1 The SCIJump Framework for Parallel an Distributed Scientific Computing

STEVEN G. PARKER, KOSTADIN DAMEVSKI, AYLA KHAN, ASHWIN SWAMINATHAN, CHRISTOPHER R. JOHNSON

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, Utah

1.1 INTRODUCTION

In recent years, software component technology has been a successful methodology for large-scale commercial software development. Component technology combines a set of frequently used functions in an easily reusable component and makes the implementation transparent to the users or other components. Developers create new software applications by connecting groups of components. Component technology is becoming increasingly popular for large-scale scientific computing to help tame software complexity resulting from coupling multiple disciplines, multiple scales, and/or multiple physical phenomena.

In this chapter, we discuss our SCIJump Problem Solving Environment (PSE), that builds on its successful predecessor SCIRun and the DOE Common Component Architecture (CCA) scientific component model. SCIJump provides distributed computing, parallel components and the ability to combine components from several component models in a single application. These tools provide the ability to use a larger set of computing resources to solve a wider set of problems. For even larger applications that may require thousands of computing resources and tens of thousands of component instances, we present our prototype scalable distributed component framework technology called CCALoop. When the technology described in CCALoop matures, it will be included in SCIJump.

SCIRun is a scientific PSE that allows interactive construction and steering of large-scale scientific computations [25, 27, 26, 18, 17, 19]. A scientific application is constructed by connecting computational elements (modules) to form a program (network), as shown in Figure 1.1. The program may contain several computational

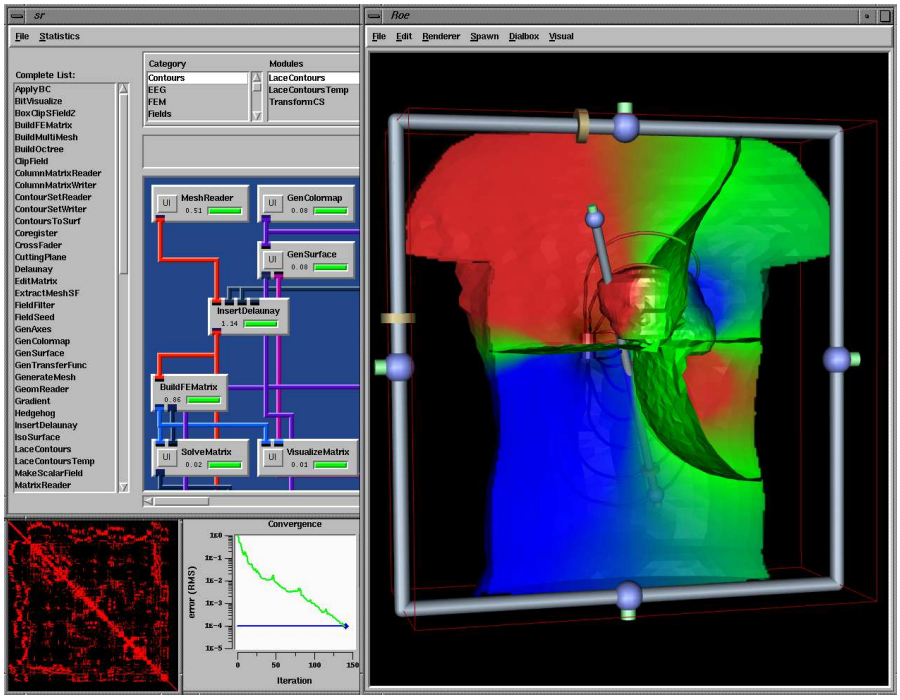

Fig. 1.1 The SCIRun PSE, illustrating a 3D finite element simulation of an implantable cardiac defibrillator.

elements as well as several visualization elements, all of which work together in orchestrating a solution to a scientific problem. SCIRun is designed to facilitate large-scale scientific computation and visualization on a wide range of architectures from the desktop to large supercomputers. Geometric inputs and computational parameters may be changed interactively, and the interface provides immediate feedback to the investigator.

The CCA model consists of a framework and an expandable set of components. The framework is a workbench for building, connecting and running 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** and **provides**. 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, which is specified through the Scientific Interface Definition Language (SIDL). SIDL is compiled to specific language bindings using compilers such as Babel [12], which supports a number of languages such as C/C++, Java, Fortran, Python etc.

