Component Architectures for Distributed Scientific Problem Solving

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

Abstract

Component based technologies will play an increasingly significant role in the design and implementation of large scale distributed software systems during the next few years. The advantages of this model of programming include rapid prototyping of complex, distributed applications and a framework that provides a natural way to incorporate our existing scientific software code base as components of larger problem solutions. In this paper, we survey the design issues and theory of component based software. To illustrate the ideas we also present a prototype component based problem-solving environment (PSE) called the Linear System Analyzer (LSA) which we have built for the manipulation and solution of large sparse linear systems of equations.

1 Introduction

Suppose that a scientist working in fluid dynamics finds that going from 2D to 3D models exceeds the storage capacity on the workstation. Tracking down the memory high-water mark shows it occurs in the linear solver, which is based on a "skyline" storage format. The scientist decides to use a sparse matrix storage scheme and iterative solvers instead. Knowing better than to re-invent the wheel, she checks NIST's Guide to Mathematical Software (http://gams.nist.gov/) and ORNL's Netlib (http://www.netlib.org/), and almost gives up hope.

The problem faced by this programmer is not a lack of software, but too much: a veritable alphabet soup of solvers and packages is available (SMMD, QMRPACK, SPLIB, ITPACK, SPARSKIT, ...) Furthermore, colleagues battle-scarred from dealing with sparse iterative solvers have warned: iterative solvers don't work without a good preconditioner; reordering the equations and unknowns can have drastic effects on the solution robustness and efficiency; scaling the linear systems is critically important. The veterans all agree that finding the right combination of solver, preconditioner, scaling, and reordering is an art form and requires experimentation with problems from the new application area. The veterans all disagree about which solution strategy is most likely to work for these problems, and the

only one willing to actively help on the problem is located at a national lab halfway across the continent.

The complexity of the software selection problem is only half the battle. Perhaps the problem size exceeds the workstation memory even with iterative methods. Alternatively, our researcher may discover that the best possible solution may involve a code that only runs on one particular remote server. Or, combining these two scenarios, the only feasible solvers are specialized parallel algorithms on a remote distributed memory MPP system.

In another example, a scientist wants to look at information being obtained in "real time" from a networked instrument such as a telescope. Because of the length of the observation and the size of the data set, it may be unrealistic to save the data for later analysis. However, a compute server at a third location is available that can analyze the information as it comes off the instrument. The problem is then routing the data from the telescope to the remote analysis program and then routing the result back to the scientist's desktop.

In a third example, a large data base in San Diego is being mined to extract biochemical data that must be compared to a simulation that is being run on a supercomputer in Urbana. Our scientist is interested in matching a set of important patterns which will be fed to an tele-immersive visualization session taking place between two researchers located in different laboratories.

In each of the cases above, the scientific programmer described above is building applications by composing existing software components which exploit specialized computing and algorithmic resources. This programming model is actually at the heart of a very powerful software construction paradigm that is having a substantial impact on the software industry. In this paper we examine the concepts behind the component technology programming paradigm and illustrate them with a prototype system for solving the linear algebra algorithm selection problem.

We begin with a more detailed look at the user level programming model for the prototype system (called the Linear System Analyzer (LSA)) which illustrates the component composition model of distributed computing. Section three of this article discusses the general characteristics of components and their composition architectures, and surveys some of the existing commercial and experimental implementations.

Component based systems actually involve two levels of programming. At the user level, programming consists of selecting and connecting components together into a *component network*. This is often accomplished with a graphical user interface that is modeled on circuit design and layout metaphors. Scripting languages such as Python and Perl provide another component integration technology that is frequently used. The next level of programming is encountered when the user wishes to build a new component. In section four we describe the process of building components using a C++ based distributed and parallel computing library known as HPC++ [9] and Fortran.

2 LSA: A System for Composing Distributed Linear Algebra Solvers

Component architecture concepts can be illustrated by a concrete example of a system we have developed to help the hypothetical scientist who needs to develop a solution strategy for her systems of equations. In general solving large, sparse, nonsymmetric linear systems $Ax = b$ is an increasingly common computational subproblem in CS&E, and much research activity in the past thirty years has targeted solving them. Several sophisticated techniques and codes are now freely available for matrix manipulations and solution methods. Furthermore, many of those take a standard data structure for sparse linear systems, allowing them to be represented uniformly. Applications now routinely generate and solve linear systems which require tens or hundreds of Mbytes of internal memory to represent. The daunting task for a user, however, is stringing together packages and navigating a combinatorially large parameter space to form an effective solution strategy. Some of those packages are difficult to port, or use system-specific features which makes them more efficient on their "native" computer. This makes the problem ripe for a distributed component architecture solution.

2.1 Overview of Linear Systems

A brief overview of the options available in modern numerical linear systems will aid in understanding the requirements. Although in theory "direct" methods (versions of Gaussian elimination) work for any nonsingular system, for many problems of practical interest they need too much additional storage. Storage requirements can be reduced some by reordering methods which permute the rows and/or columns of the matrix A. Since pivoting in Gaussian elimination implicitly permutes the rows of the matrix also, the benefits of reordering can be undone during the factorization. Strategies such as Markovitz pivoting balance pivoting for numerical stability against retaining a preliminary lower storage ordering by introducing a parameter varying from 0 (no pivoting) to 1 (complete pivoting). Modern sparse direct solvers also are parameterized to allow efficient block linear algebraic operations for high computation rates. All of these techniques (reordering, Markovitz pivoting, blocking parameterization) can lead to large differences in the numerical quality of solution, amount of additional storage required, and computational efficiency achieved.

