SCIRun2: A CCA Framework for High Performance Computing

Keming Zhang, Kostadin Damevski, Venkatanand Venkatachalapathy, Steven G. Parker
SCI Institute, University of Utah

{*kzhang,damevski,venkat,sparker*}@*sci.utah.edu*

Abstract

We present an overview of the SCIRun2 parallel component framework. SCIRun2 is based on the Common Component Architecture (CCA)[2, 5] and the SCI Institutes' SCIRun[10]. SCIRun2 supports distributed computing through distributed objects. Parallel components are managed transparently over an MxN method invocation and data redistribution subsystem. A meta component model based on CCA is used to accommodate multiple component models such as CCA, CORBA and Dataflow. A group of monitoring components built on top of the TAU toolkit[12] evaluate the performance of the other components.

1. Introduction

In recent years, component technology has been a successful methodology for large-scale commercial software development. Component technology combines a set of frequently used functions in a component and makes the implementation transparent to the users. Application developers typically connect a group of components from a component repository, connecting them to create an application.

Problem-Solving Environments (PSEs) often employ component technology to bring a variety of computational tools to an engineer or scientist that is solving a computational problem. In this scenario, the tools should be readily available and simple to combine to create an application.

SCIRun[1] is a scientific Problem-Solving Environment that allows the interactive construction and steering of large-scale scientific computations [19, 18, 20, **?**, **?**]. A scientific application is constructed by connecting computational elements (modules) to form a program (network), as shown in Figure 1. This program may contain several computational elements as well as several visualization elements, all of which work together in orchestrating a solution to a scientific problem. Geometric inputs and computational parameters may be changed interactively, and the results of these changes provide immediate feedback to the investigator. SCIRun is designed to facilitate large-scale scientific computation and visualization on a wide range of machines from the desktop to large supercomputers.

A number of component models have been developed. Java Beans[3], a component model from Sun, is a platform-neutral architecture for the Java application environment. However, it requires a Java Virtual Machine as the intermediate platform and the components must be written in Java. Microsoft has developed the Component Object Model (COM)[15], a software architecture that allows applications to be built from binary software components on the Windows platform. The Object Management Group (OMG) developed the Common Object Request Broker Architecture (CORBA)[16], which is an open, vendor-independent architecture and infrastructure that computer applications can use to work together in a distributed environment.

Many Problem-Solving Environments, such as SCIRun, employ these component models, or one of their own. As an example, SCIRun provides a dataflow-based component model. The Common Component Architecture (CCA) Forum, a group of researchers from several national laboratories and academic institutions, has defined a standard component architecture[2] for high performance parallel computing. The CCA forum has defined a minimal set of standard interfaces that a high-performance component framework should provide to implement high-performance components. This standard promotes interoperability between components developed by different teams across different institutions. However, CCA has not yet fully addressed the architecture of parallel components combined with distributed computation.

A few research groups have implemented CCA frame-

[1] Pronounced "ski-run." SCIRun derives its name from the Scientific Computing and Imaging (SCI) Institute at the University of Utah.

