Chapter 15

Integrating Component-Based Scientific Computing Software

Steven G. Parker, Keming Zhang, Kostadin Damevski, and Chris R. Johnson

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 users. Software application developers typically connect a group of components from a component repository, connecting them to create a single application.

SCIRun[1] is a scientific PSE that allows the interactive construction and steering of large-scale scientific computations [20, 19, 21, 11, 23, 25, 10]. A scientific application is constructed by connecting computational, modeling, and visualization elements [8]. This application 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.

Problem solving environments, such as SCIRun, often employ component technology to bring a variety of computational tools to an engineer or scientist for solving a computational problem. In this scenario, the tools should be readily available and simple to combine to create an application. However, these PSEs typically use a single-component model (such as Java Beans, Microsoft COM, CORBA, or CCA) or employ one of their own design. As a result, components designed for one PSE cannot be easily reused in another PSE or in a stand-alone program. Software developers must *buy in* to a particular component model and produce components for one particular system. Users must typically select a single system or face the challenges of manually managing the data transfer between multiple (usually) incompatible systems.

---

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

SCIRun2 [27], currently under development, addresses these shortcomings through a *meta-component model*, allowing support for disparate component-based systems to be incorporated into a single environment and managed through a common user-centric visual interface.

In this chapter, section 15.1 discusses the SCIRun and BioPSE PSEs. Other scientific computing component models are discussed in section 15.2. The remainder of the chapter discusses the design of SCIRun2, including a discussion of meta-components, support for distributed computing, and parallel components. We present conclusions and future work in section 15.6.

15.1   SCIRun and BioPSE

SCIRun is a scientific PSE that allows the interactive construction and steering of large-scale scientific computations [20, 19, 21, 11, 23]. A scientific application is constructed by connecting computational elements (modules) to form a program (network), as shown in Figure 15.1. The program may contain several computational elements as well as several visualization elements, all of which work together in orchestrating a solution to a scientific
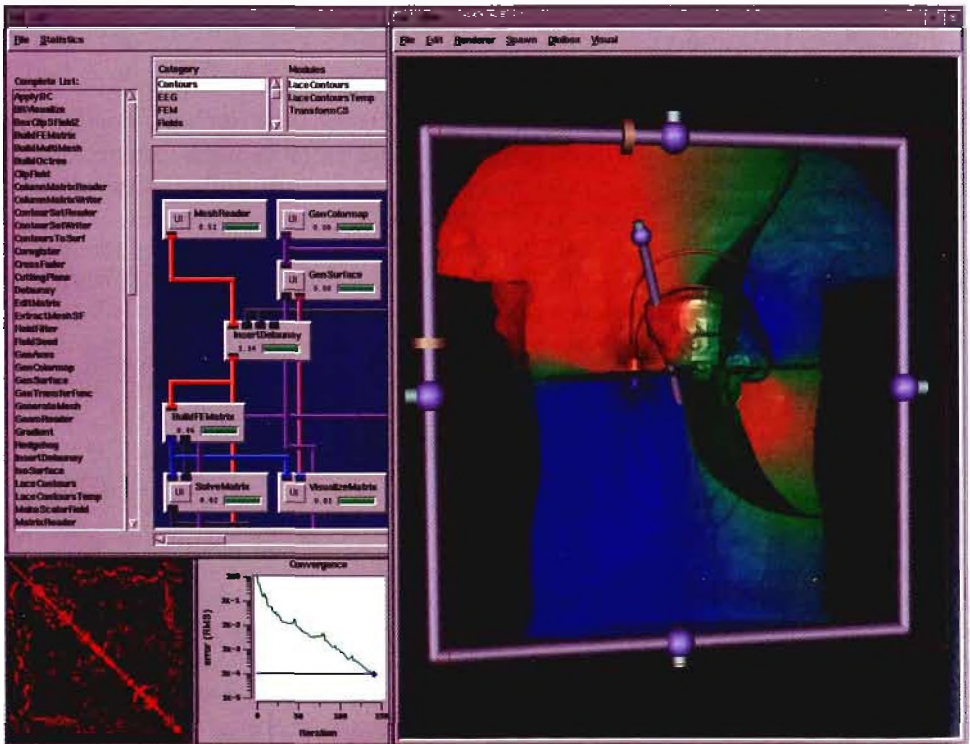

Figure 15.1. *The SCIRun PSE, illustrating a 3D finite element simulation of an implantable cardiac defibrillator.*

Figure 15.2. *BioPSE neural source localization network. The optimal dipole source is recovered using a multistart optimization algorithm.*

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 architectures from the desktop to large supercomputers.