Iterative methods, the other major category of linear solver, typically need only a few n-vectors of additional storage, but for the majority of systems they don't work: convergence is slow or nonexistent for nontrivial applications problems. Instead, the iterative method is applied to a preconditioned system $M^{-1}Ax = M^{-1}b$, where M is a cheaply computed approximation to A. Most of the important research for applied iterative solvers is in choosing a preconditioner. A typical one for general problems is based on incomplete factorization: Gaussian elimination is performed on A, but entries outside a certain sparsity pattern or below a cut-off numerical value are simply discarded during the factorization. The approximate LU factors then define the preconditioner M. Iterative solvers also present a large parameter space to navigate: solver method (BiCG, CGS, Bi-CGstabilized, GMRES, GCR, OrthoMin, etc.), preconditioner method (ILU, MILU, RILU, ILUT, SSOR, etc.), target sparsity patterns for the preconditioners, cut-offs for numerical dropping of small terms, stopping tests

and tolerances, and so forth. Iterative methods can also use the same kind of preliminary massaging as direct methods: reordering of rows and/or columns, scaling of rows and/or columns, squeezing out small values in the data structure, and combinations of all of those.

2.2 Linear System Solution Strategies

For modern nonsymmetric linear systems of equations no current mathematical theory provides a practical guide to choosing a solution strategy, so developing one requires experimentation and testing. The commonly used methodology is to "extract" the linear system and write it to a file in a standard format. The user has 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 purported solution vector and a summary of the results of the computation (error estimates, memory usage, and time required are common quantities.) Control parameters for these transformation and solver programs are typically from an input file or command line arguments. The user mixes and matches these programs, typically trying several combinations and 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.

2.3 Linear System Analyzer

The Linear System Analyzer (LSA) encapsulates these programs as distributed components, allowing a user to quickly and easily wire together solution strategies as *component networks*. Existing public domain codes on the Internet comprise the computational engines; the current list [1] of computational components is in Table 1.

A Java-based GUI working together with an HPC++ manager incorporate the *container framework* mechanisms for a user to interact with and control the components. When the LSA Manager and a LSA Client window are started, the GUI in Figure 1 appears.

On the left is the workspace on which graphs of wired-together components will appear as they are constructed. On the right is a control window containing a list of networked

---

[1] Network sources for the computational part of the LSA components are

- SPARSKIT: http://www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html
- Matrix Market: http://math.nist.gov/MatrixMarket/
- SPARSPAK: source code from *The Computer Solution of Large Sparse Positive Definite Systems* (Prentice-Hall, Englewood Cliffs NJ), by Alan George and Joseph Liu.
- LINPACK: http://www.netlib.org/linpack/index.html
- LAPACK: http://www.netlib.org/lapack/index.html
- Berkeley: http://HTTP.CS.Berkeley.EDU/ xiaoye/
- Indiana: ftp://ftp.cs.indiana.edu/pub/bramley/
- U Texas: http://www.netlib.org/itpack/

Freely distributed software for numerical linear algebra genuinely is an embarrasment of riches. Jack Dongarra has recently compiled an incomplete listing of 31 packages at http://www.netlib.org/utk/people/JackDongarra/la-sw.html. As the URL stems above indicate, the first and best place to start looking for mathematical software is Netlib, maintained by Oak Ridge National Lab and the University of Tennessee.

Name and *Category*	Status	Source	Purpose
New System *I/O*	Current	SPARSKIT, Matrix Market	Input a system into component framework
ExtractVector *I/O*	Current	Local	Output a vector to environment outside of the framework
BasicInfo *Expert Sys*	Current	SPARSKIT	Analysis of linear system: symmetry, density, diagonal dominance, structural info, etc.
DirectInfo *Expert Sys*	Current	Local	Specialized direct methods advice: storage required, pivot likelihood, performance prediction
IterativeInfo *Expert Sys*	Development	Local	Specialized iterative advice: spectral estimates, preconditioner selection, performance prediction
Reorder *Filter*	Current	SPARSPAK	Min degree, nested dissection, rev Cuthill-McKee orderings
Ureorder *Filter*	Current	Local	Max transversal, max diagonal, dense block diag orderings
Scale *Filter*	Current	Local	Row and column scaling, equilibration
Squeeze *Filter*	Development	Local	Drop entries below a tolerance in absolute or relative size
Banded *Solver*	Current	LINPACK	Solve a system with banded coefficient matrix
Dense *Solver*	Current	LAPACK	Solve a dense linear system with HPC blocking techniques
SuperLU *Solver*	Current	Berkeley	Supernodal direct method for sparse coefficient matrix systems
SPLIB *Solver*	Current	Indiana	Iterative solver and preconditioner package

Table 1: LSA Components

machines and, for a given selected machine, a group of buttons representing the component modules available on that machine.

As an example scenario, a user might take the following steps.

1. First select the “NewSystem” button to start a process on the remote machine for reading in a linear system that an application scientist has made available, and which is causing troubles in her codes.
2. After the system has been correctly read in, a “BasicInfo” module is started on a local machine, and its input port is connected to the output port of the “NewSystem” module. The “BasicInfo” module applies some basic and easily-found analysis of the linear system; for example, its bandwidth, the number of nonzeros on the main diagonal, the presence of any rows or columns of all zeros, etc. The “BasicInfo” module also returns an HTML document summarizing the results, and then recommends trying a bandwidth-reduction reordering method.
3. Select a “Reorder” module, enter in its subGUI (a popup dialog box) the recommended reordering method, and connect its input port to the output port of the “BasicInfo” component. After reordering, a summary results HTML file returned from the “Reorder” module indicates that a banded solver may require 700 Mbytes of memory.
4. Select a “Banded” solver component, this time running on a remote machine with 2 Gbytes of main memory. After connecting it to the “Reorder” module, it is run and produces a solution.
5. Decide to also try an iterative solution. An “SPLIB” component is selected running on a local workstation, and its input port is connected to the “NewSystem” module.

