A COMPONENT ARCHITECTURE FOR HIGH-PERFORMANCE SCIENTIFIC COMPUTING

Benjamin A. Allan[2]
Robert Armstrong[2]
David E. Bernholdt[1]
Felipe Bertrand[3]
Kenneth Chiu[4]
Tamara L. Dahlgren[5]
Kostadin Damevski[6]
Wael R. Elwasif[1]
Thomas G. W. Epperly[5]
Madhusudhan Govindaraju[4]
Daniel S. Katz[7]
James A. Kohl[1]
Manoj Krishnan[8]
Gary Kumfert[5]
J. Walter Larson[9]
Sophia Lefantzi[10]
Michael J. Lewis[4]
Allen D. Malony[11]
Lois C. McInnes[9]
Jarek Nieplocha[8]
Boyana Norris[9]
Steven G. Parker[6]
Jaideep Ray[12]
Sameer Shende[11]
Theresa L. Windus[13]
Shujia Zhou[14]

Abstract

The Common Component Architecture (CCA) provides a means for software developers to manage the complexity of large-scale scientific simulations and to move toward a *plug-and-play* environment for high-performance computing. In the scientific computing context, component models also promote collaboration using independently developed software, thereby allowing particular individuals or groups to focus on the aspects of greatest interest to them. The CCA supports parallel and distributed computing as well as local high-performance connections between components in a language-independent manner. The design places minimal requirements on components and thus facilitates the integration of existing code into the CCA environment. The CCA model imposes minimal overhead to minimize the impact on application performance. The focus on high performance distinguishes the CCA from most other component models. The CCA is being applied within an increasing range of disciplines, including combustion research, global climate simulation, and computational chemistry.

Key words: component architecture, combustion modeling, climate modeling, quantum chemistry, parallel computing

[1]COMPUTER SCIENCE AND MATHEMATICS DIVISION, OAK RIDGE NATIONAL LABORATORY, P. O. BOX 2008, OAK RIDGE, TN 37831

[2]SCALABLE COMPUTING R & D, MS 9915, PO BOX 969, SANDIA NATIONAL LABORATORIES, LIVERMORE, CA 94551–0969

[3]COMPUTER SCIENCE DEPARTMENT, 215 LINDLEY HALL, INDIANA UNIVERSITY, 47405

[4]DEPARTMENT OF COMPUTER SCIENCE, STATE UNIVERSITY OF NEW YORK (SUNY) AT BINGHAMTON, BINGHAMTON, NY 13902

[5]CENTER FOR APPLIED SCIENTIFIC COMPUTING, LAWRENCE LIVERMORE NATIONAL LABORATORY, P.O. BOX 808, L-365, LIVERMORE, CA 94551

[6]SCIENTIFIC COMPUTING AND IMAGING INSTITUTE, UNIVERSITY OF UTAH, 50 S. CENTRAL CAMPUS DR., ROOM 3490, SALT LAKE CITY, UT 84112

[7]JET PROPULSION LABORATORY, CALIFORNIA INSTITUTE OF TECHNOLOGY, 4800 OAK GROVE DRIVE, PASADENA, CA 91109

[8]COMPUTATIONAL SCIENCES AND MATHEMATICS, PACIFIC NORTHWEST NATIONAL LABORATORY, RICHLAND, WA 99352

[9]MATHEMATICS AND COMPUTER SCIENCE DIVISION, ARGONNE NATIONAL LABORATORY, 9700 SOUTH CASS AVE., ARGONNE, IL 60439–4844

[10]REACTING FLOW RESEARCH, MS 9051, PO BOX 969, SANDIA NATIONAL LABORATORIES, LIVERMORE, CA 94551–0969

[11]DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE, UNIVERSITY OF OREGON, EUGENE, OR 97403

[12]ADVANCED SOFTWARE R & D, MS 9051, PO BOX 969, SANDIA NATIONAL LABORATORIES, LIVERMORE, CA 94551–0969

[13]PACIFIC NORTHWEST NATIONAL LABORATORY, ENVIRONMENTAL MOLECULAR SCIENCES LABORATORY, P.O. BOX 999, MS-IN: K8–91, RICHLAND, WA 99352

[14]NORTHROP GRUMMAN CORPORATION, INFORMATION TECHNOLOGY SECTOR, 4801 STONECROFT BLVD, CHANTILLY, VA 20151

The International Journal of High Performance Computing Applications, Volume 20, No. 2, Summer 2006, pp. 163–202
DOI: 10.1177/1094342006064488

1 Introduction

Historically, the principal concerns of software developers for high-performance scientific computing have centered on increasing the scope and fidelity of their simulations, and then increasing the performance and efficiency to address the exceedingly long execution times that can accompany these goals. Initial successes with computational simulations have led to the desire for solutions to larger, more sophisticated problems and the improvement of models to reflect greater levels of detail and accuracy. Efforts to address these new demands necessarily have included improvements to scientific methodology, algorithms, and programming models, and virtually always each advance has been accompanied by increases in the complexity of the underlying software. At the same time, the computer industry has continued to create ever larger and more complex hardware in an attempt to satisfy the increasing demand for simulation capabilities. These architectures tend to exacerbate the complexity of software running on these systems, as in nearly all cases, the increased complexity is exposed to the programmer at some level and must be explicitly managed to extract the maximum possible performance. In scientific high-performance computing, relatively little attention has been paid to improving the fundamental software development process and finding ways to manage the ballooning complexity of the software and operating environment.

Simultaneously, in other domains of software development, complexity rather than runtime performance has been a primary concern. For example, in the business area the push has been less for increasing the size of "the problem" than for interconnecting and integrating an ever-increasing number of applications that share and manipulate business-related information, such as word processors, spreadsheets, databases, and web servers. More recently, with the fast pace of the internet boom, the flood of new software technology used in business and commercial applications has increased both the degree of interoperability desired and the number of applications and tools to be integrated. This situation has led to extreme software complexity, which at several levels is not unlike the complexity now being seen in high-performance scientific computing. For example, scientific codes regularly attempt the integration of multiple numerical libraries and/or programming models into a single application. Recently, efforts have increased to couple multiple stand-alone simulations together into multi-physics and multi-scale applications for models with better overall physical fidelity.

One of the approaches that the business/internet software community has found invaluable in helping to address their complexity conundrum is the concept of component-based software engineering (CBSE). The basic tenet of CBSE is the encapsulation of useful units of software functionality into *components*. Components are defined by the interfaces that they present to the outside world (i.e. to other components), while their internal implementations remain opaque. Components interact *only* through these well-defined interfaces, and based on these interfaces can be composed into full applications. Using this methodology enables use of the "plug-and-play" software approach for creating complex applications. The smaller, task-specific units of software are more easily managed and understood than a complete application software structure. Logically, many components can provide functionality that is generally useful in a variety of different applications. In such cases, suitably well-designed components might be *reusable* across multiple applications with little or no modification. It is also possible that a number of different components can export the same interface and provide the same essential functionality, but via different implementations, thereby allowing these *interoperable* components to be swapped within an application in a plug-and-play fashion.

These ideas have spawned a number of component architectures, some of which have become so widely used as to have reached "commodity" status: Microsoft's Component Object Model (COM) (Microsoft Corporation 1999), the Object Management Group's Common Object Request Broker Architecture (CORBA) Component Model (Object Management Group 2002), and Sun's Enterprise JavaBeans (Sun Microsystems 2004a).

While having proven quite popular and successful in the areas in which it originated, CBSE has made few inroads into the scientific computing community. Part of the reason is that serious scientific applications tend to be large, evolving, and long-lived codes whose lifetimes often extend over decades. In addition to a natural inertia that slows the adoption of new software engineering paradigms by scientific software developers, current commodity component models present a variety of issues ranging from the amount of code that must be changed and added to adapt existing code to the component environment, to performance overheads, to support for languages, data types, and even operating systems widely used in scientific computing.

The Common Component Architecture (CCA) Forum was launched in 1998 as a grass-roots effort to create a component model specifically tailored to the needs of high-performance scientific computing. The group's goals are both to facilitate scientific application development and to gain a deeper understanding of the requirements for and use of CBSE in this community so that they can feed back into the development of future component models, in the hope that eventually this community too may be adequately served by "commodity" tools. In the intervening years, the Forum has developed a specification for the CCA as well as prototype implementations of

many associated tools, and the CCA is experiencing increasing adoption by applications developers in a number of scientific disciplines. This paper updates our previous overview paper in 1999 (Armstrong et al. 1999) to reflect the evolution of the CCA and our much more detailed understanding of the role of CBSE in scientific computing.