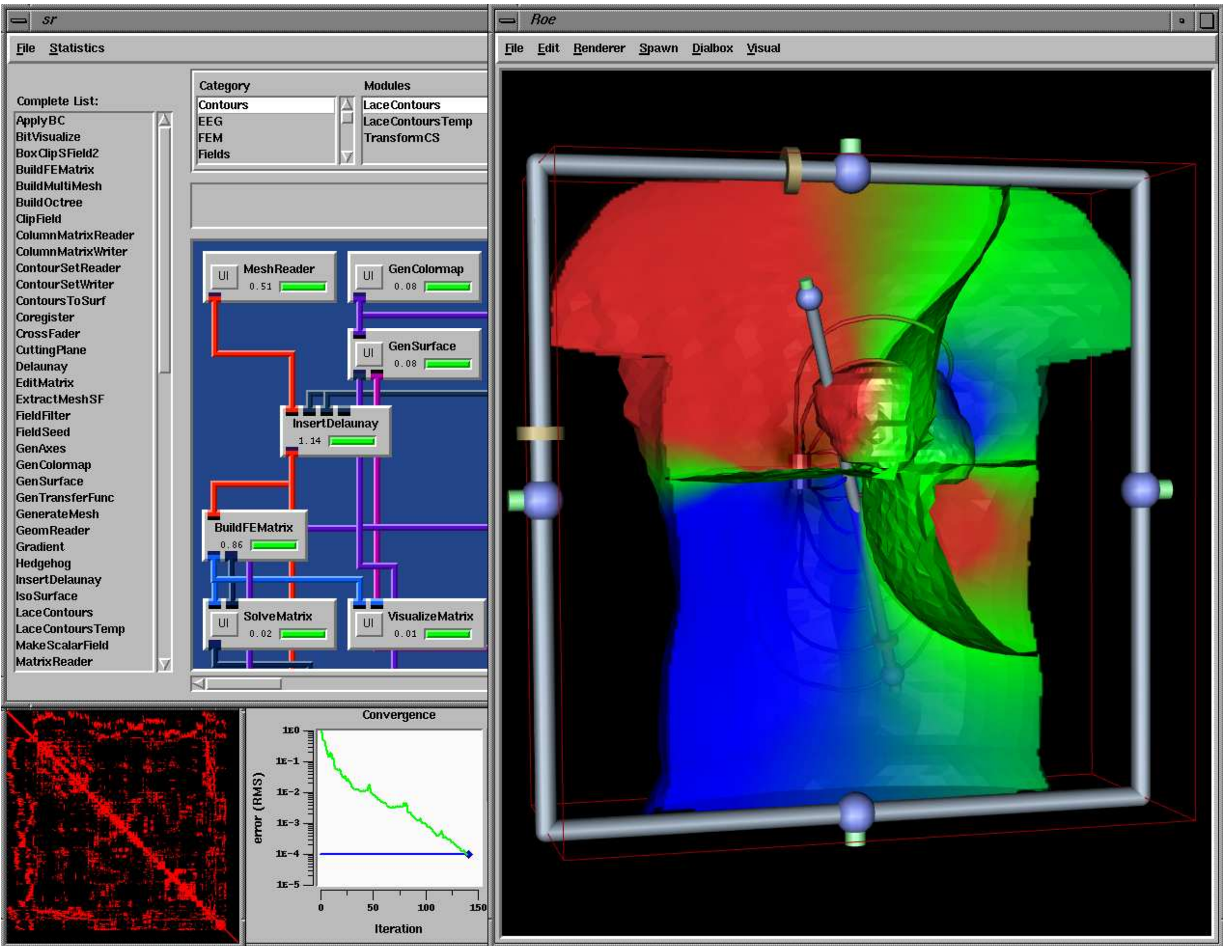

Figure 1: A picture of the SCIRun PSE, showing a 3D finite element simulation of an implantable cardiac defibrillator.

works. XCAT[4], from Indiana University, supports applications built from components that are distributed over a wide-area Grid of resources and distributed services. Caffeine[1], developed at Sandia National Laboratory, is considered the CCA "reference framework" implementation and supports distributed-memory parallel computing with SPMD-style components.

In this paper, we present an overview of the architecture of SCIRun2, a framework that combines CCA compatibility with connections to other commercial and academic component models. SCIRun2 is based on the SCIRun [10] infrastructure and the CCA specification. It utilizes parallel-to-parallel remote method invocation to connect components in a distributed memory environment, and is multi-threaded to facilitate shared memory programming. It also has an optional visual-programming interface.

Although SCIRun2 is designed to be fully compatible with CCA. It aims to combine CCA compatibility with the strength of other component models. A few of the design goals of SCIRun2 are:

1. It is fully CCA compatible, thus any CCA components can be used in SCIRun2 and CCA components developed from SCIRun2 can also be used in other CCA frameworks.

2. It accommodates several useful component models. In addition to CCA components and SCIRun Dataflow Modules, CORBA components, Microsoft COM components, and Vtk[21] modules will be supported in SCIRun2.

3. It builds bridges between different component models, so that we can combine a disparate array of computational tools to create powerful applications with cooperative components from different sources.

4. It supports distributed computing. Components created on different computers can work together through a network and build high performance applications.