SCIJump is a framework built on SCIRun [11] infrastructure that combines CCA compatible architecture with hooks for other commercial and academic component

models. It provides a broad approach that will allow scientists to combine a variety of tools for solving a particular problem. The overarching design goal of SCIJump is to provide the ability for a computational scientist to use the right tool for the right job. SCIJump utilizes parallel-to-parallel remote method invocation (RMI) 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. A few of the design goals of SCIJump are:

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

2. SCIJump accommodates several useful component models. In addition to CCA components and SCIRun Dataflow modules, CORBA components, and Vtk[20] modules are supported in SCIJump, which can be utilized in the same simulation.

3. SCIJump 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. SCIJump supports distributed computing. Components created on different computers can be networked to build high performance applications.

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

Figure 1.2 shows a SCIJump application that demonstrates bridging multiple component models. SCIJump is currently released under the MIT license and can be obtained at http://www.sci.utah.edu.

As scientific computing experiences continuous growth of the size of simulations, component frameworks intended for scientific computing need to handle more components and execute on numerous hardware resources simultaneously. Our distributed component framework CCALoop presents a novel design that supports scalability in both number of components in the system and distributed computing resources. CCALoop also incorporates several other beneficial design principles for distributed component frameworks such as fault-tolerance, parallel component support and multiple user support.

In this chapter, Section 1.2 discusses meta-components, while Section 1.3 and Section 1.4 explain the support SCIJump provides for distributed computing and parallel components. The design of our highly scalable component framework CCALoop is discussed in Section 1.5. We present conclusions and future work in Section 1.6.

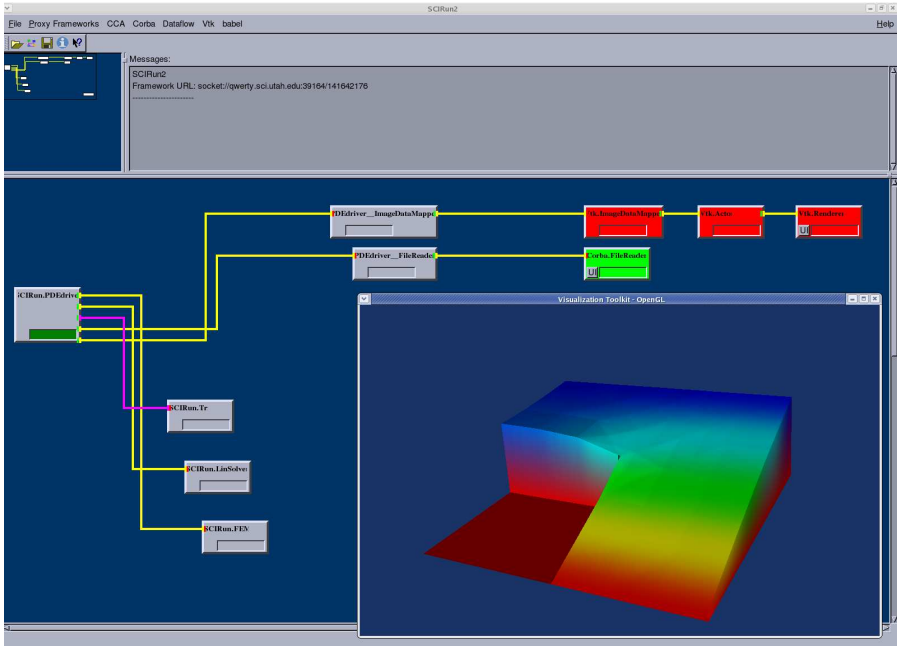

Fig. 1.2 Components from different models cooperate in SCIJump

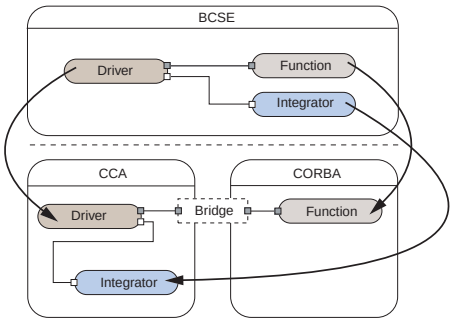

Fig. 1.3 Bridging components from different models in SCIJump

1.2 META-COMPONENT MODEL

Component software systems for scientific computing provide a limited form of interoperability, typically working only with other components that implement the same underlying component model. As such, we propose a next generation concept of meta-components where software components can be manipulated in a more abstract manner, providing a plug-in architecture for component models and bridges between them, allowing for interoperability between different component models. These abstract, meta-components are manipulated and managed by the SCIJump framework, while concrete, standard component models perform the actual work. Thus components implemented with disparate component models can be orchestrated together. As an example of a multi-component system, we have used this system to connect components from SCIRun, the Visualization Toolkit (Vtk), and the CCA into a single application (see Figure 1.2).