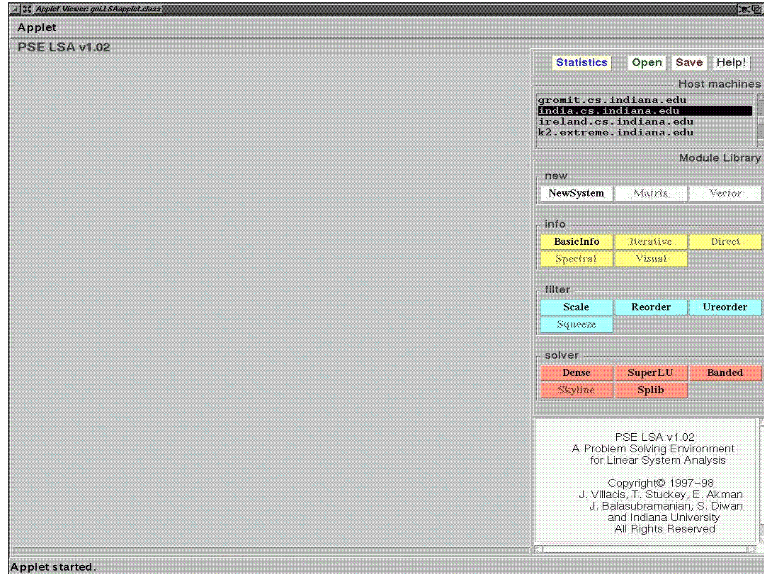

Figure 1: Startup GUI Interface for the LSA

Since the "NewSystem" module is still active, it sends the original system to "SPLIB" which solves it.

6. Decide to compare "SPLIB" on the reordered system. Another "SPLIB" component is instantiated, possibly on a different workstation, and its system input port is connected to the "Reorder" module.

Figure 2 shows a snapshot of the LSA based on this scenario.

Further experiments can be run using this already-connected graph of components. If a control parameter for an upstream component changes, the changes automatically propogate downstream through the graph. So if the user opens the "Reorder" component's subGUI and changes the method applied, the differently reordered linear system will be output to the "Banded" and "SPLIB" modules, which then run. At any time the summary results HTML files can be retrieved from the solvers and compared.

Once a solution strategy is developed as a component network, the "NewSystem" module can be used to read in additional test problems from the application area, checking the strategy for robustness and reliability. The component-based architecture of the LSA allows further useful extensions. For example, the "NewSystem" module can easily allow input from another active process - possibly a large running application code. Similarly, its solution ouput module can be hooked back into the running application, allowing the LSA to be "wired" directly into an application, the same way a hardware component can be wired into an integrated circuit.

This example helps show the usefulness of a component architecture, particularly compared to the usual approach of running different programs by hand and trying to patch together a meaningful picture of the problem-solving process. As a complete problem-solving environment, the LSA includes several features not described here - expert systems and case-based reasoning systems to guide a user in choosing strategies, systems to organize and preserve the summary results files coherently, dynamic resource allocation systems to help

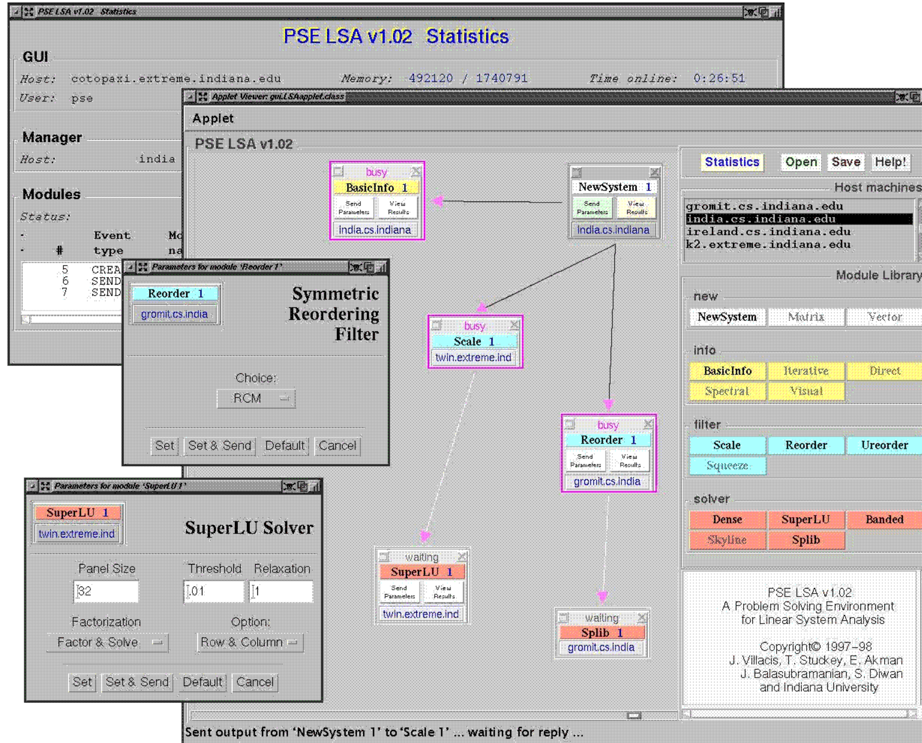

Figure 2: Snapshot of LSA for the Sample Scenario

choose a computer on which to run a module, and collaboration capabilities which allow users distributed across the network to share in developing a single component network. However, those subsystems are also implemented as components - making the entire design modular and useful for other scientific and engineering application areas. To further understand the possibilities for component systems in computational science and engineering, we first need to examine the underlying infrastructure supporting systems like the LSA.