BioPSE [10] is a PSE based on SCIRun that is targeted to biomedical computing research problems. BioPSE provides the modeling, simulation, and visualization modules; data structures; data converters; and application networks for bioelectric field applications. With these components, researchers can investigate biomedical research issues, such as bioelectric fields produced from epileptic seizures in the brain to atrial arrhythmias in the heart.

An example electroencephalography (EEG) neural source localization application is shown in Figures 15.2 and 15.3. Figure 15.2 contains the dataflow network that implements the inverse EEG application. At the top of the network, the input data files are loaded; these include the finite element mesh that defines the geometry and electrical conductivity properties of the model and a precomputed lead-field matrix that encodes the relationship between electric sources in the domain and the resulting potentials that would be measured at the electrodes. Further down in the network, we have a set of modules that optimize the dipole location in order to minimize the misfit between the measured potentials from the electrodes and the simulated potentials due to the dipole. Finally, we have visualization and
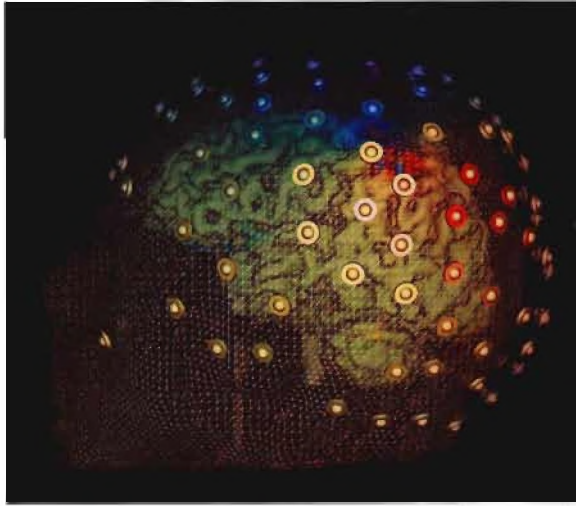
Figure 15.3. *Visualization of the iterative source localization. The voltages of the true solution (disks) and the computed solution (spheres) are qualitatively compared at the electrode positions as the optimization (shown as arrows) converges on a neural source location. The solution misfit can be qualitatively interpreted by pseudocolored voltages at each electrode.*

rendering modules, which provide interactive feedback to the user. The visualization that accompanies this network is shown in Figure 15.3. The potentials that were measured at the electrodes on the scalp are rendered as pseudocolored disks; the potentials originating from the simulated dipole source are shown as pseudocolored spheres embedded within the disks. The rainbow colors of the disks and spheres correspond to voltages, with red mapping to positive potentials, blue mapping to negative potentials, and green mapping to ground. The difference in the color of the sphere and the color of the disk at any particular electrode indicates the misfit between the measured and simulated potentials at that site. The dipoles that are iteratively approaching the true dipole location are shown as gray and blue arrows, and the outside of the head model has been rendered with wire-frame cylinders.

PowerApps

Historically, one of the major hurdles to SCIRun becoming a tool for the scientist as well as the engineer has been SCIRun's dataflow interface. While visual programming is natural for computer scientists and engineers, who are accustomed to writing software and building algorithmic pipelines, it can be overly cumbersome for many application scientists. Even when a dataflow network implements a specific application (such as the forward bioelectric field simulation network provided with BioPSE and detailed in the BioPSE tutorial), the user interface (UI) components of the network are presented to the user in separate UI windows, without any semantic context for their settings. For example, SCIRun provides file browser user interfaces for reading in data. However, on the dataflow network all the file browsers

have the same generic presentation. Historically, there has not been a way to present the filename entries in their semantic context, for example, to indicate that one entry should identify the electrodes input file and another should identify the finite element mesh file.