The success of Java Beans, COM, CORBA and CCA stems from allowing users to rapidly assemble computational tools from components 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 re-deploy them in another. Software developers must *buy in* to a particular 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. SCIJump 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.

Figure 1.3 demonstrates a simple example of how SCIJump bridges different component models. Two CCA components (**Driver** and **Integrator**) and one CORBA component (**Function**) are created in the SCIJump framework. In this simple exam-

ple, the Driver is connected to both the Function and Integrator. Inside SCIJump, 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 allow a connection from a component of a different model, so a bridge component is created instead. Bridges belong to a special internal component model used to build connections between components of different component models. In this example, Bridge has two ports: one CCA port and one CORBA port allowing it to be connected to both the CCA component and the CORBA component. The CORBA invocation is converted to a 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 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.

A generalized translation between the component models is needed to automatically generate a bridge component. Typically, a software engineer determines how two particular component models will interact; this task can require creating methods of data and controlling translation between the two models, which can be quite difficult in some scenarios. The software engineer implements the translation as a compiler plugin, which is used as the translation specification as it 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 [13] scripts. Ruby scripts are useful for situations where translation requires more sophistication than regular text (such as control structures or additional parsing). The scripted plugin provides us with better flexibility and more power with the end goal of supporting translation between 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 member 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.

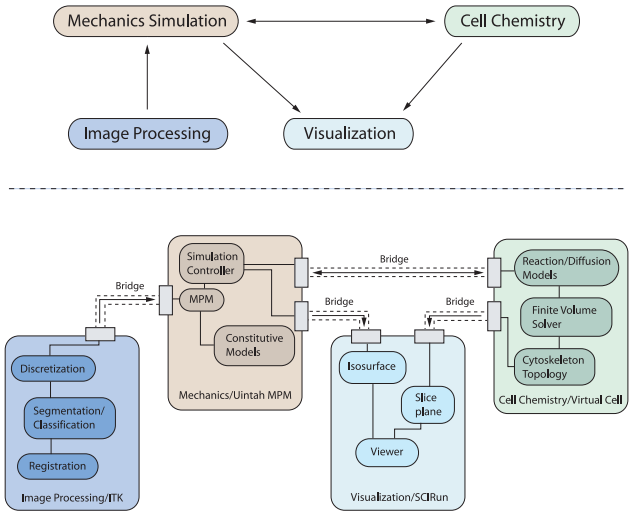

Fig. 1.4 A more intricate example of how components of different models cooperate in SCIJump. The application and components shown are from a realistic (albeit incomplete) scenario.

Figure 1.4 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 native mechanisms ensures that performance bottlenecks are not introduced and that the original semantics are preserved.

1.3 DISTRIBUTED COMPUTING

SCIJump provides support for 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 implements a set of interfaces defined in SIDL that can be referenced remotely. 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. We implemented a straightforward distributed object system by extending the SIDL language and building upon this system for implementing parallel to parallel component connections, which will be demonstrated 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 a user-

made method call to a call by the concrete object. The proxy classes are described in a SIDL file which is parsed by a compiler that recognizes the SIDL extensions to generate the proxy classes. The proxy classes are defined as abstract classes with a set of pure virtual functions. The concrete classes extend 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 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, 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 the Data Transmitter, which is a communication layer used by 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 [8]. 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. An array size is marshaled in the beginning of an array argument so that 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.

1.4 PARALLEL COMPONENTS

This section introduces the CCA parallel component design and discusses issues arising from 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 processes 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 [15] or an equivalent message passing library.

PComs communicate with each other through CCA-style RMI ports. We developed a prototype parallel component infrastructure [3] 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 registers 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 [5] for details.

If a M-member PCom A obtains a pointer ptr pointing to a N-member PCom's B collective port pB. Then $ptr \rightarrow foo(args)$ 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 calls $foo(args)$ 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,

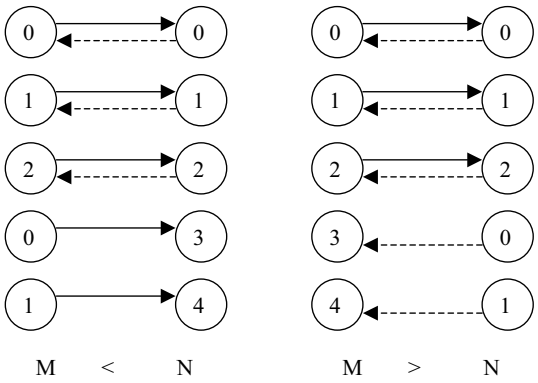

Fig. 1.5 MxN method invocation, with the caller on the left and the callee on the right. In the left scenario, the number of callers is fewer than the numbers of callees, so some callers make multiple method calls. In the right, the number of callees is fewer, so some callees send multiple return values.