3 Component-Based Programming Paradigms

An appropriate metaphor for the distributed component composition model used in the LSA system is hardware design from integrated circuits (ICs). The designer of an electronic digital system builds by selecting IC components from a family of parts which belong to a compatible technology. Each part has a specification sheet which describes the precise behavior of the device and the rules for connecting it to other components, including the protocol for its input and output buffers. However, this metaphor is not new [5]. In visualization, systems like IRIS Explorer [11] use it to construct specialized visualization tools by composing various filters, image preprocessors and rendering systems. More recently, Java Studio from Sun provides a complete system for composing Java *components* to build complete graphical user interfaces for Java applets. Microsoft's ActiveX technology also builds upon a component architecture to construct applications for Windows95 and NT. A system called OpenDoc is another component architecture from IBM and Apple which is based on the Object Management

Group's Common Object Request Broker Architecture (OMG's CORBA).

These commercial software component technologies are rapidly converging to two industry standards: Microsoft and a merged Java/CORBA solution (see sidebar). While these technologies provide some of the technology we need for scientific and engineering systems, they fall short when it comes to high performance scientific applications. To understand the missing pieces, we first need to consider the basic concepts that all these software technologies share.

3.1 Basic Concepts

The core technology used to build component based software is based on object oriented design principles. More specifically, a component architecture consists of a set of classes and behaviors defining component objects and an infrastructure which allows components to be composed.

A component is an object which encapsulates a computation and responds to messages received. There are several communication mechanisms that are commonly used in component architectures. These include events, remote procedure calls (RPC) and remote object method invocations (RMI) and a type of large grain data-flow. In this paper we will consider the data-flow model in more detail because it is a natural fit for many of the large matrix computations that are used by the LSA. Also it is a model that is easily constructed from RMI and RPC style communications. The concept is simple. When messages are received at the component's *input ports*, state changes in the internal computation can then cause messages to be sent out the component's *output ports*. While this concept is sufficient for a complete distributed programming paradigm, most systems also implement an interface for control signals and type and status queries based on RMI communication.

Input and output ports are each defined by an interface specification which describes the exact types of message that can be passed though the port. A port can be defined to handle messages of multiple types; e.g. a single input port of an LSA component may be defined to handle messages containing vectors and matrices. The interface specification also describes the types of exceptions that can be raised when an error is detected.

In most component systems, the interfaces are defined with an *Interface Definition Language* (IDL) which provides a concise way to define types, message names and exceptions.

The IDL specification is in turn used to generate the classes which represent the corresponding input and output port objects. For example, if a component design requires an output port of type *OutputVectorPort*, the programmer can add this into the system as

```
OutputVectorPort myOutPort;
```

The class OutputVectorPort has a method *send(floatVector)* which is used by the component to send a message out the port by means of a call of the form

```
myOutPort.send(myVector);
```

which initiates an action on every component with an input port connected to `myOutPort`. On the receiving side, a component can declare an input port similarly. Every input port must be bound to a handler member function for that component. For example, to create an input port and bind it to the function *consume_vector*, one could write

```
void consume_vector(doubleVector);

InputVectorPort myInPort;
myInPort.setHandler(consume_vector);
```

When a message is received by the component at a particular port, the component wrapper calls the handler function.

A component needs a way in which it can identify its ports to the outside world. We will return to this topic later. In addition, the careful reader will note that while we have indicated that ports can handle multiple data types, we have not described the conversion (cast) problems that must be solved to make this work.

3.2 Container Frameworks

To connect a set of components together into a working system, we need an architectural framework or *component container.* The important tasks for the container framework include:

- *Component identification and handling.* This refers to the way an object is recognized as a component and integrated into the system. This includes the recognition of the data types associated with the ports as well as handling the interface to the object initialization and parameter settings.

- *Interface registry.* The container framework must maintain a database of component interfaces and associated implementations. In CORBA terms, this is the interface registry function of the object request broker.

- *Connection management.* This gives the specification of which output ports are currently connected to which input ports as well as providing the protocols for communication of the typed data streams on these connections.

- *User interface.* Programmers can build systems based on connecting objects together with a graphical circuit-building tool such as Iris Explorer or Java Studio. Alternatively, a script language can be used to list the components to be activated and their connections. In the case of graphical systems, there is usually an icon that represents each object as a member of the container.

- *Exception and event management.* In most cases an exception raised by an input port should be propagated back to the corresponding output port. In some cases the exception also must be propagated to the user interface system. Other types of events are associated with the user interaction with the component icons. For example, selecting an icon with a mouse should open a control panel window for parameter settings for that component.

- *Resource management.* In the case of distributed component systems, an important factor is the identification of suitable computational and network resources upon which the components will run. This is especially true for the high performance component

systems considered here. A closely related set of concerns are those of authentication, security and privacy, which are not discussed in this paper.

- *Debugging.* When building and testing a distributed application it is essential to be able to interactively test and monitor the system under construction. It is part of the task of the component container framework to make that possible.
- *Component substitution.* A component architecture allows dynamic replacement of modules. As long as the input/output ports do not change, the software equivalent of a "hot swap" is possible.
- *Collaboration.* A piece that is missing from almost all container architectures is a mechanism that will incorporate collaboration into both the design of a component network as well as its use. By collaboration, we mean two or more users in different locations working together to build, use and modify a distributed computation.