A recent release of BioPSE/SCIRun (in October 2003) addressed this shortcoming by introducing PowerApps. A PowerApp is a customized interface built atop a data flow application network. The data flow network controls the execution and synchronization of the modules that comprise the application, but the generic user interface windows are replaced with entries that are placed in the context of a single application-specific interface window.

BioPSE contains a PowerApp called BioFEM. BioFEM has been built atop the forward finite element network and provides a useful example for demonstrating the differences between the dataflow and PowerApp views of the same functionality. In Figure 15.4, the dataflow version of the application is shown: the user has separate interface windows for controlling different aspects of the simulation and visualization. In contrast, the PowerApp version is shown in Figure 15.5: here, the application has been wrapped up into a single
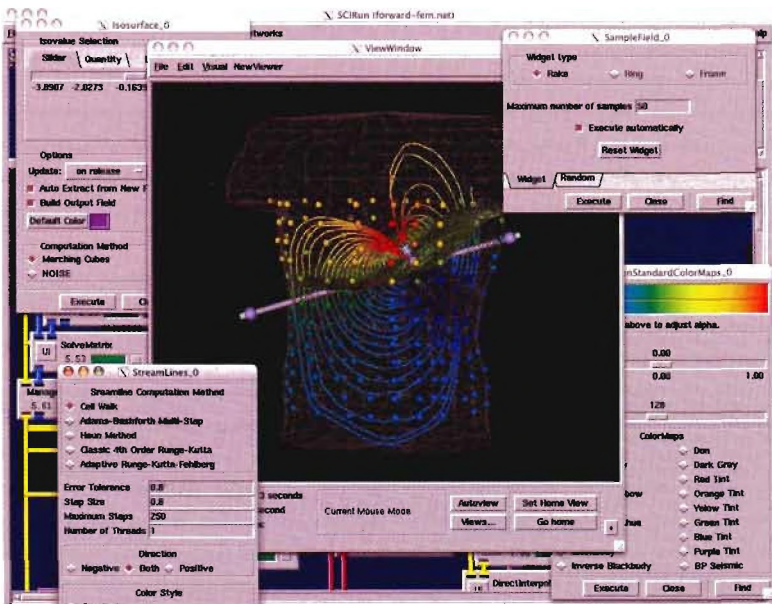
Figure 15.4. *BioPSE dataflow interface to a forward bioelectric field application. The underlying dataflow network implements the application with modular interconnected components called modules. Data are passed between the modules as input and output parameters to the algorithms. While this is a useful interface for prototyping, it can be nonintuitive for end users; it is confusing to have a separate user interface window to control the settings for each module. Moreover, the entries in the user interface windows fail to provide semantic context for their settings. For example, the text-entry field on the SampleField user interface that is labeled "Maximum number of samples" is controlling the number of electric field streamlines that are produced for the visualization.*
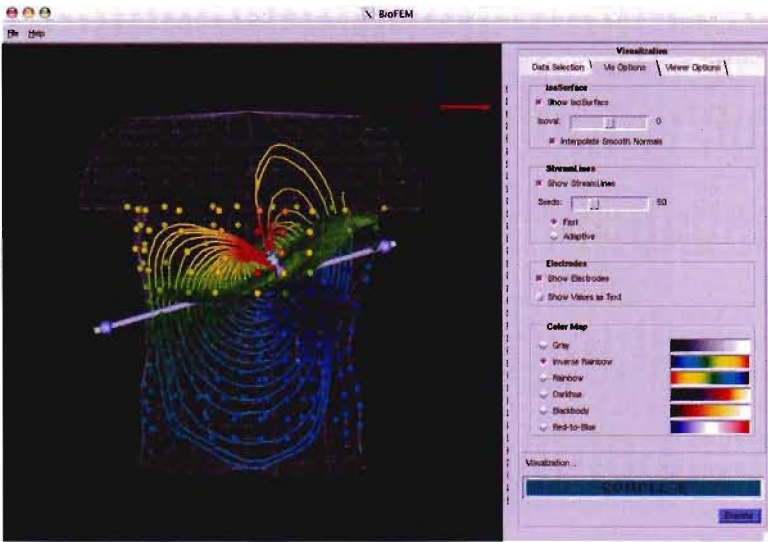
Figure 15.5. *The BioFEM custom interface. Although the application is the functionality equivalent to the data flow version shown in Figure* 15.4, *this PowerApp version provides an easier-to-use custom interface. Everything is contained within a single window. The user is lead through the steps of loading and visualizing the data with the tabs on the right; generic control settings have been replaced with contextually appropriate labels, and application-specific tooltips (not shown) appear when the user places the cursor over any user interface element.*

