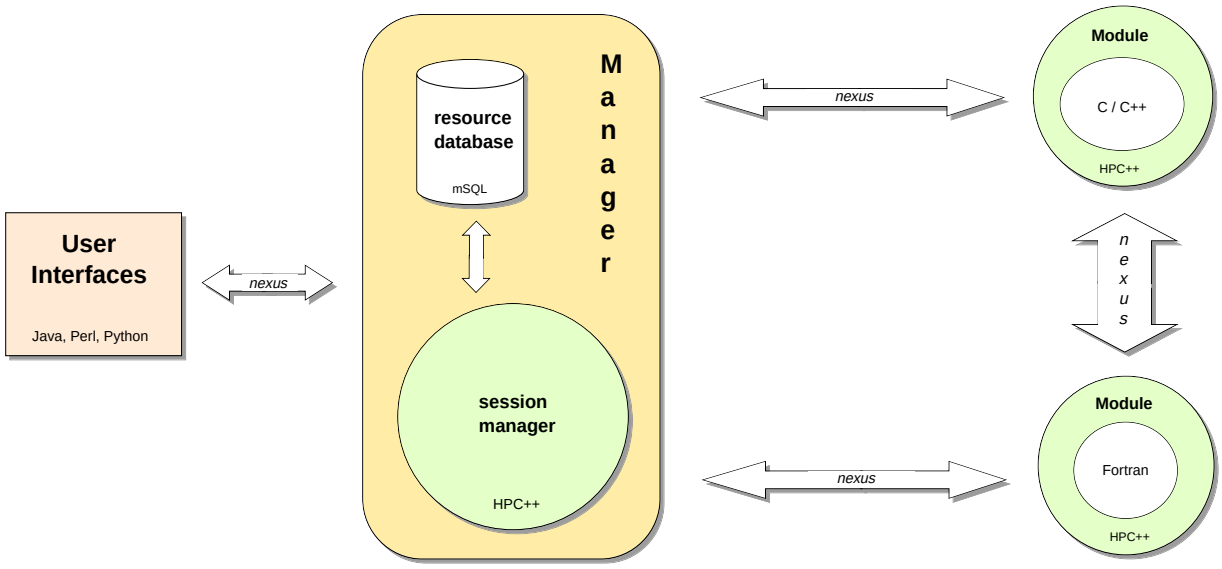

Figure 2: Top-level LSA Architecture

4.1 User Control

The current user interface and control is a Java GUI. Component icons all have the same set of buttons, which are intended to provide the basic functionality associated with computational modules in scientific computing: setting internal control parameters, viewing summary results, and setting input sources. Arrowed lines indicate the flow of data between connected component icons. Clicking on an icon's "control parameters" button brings up a subGUI tailored to the components parameter space. During session activity, each component icon may change its appearance based on its internal process state. The "Save" button saves the connectivity structure of the components, allowing a user to later retrieve a complicated network using the "Open" button. Note that the save operation does not save the internal state of the components, because doing so may involve storing Gbytes of data on remote machines. This design separates the base functions of the problem solving environment from the particular features of the application domain, thereby enhancing reusability of the PSE infrastructure.

4.2 LSA Manager

The presence of a manager program is contrary to the peer-to-peer character of component systems. However, the LSA manager is minimal and serves only two roles: collaboration

control and resource management. In its first role, the LSA manager has been designed to allow collaborative construction of a component network, so that multiple user control systems can interact with a single LSA session. The second role is to assign unique identifiers, as well as maintain and access the database of available machines and components. Having a minimal modular manager will facilitate its future replacement by the more sophisticated resource management system in Globus [9].

4.3 Communication Subsystem

The basic communication runtime system in the LSA is Nexus [10] from Argonne National Laboratory. This cross-platform system is designed for parallel applications and wide-area distributed computing. Nexus uses multi-threading, can take advantage of multiple communications protocols, and provides a bridge between Java and C++. This bridge is needed because the LSA is a mixed language system. The user control system is in Java, but each computational component is provided with a light-weight wrapper in HPC++ [11], which interacts with a generic control module.