Figure 3 shows a set of components contained inside the Component Container Framework, along with the Framework Controller. The components communicate with each other through their input and output ports. The framework controller communicates with a component through the component's functional interface. This interface is used by the controller for tasks such as component firing control, setting a component's initial parameters, querying the component's port types, etc. It is also used by the component to forward events and exceptions to the framework controller. The framework controller, apart from controlling the operation of the components, also supports tasks such as User Interface, Collaboration, Resource Management and Scripting. Although the figure shows the framework controller as a single monolithic entity, it could be implemented as a set of distributed modules working together.

As shown in figure 4, a component itself is divided into two parts. One part implements generic component functions, such as initialization, port handlers functions, framework exception handling etc. The other part contains component specific functionality, such as the Fortran/C++ code of a linear system solver. The two parts are connected through a component specific internal interface.

To see how one can build a system with all of the above mentioned properties, it is necessary to take a closer look at the design of the base port and component classes.

3.3 Container Input/Output Ports

As described earlier, ports are objects that must be accessible from outside the component. We can represent a connection from an output port to an input port by means of a Java remote reference or CC++/HPC++ global pointer to the input port. By a remote reference or global pointer, we mean an object that acts like a reference or pointer, but really represents a proxy (often called a *stub*) for a remote object. Invoking a method of the remote object class on the proxy causes a remote procedure call to the same method on the associated remote object.

Each output port must contain

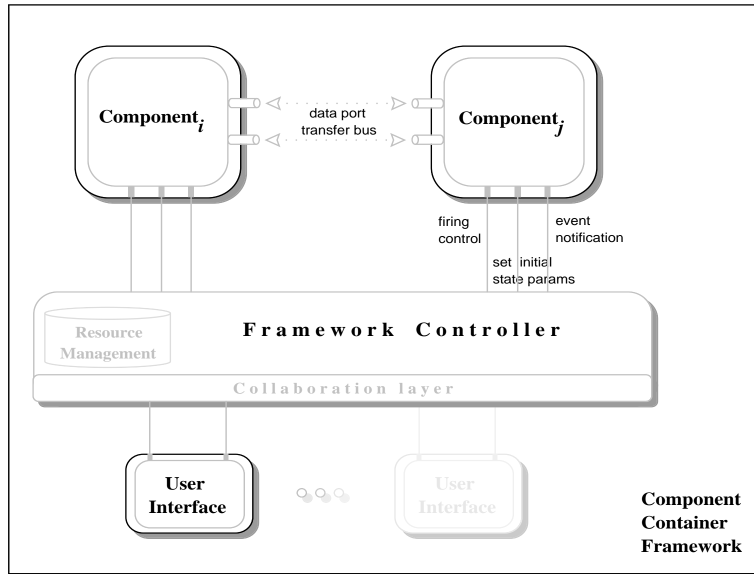

Figure 3: Overview of Container Framework

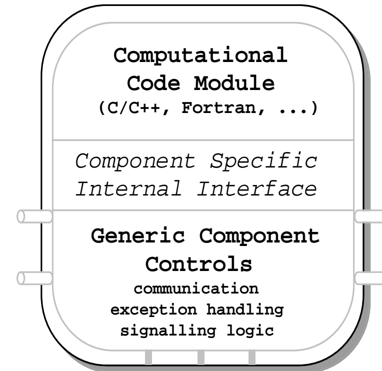

Figure 4: Internal Component Structure

- A list of global pointers or remote references to the input ports which are connected to this port.
- A method to add a connection to this list.
- A method to remove a connection.
- A method to inquire about the data types that this port serves.
- A "send" method that can be called by the component to transmit the message to the connected input ports.

An input port requires

- A binding to a handler function in the component and a method to set this binding.
- A method which returns an encoding of the types of messages that this port expects.

In addition, each component must define the rules for port data management. For example, does an input port maintain a queue of data values or does it have a single buffer which can be overwritten?

The base class of a component provides the minimum functionality that a component needs to be a citizen of the component architecture. In particular, components need to provide

- A function which returns an array of global references to the component's ports.
- A function which provides information about the component's functionality. For example, this could be a URL to the component's documentation and an encoding of the interface specifications of the ports it contains.
- A function which can provide a snapshot of important features of the component's internal state at any given time.

- Persistence operators including a mechanism to "freeze" or "thaw" a component for later use.
- A method to clone a component, perhaps on a different machine.

Some modern research systems such as Infospheres [4] and WebFlow [8] use a similar approach but also address the concerns of high performance scientific and engineering applications. Legion [15] from the University of Virginia supports an extensive object model that meets many of these objectives. Another system that could have been used to build this system is the ACE framework from Washington University [10]. Although based on a client-server-agent model, NetSolve [2] can also be used to develop solution strategies for large linear systems. The NetSolve system closely targets the dynamic resource management problem of finding suitable hardware and software resources on which to run a module.

Two other systems target high-performance distributed, parallel computation in a manner similar to LSA. The first is PELLPACK [12], a PDE based distributed PSE, and PAWS [14], a parallel application workspace for scientists. The software architecture of PSEs is characterized by the integration model used to connect the software parts, underlying execution model used, the user interface, analytical tools, communication model and the medium used. PELLPACK and PAWS also provide graphical user interface for machine and data selection and visualisation. This is independent of the target machine architecture and its native programming environment. Both these frameworks support libraries in multiple high level languages.

PAWS provides for computational steering and a system to integrate loosely connected modules. Application codes such as those derived from POOMA framework can be directly plugged into thier workspace. In the PAWS model the connection between two different tasks is usually achieved via a producer-consumer or client-server model. However in our model, communication demands and the need for concurrency requires that components communicate as peers. The typical style to achieve connection between applications in PAWS is to exchange their "PawsConnectHandle". The LSA uses HPC++ methods to achieve this. However, both PAWS and HPC++ are built on top of the Nexus communication library which is part of the Globus infrastructure. In the future, we will provide for the integration of PAWS components with the LSA framework.

