Final Report
Covering Period: 01/15/2006 - 01/14/2009
Date of Report: 12/24/2013

Award Number: DE-FC02-06ER54848

Project Title: SciDAC-Center for Plasma Edge Simulation Report
Project Period: 01/15/2006 - 01/14/2009

Recipient Organization: SCI Institute, University of Utah
72 south Campus Drive, Room 3750
Salt Lake City, Utah 84112

Partners: Steven Parker, PhD
report written by: Greg M. Jones, MBA, PhD

Technical Contact: Greg M. Jones, MBA, PhD
72 south Campus Drive, Room 3750
Salt Lake City, Utah 84112
(801) 585-1867
greg@sci.utah.edu

Business Contact: Edward Black
72 south Campus Drive, Room 3750
Salt Lake City, Utah 84112
(801) 585-1867
ed@sci.utah.edu

DOE Contracting Officer: Mr. Warren Riley
630-252-2485
Warren.Riley@ch.doe.gov

DOE Grants Management Specialist Ljubomir Nesovic
Office of Acquisition and Assistance
U.S. Department of Energy
Office of Science-Integrated Support Center-Chicago Office
630-252-2816
ljubomir.nesovic@ch.doe.gov

Project Objective:

A software component is a mechanism for encapsulating related units of functionality. A component framework provides mechanisms for composing that functionality with other components to provide a means of structuring software that benefits both the developers and users of software. This approach lowers the barrier to developer participation and can simplify collaborations among research teams by allowing each group to focus on their interests and expertise. Component-based software engineering aids in creating software that is more readily reused across applications and enables the automation of complicated tasks.

Background:

The Common Component Architecture (CCA) effort is the embodiment of a long-range program of research and development into the formulation, roles, and use of component technologies in high-performance scientific computing. CCA components can interoperate with other components in a variety of frameworks, including SCIRun2 from the University of Utah. The SCIRun2 framework is also developing the ability to connect components from a variety of different models through a mechanism called meta-components. 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 and communicate with each other directly.

Project Objectives

We will leverage the SCIRun2 framework and the Kepler system to orchestrate components in the Fusion Simulation Project (FSP) and to provide a CCA-based interface with Kepler. The groundwork for this functionality is being performed with the Scientific Data Management center. The SDM center is developing CCA-compliant interfaces for expressing and executing workflows and create workflow components based on SCIRun and Ptolemy (Kepler) execution engines, including development of uniform interfaces for selecting, starting, and monitoring scientific workflows.

Milestones:

1. Utilize the SCIRun2-Kepler interface to initiate and monitor distributed simulations, including feedback about time required in each phase of the process.
2. Utilize the SCIRun2-CCA interface to provide access to CCA-based components in the workflow.
3. Create a prototype CCA-based interface to appropriate FSP simulation components
4. Prepare presentations to advise the Center on use of component technology in scientific simulations.

Accomplishments

Introduction to CCA and Simulation Software Systems

Component technology has seen success has a methodology for large-scale commercial software development. Component technology encapsulates a set of frequently used functions into a component and makes the implementation transparent to the users. Application developers typically use a group of components, connecting them to create an executable application. The components are managed by a component framework that exists on each computing node where components may be instantiated or executed. A component framework provides a set of services to components: locating other components in the system, instantiating, connecting, executing, reporting error messages or results, etc. It can also provide a user interface, often a Graphical User Interface (GUI), to compose, execute and monitor components. In order to manage a large component application that uses many components and utilizes sets of distributed computing resources, one or more component frameworks have to exist on each separate computing resource. This requires that multiple frameworks cooperate in some fashion to manage and monitor a large component application.