The first half of the paper presents the Common Component Architecture itself in some detail. We will discuss the special needs of this community (Section 2), and we will describe the CCA model and how it addresses these needs (Section 3). Section 4 presents the CCA's approach to language interoperability, and Sections 5–7 describe in detail how Common Component Architecture handles local, high-performance parallel, and distributed component interactions, respectively. Section 8 summarizes the currently available tools implementing the CCA, and Section 9 describes related work.

The remainder of the paper provides an overview of the the CCA in use. Section 10 outlines the typical way in which software is designed in a component-based environment, and the process of componentizing existing software. Section 11 discusses some of the existing work towards the development of common interfaces and componentized software in the CCA environment. Section 12 describes progress on tools to simplify data exchange in coupled simulations. Section 13 presents an overview of CCA-based applications in the fields of combustion research, global climate modeling, and quantum chemistry. We conclude the paper with a brief look at the people and groups associated with the Common Component Architecture effort (Section 14) and a few ideas for future work in the field (Section 15).

2 Components for Scientific Computing

Component-based software engineering is a natural approach for modern scientific computing. The ability to easily reuse interoperable components in multiple applications, and the plug-and-play assembly of those applications, has significant benefits in terms of productivity in the creation of simulation software. This is especially so when the component "ecosystem" is rich enough that a large portion of the components needed by any given application will be available "off the shelf" from a component repository. The simplicity of plug-and-play composition of applications and the fact that components hide implementation details and provide more manageable units of software development, testing, and distribution all help to deal with the complexity inherent in modern scientific software. Once the overall architecture of a software system and interfaces between its elements (components) have been defined, software developers can then focus on the creation of the components of particular scientific interest to them, while reusing software developed by others (often experts in their own domains) for other needed components of the system. Componentization also indirectly assists with the performance issues that are so critical in this area; by providing the ability to swap components with different implementations that are tailored to the platform of interest. Finally, components are a natural approach to handle the coupling of codes involving different physical phenomena or different time and length scales, which is becoming increasing important as a means to improve the fidelity of simulations.

In the scientific software world, CBSE is perhaps most easily understood as an evolution of the widespread practice of using a variety of software libraries as the foundation on which to build applications. Traditional libraries already offer some of the advantages of components, but a component-based approach extends and magnifies these benefits. While it is possible for users to discover and use library routines that were not meant to be exposed as part of the library's public interface, component environments can enforce the public interface rigorously. It is possible to have multiple instances (versions) of a component in a component-based application, whereas with plain libraries this is not generally possible. Finally, in a library-based environment, there are often conflicts of resources, programming models, or other dependencies, which are more likely as numbers of libraries increase. With components, these concerns can often be handled by hooking up each "library" component to components providing the resource management or programming model functionality.

The use of domain-specific computational frameworks is another point of contact between current practice in scientific computing and CBSE. Domain-specific frameworks have become increasingly popular as environments in which a variety of applications in a given scientific domain can be constructed. Typically, the "framework" provides a deep computational infrastructure to support calculations in the domain of interest. Applications are then constructed as relatively high-level code utilizing the domain-specific and more general capabilities provided by the framework. Many frameworks support modular construction of applications in a fashion very similar to that provided by component architectures, but this is typically limited to the higher-level parts of the code. However, domain-specific frameworks have their limitations as well. Their domain focus tends to lead to assumptions about the architecture and workflow of the application becoming embodied in the design of the framework, making it much harder to generalize them to other domains. Similarly, since their development is generally driven by a small group of domain experts, it is rare to find interfaces or code that can be easily shared across multiple domain-specific frameworks. Unfortunately, this situation is often true even with important cross-cutting infrastructure, such

as linear algebra software, which could in principle be used across many scientific domains. Generic component models, on the other hand, provide all the benefits of domain-specific frameworks. However, by casting the computational infrastructure as well as the high-level physics of the applications as components, they also provide easier extension to new areas, easier coupling of applications to create multi-scale and multi-physics simulations, and significantly more opportunities to reuse elements of the software infrastructure.

However, scientific computing also places certain demands on a CBSE environment, some of which are not easily satisfied by most of the commodity component models currently available. As noted above, performance is a paramount issue in modern scientific computing, so that a component model for scientific computing would have to be able to maintain the performance of traditional applications without imposing undue overheads. (A widely used rule of thumb is that environments that impose a performance penalty in excess of ten percent will be summarily rejected by high-performance computing (HPC) software developers.) In contrast, commodity component models have been designed (primarily or exclusively) for distributed computing and tend to use protocols that assume all method invocations between components will be made over the network, in environments where network latencies are often measured in tens and hundreds of milliseconds. This situation is in stark contrast to HPC scientific computing, where latencies on parallel interconnects are measured in microseconds; traditional programming practices assume that on a given process in a parallel application, data can be "transferred" between methods by direct reference to the memory location in which it lives. The performance overheads of the commodity component models are often too high for scientific computing.

In scientific computing, it is common to have large codes that evolve over the course of many years or even decades. Therefore, the ease with which "legacy" code bases can be incorporated into a component-based environment, and the cost of doing so, are also important considerations. Many of the commodity component models may require significant restructuring of code and the addition of new code to satisfy the requirements of the model. At least partial automation of this process is supported for some languages, such as Java and more recently, C++; to our knowledge, however, there is no industry support for generating components from Fortran, which is used in the majority legacy scientific software. Furthermore, automated analysis of parallel code adds another dimension to the complexity of automating component generation.

Finally, also important to high-performance scientific computing are considerations including support for languages, data types, and computing platforms. The various commodity component models available may not support Fortran, may require extensive use of Java, and may not support arrays or complex numbers as first-class data types. They may even be effectively limited to Windows-based operating systems, which are not widely used in scientific HPC.

The Common Component Architecture (CCA) was conceived to remedy the fact that a suitable component environment could not be found to satisfy the special needs of this community. Since its inception, the CCA effort has grown to encompass a community of researchers, several funded projects, an increasing understanding of the role of CBSE in high-performance scientific computing, a maturing specification for the CCA component model, and practical implementations of tools and applications conforming to that specification.

3 The Common Component Architecture

Formally, the Common Component Architecture is a specification of an HPC-friendly component model. This specification provides a focus for an extensive research and development effort. The research effort emphasizes understanding how best to utilize and implement component-based software engineering practices in the high-performance scientific computing arena, and feeding back that information into the broader component software field. In addition to defining the specification, the development effort creates practical reference implementations and helps scientific software developers use them to create CCA-compliant software. Ultimately, a rich marketplace of scientific components will allow new component-based applications to be built from predominantly off-the-shelf scientific components.

3.1 Philosophy and Objectives

The purpose of the CCA is to facilitate and promote the more productive development of high-performance, high-quality scientific software in a way that is simple and natural for scientific software developers. The CCA intentionally has much in common with commodity component models but does not hesitate to do things differently where the needs of HPC dictate (see Section 9 for a more detailed comparison). High performance and ease of use are more strongly emphasized in the CCA effort than in commodity component models. However, in principle no barriers exist to providing an HPC component framework based on commodity models, or to creating bridges between CCA components and other component models, e.g. Web Services (Christensen et al. 2001; Foster et al. 2002).

The specific objectives that have guided the development of the CCA are:

1. **Component Characteristics.** The CCA is used primarily for high-performance components implemented in the Single Program Multiple Data (SPMD) or Multiple Program Multiple Data (MPMD) paradigms. Issues that must be resolved to build applications of such components include interacting with multiple communicating processes, the coexistence of multiple sophisticated run-time systems, message-passing libraries, threads, and efficient transfers of large data sets.
2. **Heterogeneity.** Whenever technically possible, the CCA must be able to combine within one application components executing on multiple architectures, implemented in different languages, and using different run-time systems. Furthermore, design priorities must be geared toward addressing the software needs most common in HPC environment; for example, interoperability with languages popular in scientific programming, such as Fortran, C, and C++, should be given priority.
3. **Local and Remote Components.** Components are local if they live in a single application address space (referred to as *in-process* components in some other component models) and remote otherwise. The interaction between local components should cost no more than a virtual function call; the interaction of remote components must be able to exploit zero-copy protocols and other advantages offered by state of the art networking. Whenever possible local and remote components must be interoperable and be able to change interactions from local to remote seamlessly. The CCA will address the needs of remote components running over a local area network and wide area network; distributed component applications must be able to satisfy real-time constraints and interact with diverse supercomputing schedulers.
4. **Integration.** The integration of components into the CCA environment must be as smooth as possible. Existing components or well-structured non-component component code should not have to be rewritten substantially to work in CCA frameworks. In general, components should be usable in multiple CCA-compliant frameworks without modification (at the source code level).
5. **High-Performance.** It is essential that the set of standard features contain mechanisms for supporting high-performance interactions. Whenever possible, the component environment should avoid requiring extra copies, extra communication, and synchronization, and should encourage efficient implementations, such as parallel data transfers. The CCA should not impose a particular parallel programming model on users, but rather allow users to continue using the approaches with which they are most familiar and comfortable.
6. **Openness and Simplicity.** The CCA specification should be open and usable with open software. In HPC this flexibility is needed to keep pace with the ever-changing demands of the scientific programming world. Related and possibly more important is simplicity. For the target audience of computational scientists, computer science is not a primary concern. An HPC component architecture must be simple to adopt, use, and reuse; otherwise, any other objectives will be moot.