... 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 1.5 shows an example of this case with M=3 and N=5. If $M>N$, we "extend" 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 1.5 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 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.

SCIJump provides the mechanism to start a parallel component either on shared memory multi-processors computers, or clusters. SCIJump 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 requests to create a parallel component,

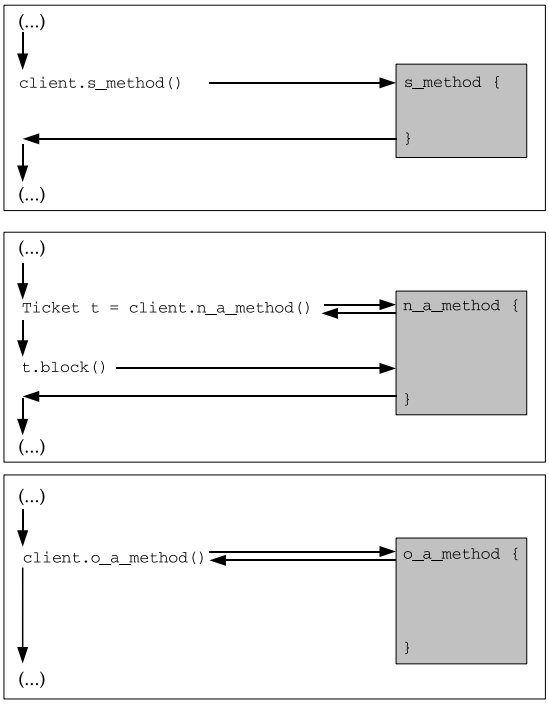

Fig. 1.6 The synchronization options given for (top to bottom) synchronous (s_method), nonblocking asynchronous (n_a_method) and oneway asynchronous (o_a_method) parallel remote method invocation.

the PCL instantiates a parallel component on its processes (or nodes) and passes a distributed pointer to the SCIJump framework. PCLs are responsible for creating and destroying components running on their nodes, but they do not maintain the port connections. The SCIJump framework maintains all component instances 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 MPI distinguishes only processes, not threads. The MPI interface allows an implementation to support threads but does not require it, allowing most MPI implementations not to be threadsafe. We provide support for both threadsafe and non-threadsafe MPI implementations so that users can choose any available MPI.

To address variability of MPI implementations and to optimize parallel-to-parallel component communication performance we provide three different locking behaviors: ”synchronous”, ”nonblocking asynchronous”, and ”oneway asynchronous” as

shown in Figure 1.6. These are currently only provided in conjunction with the Babel [12] SIDL compiler, whose facilities are integrated within SCIJump as a separate component model. When Babel uses a wire protocol without threads, we conservatively allow only ”synchronous” Parallel Remote Method Invocation (PRMI). Similarly, to allow asynchronous PRMI we require a version of the parallel library (e.g. MPI, PVM) that is thread-safe. An example of the progression of the thread of execution for each type of method is given in Figure 1.6 and an explanation of the types of PRMI synchronization option we provide follows:

• **synchronous**. These parallel invocations should work in all system environments so they are built to work without any thread support. Even if threads exist in the wire protocol, the parallel library may not be thread-safe so we synchronize conservatively in order to get the right behavior in all scenarios. The conservative synchronization style of these parallel invocation comes at a significant performance cost.

• **nonblocking asynchronous**. Nonblocking methods in Babel return immediately when invoked and each method returns a ticket that can be used to wait until the call finishes or to periodically check its status. We extend this existing ticket mechanism from Babel nonblocking calls to additionally control our nonblocking asynchronous calls. This enables the caller to use a ticket to check the status of an invocation. For parallel objects receiving invocations from multiple proxies, we provide a guarantee that two method calls coming from the same proxy (caller) will execute in order and will be non-overtaking on the callee.

• **oneway asynchronous**. These methods are defined not to return any variable or error condition to the caller, making them least syncronized. Only ’in’ arguments are allowed so the only guarantee we provide for these calls is ordering and non-overtaking behavior of invocations coming from the same caller.

When designing PRMI, it is crucial to provide the user with a consistent set of guarantees. In the case of most programming languages, execution order is something that a programmer relies on for program correctness. When programming in a middleware that is multithreaded and offers support for an operation such as PRMI, the invocation order given by a user should be preserved. In the past we have identified situations when some reordering may take place [4] with user awareness. However, in implementing general PRMI, we choose not to reorder any invocations and ensure that calls made on the proxy execute in order on the server. An object may receive calls from multiple proxies, and we see no reason why synchronization should preserve inter-proxy order. Therefore, our implementation does not guarantee invocation order from more than one proxy.