Component technology is becoming increasingly popular for large-scale scientific computing in helping to tame the software complexity required in coupling multiple disciplines, multiple scales, and/or multiple physical phenomena. The Common Component Architecture (CCA) is a component model that was designed to fit the needs of the scientific computing community by imposing low overhead and supporting parallel components. The CCA standard also provides for the inclusion of Single Program Multiple Data (SPMD) parallel components. These components exist in several address spaces and are internally managed by message passing (e.g. Message Passing Interface (MPI)). A compliant framework needs to provide the facilities to instantiate, manage, and execute this novel type of component. The CCA model selects a lightweight component framework that optimizes execution efficiency. Several software frameworks are targeted at the CCA component model, including SCIRun2, CCAFFEINE, XCAT and others. Scientific computing frameworks not based on CCA are also popular and have relatively similar structure. All of these systems enable creation of complex high-performance simulations through the assembly of software components. 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. These simulations are often targeted to execute on sets of distributed memory machines spanning several computational and organizational domains. 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.

Introduction into SCIRun2 and Bridging within SCIRun2

SCIRun2 is a component framework whose goal is to create an environment that facilitates the coupling of software components from a variety of domains in a single high-performance application. For example, parallel components based on CCA could be connected to serial or distributed components based on CORBA or other models. In this manner, the scientist is able to employ the right tool for the right job in a flexible software framework.

Component models can differ widely. For example, they can use a different network communication scheme, rely on a different communication paradigm (method invocation, publish/subscribe), be synchronous or asynchronous etc. The differences are inherent as the models were constructed to satisfy certain usability or performance requirements.

Although component model differences are large, it is possible to extrapolate a few basic similarities between typical models. For instance, each component model has a number of components that in turn contain one or more ports. These ports either push or pull some data and therefore are either an in-port or an out-port. Based on assumptions of this kind, we can design an abstract meta-component interface. The goal of this meta-component interface is straightforward: provide an interface through which component models can be plugged into and manipulated within the SCIRun2 framework. Each component model supported in SCIRun2 implements interfaces based on an abstract meta-component interface specification. The SCIRun2 framework manipulates concrete implementations of this abstract meta-component interface and through it is able to handle component instances belonging to an unlimited number of component models.

Figure 1: A UML diagram of SCIRun2s meta-component model interface that enables multiple component models to be included in the framework.

The interfaces defined by the meta-component interface primarily abstract the process of com-

ponent creation, deletion and assembly. A component model is required to implement the ComponentModel, ComponentInstance, and PortInstance interfaces. To better illustrate the concept of the meta-component interface, below we present part of it in Figure 1.

In order for a concrete component model to exist within the SCIRun2 framework it needs to implement the above meta-component interfaces. This should be done in a way that preserves the functionality and requirements of the model itself.

CCALoop: A scalable design for a distributed component framework

A component frameworks data is queried and modified by a user (through a GUI) and by executing components. In both cases it is imperative that the framework provides quick responses under heavy loads and high-availability to long running applications. The goal of our work is to present a solution to several key issues is distributed component framework design. 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). 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.

CCALoop Design Current distributed framework designs are not adequate in accommodating component applications with numerous components that use many computing resources. CCALoop implements a CCA-compliant distributed component framework 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 the following we more closely describe 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 but in this work we focus on the ones in the style of CORBA¹ that do not interfere with the execution of every component. This kind of a component framework per-

¹OMG. The Common Object Request Broker: Architecture and Specification. Revision 2.0, June 1995.

Figure 2: The data responsibilities of the CCALoop framework with four nodes.

forms 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. This allows the user to make 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 or even other systems such as Kepler for provenance management. This is done by using a 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 2. 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.

CCALoops 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.

Combining Workflow methodologies with Component Architectures

Introduction As mentioned earlier, composing scientific simulations from smaller general parts has many beneficial properties such as modularity and reuse. Component software and scientific workflows are both technologies for decomposition of scientific simulation, although each of them performs this task along a separate dimension and on a different level. Scientific workflows are concerned with high-level orchestration of a timedecomposed simulation, while components are mainly used at a lower level to enable software modularity and reuse at a small performance cost. Combining these two technologies in a seamless way enables using the benefits of both of these decomposition paradigms, resulting in more flexibility for scientific application design. In this work, we describe a design of our system that communicates parameters and results to and from the CCA framework (e.g. SCIRun2, SCIJump, CCaffeine) and the Kepler scientific workflow system.