3.2 CCA Concepts

The central task of the CCA Forum (`http://www.cca-forum.org`) is the development of a component model specification that satisfies the objectives above. That specification defines the rights, responsibilities of individual elements and the relationships among the elements of the CCA's component model, including the interfaces and methods that control their interactions. Briefly, the elements of the CCA model are as follows:

- **Components** are units of software functionality and deployment that can be composed together to form applications. Components encapsulate much of the complexity of the software inside a black box and expose only well-defined interfaces to other components.
- **Ports** are the abstract interfaces through which components interact. Specifically, CCA ports provide procedural interfaces that can be thought of as a class or an interface in object-oriented languages, or a collection of subroutines; in a language such as Fortran 90, they can be related to a collection of subroutines or a module. Components may provide ports, meaning they implement the functionality expressed in the port (called *provides ports*), or they may use ports, meaning they make calls on a port provided by another component (called *uses ports*). It is important to recognize that the CCA working group does not claim responsibility for defining all possible ports. It is hoped that the most important ports will be defined by domain computational scientists and be standardized by common consent or de facto use.
- **Frameworks** manage CCA components as they are assembled into applications and executed. The framework is responsible for connecting *uses* and *provides* ports without exposing the components' implementation details. The framework also provides a small set of standard services that are available to all components. In order to reuse concepts within the CCA, services are cast as ports that are available to all components at all times.

3.3 Overview of the CCA Specification

Formally, the CCA specification is expressed as a set of abstract interfaces (CCA Forum 2003) written in the Scientific Interface Definition Language (SIDL) (Dahlgren et al. 2004, Chapter 3). SIDL is used by the Babel language interoperability tool (discussed in depth in Section 4), which implicitly defines bindings to the various languages Babel supports. The current CCA specification is version 0.7.5, and uses the SIDL namespace `gov.cca`.

Central to the CCA specification is the `Services` interface. This is the primary means by which components interact with the framework, allowing the component to inform the framework of port interfaces that the component intends to use or provide. The `Services` object also permits a component to avail itself of other services that the framework may provide, such as information about connections between itself and other components, or the ability to instantiate and otherwise manipulate other components.

`Services` provides methods to declare what ports the component will *provide* and *use* (`addProvidesPort()`, `removeProvidesPort()`, `registerUsesPort()`, `unregisterUsesPort()`), making it possible for the CCA framework to effectively mediate port connections in applications. These methods are most often used when the component is instantiated by the framework during assembly of the application, but can also be used at other times. `Services` also allows a user to get a handle to a port in order to make method calls on it (`getPort()`, `getPortNonblocking()`, `releasePort()`). These methods are generally used in the main code of a component, bracketing regions of code where calls are made to methods on a port provided by another component. A recent addition to the `Services` interface is the ability to register a callback that the framework will invoke immediately prior to destroying the component (`registerForRelease()`). This capabiity provides the component an opportunity to perform cleanup operations, though in practice, we find that relatively few components need this capability and its use is completely optional.

When components are instantiated, they given a reference to a unique `Services` object which is owned by the framework. Since each component has a different `Services` object, the framework can separately control what each component "sees" of the rest of the application – for example, the way in which *uses* ports are connected.

This `Services` mechanism differs from other component models (e.g. CORBA CCM (Object Management Group 2002)) that require components to implement all functionality of the component model within the component itself. In the CCA, we have begun simply and augment the `Services` specification with more methods as the need arises. Since `Services` is the responsibility of the framework, extending the `Services` interface with new methods will not break existing component code. Components can later take advantage of new features as part of their normal development process.

To be a CCA-compliant component, it is necessary to implement the CCA's `Component` interface, which specifies just one method, `setServices()`. When the component is instantiated in a CCA framework, the framework invokes the component's `setServices()` method, passing it an object of type `Services`. `setServices()` is also generally where the component registers the ports it *uses* and *provides*. This definition of the `Component` interface is characteristic of the CCA's minimalist approach – the number of new methods required to become a CCA component (sometimes referred to as *surface area* (Szyperski 1999)) is as small as possible, exactly one.

At the user level, components are "typed" and given individual identifiers. Both are strings, with the type being conventionally the SIDL class name of the component. The identifier is provided by the user assembling the application, with the requirement that it be unique within the application. Internally to the CCA framework, components are identified by unique opaque component IDs, obtained through the `ComponentID` interface.

As with components, CCA ports are identified by a type and name, both strings. By convention, the type is set to the SIDL interface name of the port; the name must be unique within the component. The functions that register ports also accept a `TypeMap` (a map containing key/value pairs of any basic SIDL data type) specifying properties for the port. The `Services` interface also includes a `getPortProperties()` method to retrieve this information. The CCA specification includes a `Port` interface, which is merely a container with no required methods of its own. To implement a CCA port, the component's code must inherit from (or extend) `Port`. Components may provide multiple ports, and even multiple instances of the same port.

In addition to defining the general CCA port mechanism, the CCA specification also defines a number of specific ports. `GoPort`, `BasicParameterPort`, and `ParameterPortFactory` are specified conveniences for component writers, to standardize some basic operations. The `GoPort`, with its `go()` method (with no arguments), is a specific type of port that frameworks and application assembly GUIs can recognize as a means to initiate an operation in the component (i.e. start the application). The other two ports facilitate the implementation of user- and component-accessible interfaces to set input/control parameters that components might need.

The CCA reuses this port mechanism to export services provided by the framework. The only difference between a framework-provided service port and a port provided by

a component is availability: the framework service port is always available, while the port from a component is available only after a connection has been made. CCA-defined service ports, such as `ConnectionEventService`, `BuilderService`, `AbstractFramework`, and `ComponentRepository` are required of all frameworks. `ConnectionEventService` lets components know when connections are made and broken. `ComponentRepository` is currently a placeholder for a planned interface that will retrieve components from distributed repositories. `BuilderService` and `AbstractFramework` provide a means to programmatically assemble and modify applications (instantiate and destroy components, make and break connections between ports), and the means for arbitrary code to become a CCA framework. These services not only allow the easy construction of application builder GUIs, but also allow dynamic behavior of the application itself, for example, swapping components based on numerical or computational performance (Hovland et al. 2003; Bernholdt, Armstrong, and Allan 2004; Norris et al. 2004). `BuilderService` also enables encapsulation of groups of components so that they can be treated as a single component. This facilitates management of large component-based applications, including the assembly of multi-scale or multi-physics simulations, where complex applications representing a particular length scale or type of physics can be encapsulated and treated as a single component, exposing only a limited number of ports.

Exactly which ports and components are connected, and in what way, is mediated by the framework (see below), which is controlled by a script or interactively by the user. Because ports are nothing more than interfaces, components that reside in the same process — as they almost always do in the HPC case — have direct access to methods, just as they would in a scientific subroutine library. Except for the usually negligible overhead of virtual function calls, directly connected components have the same performance as their non-componentized counterparts. This means that once the component composition phase of an application is finished, an application created from CCA components has performance characteristics that are little different from that of non-component code.

3.4 Component Lifecycle

Figure 1 illustrates more specifically how ports are connected and used through a sequence of interactions between a component and framework via the CCA `Services` object. For technical reasons, the `Services` object is owned by the framework, but since each component gets a reference to a unique instance of `Services` which it uses throughout its lifecycle, we picture it here as being associated with the component. In Step 1, Component 1

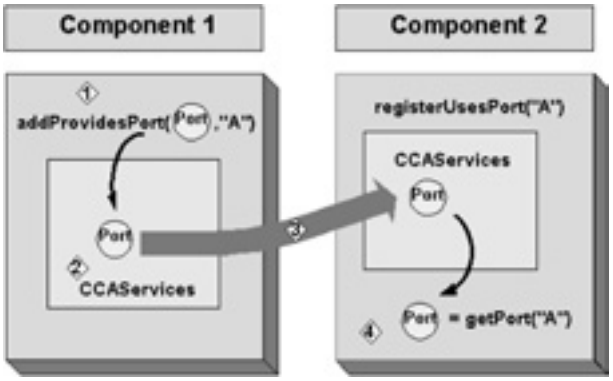

Fig. 1 A schematic representation of how ports are connected and used through a sequence of interactions between a component and framework via the CCA `Services` object.

calls `addProvidesPort()` on the `Services` object (and Component 2 calls `registerUsesPort()`) to express their intent. Then in step 2, the CCA Services object caches the information about the port that it received from `addProvidesPort()`. In the third step, the framework connects the *uses* port to the *provides* port, and the framework copies information about the *provides* port to the user's (Component 2's) CCA Services object. Finally, when Component 2 wants to invoke a method on the port provided by Component 1, it issues a `getPort()` call to obtain a handle to the port. Not shown in the diagram is the `releasePort()` call, which informs the framework that the caller is (temporarily) done using the port. A port may be used only *after* a `getPort()` call has been made for it, and *before* its companion `releasePort()` call; `getPort()` and `releasePort()` can be used repeatedly throughout the body of the component. This approach is considered better CCA programming practice than acquiring handles to all relevant ports once at the beginning of the component execution and releasing them only at the end, as it allows the use of a more dynamic component programming model.