The PELLPACK framework has the facility to make decisions to help the user by consulting a knowledge base about the user, problem domain and past solutions of similar problems. The PELLPACK model controls the both compilation and loading of the components. By contrast, component architectures such as the LSA are based on efficiently connecting existing binaries. The LSA architecture also provides facility for input from an active process and can hook an output module onto a running application.

3.4 Requirements of High Performance Distributed Object Systems

Each of the commercial desktop component systems (ActiveX, CORBA/OpenDoc, Java Studio) supports a family of component properties similar to those described above. However, they all share some serious shortcomings for distributed scientific applications. The most serious problem is the routing of all communications between the distributed components

through the front-end container application. For example, in a distributed application the components in Java Studio are proxies for the corresponding distributed computational resources. The proxies invoke remote method calls on their distributed counterparts and all of the message passing takes place in the Studio or front-end applet environment. In the case of ActiveX, the same problem exists. However, in the LSA example, the line glyphs connecting components in the front-end user interface of Figure 2 represents point-to-point, high bandwidth data communication channels between the corresponding distributed components.

The reason for this weakness in the existing distributed systems standards is that they are derived from a *client-server* model of distributed computing. In this model a client makes a request of a server and then waits for the reply. However, in the distributed scientific applications considered here, both the communication demands and need for concurrency require that all components communicate as peers. Furthermore, the communication protocols on which the systems are based must be able to adapt to multiple and dynamic transport layers. The current Java or ActiveX communication layers are unable to recognize when a high-speed pathway exists between two components (such as an ATM link) and adapt to the protocol supporting it. In particular, for components that are based on parallel implementations, the communications channels may need to be implemented as multiple concurrent pathways (see [13] for an example.)

A second problem with current commercial desktop component systems is the mechanism for encapsulating an existing scientific application (which may itself be a parallel distributed computation) into a component. Completely Java-based systems are not suitable for an application like the LSA, because a mixed language model is needed. For example, the LSA prototype is

- 15% C/C++
- 63% Fortran
- 22% Java.

A mixed language system provides more than just legacy code encapsulation - it allows a wider class of users to contribute components, and bug fixes and updates from the original code suppliers can be quickly incorporated into the system. CORBA suffers less from this mixed-language problem. Although it uses IDL as the common data description language, components must be implemented in a language for which IDL has a binding; currently no binding exists for Fortran.

A third defect with current systems is poor support for scripting. Although GUI's are useful for quick experimentation and rapid prototyping of various connected component systems, for science and engineering applications a script language interface is also needed - individual components may run for hours or days, and a user may not want to instantiate a downstream component until the upstream one completes. A scripting interface is also useful to generate and test several combinations of component systems rapidly. In the LSA example the user may want to test five different reordering methods with seven different solvers. In this case writing a short double-nested loop in a scripting language is more user-friendly than using the GUI to construct the 35 different solution strategies. Scripting languages are also necessary when a very large numbers of components need to be connected. Finally, it should

be possible to generate a script from the visual component composition tool. We note that the LSA system does not currently support scripting, but work is underway at both Indiana and Los Alamos on defining scripting interfaces to languages like Perl and Python to allow them to interact with HPC++ and Java components.

4 How to Build a Component

The previous sections have discussed the process of building applications from components in the LSA example and the general features of the component container architecture. However, for the scientific programmer who wants to add a new component, this still raises the question of how one might "wrap" an existing application so that it can become a component.

Consider the example of introducing a new linear system solver which takes as inputs a matrix A and right-hand-side vector b and produces a solution x to $Ax = b$. The library that implements this may be in Fortran or some other language. Furthermore, we must allow that the library module has special parameters that may be set by users. The task of building a new component can be broken down as follows.

- To define the component, one must build an IDL description defining the types of messages that will be sent and received at the output and input ports respectively. Alternatively, if a generic component of the desired type already exists, one may use class inheritance to define the basic port structure.
- The control interface for the messages that the component will receive to set parameters or other signals the container will use to interact with the component must be defined with an IDL interface.
- The IDL compiler will generate the standard interface protocol code. But the user must implement the skeleton of the new component class.

In our case the solver is an instance of a generic solver for the LSA. This generic *SolverComponent* only creates and instantiates the input and output ports and labels each port with a type string as shown below.

```
class SolverComponent: public Component{
  public:
     InputVectorPort b_input;
     InputMatrixPort A_input;
     OutputVectorPort x_output;

     SolverComponent(){
         addOutputPort(x_output, "Vector");
         addInputPort(b_input, "Vector");
         addInputPort(A_input, "Matrix");
       }
};
```

The user code for the new solver would only need the IDL to specify the control signals that are exported to the object container environment. This IDL description will also identify the types of exceptions that are thrown when an error is encountered.

```
interface MyComponent: SolverComponent{
   void modifyParameters(ParameterBlock) throws ParameterException;
   void kill();
   void run() throws SolverException;
};
```

The IDL compiler generates the class header for *MyComponent*. The programmer then only needs to implement these operations.

In the current LSA prototype we use HPC++Lib [9] to program the component wrapper code.