5. It supports parallel components in a variety of ways for maximum flexibility. This is not constrained to only CCA components, because SCIRun2 employees a M process to N process method invocation and data redistribution (MxN) library [7] that potentially can be used by many component models.

Overall, SCIRun2 provides a broad approach that will allow scientist combine a variety of tools for solving a particular problem. The overarching design goal of SCIRun2 is to provide the ability for a computational scientist to use the right tool for the right job.

Section 2 briefly reviews the Common Component Architecture. The SCIRun2 meta-component architecture is described in Section 3. In Section 4 we describe our approach to distributed components. The parallel component architecture is discussed in Section 5, and integrated performance evaluation in Section 6. Conclusions and future work are discussed in Section 7.

2. CCA Overview

This section briefly reviews the Common Component Architecture(CCA) for readers who are not familiar with the CCA. It also provides some background knowledge for the SCIRun2 architecture.

CCA consists of a framework and a expandable set of components. The framework is a workbench for building, connecting and running the components. A component is the basic unit of an application. A CCA component consists of one or more ports, and a port is a group of method-call based interfaces. There are two types of ports: **uses** port and **provides** ports. A provides port (or callee) implements its interfaces and waits for other ports to call them. A uses port (or caller) issues method calls that can be fulfilled by a type-compatible provides port on a different component.

A CCA Framework provides a number of services for the components. It maintains the component repository, creates and destroys components. It gathers the port information and maintains the connections between ports.

CCA defines interfaces through a scientific interfaces definition language (SIDL). For example, a port interface can be defined by SIDL as follows,

```
package pkg_name{
    interface string_port{
        string getString(in int index);
    }
}
```

where method getString is defined in the string_port interface. getString takes an input argument of integer type, and returns a string type. For more information about SIDL, see [11].

CCA places few constraints on how an interface is implemented. An IDL compiler is usually used to compile a SIDL interface description file into specific language bindings. Generally, component language binding can be provided for many different languages such as C/C++, Java, Fortran, Python etc. The Babel [11] compiler group is working on creating this support for different languages within CCA.

Building an application with CCA components is quite simple if a sufficient component set is available: the user simply creates a set of components, connects the relevant ports, and starts a special port (go port) on the driver component.

3. Meta Component Model

The key concept behind SCIRun2 is a meta component model that allows components to be used together even if they do not share the same underlying component architecture. SCIRun2 provides support for the DOE Common Component Architecture, a model that is more familiar to programmers who have used the Object Management Group's (OMG) CORBA[16], or Microsoft's COM[15]. SCIRun2 also combines support for old-style SCIRun dataflow components, and we are planning support for CORBA, COM and Vtk. As a result, SCIRun2 can utilize SCIRun components, CCA components and others in the same simulation. This provides a variety of components available to the application developer, without requiring buy-in to any particular methodology. Furthermore, systems that are not traditionally thought of as component models, but have a well-designed regular structure can be mapped to a component model and manipulated dynamically.

The meta component model operates by providing a plug-in architecture for component models. Abstract components are manipulated and managed by the SCIRun2 framework, while concrete component models perform the actual work. This facility allows components implemented with disparate component models to be orchestrated together.

Figure 2 demonstrates with a simple example how SCIRun2 handles different component models. Two CCA components, **Driver** and **Integrator**, and one CORBA component, **Function**, are created in the SCIRun2 framework. In this simple example, the Driver is connected to both the Function and Integrator. Inside SCIRun2, two frameworks are hidden: the CCA framework and the CORBA Object Request Broker (ORB). The CCA framework creates the CCA components, Driver and Integrator.