Finally, when not within a `getPort()`/`releasePort()` block, connections between *uses* and *provides* ports may be broken and reconnected, and components may be destroyed. In general, components cannot use ports on other components during the composition phase (e.g. within the component's `setServices()` routine) because there is no guarantee that the components providing those ports have been instantiated and connected to this component's *uses* port. Framework service ports are always available and can be used at any time.

The framework and the CCA model do *not* require that all *uses* ports declared by a component be connected to

provides ports. While there are many cases in which this is an error (akin to a traditional application not being linked to a library it needs), it also provides a useful flexibility to the component developer. For example, needed functionality might be provided by any of several different ports, and this approach allows the component to check which are available (i.e. connected) and use the best one. Some *uses* ports might be considered optional by the component developer – to be used if present, but if not, the component should continue without failing. Examples include an optimization component that will make use of the function's Hessian (second derivatives) if connected to a port providing it, but will otherwise use an internally computed approximate Hessian, or a component that will send its text output to another component if connected (perhaps one that collects and annotates the output of a parallel application with information about the originating process) but simply will use `stdout` if not connected.

In everyday use of the CCA model, steps 1–3 above would take place during the composition phase of the applications. Specifically, steps 1 and 2 would occur in the component's `setServices()`, invoked by the framework when the component is instantiated, and step 3 would take place as the user instructs the framework how to connect the *uses* and *provides* ports for the application, which is typically done through a script or GUI interface that guides assembly of the application. Step 4 would take place during execution of the component's code.

While this explanation has portrayed the phases of the lifecycle as *collective*, with the entire application being assembled, executed, and then disassembled, this is not necessarily the case. As mentioned above, the `BuilderService` framework service port allows all lifecycle operations to be under the programmatic control of any component. This behavior allows the component to proxy the behavior of the framework (e.g. to recover the concept of a "main" program, which otherwise is not part of a general component model like the CCA) but is considered advanced use. The developer of such a component must be keenly aware of the CCA rules and practices governing component behavior to ensure that all components in the application "see" the expected CCA environment.

4 Language Interoperability

Multi-language programming is a fact of life in scientific computing, making programming language interoperability an important issue that the CCA must address. Scientific applications often contain combinations of compiled languages, such as C, C++, FORTRAN 77 and Fortran 90/95, and scripting languages, such as Perl, Python and Tcl. An informal survey of scientific coding groups, mostly from DOE labs, found that the average project used more than three computer languages (Kumfert and Epperly 2002). Furthermore, as the community builds more CCA-compliant components, some based on legacy codes, there is every reason to expect the trend toward multi-language applications to continue.

Unfortunately, getting codes written in different programming and scripting languages to interoperate can be quite difficult if there is no native support in one or more of the languages. Historically, developers had to write additional code to "glue" multi-language applications together, most commonly in a pairwise fashion. As a result, changing interfaces between the languages required changes to the glue code. As the number n of supported languages increased, the language interoperability burden increased n^2.

This maintenance nightmare has led to a number of efforts to ameliorate the problem. For example, Simplified Wrapper and Interface Generator (SWIG) (Beazley 2003) is a tool that enables interactions among a select group of languages. SWIG allows code written in Perl, Python, Ruby and Tcl to call software written in C or C++, but it does not enable C/C++ to call Perl, Python, Ruby or Tcl.

Rather than rely on the current norm of pairwise interoperability between a few languages, the CCA has adopted a tool that reduces the language interoperability burden on component developers and users from n^2 to $2n$. This tool, called *Babel* (Lawrence Livermore National Laboratory 2004), is able to substantially reduce the interoperability effort through the use of an intermediate object representation (IOR) implemented in ANSI C (Dahlgren et al. 2004). Currently, Babel supports C, C++, Fortran 77, Fortran 90/95, and Python. Actually Babel is a "stand-alone" tool in that it can be used outside the Common Component Architecture environment.

Babel is an interface definition language (IDL)-based tool that automatically generates code to glue multi-language components together. It relies on the Scientific Interface Definition Language (SIDL) (Dahlgren et al. 2004) for the definitions of calling interfaces through defined types (i.e. interfaces and classes) and declared methods. A SIDL file contains only type and method declarations; a SIDL type is implemented by running the Babel compiler on the SIDL file and editing the implementation file in one of Babel's supported languages (i.e. C, C++, etc.). Both SIDL *interface* and *class* are used to define collections of methods representing the calling interface. The difference between the two is interfaces cannot be implemented directly while classes can — provided they are not abstract. The implementation of one or more methods of an abstract class must be deferred to a concrete subclass. Interfaces and classes are grouped together in packages to provide a namespace concept to prevent symbol name collision. Using a SIDL file, the Babel compiler generates glue code to handle the translation of arguments and method calls between caller and callee for any supported language.

Like similar technologies such as CORBA (Object Management Group 2002) and COM (Microsoft Corporation 1999), Babel provides the standard hallmarks of object-oriented programming (e.g. object identity, inheritance, polymorphism, and encapsulation) in all supported languages even when the features are not native to the language. These concepts map differently into different languages. For example, object identity is handled with a struct pointer in C, wrapper objects in C++, Java and Python, 64-bit integers in FORTRAN 77, and a derived type containing a 64-bit integer in Fortran 90/95.

Babel's inheritance model is analogous to that of Java (Gosling, Joy, and Steele 1996). A SIDL class can extend, or inherit from, at most one class and can implement an arbitrary number of interfaces. Single inheritance of classes was chosen to avoid the confusion that sometimes arises with multiple inheritance. Interfaces can extend multiple interfaces. With Babel, it is possible to implement each level of a type hierarchy in a different language. This is inherent in Babel's object model because all classes specified in SIDL, regardless of their language of implementation, implicitly extend the SIDL base class, `sidl.BaseClass`, which is implemented in C.

What makes Babel and SIDL different from other IDL-based tools, such as CORBA and COM, is that they are being tailored for use in scientific computing. For example, SIDL has built-in support for scientific data types, including complex types and dynamically allocated, multi-dimensional, arbitrarily strided arrays.

Furthermore, having the IOR implemented in ANSI C has several important benefits for HPC. First, it enables the generation of glue code that will support fast, single-process communication. Second, using ANSI C increases the portability of the glue code across high-end platforms. Finally, Babel supports Fortran as a first class language. Although Fortran is now largely ignored by most of the computing world, including most of the commodity component models, it still has a large following in scientific computing, particularly among application developers. Babel provides native Fortran 90/95 array access to Babel's internal array data structure using the CHASM array descriptor library (Rasmussen et al. 2001, 2003). Providing native access to array descriptors is challenging because array descriptor internals are usually compiler specific and undocumented.

Since component technologies are an evolutionary step beyond object-oriented programming, the CCA has been able to leverage Babel and SIDL in the development of its component framework. The CCA specification is written in SIDL, and component developers write SIDL files to describe their ports and the classes that use or provide those ports. Using SIDL enables the encapsulation of implementation details of CCA-compliant components. Wrapping the CCA framework in SIDL makes it a relatively simple exercise to create Python components serving as application drivers, GUIs, and other tasks for which scripting languages are often used in traditional programming environments (Kumfert 2003).

The benefits of Babel technologies include its environment, programming language support scalability, and its potential as a basis for future research. Babel is a portable, language independent, object-oriented programming environment that enables the CCA to incorporate components written in the most common scientific programming languages. Furthermore, it provides a scalable approach to support new languages as they rise to prominence. Other advantages include the ability to leverage Babel to facilitate scientific computing research in areas such as remote method invocation, specification-level parallelism, and dynamic code insertion. More information about current efforts on these research areas can be found in Section 15.

5 Fine-Grained Component Interactions

As mentioned previously, one of the major requirements of the Common Component Architecture is that it impose minimal performance overheads on both local component interactions and component-based parallel applications. While not explicit in the CCA specification, the design of the CCA makes it possible for straightforward implementations of the specification to obtain high performance in both contexts.