The complexity of modern scientific simulations has necessitated methods of decomposing the application into modular reusable parts. These interchangeable modules can be connected to create applications and subsequently executed. Scientific workflows and components are technologies aimed at better modularity and reuse of scientific software that are supported by growing user communities and growing numbers of reusable modules available on each platform. However, each of these technologies is different in its approach: components communicate through method invocation and have a spatial composition style, while workflows use dataflow to communicate and are composed temporally. Also, scientific workflows are intended to guide an application from the highest level including tasks such as communicating with outside data sources, while components are used for slightly finer-grain tasks. We argue that scientific workflows and components are complimentary methodologies in that scientific programming would benefit from the ability to combine these two decomposition schemes into one working application.

In this work we present our prototype solution that enables cooperation between the Kepler workflow system², see Figure 3, and any CCA component framework resulting in a hybrid decomposition model that can use both component architectures and workflows.

The decomposition paradigm of each of these technologies prescribes a hierarchical design for interoperability: workflows as top level application control and components as the means of implementing a part of it. In this hierarchical model, a full duplex (two-way) method of communication is necessary as a workflow may need to adapt based on (intermediate) results of a component simulation. Also, it is important to provide the user with the ability to enable both coarser and finer grain communication between the two systems. The system we architected provides a set of Kepler actors that directly communicate with a component framework and other actors that are able to directly communicate to individual components. The actor to individual components connection may only need to be used to execute a set of connected components, however, a more complex scenario where components and actors communicate at multiple points of an application is possible

²I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. International Conference on Scientific and Statistical Database Management, 00:423, 2004

Figure 3: The Kepler system for creating scientific workflows from dataflow composed actors.

(e.g. to communicate intermediary results, gage progress or for computational steering). We have a prototype implementation of our design that allows this kind of hybrid application decomposition between Kepler and the SCIJump³ component framework.

Related Work Earlier work by Lu et al.⁴ defines a JobProxy and JobFactory web services and corresponding actors in Kepler to control the CCA framework CCAFEINE. Like their work, our approach uses workflows as higher level control of lower level component simulations. However, our goal differs significantly in that we want to provide a finer-grained approach to communication between components and workflows, while the communication that Lu et al. provide is very coarse-grained. Their approach uses a job mechanism to control the a componentized simulation and provides the ability to expose one CCA port type in Kepler as a web service to start the simulation. The job abstraction actors provided in Kepler for interoperability with components by Lu et al. partially duplicate the functionality of a similar set of general job actors available in Kepler. Our choice is to allow application designers finer level of control if they need it and allow a wider set

³ The SCIJump Framework for Parallel and Distributed Scientific Computing, Manish Parashar BE, MS, PhD Professor Associate Director visiting fellow senior member Vice Chair, Xiaolin Li PhD Assistant Professor visiting scholar chair member, Steven G. Parker, Kostadin Damevski, Ayla Khan, Ashwin Swaminathan, Christopher R. Johnson, SCIJump is a framework built on the SCIRun infrastructure the combines CCA compatible architecture with hooks for other commercial and academic component models. It provides a broad approach allowing 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 multithreaded to facilitate shared memory programming. It also has an optional visual programming interface.

⁴ Lu, K. Chiu, S. Shirasuna, , and D. Gannon. A hybrid decomposition scheme for building scientific workflows. Proceedings of High Performance Computing Symposium (HPC 2007), Norfolk, Virginia, March 25-29, 2007

of use scenarios. In addition, Lu et al. choose to translate CCAs Scientific Interface Definition Language (SIDL) into the Web Service Definition Language (WSDL) that workflows use in order to use web service protocols for communication between the two frameworks. This translation presents a unnecessary level of complexity. We use a more straightforward and efficient approach that does no translation and communicates natively through the RMI protocol defined by the Babel SIDL compiler.