interface window, with logically arranged and semantically labeled user interface elements composed within panels and notetabs.

In addition to bioelectric field problems, the BioPSE system can also be used to investigate other biomedical applications. For example, we have wrapped the tensor and raster data processing functionality of the Teem toolkit into the Teem package of BioPSE, and we have used that increased functionality to develop the BioTensor PowerApp, as seen in Figure 15.6. BioTensor presents a customized interface to a 140-module data flow network. With BioTensor the user can visualize diffusion weighted imaging (DWI) datasets to investigate the anisotropic structure of biological tissues. The application supports the import of DICOM and Analyze datasets and implements the latest diffusion tensor visualization techniques, including superquadric glyphs [13] and tensorlines [26] (both shown).

15.2   Components for scientific computing

A number of component models have been developed for a wide range of software applications. Java Beans[9], 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)[17], a software architecture that allows applications to
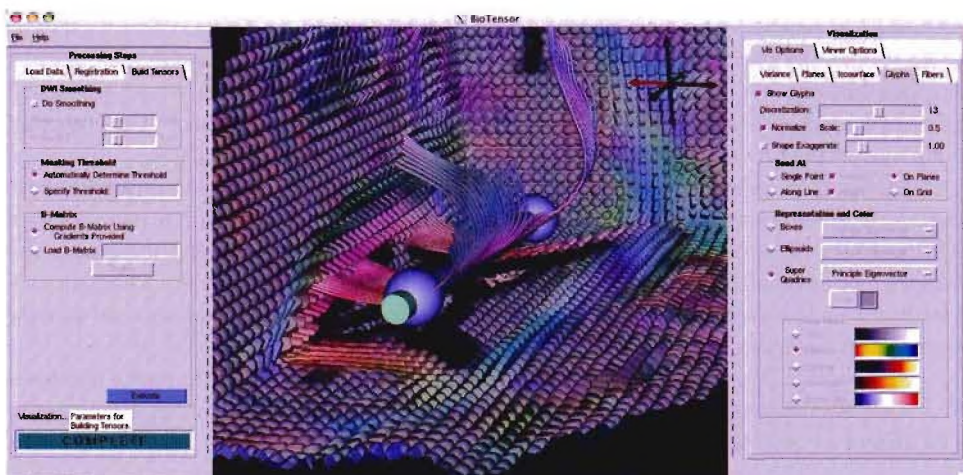
Figure 15.6. *The BioTensor PowerApp. Just as with BioFEM, we have wrapped up a complicated data flow network into a custom application. In the left panel, the user is guided through the stages of loading the data, coregistering MRI diffusion weighted images, and constructing diffusion tensors. On the right panel, the user has controls for setting the visualization options. In the rendering window in the middle, the user can render and interact with the dataset.*

be built from binary software components on the Windows platform. The Object Management Group (OMG) developed the common object request broker architecture (CORBA) [18], 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 CCA forum, a group of researchers from several DoE national laboratories and academic institutions, has defined a standard component architecture [1] 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.

CCA is discussed in more detail in Chapter 14, but we present an overview here. The CCA model consists of a framework and an 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 port is represented by an interface, while interfaces are specified through a SIDL. A 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, or Python. The Babel [14] compiler group is working on creating this support for different languages within CCA.