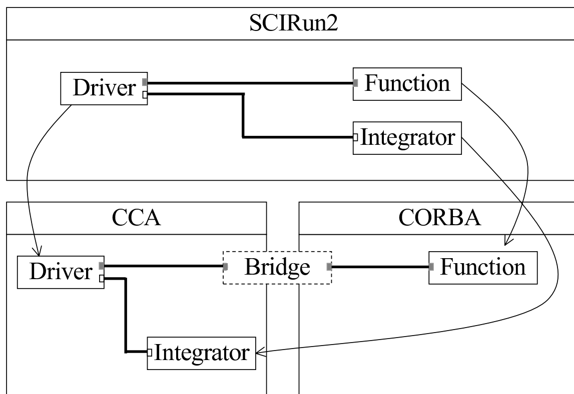

Figure 2: Components of different models cooperate in SCIRun2

The CORBA framework creates the CORBA component, Function. The two CCA components can be connected in a straightforward manner through the CCA component model. However, the components Driver and Function cannot be connected directly, because neither CCA nor CORBA allow a connection from a component of a different model. Instead, a bridge component is created. Bridges belong to a special internal component model that is used to build a connection between components of different component models. In this example, Bridge has two ports: one CCA port and one CORBA port. In this way it can be connected to both CCA component and CORBA component. The CORBA invocation is converted to request to the CCA port inside the bridge component.

Bridge components can be manually or automatically generated. In situations where interfaces are easily mapped between one interface and another, automatically generated bridges can facilitate interoperability in a straightforward way. More complex component interactions may require manually generated bridge components. Bridge components may implement heavy-weight transformations between component models, and therefore have the potential to introduce performance bottlenecks. For the few scenarios that require maximum performance, reimplementation of both components in a common, performance-oriented component model may be required. However, for rapid prototyping, or for components that are not performance-critical, this is completely acceptable.

To automatically generate a bridge component that translates a given pair of components, a generalized translation must be completed between the component models. A software engineer designs how two particular component models will interact. This task can require creating methods of data and control translation between the two models and can be quite difficult in some scenarios. The software engineer expresses the translation into a compiler plugin, which is used as a specification of the translation process. A plugin abstractly represents the entire translation between the two component models. It is specified as an XML document that can be augmented by Ruby [13] scripts. The Ruby scripts are useful for situations where the translation requires more sophistication than regular text (such as control structures or additional XML parsing). This provides us with better flexibility and more power inside the plugin, with the end goal of being able to support the translation of a wider range of component models. By using the plugin and an interface of the ports we want to bridge (usually expressed in an IDL file), the compiler is able to generate the specific bridge component. This component is automatically connected and ready to broker the translation between the two components of different models.

4. Distributed Computing

SCIRun2 provides support for Remote Method Invocation (RMI) based distributed objects. This support is utilized in the core of the SCIRun framework in addition to distributed components. This section describes the design of the distributed object subsystem.

A distributed object is a set of interfaces defined by SIDL that can be referenced over network. The distributed object is similar to the C++ object: it utilizes similar inheritance rules and all objects share the same code. However only methods (interfaces) can be referenced, and the interfaces must be defined in SIDL. Using the SIDL language, we implemented a straightforward distributed object system. We extend the SIDL language and build upon this system for implementing parallel to parallel component connections, as discussed in the next section.

A distributed object is implemented by a concrete C++ class, and referenced by a proxy class. The proxy class is a machine-generated class that associates the user-made method calls to a call by the concrete object. The proxy classes are described in a SIDL file, and a compiler compiles the SIDL file and creates the proxy classes. The proxy classes define the abstract classes with a set of pure virtual functions. The concrete classes extends those abstract proxy classes and implement each virtual functions.

There are two types of object proxies. One is called server proxy, the other is called client proxy. The server proxy (or skeleton) is the object proxy created in the same memory address space as the concrete object. When the concrete object is created, the server proxy starts and works as a server, waiting for any local or remote meth-

ods invocations. The client proxy (or stub) is the proxy created on a different memory address space. When a method is called through the client proxy, the client proxy will package the calling arguments into a single message, and send the message to the server proxy, and then wait for the server proxy to invoke the methods and return the result and argument changes.

We created Data Transmitter, a separate layer, that is used by the generated proxy code for handling messaging. We also employ the concept of a Data Transmission Point (DTP), which is similar to the start point and end points used in Nexus [9]. A DTP is a data structure that contains a object pointer pointing to the context of a concrete class. Each memory address space has only one Data Transmitter, and each Data Transmitter uses three communication ports (sockets): one listening port, one receiving port and one sending port. All the DTPs in the same address space share the same Data Transmitter. A Data Transmitter is identified by its universal resource identifier(URI): IP address + listening port. A DTP is identified by its memory address together with the Data Transmitter URI, because DTP addresses are unique in the same memory address space. Optionally, we could use other type of object identifiers.