In the CCA, the framework mediates all connections between *uses* and *provides* ports. This makes it possible for CCA implementations to transparently support both distributed and local connection models. In the distributed case, `getPort()` returns a pointer to a proxy for the *provides* port provided locally by the framework; the framework itself is responsible for conveying the method invocations to the actual remote port (see Section 7).

In the local case, CCA implementations generally load components into the same *process* so that they share the same memory space, while putting them into different *namespaces* to preserve the separations expected between components. In this case, `getPort()` can simply return a pointer to the virtual function table for the port, thereby allowing methods on the port to be invoked directly (without further intervention by the framework), and data to be passed by reference if desired. The CCA term for this approach is *direct connection*. In component models, most of which were designed primarily with distributed computing in mind, direct connection optimizations would violate the component model's specifications. Some CORBA ORBs provide a less efficient collocation optimization, which does maintain compatibility with the component model (Schmidt, Vinoski, and Wang 1999).

Experiments have shown that the overheads for calls between directly connected CCA components are small

and quite manageable in the context of scientific computing (Bernholdt et al. 2002; Norris et al. 2002; Benson et al. 2003). Calls using simple data types (those requiring no translation by Babel) cost the equivalent of a C++ virtual function call, exactly what is expected for the direct connection implementation described above. This is roughly three times the cost of a function call in Fortran or C, or about 50 ns on a 500 MHz Pentium III system. For comparison, the same test with the omniORB CORBA implementation on the same platform required approximately twenty-six times longer.

Of course there are data types that require more effort on Babel's part to translate from one language to another – effort that varies depending on the languages being connected. For example, nearly every language represents strings differently, so it is generally necessary to allocate memory and copy for every string argument. Babel also allows the user to specify that arrays always be in a fixed order (row- or column-major), and between some language pairs must allocate and copy the entire array to enforce this. However, for general scientific computing, it is not difficult to avoid excess overheads associated with componentization, or to amortize them with a sufficient amount of computational work in each function invocation. The advice given to developers of CCA-based applications is to be aware of the costs and factor them into the design of the individual components and interfaces and the overall design of the application.

6 Parallel Components

The CCA approach to supporting high-performance parallel computing is guided primarily by the "Integration" objective of Section 3.1. The simplest approach to integrating existing parallel code in a CCA environment is to accept the parallel programming model used in the original code, rather than trying to impose a new CCA-specific one. This approach offers a number of additional benefits. First and foremost, software developers can continue to take advantage of the investment they have made in learning how to produce high-performance software in their favorite programming model. For many users, this is probably as important as the more obvious fact that our approach requires a minimal modification to existing code. Thus, the CCA does not require yet another parallel programming model, which also greatly simplifies the task of implementing a CCA-compliant environment.

There are a variety of CCA-compliant frameworks that vary in their support of parallel and distributed computing. At present, a given framework supports *either* HPC parallel *or* distributed computing, and parallel plus distributed applications must be assembled from pieces running in different frameworks, using the `BuilderService` and `AbstractFramework` services defined in the CCA specification. As discussed further below, we expect to move toward higher degrees of integration and transparency in the coupling of parallel and distributed applications. For now, however, we focus on the current situation with respect to parallel and distributed computing.

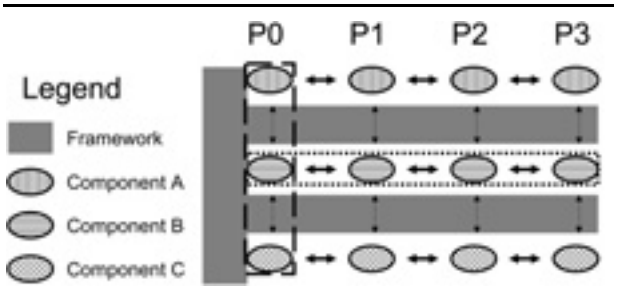

Fig. 2 A schematic representation of the CCA parallel programming environment in the single component/multiple data (SCMD) paradigm. Different parallel processes are labeled P0P3, and the same set of three components has been loaded into each process. Components in the same process (vertical dashed box) interact using standard CCA port-based mechanisms, while parallel components of the same type (horizontal dotted box) interact using their preferred parallel programming model, such as MPI, PVM, Global Arrays, etc.

Ccaffeine (Allan et al. 2002, 2003), the main CCA framework implementation for HPC parallel computing, supports the component analogs of both the single program/multiple data (SPMD) and multiple program/multiple data (MPMD) models. We refer to these as single or multiple *component*/multiple data (SCMD or MCMD) models. Figure 2 depicts the SCMD case; each process is loaded with the same set of components wired together in the same way. Interactions among components within a given process (vertical direction) take place through the normal CCA means — by declaring ports and using `getPort()`/`releasePort()` around calls to other ports. Interactions among a component's parallel *cohort* (parallel instances of the same component, the horizontal direction in the figure) take place via the parallel programming model that *component* uses. Different components within the same application may use different parallel programming models (with certain restrictions, discussed below). "Diagonal" interactions – between component A on one process and component B on another process – are not prohibited by the CCA, but require a degree of coordination that may not be consistent with the notional independence of components (unless a componentization has been carried to a degree that groups of closely inter-

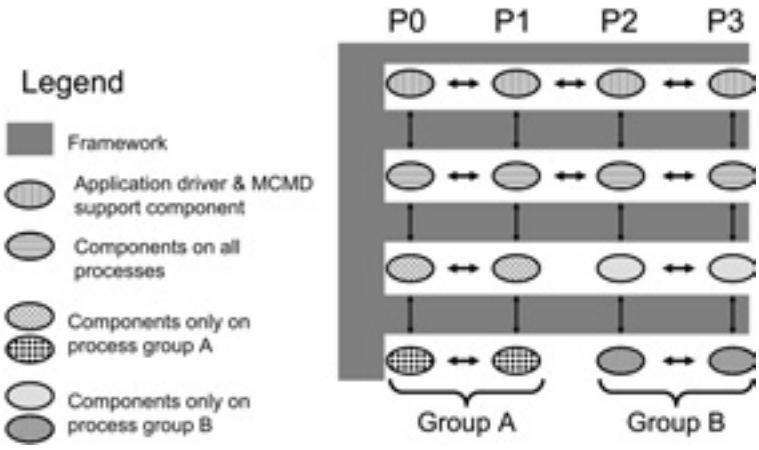

Fig. 3 A schematic representation of the CCA parallel programming environment in the multiple component/multiple data (MCMD) paradigm. Different parallel processes are labeled P0P3. Some components are loaded into all processes (top two rows), while others (bottom two rows) are loaded only into subsets of the processes (Group A and Group B).

twined components are used to carry out certain operations).

In the MCMD case (Figure 3), some components are typically loaded into all processes, while others are loaded only into subsets of the processes. An example of an MCMD application might be a coupled global climate model, where atmospheric, oceanic, land, and other elements of the model are run simultaneously on different subsets of the available processes. These separate elements are managed by a driver component and supported by other components, which handle the data exchange. In contrast to the disparate elements (atmospheric, oceanic, etc.), the management and support components are loaded into all processes of the simulation. Because of the inherent complexity of partitioning the process space and of launching parallel jobs with different inputs or executables on every process, MCMD applications can be most easily created using the CCA's `BuilderService` from a so-called builder component. This component, which might be written in a scripting language, such as Python, would compute the desired partition of the available processes and then use `BuilderService` on each process to load and connect the appropriate set of components. The builder would then invoke the `GoPort` on the climate model's driver component to initiate the simulation.

Ccaffeine also supports threaded parallelism, with the restriction that on each process, only the main thread is allowed to interact with the framework to manage connections between components. The SCIRun2 framework (Zhang et al. 2004) is fully thread-safe, thus allowing combinations of MPI-based and thread-based parallelism within the same component. Currently, neither of these frameworks provides a mechanism for managing resources when more than one multi-threaded component is competing for processing resources.