The wrapper is implemented as a C++ class that inherits its interface from two abstract base classes. These base classes define the interaction between the generic module and the computational code. The amount of programming required for the wrapper can vary depending on the complexity of the computational component. For LAPACK [1] functions, the wrapper is simple and primarily converts error code values to text messages. However, a library like SPLIB [3] requires an extensive wrapper that must interact with the local file system. In general, the complexity of the wrapper code depends on whether the original library code is reentrant. Tasks usually performed in the wrapper include interaction with the computational code, parameter settings, memory allocation/management, and receiving and interpreting computational error flags.

The generic control module is identical for all components, and provides interactions with the PSE system: firing logic, communications control, errors signaling, etc. This module consists of a main loop and several functions that the manager can invoke remotely. The primary purpose of the module is to maintain process state information and to provide a standard set of functions that can be invoked remotely. The computational code is required to implement two interfaces. The first interface is a generic computational component interface. The second is an LSA-specific interface. The LSA interface customizes the PSE to the LSA problem domain, while the first interface is required of all computational components regardless of problem domain. Figure 3 illustrates the relationships between the wrapper interface and the generic component control.

HPC++ is used because it allows a natural interface to procedural and object-oriented languages for high-performance applications. Furthermore, HPC++ has global pointers (a generalization of the C pointer type), which can be used to perform remote method invocation (RMI). For example, when component A needs to send its sparse system to component B, A remotely invokes a RecvSystem() function on B, which then gets the system from A. Some details of how this is handled are in [4].

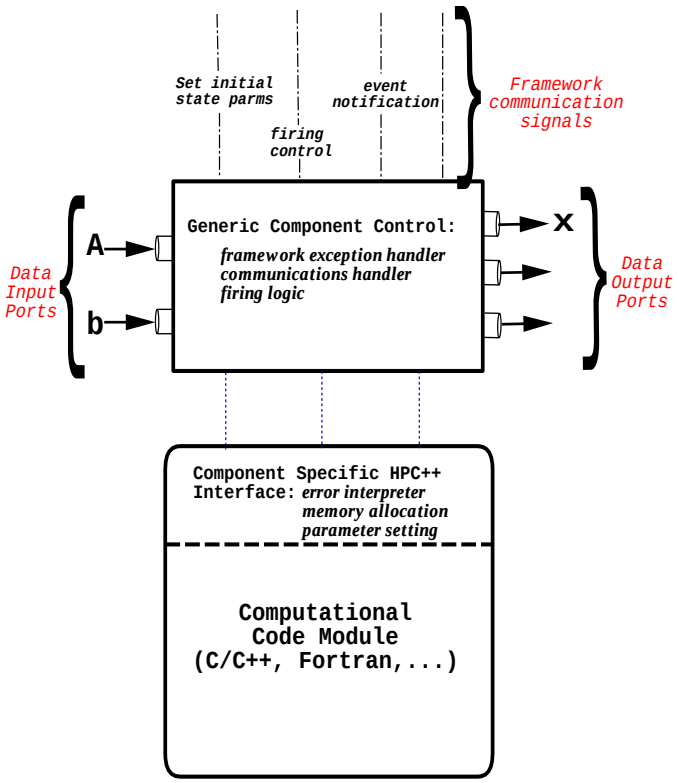

Figure 3: Diagram of an LSA Component

Figure 4: Sample Component Icon

4.4 Information Subsystem

Providing information about the solution process to the user is critical for any problem-solving environment. However, the component architecture idea works against the integrated approaches typically used when building PSEs. We have handled this by limiting the flow of unsolicited information from the components to the user. For every request sent to a component, a summary of the results is returned along with performance metrics for that event. The summary results may contain one or more elements. Each element consists of a one line text description of a specific result, a flag indicating the category of the result (success, warning, failure, etc), and the location of more detailed information if available. The user interface displays the text results in the status window and alerts the user if failure results are received. A web browser is used to view and navigate through the complete set of results generated by the LSA.

Each component creates a directory on its local machine to store results and data files. Since the component is built around legacy code that may create new files when it executes, a separate directory is created for each execution. All the information in this directory structure is tied together by an HTML document created by the component as it runs. This *process log* HTML document contains all the summary results information, links to the detailed information, and links to the index documents of other components connected to the current component. The links between index documents go in both directions so that the browser can traverse up and down the component tree structure. An example of this is

shown in Figure 5, which shows the process log page for the Reorder component in Figure 1.