The following implementation example uses synchronized queues for managing the input data streams. HPC++ is multithreaded. Synchronized queues provide a simple way for a *producer* thread to send data to a *consumer* thread. In our case, the producers are the input port objects. When a port receives a new object it calls the component function that has been bound to the port by the *setHandler()* function. This activity takes place in a separate thread from the main thread of the component. The main thread will be running the *run()* function and is the consumer. When a producer writes to a synchronized queue object, the new value is appended to the end of a queue. When a consumer reads from a synchronized queue it removes the value at the head of the list. If there are no values there, it waits. In the example below, the *run()* thread waits on the input queues until both a matrix and a right hand side are available, calls the Fortran solver, and then sends the solution out the output port.

```
class MyComponent: public SolverComponent{
     HPCxx_SyncQ< Vector > b;
     HPCxx_SyncQ< Matrix > A;
     Vector x;
     int kill;
     ParameterBlock parameters;
 public:
     void set_rhs(Vector b_in){ b = b_in; }
     void set_A(Matrix A_in){ A = A_in; }

     MyComponent(): SolverComponent(), parameters(default){
         kill = 0;
         b_input.setHandler(this, set_rhs);
         m_input.setHandler(this, set_M);
         }

     void modifyParameters(ParameterBlock parameter){
         if( bad_parameters(parameter) ) throw ParameterException();
         else parameters = parameter;
        }
     void kill(){ kill = 1;}
```

```
    void run(){
        while(!kill){
           Vector b_in = b; // causes a wait until b set by port.
           Vector A_in = A; // causes a wait until A set by port.
           error = fortran_solver(&b_in, &A_in, &x, &parameters);
           if(error != 0) throw SolverException(error);
           try{ x_output.send(x);
           }
           catch(SendException e){
              // analyze problem and throw appropriate
              // exception up to the container.
          }
    }
};
```

This particular example has too many data copies, but it illustrates the key concepts. A more efficient implementation would keep a single buffer to hold the matrix and vectors. When an input port wants to deposit a new value, a lock must be acquired. Once acquired, the lock is set and the value placed in the buffer. The component then waits for the buffer to be filled, does its computation, marks the buffers as empty and finally frees the locks. The exact semantics of how a component handles its inputs is defined by that component, in what is called the "firing logic" in Figure 3.

4.1 Adding the Component to the Framework.

To install the component, the class above is linked with a simple driver program, which when run, will create an instance of the class and, upon command from the container framework, return a remote reference to the instance and start the "run()" method for the object.

The container framework must then load a URL that can be used to start the driver program on the target host server. Once the container has access to the remote reference to the component, it sends a series of queries to the component to identify the control interface specification and the number and types of ports.

The current implementation of the LSA was written prior to the completion of the IDL compiler described here. Consequently the mechanisms described here are hand coded in HPC++. The IDL compiler is scheduled for completion by the end of 1998.

4.2 Implementing the Component Framework

Java was used to build the Graphical User Interface for the LSA component framework. A multithreaded implementation of CORBA could have been used to build everything else, because CORBA now has a Java binding. However, because of our performance concerns with CORBA on specialized networks and the lack of implementations of CORBA on some of our parallel computing platforms another approach was used. HPC++ provides us support on all parallel processing platforms and it has a rich set of thread and remote object primitives. HPC++Lib has been implemented using the Nexus [7] runtime system which is part of the

Globus [6] metacomputing infrastructure. To implement the Java to HPC++Lib/Nexus library we have developed a version of the Java RMI layered over Nexus. HPC++Lib uses the concept of global pointers from CC++ [3] for remote method invocations. Consequently, a substantial subset of the RMI remote reference semantics can be converted to HPC++Lib global pointer operations. In other words, Java can see a remote HPC++ object as a remote Java object instance and execute RMI calls as remote method calls on that object. There are some problems with implementing the complete RMI semantics using non-Java objects that arise due to the Java serializability and reflection properties, but it is easy to avoid problems in practice [1].

Recently Sun has announced that RMI will be ported to run over the CORBA IIOP protocol to achieve exactly this sort of Java to foreign object interoperability. We are currently working on a bridge between IIOP based CORBA systems and HPC++ and a complete implementation of HPC++ over IIOP is under development. The good news is that as these remote method call protocols continue to converge, it will become even easier to implement the future component architectures that they will support.

5 Conclusion

Although a useful model for many systems, the standard client-server architecture is not adequate for many modern CS&E applications. Instead, a peer-to-peer model based on a component architecture is more appropriate for systems with many distributed modules, complicated inter-module interactions, multilanguage code, collaboration subsystems, and dynamically managed network communications protocols.

In this paper we have outlined the basic features of a component system architecture, including an example of how a simple component and container framework can be built. The potential usefulness for large scale computing was illustrated with a prototype system for quickly developing solution strategies for large sparse linear systems of equations, and we showed how it can be used to build a distributed linear system solver. Programming the LSA consists of selecting components from a toolbox of matrix analysis modules, filters, and direct and iterative solvers. The components can be instantiated and run on remote systems and may be parallel applications and implemented in any mixture of languages. The LSA component framework then allows "wiring" together the components in much the same way a design engineer wires together integrated circuits to build an application.

The broad family of component architectures is becoming increasingly important in the desktop and Internet computing environments. Over the next year we can expect that the software industry will see a maturation and convergence of basic component architectures. Systems such as Microsoft's ActiveX technology and the Java/CORBA component models will become fully interoperable. However, the economics of the computer industry assure that the high performance scientific and engineering applications of these industrial trends will be slow in coming. Consequently, it is the task of the CS&E community to push this technology up the Branscomb pyramid, from desktops to high-end and large-scale scientific and engineering computing. Our problem solving technology really must be interoperable with desktop standards.