SCIRun2 is a new software framework that combines CCA compatibility with connections to other commercial and academic component models. SCIRun2 is based on the SCIRun [12] infrastructure and the CCA specification. It utilizes parallel-to-parallel remote method invocation to connect components in a distributed memory environment and is multithreaded 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 as follows:

1. SCIRun2 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. SCIRun2 accommodates several useful component models. In addition to CCA components and SCIRun dataflow modules, CORBA components, Microsoft COM components, ITK, and Vtk[24] modules will be supported in SCIRun2.

3. SCIRun2 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. SCIRun2 supports distributed computing. Components created on different computers can work together through a network and build high performance applications.

5. SCIRun2 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 [3] that potentially can be used by many component models.

Overall, SCIRun2 provides a broad approach that will allow scientists to combine a variety of tools for solving a particular computational 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, a goal motivated by the needs of our biomedical and other scientific users.

15.3   Metacomponent model

Systems such as Java Beans, COM, CORBA, CCA, and others successfully employ a component-based architecture to allow users to rapidly assembly computational tools in a single environment. However, these systems typically do not interact with one another in a straightforward manner, and it is difficult to take components developed for one system and redeploy them in another. Software developers must *buy in* to a particular model and
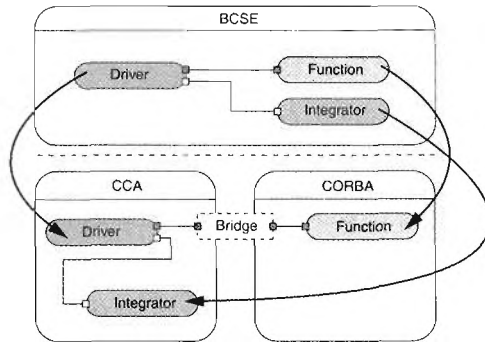
Figure 15.7. *Components of different models cooperate in SCIRun2.*

produce components for one particular system. Users must typically select a single system or face the challenges of manually managing the data transfer between multiple (usually) incompatible systems. SCIRun2 addresses these shortcomings through the meta-component model, allowing support for disparate component-based systems to be incorporated into a single environment and managed through a common user-centric visual interface. Furthermore, many systems that are not traditionally thought of as component models but that have well-designed, regular structures can be mapped to a component model and manipulated dynamically. SCIRun2 combines support for CCA components, "old-style" SCIRun data flow components, and we are planning support for CORBA, COM, and VTK. As a result, SCIRun2 can utilize SCIRun components, CCA components, and other software components in the same simulation.

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 15.7 demonstrates a simple example of 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 the 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. 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 allows 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, a 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 in which 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 by an eRuby (embedded Ruby) template document. eRuby templates are text files that can be augmented by Ruby [15] scripts. The Ruby scripts are useful for situations where the translation requires more sophistication than regular text (such as control structures or additional 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.

The only other source of information is the interface of the ports we want to bridge (usually expressed in an IDL file). The bridge compiler accepts commands that specify a mapping between incompatible interfaces, where the interfaces between the components differ in various names or types but not functionality. Using a combination of the plugin and the interface augmented with mapping commands, 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.

Figure 15.8 shows a more complex example that is motivated by the needs of a biological application. This example works very much like the last: the framework manages components from several different component models through the meta-model interface. Components from the same model interact with each other natively and interact with components in other models through bridges. Allowing components to communicate with each other through their native mechanisms ensures that no performance bottlenecks are introduced and that the original semantics are preserved.