The proxy objects package method calls into messages by marshaling objects and then waiting for a reply. Non-pointer arguments, such as integers, fixed sized arrays and strings (character arrays), are marshaled by the proxy into a message in the order that they are presented in the method. After the server proxy receives the message, it unmarshals the arguments in the same order. A array size is marshaled in the beginning of an array argument, so the proxy knows how to allocate memory for the array. SIDL supports a special opaque data type that can be used to marshal pointers if the two objects are in the same address space. Distributed object references are marshaled by packaging the DTP URI (Data Transmitter URI and object ID). The DTP URI is actually marshaled as a string and when it is unmarshaled, a new proxy of the appropriate type is created based on the DTP URI.

C++ exceptions are handled as special distributed objects. In a remote method invocation, the server proxy tries to catch an exception (also a distributed object) before it returns. If it catches one, the exception pointer is marshaled to the returned message. Upon receiving the message, the client proxy unmarshals the message and obtains the exception. The exception is then re-thrown by the proxy.

5. Parallel Components

This section introduces the CCA parallel component design and discusses issues of the implementation. Our design goal is to make the parallelism transparent to the component users. In most cases, the component users can use a parallel component as the way the use sequential component without knowing that a component is actually parallel component.

Parallel CCA Component (PCom) is a set of similar components that run in a set of processes respectively. When the number of process is one, the PCom is equivalent to a sequential component. We call each component in a PCom a *member component*. Member components typically communicate internally with MPI [14] or an equivalent message passing library.

PComs communicate with each other through CCA-style RMI ports. We developed a prototype parallel component infrastructure [6] that facilitates connection of parallel components in a distributed environment. This model supports two types of methods calls: *independent* and *collective*, and as such our port model supports both independent and collective ports.

An independent port is created by a single component member, and it contains only independent interfaces. A collective port is created and owned by all component members in a PCom, and one or more of its methods are collective. Collective methods require that all member components participate in the collective calls in the same order.

As an example of how parallel components interact, let pA be a uses port of component A, and and pB be a provides port of component B, Both pA and pB have the same port type, which defines the interface. If pB is a collective port, and has the following interface,

```
collective int foo(inout int arg);
```

Then getPort("pA") returns a collective pointer that points to the collective port pB. If pB is an independent port, getPort("pA") returns a pointer that points to an independent port.

Component A can have one or more members, so each member might obtain a (collective/independent) pointer to a provides port. The component developer can decide what subset (one, many, or all components) participate in a method call foo(arg).

When any member component register a uses port, all other members can share the same uses port. But for a collective provides port, each member must call addProvidesPort to register each member port.

The MxN library takes care of the collective method invocation and data distribution. We repeat only the essentials here, one can reference [7] for details. If

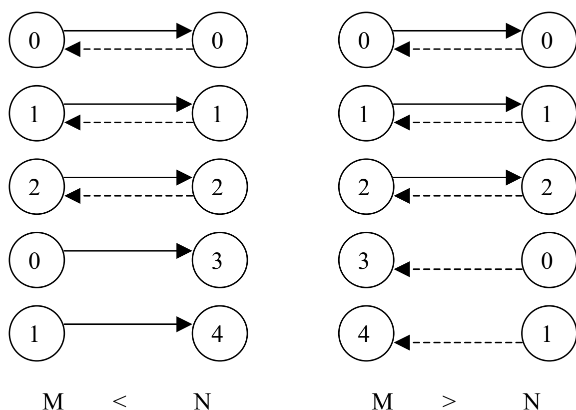

Figure 3: MxN method invocation, with the caller on the left and the callee on the right. In the left scenario, the number of callers is fewer than the numbers of callees, so some callers make multiple method calls. In the right, the number of callees is fewer, so some callees send multiple return values.

a M-member PCom A obtains a pointer ptr pointing to a N-member PCom's B collective port pB. Then ptr→foo(args) is a collective method invocation. The MxN library index PCom members with rank 0,1,...,M-1 for A and 0,1,...,N-1 for B. If M=N, then the i-th member component of A call foo(args) on the i-th component of B. But if M<N, then we "extend" the A's to 0,1,2,...,M, 0, 1,2,...M, ... N-1 and they call foo(args) on each member component of B like the M=N case, but only the first M calls request returns. The left panel of Figure 3 shows an example of this case with M=3 and N=5. If M>N, we "extends" component B's set to 0, 1, ..., N, 0, 1,...,N, ...,M-1 and only the first N member components of B are actually called, and the rest are not called but simply return the result. We rely on collective semantics from the components to ensure consistency without requiring global synchronization. The right panel of Figure 3 shows an example of this case with M=5 and N=3.