Design As we have already mentioned, a Kepler workflow is very different from a CCA component assembly. Keplers actors are composed in a dataflow fashion, representing a temporal relationship (*e.g.* execute actor X after actor Y finishes). Components are composed via ports and represent a spatial relationship (*e.g.* component X executes component Y and sends results to component Z). The spatial composition is more powerful, enabling more complex relationships, while the dataflow models is simpler, easier to debug and measure performance. Workflows are limited in expressing control-flow semantics such as webservice retry. Basically, everything that has an if-statement like semantics requires a needlessly complicated workflow, which is inconvenient to write as well as read. Since CCA components are very good at control-flow, combining the two technologies results in more expressive and better workflows.

The applications written for components are much lower level than those for workflows. The dataflow scheme of the workflows lends itself to defining a high level application flow, and the user community supports this momentum. Based on the above observation, we posit that creating workflow actors that are able to communicate to components is a reasonable way to combine these units into one application. We provide a few actors in Kepler that would enable a scientific workflow to manage and execute a CCA component application. For instance, an application of this kind would use preexisting Kepler actors to fetch the input data and begin the component framework, and use the componentized actors to setup and execute a component simulation. All the while another actor that we provide would be used to monitor the simulations progress from Kepler, and communicate back to CCA to adjust the simulations parameters. The task of componentizing actors is not practically difficult. To communicate to a component service that is defined by a specific interface, an actor needs to contain the client stub code. By using the Babel compiler, we are able to do this task easily, and use the distributed capabilities of RMI to communicate across machine domains.

By using this new set of componentized Kepler actors, one can communicate with any CCA framework that is specification compliant and exposes its framework services for RMI access. Through the interfaces provided by the services, we control various aspects of the framework (*e.g.* component instantiation, component connection, events reporting various messages etc...). Using a special actor that is able to gain control of the CCA framework and manage the creation and connection of components a workflow can initialize a component application. Once a set of components have been instantiated and connected, they need to be executed. For this purpose, we designed an actor that is able to bind to any component and execute methods of its interface dynamically, provided that its given the method name, parameters, port name and component name. We can monitor the state of the running applications by listening to events sent by the components via the frameworks event service. Figure 4 gives a graphical display of the communication pattern between the Kepler actors we introduced and a CCA framework and components. We identify three actors that are required in order to achieve interesting application scenarios: one that is able to initialize the components, one able to control and begin the execution of the components, and one

Figure 4: Our design uses special Kepler actors that are able to communicate with the CCA framework as well as with individual components facilitating both a coarse and fine grained interoperability.

that is able to communicate status messages and results. Two of these communicate with the component frameworks services and one of them communicates directly to an individual components port: *CCAEventListenerActor*, *SCIJumpOpenNetworkActor*, and *CCAInvokePortMethodActor*. Other component-friendly actors may be useful to be added in the future, but these three define the core for many applications.

CCAEventListenerActor The CCA specification defines an event service to be implemented by specification-compliant component frameworks in order to allow communication of status messages and various other information from any component to any one or more listening components. The event service uses a publish and subscribe mechanism. A topic abstraction is used to define a specific communication channel.

The *CCAEventListenerActor* is able to subscribe to specific topic(s) of the CCA event service and execute when it reads any messages. The actor uses Babel RMI as the underlying protocol to directly communicate to the framework implementation. In our prototype example, we use this actor to communicate from the CCA realm back to the workflow in order to report the result of the component computation.

SCIJumpOpenNetworkActor This actor is used to open a saved network of CCA components, ports and connections, which is accomplished through an open network service provided by the *SCIJump* component framework. This service is not yet standardized by the CCA. However, the *BuilderService*, a similar service which is a core part of the CCA standard, could be exposed in order to accomplish the same task as the open network service. However, using the *BuilderService* would be a more tedious task, requiring that each individual instantiated component and connection to be defined. *SCIJumpOpenNetworkActor* loads a file that defines all of this information; creating this file can be done through the *SCIJump* GUI.