The simplicity of the CCA approach to parallel computing offers another significant advantage besides the ease of incorporating existing parallel software into the CCA environment: the performance of parallel components is virtually identical to that of the original code. The only overheads imposed by the CCA are those on the local component interactions, which, as discussed in the previous section, are small, and quite managable. A number of studies have verified this experimentally:

- Norris and coworkers (Norris et al. 2002) have compared a CCA component implementation and a non-component library implementation of a nonlinear unconstrained minimization problem solved using an inexact Newton method with a line search (Norris et al. 2002). Each iteration of the Newton method required function, gradient, and Hessian evaluations, as well as an approximate solution of a linear system of equations. The overhead of the component implementation resulted from the

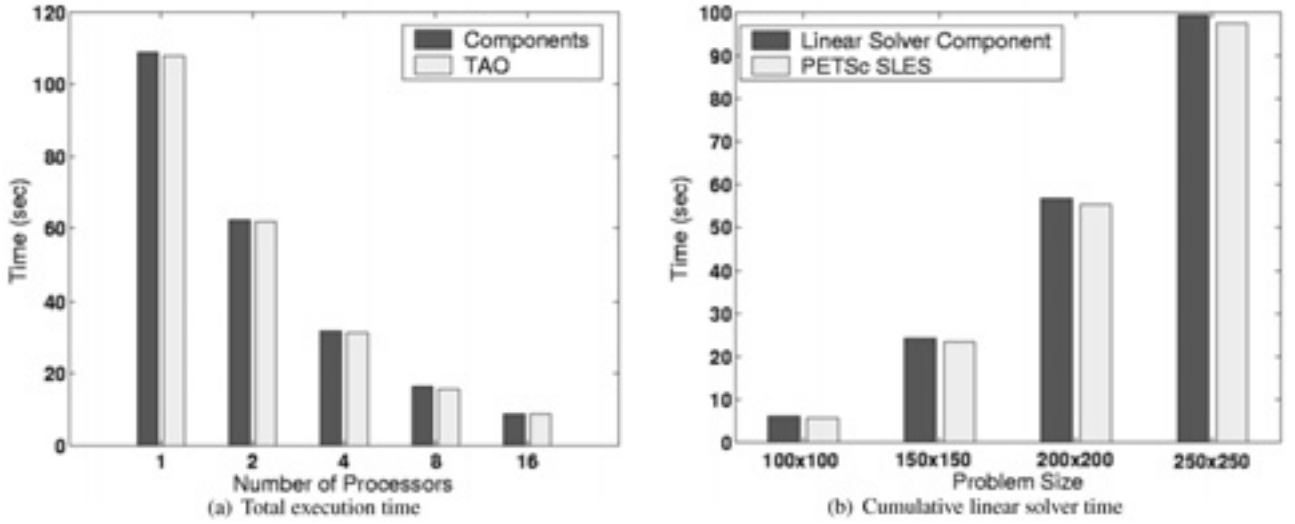

Fig. 4 Component overhead in an unconstrained minimization application on a 250 × 250 mesh: (a) total nonlinear solution time; (b) linear solution time.

extra layer of abstraction (more virtual function calls) and was most significant for very small problem sizes. The parallel performance on 1 to 16 processes of a Linux cluster also demonstrate that the SPMD component implementation does not adversely affect parallel performance and scalability. Figure 4 compares the component and non-component costs for the overall minimization simulation and the linear solve phase.

- Katz, Tisdale, and Norton (2002) constructed a simple component-based two-dimensional parallel unstructured Adaptive Mesh Refinement (AMR) application and compared it with the library on which the component was based. Repeated timings (5–10 runs each) on between 2 and 32 processes showed no effect on the parallel scalability of the application and essentially no difference in run times (some runs showed the library version slightly faster, while for others the component version was slightly faster).
- Benson et al. (2003) analyzed the performance of a molecular optimization problem (Hoare 1979) similar to the one described in Section 13.3. This application used a two-dimensional Lennard-Jones potential together with the Toolkit for Advanced Optimization (TAO) (Benson, McInnes, and Moré 2001; Benson et al. 2003) for the optimization and Global Arrays (GA) (Nieplocha, Harrison, and Littlefield 1996; Pacific Northwest National Laboratory 2004a) for the linear algebra. Tests were performed with up to 65536 atoms on up to 170 processes to evaluate the overhead of the CCA component version with respect to the non-component library-based version of this application. The largest overhead penalty was less than 0.2% of the total execution time (Figure 5).

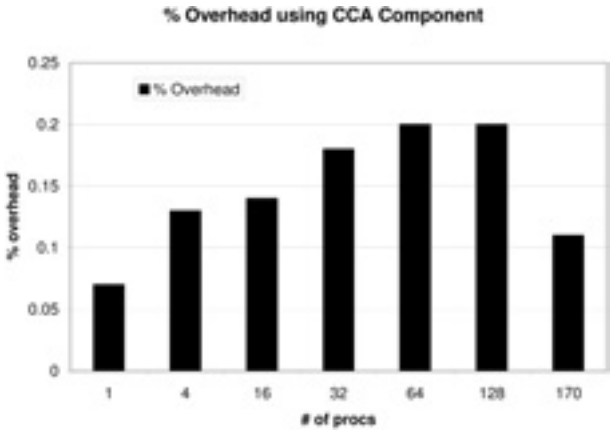

Fig. 5 Measured overhead as a percentage of execution time for component- vs. library-based implementations of a Lennard-Jones energy optimization problem.

The simplicity of the CCA approach to parallel computing also brings with it some issues and possible disadvantages that must be mentioned. First, the programming models (and specific implementations) used in the various components assembled into an application must be "composable", or able to work together without interference. In many cases, this issue does not really pose a problem. For example, MPI (MPI Forum 1994), PVM (Geist et al. 1994), and the Global Array Toolkit (Nieplocha, Harrison and Littlefield 1996; Pacific Northwest National Laboratory 2004a) can coexist without problems thanks to their compatible execution environments. On the other hand, connecting an MPI-based component to one based on OpenMP is unlikely to produce the desired result without careful thought, and perhaps additional programming. While the composition of parallel programming models in a single application is perhaps more likely to occur in a component environment than in traditional application development, this issue is in no way specific to component environments and is outside the scope of the CCA.

In addition, although SIDL bindings exist for MPI, PVM, and Global Arrays, no parallel programming model has yet been formally componentized in the CCA environment. This means that they must be treated as traditional libraries, which raises subtle issues in a component environment. Basically, whether the parallel programming (or other) library is linked to the component instead of the framework itself, and whether the linkage is static or dynamic, affect whether each component sees the same parallel environment or a completely separate instance of it (Bernholdt, Elwasif, and Kohl 2002). In most cases, the desired result of a shared parallel environment requires that the library be linked into the CCA framework, which is a minor inconvenience (in so far as the framework build must be modified to include the libraries required by the application(s) of interest), and begs the question of who takes responsibility for initializing and finalizing such libraries. The present solution is that the framework is also responsible for these actions, which has proven adequate so far. In principle, they could be moved into a component, but MPI, for example, requires the application's command line arguments for its intialization procedure. (This suggests the need for a standardized means by which components can obtain a copy of the commandline arguments from the framework.)

Finally, for any programming model that provides support for multiple contexts, code is much easier to reuse when it accepts an externally determined context rather than simply using the default global context (such as MPI's `COMM_WORLD`). This is advice often given to those programming in message-passing environments (and just as often ignored, it seems), that becomes all the more important in component environments, particularly for components that might be used in the MCMD fashion.

7 Distributed Components

Historically, high-performance scientific computing has focused primarily on local sequential and parallel computing. However, there is an increasing interest in distributed computing, reflecting the rise of the Internet, web-based "application service providers", and Grid computing (Foster and Kesselman 1998). The CCA's design objectives (Section 3.1) support both parallel and distributed computing. The primary distinction between the two paradigms, as far as the CCA is concerned, is whether or not components can be directly connected, sharing memory within the same process. As described above, such direct connection is the primary means by which CCA environments can assure the highest possible local/parallel performance; in distributed computing, although efficient tools are still important, the emphasis is on flexibility and the ability to integrate with existing distributed computing standards and environments. Beyond this, the integration of the HPC parallel and distributed paradigms within a single component-based environment is an interesting and challenging research issue.

Distributed computing environments raise a number of issues that differ from those in the direct-connect parallel paradigm:

- **Component Interaction.** When components cannot be directly connected because they are in separate processes, generally in disjoint address spaces, it is necessary for the framework to mediate calls on ports through mechanisms such as remote method invocation (RMI) (Sun Microsystems 2004b), remote procedure calls (RPC) (Birrell and Nelson 1984), or message passing (MPI Forum 1994). Numerous such systems are available (and new ones are continually being developed), with a range of performance and capabilities. In each system, there is a specialized representation for "handles" to the ports and services of components. These must be globally unique references which point to objects in remote address spaces. Often these handles are objects that can function as proxies for remote objects.
- **Registration and Discovery.** This is the term for the mechanisms that allow components to announce their availability and to find each other so that they can cooperate within the same application. Though this capability is necessary for both local/parallel and distributed computing, in the former case, very simple solutions are quite effective – components are typically loaded from pre-specified locations on a (shared) file system into a single framework instance. In the distributed case, one cannot count on shared file systems, or even ones that

have similar structure, and the CCA framework itself must execute in distributed fashion, so that component co-location cannot be the basis of discovery. Registration and discovery (R&D) mechanisms are very much influenced by the the broader distributed computing environments into which CCA may be integrated.
For example, in an Open Grid Services Architecture (OGSA) (Foster et al. 2002) or Web Services-based environments, components can be described by Web Services Description Language (WSDL) documents (Christensen et al. 2001), and registered and discovered in registries such as the Lightweight Directory Access Protocol (LDAP) (OpenLDAP Foundation 2003) and Universal Description, Discovery, and Integration (UDDI) (UDDI.org 2003). For Legion-based environments (Grimshaw et al. 1999; Lewis et al. 2003), components have names in a globally accessible context-space, and can be discovered and filtered based on where they are running, to which class they belong, their creator, and a range of other component attributes.