Some synchronization is necessary to provide single proxy invocation ordering. Another choice we make is to synchronize on the object (callee) side and never on the caller side. Previous implementations have synchronized on the proxy (caller) side, but in a threaded environment this is not necessary. We eliminate any proxy synchronization for all but the conservative “collective” invocations which have to be deadlock free in the absence of threads.

1.5 CCALOOP

Component frameworks aimed at scientific computing need to support a growing trend in this domain toward larger simulations that produce more encompassing and accurate results. The CCA component model has already been used in several domains, creating components for large simulations involving accelerator design, climate modeling, combustion, and accidental fires and explosions [14]. These simulations are often targeted to execute on sets of distributed memory machines spanning several computational and organizational domains [1]. To address this computational paradigm a collection of component frameworks that are able to cooperate to manage a large, long-running scientific simulation containing many components are necessary.

In each of the CCA-compliant component frameworks, facilities are provided to support the collaboration among several distributed component frameworks. However, existing designs do not scale to larger applications and multiple computing resources. This is due to a master-slave (server-client) communication paradigm. In these systems, the master framework manages all the components and their metadata while also handling communicating with the user through the GUI. The slave frameworks act only as component containers that are completely controlled by the master. This centralized design is simple to implement and its behavior is easy to predict. As component and node numbers grow however, the master framework is quickly overburdened with managing large quantities of data and the execution time of the entire application is affected by this bottleneck. Moreover, as simulation size grows even further, the master-slave design can inhibit the efficiency of future scientific computing applications. We present an alternative design that is highly scalable and retains its performance under high loads. In addition to the scalability problem, the master framework presents a single point of failure that presents an additional liability for long-running applications and simulations.

A component framework's data may be queried and modified by a user through provided user interfaces and by executing components. In both cases it is imperative that the framework is capable of providing quick responses under heavy loads and high-availability to long running applications. The goal of our work is to present distributed component framework design as the solution to several key issues. The system described in this paper is architected to:

1. Scale to a large number of nodes and components.

2. Maintain framework availability when framework nodes are joining and leaving the system and be able to handle complete node failures.

3. Facilitate multiple human users of the framework.

4. Support the execution and instantiation of SPMD parallel components. Our distributed component framework, CCALoop, is self-organizing and uses an approach that partitions the load of managing the components to all of the participating distributed frameworks.

The responsibility for managing framework data is divided among framework nodes by using a technique called Distributed Hash Tables (DHT) [9]. CCALoop uses a hash function available at each framework node that maps a specific component type to a framework node in a randomly distributed fashion. This operation of mapping each component to a node is equally available at all nodes in the system. Framework queries or commands require only one-hop routing in CCALoop. To provide one-hop lookup of framework data we keep perfect information about other nodes in the system, all the while allowing a moderate node joining/leaving schedule and not impacting scalability. We accommodate the possibility that a framework node may fail or otherwise leave the system by creating redundant information and replicating this information onto other frameworks.

1.5.1 Design

Current distributed framework design is inappropriate in accommodating component applications with numerous components that use many computing resources. We implemented a CCA-compliant distributed component framework called CCALoop that prototypes our design for increased framework scalability and fault tolerance. CCALoop scales by dividing framework data storage and lookup responsibilities among its nodes. It is designed to provide fault-tolerance and uninterrupted services on limited framework failure. CCALoop also provides the ability to connect multiple GUIs in order for users to monitor an application from multiple points. While providing these capabilities CCALoop does not add overwhelming overhead or cost to the user and satisfies framework queries with low latency. In this section we will examine the parts that form the structure of this framework. We begin by looking more closely at the tasks and roles of a CCA-compliant component framework.

The main purpose of a component framework is to manage and disseminate data. Some frameworks are more involved, such as Enterprise Java Beans [7], but in this work we focus on the ones in the style of CORBA [16] that do not interfere with the execution of every component. This kind of a component framework performs several important tasks in the staging of an application, but gets out of the way of the actual execution. Executing components may access the framework to obtain data or to manage other components if they choose to, but it is not usually necessary. CCA-compliant frameworks also follow this paradigm as it means low overhead and better performance.

CCA-compliant frameworks store two types of data: static and dynamic. The majority of the data is dynamic, which means that it changes as the application changes. The relatively small amount of static data describes the available components in the system. In a distributed setting, static data consists of the available components on each distributed framework node. The dynamic data ranges from information on instantiated components and ports to results and error messages. A significant amount of dynamic data is usually displayed to the user via a GUI. In our design, we distribute management of framework data without relocating components or forcing the user to instantiate components on a specific resource. The user is allowed to make his or her own decisions regarding application resource usage.