While greater productivity and user ease can come from component architecture ideas

developed at the bottom of the pyramid, the currently emerging standards do not address many areas of vital interest in CS&E. This is the flow of ideas and technology which the CS&E community can contribute from the top of the Branscomb pyramid, where the rapid development of gigabit communication technologies is being coupled with advanced high performance computing methodologies for metacomputing. What the CS&E community learns there will work it way down to the desktop.

References

[1] 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.

[2] H. Casanova, J. Dongarra, C. Johnson, and M. Miller. Tools for Building Distributed Scientific Applications and Network Enabled Servers, 1998. In *Computational Grids*.

[3] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation, 1993. In *Research Directions in Concurrent Object Oriented Programming*, MIT Press.

[4] K. Mani Chandy, Adam Rifkin, Paolo A.G. Sivilotti, Jacob Mandelson, Matthew Richardson, Wesley Tanaka, and Luke Weisman. A World-Wide Distributed System Using Java and the Internet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, August 1996.

[5] B. J. Cox. *Object Oriented Programming*. Addison-Wesley, Reading, Mass, 1986.

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

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

[8] G. Fox and W. Furmanski. Web Technologies in High Performance Distributed Computing, 1998. In *Computational Grids*.

[9] 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, 1998.

[10] A. Gokhale, D.C. Schmidt, T. Harrison, and G. Parulkar. Towards real-time CORBA. *IEEE Communications Magazine*, 14(2), Feb 1997.

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

[12] Elias N. Houstis and John R. Rice. Parallel Ellpack, 1997. see http://www.cs.purdue.edu/research/pellpack.html.

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

[14] LANL Advanced Computing Laboratory. PAWS: Parallel Application WorkSpace, 1997. see http://bluemountain.acl.lanl.gov/PAWS/docs/proposal.htm.

[15] Michael J. Lewis and Andrew Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, August 1996.

6 Sidebar: CORBA and Java Component Technology

The Common Object Request Broker Architecture (CORBA) is the result of eight years of work by the Object Management Group (OMG) to define a standard for distributed object systems. OMG is a consortium of over 700 companies, universities and research organizations that are participating in this process. CORBA consists of core object server called an Object Request Broker (ORB) which is used by client application to connect to remote object instances. For example, every copy of Netscape Communicator contains a CORBA ORB which allows communicator utilities to talk to specialized applications on the network. In addition, CORBA defines a standard interface repository which allows servers to register interface definitions specified in the Interface Definition Language IDL. This allows CORBA clients the ability to dynamically discover the properties of a server object at runtime. The CORBA architecture provides a complete set of services such as naming, transactions and security management as well as a set of vertical market interfaces. At the lowest level CORBA ORBs communicate with each other using a protocol known as the Internet Inter-Orb Protocol (IIOP).

More recently Sun introduced the Java programming language and runtime environment to facilitate application development on the world-wide web. As part of the Java environment, Sun also introduced a special Java-to-Java communication system based called Remote Method Invocation (RMI) and a simple object registry service. The basic client-server model is identical to that of CORBA and can be described as follows.

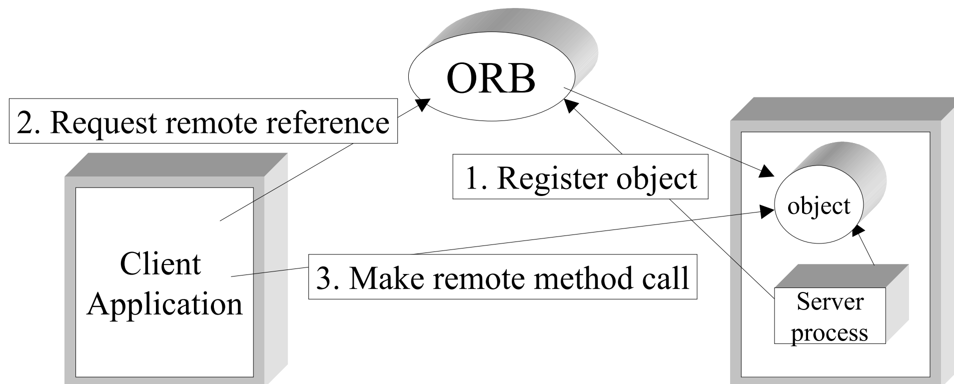

Figure 5: Basic Client Server Protocol.

1. A server creates an instance of an object and passes a reference to this object to the registry or ORB. The reference takes the form of a simple object that can be used to locate the object on the network and activate the objects methods by “remote control” from a client.

2. A client program request a reference to the object by sending a request to the local ORB. That ORB returns the reference (or it contacts another ORB which knows how to find the object).

3. The client receives the reference to the remote object and makes remote calls on the objects method functions and receives the returned values.

While this basic functionality is identical, Java RMI supports a wider range of parameter passing and object value return models. This is because, as a java-to-java solution, RMI can take advantage of many special features of Java such as reflection. On the other hand CORBA has a much richer architecture for distributed services and interfaces. Consequently, Sun, JavaSoft and OMG have agree to build a version of RMI that is layered on top of IIOP. Consequently, Java RMI can be used in conjunction with CORBA objects.

The third party to this activity is Microsoft. Their OLE/DCOM technology also provides a distributed object model. While the experts argue about the relative merits of DCOM, Java RMI and CORBA, they are all important.

However, component systems require more than a robust remote method invocation service as described in the paper. Microsoft has introduced ActiveX as a component architecture. ActiveX is now built into the next releases of Windows98 and NT and it is the foundation for their current generation desktop software. Sun has also released a component architecture consisting of Java beans and Java Studio. CORBA, on the other hand, does not as yet have an official component architecture, but OpenDoc from Apple and IBM is a CORBA based component system for the design of graphical user interfaces and documents. CORBA will probably release a specification for a component architecture within a year.