- **Component Creation.** In a distributed environment, the creation of components needed for an application typically involves remote resources, whereas in a local/parallel situation the resources are local, and typically pre-allocated (for example, by a queuing system). Distributed frameworks must therefore support dynamic remote creation and placement (i.e. scheduling) that respects the security (authentication and authorization) requirements of the component, the application, and the supporting environment.

Given the range of issues and the options for addressing them, including choices of a number of well-established distributed computing environments into which CCA's CBSE approach might be integrated, it is not surprising that there are currently several distributed CCA frameworks supporting a range of capabilities.

7.1 XCAT

XCAT (Govindaraju et al. 2003; Krishnan and Gannon 2004) is a CCA-based distributed component framework that uses the Web services model as its basic architecture. It has been implemented in both C++ and Java. Each *provides* port in an XCAT component is described using an XML schema and is designed as a Web Service that has one port type. The XCAT team is currently working on using a WSDL (Christensen et al. 2001) document for this purpose. WSDL documents will provide a description of the interface and also provide information pertaining to the communication protocols to which the service is bound. The use of WSDL for XCAT components will enable the use of well known Web service client toolkits to invoke methods on XCAT *provides* ports.

In XCAT-Java (version 3.0.6 alpha), component creation and launching mechanisms on remote locations are based on Globus GRAM via the Java CoG kit (von Laszewski et al. 2001). XCAT-C++ (1.0 beta) uses SSH to instantiate components on remote locations. Currently, it expects the component executable to be deployed on the resource on which it needs to be instantiated.

For components in the same address space, XCAT has a built-in mechanism that is optimized for co-location optimization. Communication among XCAT-Java components is enabled by the XSOAP (Slominski et al. 2001) toolkit, which provides an elegant model based on the Java-RMI (Remote Method Invocation) specification (Sun Microsystems 2004b). XSOAP and XCAT-Java use the dynamic proxy feature of Java. This feature allows the generation of stubs and skeletons dynamically. XCAT-C++ uses the Proteus (Chiu, Govindaraju, and Gannon 2002) multi-protocol library to choose the most appropriate communication protocol for each pair of interacting components.

While the XCAT API can be used directly in programs for composition of applications using the component assembly model, the program must be recompiled whenever the component wiring is modified. XCAT also provides a binding to Python for XCAT-C++ and Jython (an implementation of Python in Java) (Anonymous 2003) for XCAT-Java that can be used to access the XCAT libraries, thereby enabling scientists to change their application composition without the need for recompilation. This approach also has the added advantage of launching applications from web-portals.

7.2 SCIRun2

SCIRun2 (version 1.90) (Zhang et al 2004) is a framework that combines CCA compatibility with connections to other commercial and academic component models. Based on SCIRun (Johnson and Parker 1995; Parker and Johnson 1995; Parker et al. 1997) and the CCA specification, SCIRun2 utilizes parallel-to-parallel remote method invocation to connect components in a distributed memory environment. It is multi-threaded to facilitate shared memory programming. SCIRun2 also has an optional visual-programming interface. Overall, this framework provides a broad approach that allows scientists to combine a variety of tools for solving a particular problem. The overarching design goal of SCIRun2 is to provide the ability for a computational scientist to use the right tool for the right job, a goal motivated by the needs of biomedical and other scientific users.

The primary innovative design feature of SCIRun2 is a meta-component model that facilitates integration of a number of classes of tools from various, previously incompatible systems. In the same way that components plug into CCA or other component-based systems, SCIRun2

allows entire component models to be incorporated dynamically. Through this capability, SCIRun2 facilitates the coupling of multiple component models, each of which can bring together a variety of components. This feature allows the coupling of single-address-space components based on Babel (see Section 4), and components from SCIRun, as well as CCA components that use the SCIRun2 distributed computing infrastructure. Special components, called bridges, facilitate interactions between components belonging to different models. These bridges can be generated automatically or semi-automatically.

The SCIRun2 framework enables a variety of distributed computing mechansisms to be used together. Natively, SCIRun2 provides support for RMI-based distributed objects. This support is utilized in the core of the SCIRun2 framework in addition to distributed components. Using the SIDL language, we compile proxy objects that marshal method calls to remote objects and manage distributed references. For objects that exist within the same address space, this marshalling framework is completely bypassed, making fine-grained component interactions possible with only the cost of a C++ virtual function call. For distributed components, the SCIRun2 framework employs a remote *slave framework* that manages component creation, registration and discovery on a remote, possibly parallel, computing resource. We build on the SCIRun2 RMI system and extend the SIDL language to implement parallel-to-parallel component connections, discussed further in Section 12.

7.3 Legion

Legion is an object-based Grid computing system (Grimshaw et al. 1999; Lewis et al. 2003). The Legion-CCA framework (Lewis et al. 2003) makes use of the Legion object model and run-time system for implementing components. Legion objects contain their own address space, and are named within a global namespace by Legion Object Identifiers (LOIDs). Legion class objects define the interface to their instances, and manage them at runtime. Currently, each CCA component maps to its own Legion object and therefore has its own address space. Components make calls on others in the CCA framework by invoking methods on the Legion objects that represent them. Programmers specify CCA components using SIDL, and a stub generator produces wrapper code that allows each component to communicate with the others in a Legion-based framework. Programmers fill in port definitions and compile the code into a Legion object that can then run in the Legion-CCA framework as a component.

Each component is linked with both the Legion runtime library and a Legion-CCA library, which delivers the services required by the CCA specification. For example, the library furnishes an interface for registering and obtaining *provides* and *uses* ports, for instantiating components via a BuilderService, and for maintaining a table of connections to other components. Many of the library's functions require outcalls to remote services, but the interface to the component programmer is provided as a local API. This approach keeps the programming of components that are intended to run in a distributed environment as similar as possible to the programming of local components.

Legion provides services that cover all of the distributed computing issues raised at the beginning of this section. Thus, implementing a particular service is a matter of mapping the CCA request onto the Legion implementation. For example, component creation is realized by invoking the `createInstance()` function in the Legion library. This, in turn, calls the `CreateInstance()` method exported by the appropriate class object. Component interaction is implemented on top of the Legion messaging system, which supports remote method invocation. Soon, the registry and discovery service, although not currently implemented, will be able to take advantage of the Legion context space.

7.4 Distributed Interoperability

Interoperability among distributed frameworks differs significantly from other kinds of interoperability and requires an additional specification that is not a contract between the framework and the component, but rather a contract among frameworks. Some of the capabilities required for interoperability among distributed CCA frameworks include:

- **Common Communication Protocols.** Before two components can communicate, there must be some agreement on the protocols they use. For example, they may choose to use something based on the SOAP standard (Gudgin et al. 2003) being developed by the World Wide Web Consortium (W3C).
- **Common Remote Reference Format.** To connect a *uses* port to a *provides* port, there must be some way to refer to the *provides* port. To connect a *uses* port from one framework to a *provides* port of another, the ports must have a common reference format.
- **Common Creation Protocol.** The current CCA specification provides a BuilderService to be used for creating components. A `ComponentID` is returned upon successful instantiation of the component. In a distributed framework, once a component has been instantiated, the BuilderService and the new component need to interact so that the `ComponentID` can be returned. This interaction requires an additional protocol above that used for basic communication.
- **Common Connection Protocol.** When two components are connected, a reference to the *provides* port is

placed in the *uses* port of the other component. In a distributed framework, this reference may be to a component in a different framework. Different frameworks must agree upon the set of RPC calls and the on-the-wire format of parameters that will be needed to place a reference to the *provides* port in the *uses* port. Note that this protocol is layered above the common communication protocols.

Part of the ongoing research activity in the distributed CCA effort is aimed at establishing the full requirements for distributed framework interoperability and determining the most effective way to specify each aspect for inclusion in the formal CCA specification. Rather than "blessing" specific protocols for each capability, we anticipate an environment that allows frameworks to negotiate among multiple supported options for each capability.

8 Current Status of the Common Component Architecture

Although active research and development of the Common Component Architecture continues, a variety of tools are currently available that provide a highly functional environment for the development and use of CCA-compliant components and applications. Indeed, a number of applications groups have already begun such work, some of which is described below.

Although the CCA specification, currently at version 0.7.5, continues to evolve and mature, most of the recent changes have either been clarifications or the addition of capabilities that are considered advanced, and therefore rarely employed by typical users. For mainstream use, the CCA specification has been largely stable for the last several years. Backward compatibility of the specification is not currently a requirement, although the impact on user code is carefully considered before changes are made. For example, the recently added `ComponentRelease` interface was one of three alternatives proposed, and was chosen in large part because it had the fewest backward compatibility problems.