Figure 5: Sample Component HTML Page

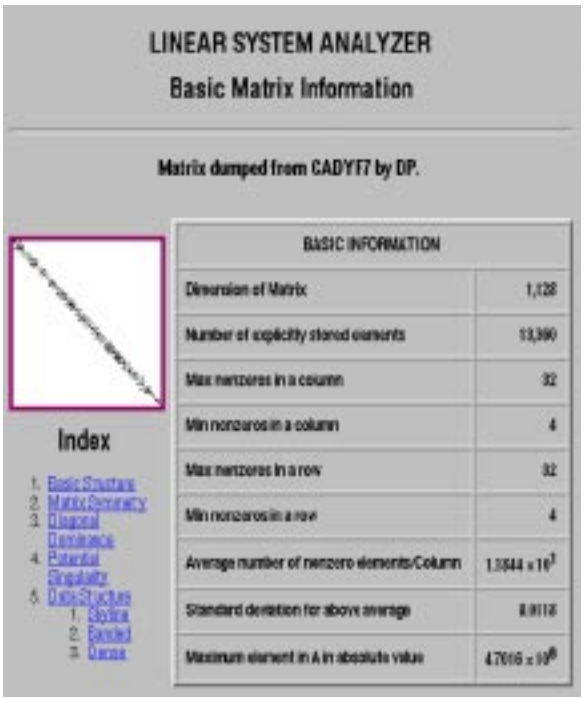

Figure 6: Sample Results HTML Page for a BasicInfo Component

The page shows the results of two linear systems fed consecutively through the component. The blocked tables show external information garnered from the component's interfaces and performance: the operation performed (which comes from its subGUI), vectors generated, and timings. The table is followed by links to components to which the resulting linear system is sent (SPLIB in this case), and preceded by a link to the component from which it received its input (NewSystem in this example).

Each component is free to write other summary results out to files, which are stored on the component host machine. These files can also be linked in with the Process Log pages, as illustrated in Figure 6 for some output from a BasicInfo component. On the left is a small GIF image of the nonzero structure of the coefficient matrix, followed by several tables giving detailed analysis of the system (only the top of the HTML page is shown).

An important part of this system is that all the data files are connected together by HTML links. When the user clicks on the "View Results" button on the component's icon, the browser reads the corresponding HTML index document for that component. The user can then follow links in the HTML document to retrieve results without regard to file system or network layout. Furthermore, this information system is independent of the application domain and is reusable for any kind of component added to the framework.

Later runs create new directories, so a postmortem analysis can be performed on results from earlier runs of the PSE. An environment variable set by the user specifies the root directory for the results files to use on each machine, so these can be archived - or assigned to a temporary workspace to be automatically cleaned up by the operating system.

The future of the information subsystem is to incorporate Java applet based data analysis tools into the web pages, and to add search capabilities into the index document pages. The goal is to have the components produce raw data and allow the user to view this data according their personal preferences. For example, instead of a component producing a 2D graph, it would produce an *n x 2* table of data. A Java applet would read this data and produce a graph and allow interactive operations like zoom in/out, scale adjustment, etc.

5 Results and Future Work

Although the LSA is still under development, the component architecture approach to building a PSE has proven its utility. First, the separation of *generic* PSE functions such as sending data between components and notifying the user interface of events from *problem domain specific* functions has allowed a consistent and comprehensive view of a component network independent of the actual computations done within each component. This does not inhibit a user from using components that create problem-specific information, and the structure used for generic PSE functions has provided hooks on which to place the domain-specific information. This provides a a natural conceptual model of the overall solution process for the application.

Although the component architecture itself is written in an object-oriented fashion with Java and HPC++, it has allowed us to integrate both procedural and object-oriented computations in a single framework. This allows users to develop and work with components in whatever programming style is most natural and productive for them.

The system provides immediate large-grain parallelism between components running on different machines. This parallelism has also allowed us to work on significantly larger problems, by having simultaneous components on different machines. In addition multi-threading has provided parallelism within the component infrastructure itself, allowing overlapping of communications and computations for better utilization of overall system resources, both network and computational.