One of the principal design goals of CCALoop is to balance the load of managing component data and answering queries to all participating frameworks. This is done by using the DHT mechanism where each node in the system is assigned a unique identifier in a particular identifier space. This identifier is chosen to ensure an even distribution of the framework identifiers across the identifier space. We provide an operation that hashes each component type to a number in the identifier space. All metadata for a given component is stored at the framework node whose identifier is the successor of the component hash as shown in Figure 1.7(b). Given a random hash function the component data is distributed evenly across the framework nodes. The lookup mechanism is similar to the storage one: to get information about a component, we compute its hash and query the succeeding framework node.

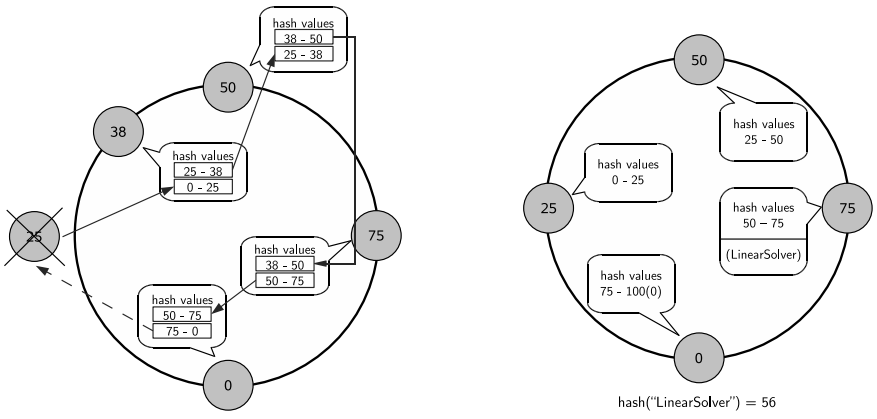

Fig. 1.7 (a) Framework data replication across node's two successors. Shown as node 25 is leaving the system and before any data adjustments have been made. (b) The data responsibilities of the CCALoop framework with four nodes.

1.5.1.1 Loop Structure CCALoop's framework nodes are organized in a ring structure in topological order by their identifier numbers. Each framework node has a pointer to its successor and predecessor, allowing the ring to span the identifier space regardless of how the system may change or how many nodes exists in a given time. CCALoop also facilitates a straightforward way of recovering from node failure, by naturally involving the successor of the failed node to become new successor to the queried identifier. Use of a loop structure with a DHT lookup is commonly found in several peer-to-peer systems such as Chord [21].

Adding framework nodes to the system splits the responsibility for framework data between the joining node and its current owner. It is a two step process that begins at any already connected node in the system. The first step is assigning an

identifier to the joining node that best distributes the nodes across the identifier space. The identifier in the middle of the largest empty gap between nodes is selected based on the queried framework node's information. Later, we will explain how every node in the system contains perfect information about the existence of other nodes in the distributed framework. Apart from a well-chosen identifier, the node is given the address of its predecessor and successor. The second step is for the joining node to inform the predecessor and successor that it is about to join the network so that they can adjust their pointers. This step also resolves conflicts that may occur if two joining frameworks are assigned the same identifier in the first step by two separate nodes. If this conflict occurs, one of the nodes is assigned a new identifier and forced to repeat the second step. Removing nodes in the system has the opposite effect as adding nodes: the successor of the leaving node becomes responsible for the vacated identifier space and associated framework data.

Periodically, nodes invoke a *stabilize()* method that ensures that the successor and predecessor of that node are still alive. If one or both has failed, the node adjusts its predecessor or successor pointer to the next available node. Since perfect loop membership information is kept at every node, finding the next node in the loop is straightforward. The process of updating predecessor and successor pointers ensures that the loop structure is preserved, even when framework nodes leave the system. When a node leaves and this readjustment takes place, the identifier distribution may become unbalanced. This will last until a new framework joins; when it will be instructed to fill the largest current gap in the identifier space.

In order to enable seamless functioning of CCALoop during framework node failure we replicate the data across successor nodes, so that if a framework fails, its successor is able to assume its responsibilities. To be able to handle multiple simultaneous failures we can increase the number of successors to which we replicate the framework data. This incurs a bandwidth cost proportional to the number of replicas. If a node joins or leaves the system, some data readjustment is performed to ensure that the replication factor we have chosen is restored. CCALoop targets the high-performance scientific computing domain, which uses dedicated machines with high availability. Machine failures or intermittent network failures are possible, but infrequent. Because of this, we are content with providing two or three replicas for a particular framework node's data. Figure 1.7(a) shows an example of data replication across two successors as a node leaves the framework.