CCAInvokePortMethodActor This actor is different from the previous two in that it does not communicate to a service provided by the CCA framework, but directly to an instantiated CCA component, which allows finer grained communication between components and workflows. To

directly communicate with an individual component, one needs to invoke a method on a components port. In separate applications, this is usually a different port and a different method with different arguments. This is a difficult task in practice, as we are adapting to each component dynamically and not relying on predefined stubs compiled with the actor. We designed this actor with this generality in mind. It relies on reflection mechanisms heavily to discover whether ports and methods with the name the user has specified exists and then invokes them passing user specified arguments. We expect that multiple instances of this actor would exist in Kepler when interoperability with components is needed.

Results Using the CCA actors in Kepler described in the previous section, we designed a proof of concept application. This hybrid application uses Kepler and the SCIJump frameworks to start and execute the CCA tutorial application. Figure 5 shows the Kepler workflow that integrates the three componentized actor that we defined. The tutorial application contains a few components that use Monte Carlo integration to calculate an estimate for the number PI. Pre-existing Kepler actors connect (using ssh) to a remote machine containing the SCIJump executable and start the framework. We use the SCIJumpOpenNetworkActor to load the components belonging to our tutorial application. After the loading has finished we use the CCAInvokePortMethod actor to begin the simulation by discovering the starting port. The simulation is independent after this step and we use the CCAEventServiceListener to communicate various status messages as well as the final result.

This work presents a method of designing hybrid scientific applications composed of both workflow actors and components. Because each of these technologies provides a different kind of decomposition this approach is useful for new application design. Since workflows are intended to be coarser grained than components we designed a hierarchical interoperability scheme, by adding actors that are able to communicate with components. This communication is made easier by the advent of RMI in the Babel compiler used heavily in scientific component technology. We designed three actors that are necessary for a basic hybrid application: one that is able to initialize the components; one able to control and begin the execution of the components; and one that is able to communicate status messages and results.

The future of this project is to create a wider array of interesting hybrid applications and add a few more actors that may be useful to application designers. We already have several potential candidates in mind: a CCAEventPublisherActor would be able to send status messages from Kepler to the CCA framework, and a CCABuilderServiceActor would generalize the component instantiation task from SCIJump to all CCA frameworks.

Publications

1. Kostadin Damevski, Ashwin Deepak Swaminathan, and Steven Parker, *CCALoop: Scalable Design of a Distributed Component Framework.*, HPDC07, June 2529, 2007, Monterey, California, USA., ACM 978-1-59593-673-8/07/0006.
2. Steven Parker, Kostadin Damevski, Ayla Khan, Ashwin Deepak Swaminathan and Chris Johnson. The SCIJump Framework for Parallel and Distributed Scientific Computing. In M. Parashar and X. Li and S. Chandra, (eds.) *Advanced Computational Infrastructures for Parallel/Distributed Adaptive Applications* Wiley Press. 2007.

Figure 5: Our proof of concept example Kepler workflow, which invokes the CCA tutorial application running in SCJump.

3. Kostadin Damevski, Keming Zhang, and Steven Parker. Practical Parallel Remote Method Invocation for the Babel Compiler. In Proceedings of the 1st Joint HPC-GECO/CompFrame Workshop ACM, Montreal, Canada. 2007.
4. Kostadin Damevski, Ashwin Deepak Swaminathan, and Steven Parker. Highly Scalable Distributed Component Framework for Scientific Computing. In Proceedings of the 3rd International Conference on High Performance Computing and Communication 2007.
5. Ashwin Deepak Swaminathan. CCALoop - A Scalable Distributed Component Framework for Scientific Computing. Masters thesis, University of Utah. 2008.
6. Kostadin Damevski, Ayla Khan, and Steven Parker. Scientific Workflows and Components: Together at Last! In Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance computing 2008.

Personnel

- Dr. Steve Parker, 1.5 Months
- Bradley Grimm, 1.0 Months
- Santiago Ize, 12.5 Months

- Ayla Yasmin Khan, 11.5 Months
- Leena Hanamantrao Kora, 3.5 Months
- Chelsea Noelle Robertson, 4.5 Months
- Dr. Claudio Silva, 1.0 Months
- Ashwin Deepak Swaminathan, 1.0 Months