Within the problem domain of large sparse linear systems, the LSA has allowed dynamic side-by-side comparisons of methods. For example, to explore the effects of SuperLU's panel size on a given linear system, we can start a dozen SuperLU modules on different machines, each receiving input from a single Reorder component and each with a different panel size parameter setting. After comparing this the reordering method can be changed, with the results automatically sent to the same dozen SuperLU components. This greatly improves the speed of such parameter studies.

In addition to a latitudinal study of methods, the LSA allows a longitudinal study by feeding a stream of linear systems to a fixed component network, and examining for which ones the solution strategy succeeds or fails. Both kinds of study are greatly enhanced by the information subsystem, which also provides primitive archiving capabilities.

Several research directions are extending the LSA. Among these are the development of methods for components implementing parallel algorithms - for this a critical problem is parallel communications between components [14]. The LSA system currently has methods for gathering statistics and event logging for analysis and evaluation of the component system itself.

The availability of event logging allows the addition of reasoning tools, which can learn from earlier runs of the LSA to help guide a user for future problems. Because we wish to keep a separation between the generic utilities of the component architecture and any problem-specific ones, case-based reasoning methods seem the most likely ones to use for the generic infrastructure, with provisions for users to add expert systems for particular components.

Other future work targets collaborative versions of the LSA, allowing an application scientist to work with a numerical linear algebraist on a single LSA session. Also, other user control mechanisms, particularly scripting language interfaces such as Perl or Tcl, will be added.

References

[1] E. Anderson et al. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1 edition, 1992.

[2] Scientific Computing Associates. Paradise Product Information, visited 7 March 98. http://www.sca.com/paradise.html.

[3] Randall Bramley and Xiaoge Wang. SPLIB: A library of iterative methods for sparse linear system. Technical report, Indiana University–Bloomington, Bloomington, IN 47405, 1995.

[4] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI performance and object model interoperability: Experiments with Java/HPC++. *Concurrency and Experience*, 1998. Presented at 1998 ACM Workshop on Java for High-Performance Network Computing.

[5] James Demmel, Stanley Eisenstat, John Gilbert, Xiaoye Li, and Joseph Liu. A supernodal approach to sparse partial pivoting. Technical Report 883, University of California–Berkeley, 1995. Submitted to SIAM J. Sci. Comp., and available at ftp.cs.berkeley.edu.

[6] J. Dongarra, C. Moler, R. Bunch, and G.W. Stewart. *Linpack User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1 edition, 1979.

[7] Iain S. Duff, Roger G. Grimes, and John G. Lewis. "Users Guide for the Harwell-Boeing Sparse Matrix Collection (Release I)". Technical report, Cedex and Boeing Computer Services, October 1992.

[8] Center for Research on Parallel Computing. Nhse: National hpcc software exchange, visited 7 March 1998. http://www.nhse.org/.

[9] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 1997. To appear.

[10] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *J. Parallel and Distributed Computing*, 37:70–82, 1996.

[11] D. Gannon, P. Beckman, E. Johnson, and T. Green. *Compilation Issues on Distributed Memory Systems*, chapter 3 HPC++ and the HPC++Lib Toolkit. Springer-Verlag, 1997.

[12] A. George and W-H Liu. *The Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Engelwood Cliffs, NJ, 1981.

[13] Numerical Algorithms Group. IRIS Explorer, visited 8-20-97. http://www.nag.co.uk/Welcome_IEC.html.

[14] Kate Keahey and Dennis Gannon. PARDIS: A Parallel Approach to CORBA. In *6th IEEE International Symposium on High Performance Distributed Computation*, August 1997.

[15] David Krieger and Richard M. Adler. The emergence of distributed component platforms. *IEEE Computer*, 31(3):43–53, March 1998.

[16] National Institute of Standards and Technology. Matrix market, visited 8 March 1998. http://math.nist.gov/MatrixMarket/.

[17] PSEware Research Group. Pseware home page, 1 October 1997. http://www.extreme.indiana.edu/pseware/.

[18] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations. Technical report, Center for Supercomputing Research and Development, University of Illinois, Urbana, Illinois, 1990.

[19] A. Thomas. A comparison of component models. *Distributed Object Computing*, pages 55–57, July 1997.