The MxN library also does most of the work for the data redistribution. An multi-dimensional array can be defined as a distributed array that associates a distribution scheduler with the real data. Both callers and callees define the distribution schedule before the remote method invocation, using an first-stride-last representation for each dimension of the array. The SIDL compiler creates the scheduler and scheduling is done in the background.

With independent ports and collective ports, we cover the two extremes. Ports that require communication among a subset of the member components present a greater challenge. Instead, we utilize a sub-setting capability in the MxN system to produce ports that are associated with a subset of the member components, and then utilize them as collective ports.

SCIRun2 provides the mechanism to start a parallel component either on shared memory multi-processors computers, or clusters. SCIRun2 consists of a main framework and a set of Parallel Component Loaders (PCLs). A PCL can be started with ssh on a cluster, where it gathers and reports its local component repository and registers to the main framework. The PCL on a N-node cluster is essentially a set of loaders, each running on a node. When the user creates a parallel component, it can dynamically choose a subset of nodes that the parallel component runs on, in which case a separate communicator is created for the subset. PCLs are responsible to create and destroy components running on their nodes, but they do not maintain the port connections. The main framework maintains all component status and port connections.

6. Performance Evaluation

This section discusses the performance monitoring and modeling facility provided by a group of SCIRun2 components, together referred to as PERFume. PERFume is designed to help SCIRun2 developers and users in quantifying and understanding the performance of its numerical components including the overhead imposed by the component model abstraction of CCA. Though designed primarily for use with the SCIRun2 framework, the PERFume components adhere to the CCA specification and should be interoperable with other CCA compliant frameworks. The PERFume group of components consist of components for performance monitoring, instrumentation, performance data storage, performance modeling and performance data visualization.

PERFume's monitoring and instrumentation components provide a facility for measuring a components' performance, resource usage (processor, memory, network), inter-component interactions so forth. This data is recorded in a database, and used to construct a performance model/metric and provide a performance assessment to the application developer with graphical visualizations. The performance monitoring and instrumentation components are built over the TAU library [12]. Using the TAU library, instrumentation can be done at various granularity (whole component performance, function calls, loop level measurements) and also at various stages in component and application development. A notable feature is that performance monitoring component can be used for dynamic instrumentation of components in execution. Thus the performance monitoring component can be turned on or off while the application is in execution.

This would enhance the ability to understand performance bottlenecks that might be encountered at any stage of a component's execution. Also, the performance monitoring and instrumentation components could be configured easily to collect various performance data - memory usage patterns, function and loop execution profiles, specific event traces etc. In the case of parallel SCIRun2 components that use MPI, PERFume makes use of MPI's performance monitoring interface [14] .

The data gathered using the performance monitoring and instrumentation components are stored in a database managed by the PERFume database component. The database component basically provides for storing and retrieving the TAU-generated data along with the details of the environment such as the interacting component group and so forth. Performance data can be stored for either individual components or a group of components and can be accessed using the component or component net ID. In the latter case, we store a series of wait times – the time spent by each component in waiting for the results of another. This data would provide insight into remodeling individual or a set of components to provide more parallelism. Also by default, we also record the time spent in waiting for the services provided by the framework (memory allocation, communication library). This data is useful for understanding the overall as well as component specific performance of the framework services.