1.5.1.2 One-hop Lookup An advantage of our distributed framework design is the ability for nodes to contact other nodes directly, or through "one-hop". The one-hop mechanism was initially designed as an alternative to Chord's multi-hop query design [10]. One-hop lookup enables low latency querying, which is important in maximizing performance to components in a distributed framework. In order to support one-hop lookups, full membership awareness is required in the framework; every node needs to keep updated information about all other nodes in the system. There is certainly a cost to keeping this information current in the framework and it is proportional to the joining and leaving (turnover) rate. As with data replication, our expectation is that framework nodes comprising a CCA-compliant distributed

framework will not have the same framework turnover rate as one of the popular file sharing networks, where turnover is an issue. Therefore, our design is unlikely to encounter very high levels of node turnover. When node turnover does occur, our distributed framework would provide graceful performance degradation.

CCALoop uses a multicast mechanism to provide easy and quick dissemination of membership information. This mechanism creates a tree structure to propagate node leaving and joining information to all nodes in the system. We divide our loop structure into a number of slices, and assign a "slice leader" node to each slice. In CCALoop, the slice leader is the node with the smallest identifier in the slice. When a node joins the framework, its successor contacts the slice leader. The slice leader distributes this information to all other slice leaders as well as all the other nodes in the slice. Finally, each slice leader that received the message propagates it to all members of its slice. This hierarchy enables faster membership propagation which in turn enables CCALoop to reach a steady state faster. Additionally, this reduces errant queries as well as providing the means for low-latency access to framework node.

1.5.1.3 Multiple GUIs Providing a graphical user interface is an important role of a scientific component framework. A framework's user is interested in assembling a simulation, steering it, and analyzing intermediate and final results. In large, cross-organizational simulations several users may need to manage a simulation, and several others may be interested in viewing the results. State of the art CCA-compliant scientific computing frameworks provide the capability to attach multiple GUIs and users. However, each of these frameworks provides that capability only at the master node, which hinders scalability as previously discussed. One of the opportunities and challenges of a scalable distributed framework like CCALoop is to handle multiple GUIs.

CCALoop allows a GUI to attach to any cooperating framework node. A user is allowed to manage and view the simulation from that node. When multiple GUIs are present, we leverage the slice leader multicast mechanism to distribute information efficiently to all frameworks with GUIs. We establish a general event publish-subscribe mechanism with message caching capability and create a specific event channel for GUI messages. Other channels may be created to service other needs. GUIs on which some state is changed by the user are event publishers, while all the other GUIs in the system are subscribers. We route messages from publishers to subscribers through the system by passing them through slice leaders while ensuring that we are not needlessly wasting bandwidth. All messages are cached on the slice leader of the originating slice of nodes.

The reason we use the hierarchical mechanism to transfer GUI messages over a more direct approach is to cache the GUI state of the framework. CCALoop provides a mechanism that is able to update a GUI with the current state when this GUI joins after some user operations have already occurred. To prepare for this scenario, we continuously cache the contribution to the GUI state from each slice at its slice leader. This is advantageous since we expect GUIs to often join at midstream, then leave the system, and possibly return.

An additional concern with multiple distributed GUIs is the order of state-changing operations. We use a first-come-first-serve paradigm and allow every GUI to have equal rights. A more complex scheme is possible and would be useful, but that is outside the scope of this paper.

1.5.1.4 Parallel Frameworks To support CCA's choice of SPMD-style parallel components, a framework needs to be able to create a communicator, such as an *MPI Communicator*, which identifies the set of components that are executing in parallel and enables their internal communication. This internal (inter-process) communication is embedded in the algorithm and it is necessary for almost any parallel computation. To produce this communicator a framework itself needs to be executing in parallel: In order to execute the component in parallel we first execute the framework in parallel. A parallel framework exists as a resource onto which parallel components can execute. Parallel component communication and specifically parallel remote method invocation can be quite complex and it has been studied extensively [2].

A concern of this work is the inclusion of parallel frameworks in a distributed framework environment involving many other parallel and non-parallel frameworks. The parallel framework can be handled as one framework entity with one identifier or it can be considered as a number of entities corresponding to the number of parallel framework processes. We choose to assign one identifier to the entire parallel framework to simplify framework-to-framework interaction. This needs to be done carefully, however, to ensure that communication to all parallel cohorts is coordinated. By leveraging previous work in the area of parallel remote method invocation we gain the ability to make collective invocations. We use these collective invocations to always treat the parallel framework as one entity in a distributed framework.