15.4   Distributed computing

SCIRun2 provides support for distributed objects based on remote method invocation (RMI). 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
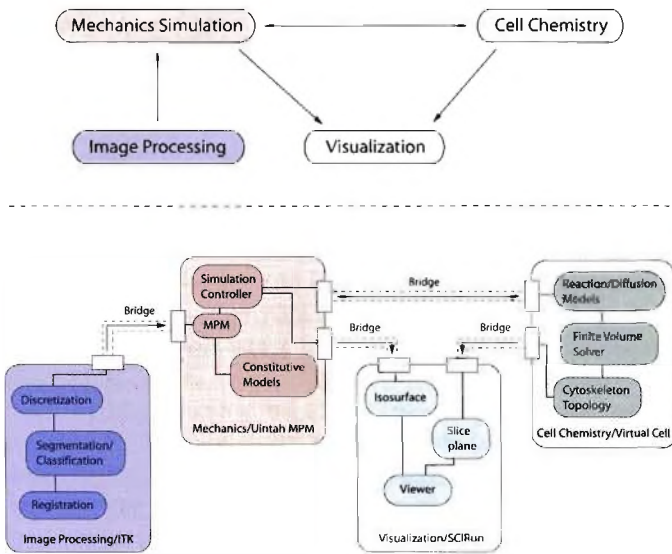
Figure 15.8. *A more intricate example of how components of different models cooperate in SCIRun2. The application and components shown are from a realistic (albeit incomplete) scenario.*

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: server proxy and 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 methods 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, 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 [7]. 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. Nonpointer arguments, such as integers, fixed sized arrays and strings (character arrays), are marshaled by the proxy into a message in the order in which 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 rethrown by the proxy.

15.5   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 they 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 [16] or an equivalent message-passing library.

PComs communicate with each other through CCA-style RMI ports. We developed a prototype parallel component infrastructure [5, 2] 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 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 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; see [3] for details. If an M-member PCom A obtains a pointer ptr pointing to an 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 $0, 1, \ldots$, M-1 for A and $0, 1, \ldots$, N-1 for B. If $M = N$, then the ith member component of A call foo(args) on the ith component of B. But if M < N, then we "extend" the A's to $0, 1, 2, \ldots$, M, $0, 1, 2, \ldots$ M, $\ldots$ 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 15.9 shows an example of this case with M = 3 and N = 5. If M > N, we extend component B's set to $0, 1, \ldots, N, 0, 1, \ldots, N, \ldots,$ M-1 and only the first N member components of B are actually called; the rest are not called but simply return the result. We rely on collective semantics
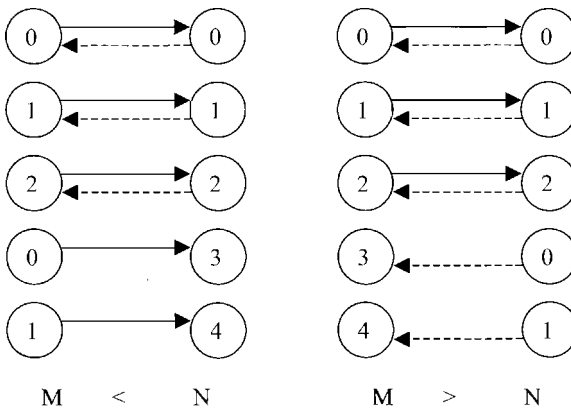

Figure 15.9. *MxN method invocation, with the caller on the left and the callee on the right. In the left scenario, the number of callers is fewer than the number of callees, so some callers make multiple method calls. In the right, the number of callees is fewer, so some callees send multiple return values.*

from the components to ensure consistency without requiring global synchronization. The right panel of Figure 15.9 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. A multidimensional 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 a 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 subsetting 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 on either shared memory multiprocessors 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 an N-node cluster is essentially a set of loaders, each running on a node. When the user requests to create a parallel component, the PCL instantiates a parallel component on its processes (or nodes) and passes a distributed pointer to the SCIRun2 framework. PCLs are responsible for creating and destroying components running on their nodes, but they do not maintain the port connections. The SCIRun2 framework maintains all component status and port connections.

Supporting threads and MPI together can be difficult. MPI provides a convenient communication among the processes in a cluster. However, if any process has more than one thread and the MPI calls are made in those threads, the MPI communication may break because MPIs distinguish only processes, not threads. The MPI interface allows an implementation to support threads but does not require it. Most MPI implementations are not threadsafe. We provide support for both threadsafe and nonthreadsafe MPI implementations so that users can choose any available MPI.