Performance modeling is designed to be offline using the data stored in the performance database. The component developer, utilizing knowledge of the algorithms employed by the component, can provide basic semantic details and the theoretical complexity of the algorithm to the performance modeler using a performance model specification file (in XML). Using this detail as well as historical performance of the component obtained from the database, the modeler predicts the performance of the component. Since performance of a component is tied very much to the services provided by the framework, the modeling component incorporates units that model the different services provided by the SCIRun2 framework (the communication library, memory allocator). The visualization component can obtain performance data from either the monitoring components or the database or the modeler. The source of the input also determines the semantics and complexity of the visualization module. The visualizations during run-time monitoring need to be fast and dynamically configurable, while those bound to the modeler or database need to operate on large scale data. Most of the work in performance modeling and visualization is currently in progress.

7. Conclusions and Future Work

We have presented an overview of the SCIRun2 component framework. SCIRun2 integrates multiple component models into a single visual Problem Solving Environment and builds bridges between components of different component models. In this way, a number of tools can be combined into a single environment without requiring global adoption of a common underlying component model. We have also described a parallel component architecture utilizing the Common Component Architecture, combined with distributed objects and parallel MxN array redistribution. By incorporating performance measuring and modeling tools we are equipping the user with the ability to get better performance out of an application.

A prototype of the SCIRun2 framework has been developed, and we are using this framework for a number of applications in order to demonstrate the SCIRun2 features. Future applications will rely more on the system, and will facilitate joining many powerful tools, such as the SCI Institutes' interactive ray-tracing system [17] and Uintah [8]. Large scale applications are under construction and are beginning to take advantage of the capabilities of SCIRun2.

Support for additional component-models, such as Vtk, CORBA, and possibly others, will be added in the future. PERFume components are an ongoing project, and we are working towards further automation of the bridging mechanisms.

References

[1] B. A. Allan, R. C. Armstrong, A. P. Wolfe, and J. Ray. The CCA core specification in a distributed memory SPMD framework. *Concurrency Computation*, 14:1–23, 2002.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, 1999.

[3] Enterprise Java Beans. http://java.sun.com/products/javabeans, 2003.

[4] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and Yechuri M. A component based services architecture for building distributed applications. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, 2000.

[5] Common Component Architecture Forum. see www.cca-forum.org.

[6] K. Damevski. Parallel component interaction with an interface definition language compiler. Master's thesis, University of Utah, 2003.

[7] K. Damevski and S. Parker. Parallel remote method invocation and m-by-n data redistribution. In *Proceedings of the 4th Los Alamos Computer Science Institute Symposium*, 2003.

[8] J. Davison de St. Germain, John McCorquodale, Steven G. Parker, and Christopher R. Johnson. Uintah: A Massively Parallel Problem Solving Environment. In *Proceedings of the Ninth IEEE International Symposium on High Performance and Distributed Computing*, August 2000.

[9] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[10] C. Johnson and S. Parker. The SCIRun Parallel Scientific Compouting Problem Solving Enviroment. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[11] C.R. Johnson and S.G. Parker. Applications in computational medicine using SCIRun: A computational steering programming environment. In H.W. Meuer, editor, *Supercomputer '95*, pages 2–19. Springer-Verlag, 1995.

[12] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 2001.

[13] Advanced Computing Laboratory. TAU: Tuning and Analysis Utilities. Technical report, Los Alamos National Laboratory, 1999.

[14] The Ruby Language. http://www.ruby-lang.org/en, 2004.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995.

[16] Component Object Model. http://www.microsoft.com/com/tech/com.asp, 2003.

[17] OMG. *The Common Object Request Broker: Architecture and Specification. Revision 2.0*, June 1995.

[18] S. Parker, M. Parker, Y. Livnat, P. Sloan, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, July-September 1999.

[19] S. G. Parker. *The SCIRun Problem Solving Environment and Computational Steering Software System*. PhD thesis, University of Utah, 1999.

[20] S.G. Parker, D.M. Beazley, and C.R. Johnson. Computational steering software systems and strategies. *IEEE Computational Science and Engineering*, 4(4):50–59, 1997.

[21] S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95*. IEEE Press, 1995.

[22] S.G. Parker, D.M. Weinstein, and C.R. Johnson. The SCIRun computational steering software system. In E. Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 1–44. Birkhauser Press, 1997.

[23] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit, An Object-Oriented Approach to 3-D Graphics*. Prentice Hall PTR, 2nd edition, 2003.