Even though we choose to treat all parallel framework processes as one parallel instance, the component's data that the framework stores is not limited to one entry per parallel component. To enable a more direct communication mechanism, we need to store an amount of data that is proportional to the number of parallel processes of the parallel component. Large parallel components are a significant reason that more attention should be directed toward scalable distributed component frameworks.

1.6 SUMMARY

We presented the SCIJump problem solving environment for scientific computing. SCIJump employs components that encapsulate computational functionality into a reusable unit. It is based on DOE's CCA component standard but it is designed to be able to combine different component models into a single visual problem solving environment.

Several features of SCIJump were discussed: multiple component models, distributed computing support, and parallel components. SCIJump 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 with several synchronization options and utilizing the common component architecture, combined with distributed objects and parallel MxN array redistribution that can be used in SCIJump. Lastly, we presented our novel design for scalability in component frameworks through our prototype framework CCALoop.

A prototype of the SCIJump framework has been developed, and we are using this framework for a number of applications in order to demonstrate SCIJump's 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 [23, 22] and the Uintah [24, 6] parallel, multi-physics system. Additional large scale computational applications are under construction and are beginning to take advantage of the capabilities of SCIJump. The design prototyped by CCALoop which will greatly increase the scalability of SCIJump in order to support the future generation application will be added in the future.

Acknowledgments

The authors gratefully acknowledge support from NIH NCRR and the DOE ASCI and SciDAC programs. SCIJump is available as open source software at www.sci.utah.edu.

REFERENCES

1. Gordon Bell and Jim Gray. What's next in high-performance computing? *Communications of the ACM*, 45(2):91–95, 2002.

2. Felipe Bertrand, Randall Bramley, Alan Sussman, David E. Bernholdt, James A. Kohl, Jay W. Larson, and Kostadin Damevski. Data redistribution and remote method invocation in parallel component architectures. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005. (Best Paper Award).

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

4. K. Damevski and S. Parker. Imprecise exceptions in distributed parallel components. In *Proceedings of 10th International Euro-Par Conference (Euro-Par 2004 Parallel Processing)*, volume 3149 of *Lecture Notes in Computer Science*. Springer, 2004.

5. K. Damevski and S. Parker. M x N data redistribution through parallel remote method invocation. *Special Issue of the International Journal of High Performance Computer Applications*, 19(4), 2005.

6. 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.

7. Enterprise Java Beans. http://java.sun.com/products/ejb, 2007.

8. 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.

9. S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2000.

10. Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. One hop lookups for peer-to-peer overlays. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 7–12, Lihue, Hawaii, 2003.

11. 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.

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

14. Lois Curfman McInnes, Benjamin A. Allan, Robert Armstrong, Steven J. Benson, David E. Bernholdt, Tamara L. Dahlgren, Lori Freitag Diachin, Manojkumar Krishnan, James A. Kohl, J. Walter Larson, Sophia Lefantzi, Jarek Nieplocha, Boyana Norris, Steven G. Parker, Jaideep Ray, and Shujia Zhou. Parallel PDE-based simulations using the Common Component Architecture. In Are Magnus Bruaset and Aslak Tveito, editors, *Numerical Solution of PDEs on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering (LNCSE)*, pages 327–384. Springer-Verlag, 2006.

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

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

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

18. 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.

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

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

21. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, Technologies, Architectures,and Protocols for Computer Communication*, pages 149–160, New York, NY, USA, 2001. ACM Press

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*, July-September 1999.

23. J. Bigler, A. Stephens and S.G. Parker. Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of The IEEE Symposium on Interactive Ray Tracing*, pages 187–196, 2006.

24. S.G. Parker. A Component-Based Architecture for Parallel Multi-physics PDE Simulation. In *Future Generation Computer Systems (FGCS)*, 22(1-2):204–216, 2006, Elsevier.

25. SCIRun: A Scientific Computing Problem Solving Environment. Scientific Computing and Imaging Institute (SCI), University of Utah, 2007. http://software.sci.utah.edu/scirun.html.

26. C.R. Johnson, S. Parker and D. Weinstein and S. Heffernan. Component-Based Problem Solving Environments for Large-Scale Scientific Computing. In *J. Conc. & Comp.: Prac. & Exper.*, (14):1337–1349, 2002.

27. D.M. Weinstein and S.G. Parker and J. Simpson and K. Zimmerman and G. Jones. Visualization in the SCIRun Problem-Solving Environment. In The Visualization Handbook, Edited by C.D. Hansen and C.R. Johnson, pages 615–632, 2005, Elsevier.