A straightforward way to support nonthreadsafe MPIs is to globally order the MPI calls such that no two MPI calls are executed at the same time. We implemented a distributed lock, which has two interfaces:

```
PRMI::lock()
PRMI::unlock()
```

The distributed lock is just like a mutex, but it is collective with respect to all MPI processes in a cluster. The critical section between PRMI::lock() and PRMI::unlock() can be obtained by only one set of threads in different MPI processes. The users must call PRMI::lock() before any MPI calls and call PRMI::unlock() after to release the lock. More than one MPI calls can be made in the critical section. In this way only one set of threads (each from a MPI process) can make MPI calls at one time. Additionally, the overhead of acquiring and releasing this lock is very high because it requires a global synchronization. However, in some cases this approach is necessary for supporting the multi-threaded software framework in an environment where a thread-safe MPI is no available.

It is fairly easier to support threadsafe MPI. Our approach is to create a distinct MPI communicator for the threads that communicate with each other and restrict that those
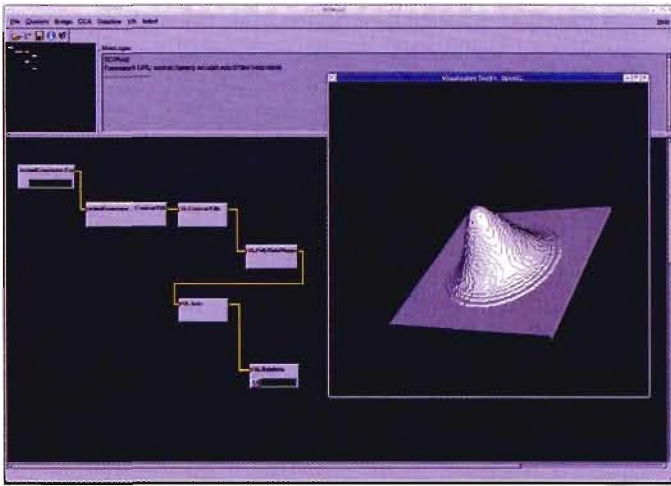
Figure 15.10. *Components of different models cooperate in SCIRun2.*

threads can use only that communicator for MPI communication. The special communicators are created by the PCL and can be obtained through a framework service interface. The threadsafe MPI allows multiple MPI calls executed safely at the same time, and the designated communicators help to identify the group of threads which initiated the MPI calls.

An efficient mechanism allows parallel components to efficient coordinate around error conditions [4].

Figure 15.10 shows a SCIRun2 application that uses bridging to Vtk visualization components. SCIRun2 is currently under development, but we expect a public release in the near future.

15.6 Conclusions and future work

We presented the SCIRun, BioPSE, and SCIRun2 problem solving environments for scientific computing. These systems all employ software components to encapsulate computational functionality into a reusable unit. SCIRun and BioPSE are open source, have biannual public releases, and are used by a number of end users for a variety of different computational applications.

Additionally, we presented an overview of the new 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 that can be used in SCIRun2.

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 [22] and the Uintah [6] parallel, multiphysics system. Additional large-scale computational 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.

Acknowledgments

Bibliography

[1] 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.

[2] F. BERTRAND, R. BRAMLEY, K. DAMEVSKI, D. BERNHOLDT, J. KOHL, J. LARSON, AND A. SUSSMAN, *Data redistribution and remote method invocation in parallel component architectures*, in Proceedings of The 19th International Parallel and Distributed Processing Symposium, Denver, CO, 2005.

[3] 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, Los Alamos, NM, 2003.

[4] K. DAMEVSKI AND S. PARKER, *Imprecise exceptions in distributed parallel components*, in Proceedings of the 10th International Euro-Par Conference, vol. 3149 of Lecture Notes in Computer Science, August/September 2004, Springer-Verlag, Berlin, New York, pp. 108–116.

[5] K. DAMEVSKI, *Parallel component interaction with an interface definition language compiler*, Master's thesis, University of Utah, Salt Lake City, UT, 2003.

[6] J. D. DE ST. GERMAIN, J. MCCORQUODALE, S. G. PARKER, AND C. 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.

[7] I. FOSTER, C. KESSELMAN, AND S. TUECKE, *The Nexus approach to integrating multithreading and communication*, Journal of Parallel and Distributed Computing, 37 (1996), pp. 70–82.

[8] C. HANSEN AND C. JOHNSON, EDS., *The Visualization Handbook*, Elsevier, Amsterdam, 2005.

[9] JAVABEANS, http://java.sun.com/products/javabeans, 2003.

[10] C. JOHNSON, R. MACLEOD, S. PARKER, AND D. WEINSTEIN, *Biomedical computing and visualization software environments*, Communications of the ACM, 47 (2004), pp. 64–71.

[11] C. JOHNSON AND S. PARKER, *Applications in computational medicine using SCIRun: A computational steering programming environment*, in Supercomputer '95, H. Meuer, ed., Springer-Verlag, Berlin, New York, 1995, pp. 2–19.

[12] C. JOHNSON AND S. PARKER, *The SCIRun Parallel Scientific Compouting Problem Solving Enviroment*, in Talk presented at the 9th SIAM Conference on Parallel Processing for Scientific Computing, 1999.

[13] G. KINDLMANN, *Superquadric tensor glyphs*, in The Joint Eurographics – IEEE TCVG Symposium on Visualization, May 2004, pp. 147–154.

[14] 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, CD-ROM.

[15] T. R. LANGUAGE, http://www.ruby-lang.org/en, 2004.

[16] MESSAGE PASSING INTERFACE FORUM, *MPI: A Message-Passing Interface Standard*, June 1995.

[17] C. O. MODEL, http://www.microsoft.com/com/tech/com.asp, 2003.

[18] OMG, *The Common Object Request Broker: Architecture and Specification. Revision 2.0*, June 1995. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[19] S. G. PARKER, *The SCIRun Problem Solving Environment and Computational Steering Software System*, Ph.D. thesis, University of Utah, Salt Lake City, UT, 1999.

[20] S. PARKER, D. BEAZLEY, AND C. JOHNSON, *Computational steering software systems and strategies*, IEEE Computational Science and Engineering, 4 (1997), pp. 50–59.

[21] S. PARKER AND C. JOHNSON, *SCIRun: A scientific programming environment for computational steering*, in Supercomputing '95, IEEE Press, Los Alamitos, CA, 1995.

[22] S. PARKER, M. PARKER, Y. LIVNAT, P. SLOAN, AND P. SHIRLEY, *Interactive ray tracing for volume visualization*, IEEE Transactions on Visualization and Computer Graphics, 5 (1999).

[23] S. PARKER, D. WEINSTEIN, AND C. JOHNSON, *The SCIRun computational steering software system*, in Modern Software Tools in Scientific Computing, E. Arge, A. Bruaset, and H. Langtangen, eds., Birkhauser Press, Basel, 1997, pp. 1–44.

[24] W. SCHROEDER, K. MARTIN, AND B. LORENSEN, *The Visualization Toolkit, An Object-Oriented Approach to 3-D Graphics*, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 2003.

[25] D. WEINSTEIN, S. PARKER, J. SIMPSON, K. ZIMMERMAN, AND G. JONES., *Visualization in the scirun problem-solving environment*, in The Visualization Handbook, C. Hansen and C. Johnson, eds., Elsevier, Amsterdam, 2005, pp. 615–632.

[26] D. WEINSTEIN, O. POTNIAGUINE, AND L. ZHUKOV, *A comparison of dipolar and focused inversion for EEG source imaging*, in Proc. 3rd International Symposium on Noninvasive Functional Source Imaging (NFSI), Innsbruck, Austria, September 2001, pp. 121–123.

[27] K. ZHANG, K. DAMEVSKI, V. VENKATACHALAPATHY, AND S. PARKER, *SCIRun2: A CCA framework for high performance computing*, in Proceedings of The 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments, April 2004, Santa Fe, NM.