The Babel language interoperability tool, currently at version 0.9.0, is independent of the CCA specification, but also is a major dependency for CCA framework implementations. Babel has been evolving fairly rapidly, including the addition of Fortran 90 support within the last year. Backward incompatible changes are announced in advance and discussed within the user community, often influencing the implementation time line. The Babel team has defined a draft set of criteria for the version 1.0.0 release, which includes a guarantee of longer-term stability.

The final core tool in the CCA environment is a framework. As we have described, there are a number of CCA-compliant frameworks currently available for use (and others which are primarily research vehicles). The primary frameworks, their versions, and their capabilities are summarized in Table 1. Frameworks depend on spe-

Table 1
Summary of versions and capabilities of core CCA tools as of June 2005

Category	Name	Version	Capabilities
CCA Specification	cca-spec-babel	0.7.5	
Language Interoperability	Babel	0.10.0	Supports C, C++, FORTRAN 77, Fortran 90/95, Java, Python
	Chasm	1.1.0	Used by Babel for Fortran 90 array support.
Frameworks	Ccaffeine	0.5.6	Local direct-connect and parallel computing (SCMD and MCMD paradigms)
	XCAT-Java	3.0.6α	Local direct-connect and distributed computing compatible with Grid- and web-services approaches
	XCAT-C++	1.0β	Uses the Proteus Multi-protocol library and distributed computing compatible with Grid- and web-services approaches
	Legion-CCA	1.0	Distributed computing compatible with the Legion environment
	SCIRun2	1.92	Local direct-connect, MPI/threaded parallel, and distributed computing
Graphical Interface	ccafe-gui	0.5.5	"Visual programming" interface for assembly of component-based applications in the Ccaffeine framework
Complete CCA Environment	cca-tools	0.5.7	Integrated software distribution including the CCA specification, Babel, Chasm, Ccaffeine, and ccafe-gui. It is the recommended way for most users to obtain a complete CCA environment.

cific versions of both the CCA specification and Babel, and consequently tend to evolve at a pace guided by releases of those tools.

A variety of secondary or more specialized tools for the CCA environment are also available or under development. These include tools which automate much of the relatively mechanical work of setting up the "skeleton" of the files required to create ports and components, graphical "application builder" interfaces which connect to CCA frameworks, and tools which facilitate the "componentization" of existing software.

9 Related Work

As has been discussed, the CCA is a component model specialized to meet the needs of the high-performance scientific computing community. While we are aware of no other similar effort targeting this particular group, we mention in relation to this work a variety of other component models, as well as other tools and environments that facilitate the development of large-scale scientific software.

With respect to component models, probably the most relevant ones are the commodity component models, particularly CORBA CCM (Object Management Group 2002); Microsoft COM, DCOM and .NET (Box 1997); Microsoft Visual Basic (Liberty 2003); JavaBeans (Englander 1997); and Enterprise JavaBeans (Roman 1997). These models were conceived without high-performance computing in mind and usually restrict the user to a platform that is not conducive to HPC. Java components are restricted to the Java platform, and COM, DCOM and .NET are, as a practical matter, restricted to Microsoft operating systems. Neither of these platforms is often associated with high-performance computing. There have been serious efforts to use CORBA in a high-performance setting with mixed results (Keahey 1996; Schmidt, Pyarali, and Harrison 1996; Denis et al. 2001, 2003). Commodity component models usually focus on generality and flexibility, and their designs do not emphasize performance in an HPC sense. The time scales for commodity component models are usually based on human perceptions and reaction time, which is roughly five orders of magnitude slower than modern parallel computer timescales. However, it is important to emphasize that the CCA considers long-term compatibility with commodity component models important. Indeed, the design of the CCA owes much to these component models, especially the CORBA CCM.

In the HPC scientific computing area, perhaps the most relevant comparisons can be made to the variety of popular domain-specific applications frameworks (see Section 2). Cactus (Allen et al. 2000) predates the CCA and originally grew out of the relativistic astronomy community. It is still fostered by that community but is being generalized to a more multi-purpose framework. The Earth System Modeling Framework (ESMF) (Killeen, Marshall, and da Silva 2003) is a similar effort, targeting global climate simulation (see Section 13.2.2). Both have a predefined data model, specific to their application area, which is shared by all of the participating components. The means of component composition is execution order – each component has a "run" or "exec" method that is called in a sequence of the user's choosing. In both cases there are specializations to their application area that will not be detailed here. Cactus and ESMF both rely on an execution engine (sometimes referred to as a "flow executive" (Anonymous 2004)) that is orchestrated by their respective frameworks, while the CCA relies entirely on port composition; execution order and timing are left to the way components use their ports. A third, relatively new, peer component model, Palm/PRISM (Guilyardi, Budich, and Valcke 2002), incorporates both of these ideas. While Palm is a generic component framework, PRISM is a specialization dedicated to climate simulation. These have a concept similar to ports, where components are composed by matching typed data structures that flow between components. In addition, Palm has a flow executive that schedules the activation of components during the calculation.

Beyond peer component models, there are packages that are simply labeled "libraries" or "frameworks". Hypre (Chow, Cleary, and Falgout 1999; Falgout et al. 2003), Overture (Brown, Henshaw, and Quinlan 1999; Henshaw et al. 2002), PETSc (Balay et al. 1997, 2003), POOMA (Reynders et al. 1996, 2001), and Trilinos (Heroux et al. 2003, 2004) are all object-oriented class libraries that do not embrace the peer component methodology. The goal of these software packages is to make HPC program construction and execution easier, often in a specific problem domain, e.g. numerical solution of partial differential equations (PDEs). Their approaches are to provide a variety of numerical method implementations (Hypre, PETSc, Trilinos) or to define high-level abstractions for solving a particular kind of problem (Overture, POOMA). None of these packages aims to provide a specification for a general-purpose peer component model.

10 Using the CCA

Although the CCA effort is rather young, the CCA specification and the basic tools associated with the CCA environment are sufficiently stable and robust that a wide range of groups are already using the Common Component Architecture in their software development efforts. The resulting activities range from the creation of components and interfaces encapsulating important capabilities for scientific computing, or for particular scientific domains, to the development of complete component-based scientific applications.

The development of individual components and interfaces for scientific computing is an important activity because it enriches the pool of available components upon which others may draw to assemble their applications. Often developers of a specific application recognize that particular components may be of broader interest or those associated with existing numerical or scientific libraries wish to make their tools available in the CCA environment. CCA applications are typically built by combining more general, reusable components with some constructed specifically for the application in question.

It is impossible to convey here the full breadth of work already in progress which utilizes the CCA environment. However, since the ultimate goal of the CCA effort is to change the way scientific software is developed, this overview of the Common Component Architecture would be incomplete if it did not at least present a overview of how the CCA is being used.

In the remainder of this section, we describe the general process of adapting existing code into the CCA environment, or of developing a component-based application "from scratch". Section 11 reviews some of the interface and stand-alone component development activities. This is followed, in Section 12, by a more in-depth description of another important CCA research activity, facilitating the development of coupled simulations. Finally, in Section 13 we sample the diversity of CCA-based application development.

10.1 Designing Component-Based Software

Scientific simulation codes have traditionally combined a set of numerical algorithms to solve physical and mathematical models for a specific set of problems. In traditional scientific software, these models and algorithms are integrated under a common design and tightly integrated using common data structures. Their identification as "modules" in an overall solution strategy, however, is typically straightforward.

Whether component-based or not, the natural decomposition of an application is often recursive in nature and intuitive to practitioners in the field. For example, in a multi-physics code, such as a global climate simulation, disparate physics are governed by separate sets of partial differential equations (PDEs), and the first stage of decomposition is typically to distinguish between the different physics (i.e. atmosphere, ocean, etc.). These are further decomposed based on the general numerical procedures required: solvers, domain discretizations, physics subassemblies, etc. A third step of decomposition typically involves implementing the details of the specific equations (in physics modules), mathematical processes, or numerical algorithms. Pervasive data structures and operations associated with them can often be encapsulated into data objects (in the sense of object-oriented programming) used by many modules of the application.

Componentization might naturally occur at any level of this decomposition, perhaps at different levels in different parts of the application. It may evolve as the application architects and developers modify their requirements, find new opportunities for code reuse, or wish to generalize parts of the code. A coarse-grain decomposition has the attraction of simplicity in defining interfaces and interactions and is often the first step in componentizing an application. However, this approach tends to offer fewer opportunities for code reuse because implementation details (the third stage of decomposition, above) are typically mixed into the higher-level science drivers. Finer-grain decompositions allow for better exposure of the details of specific physics equations and numerical algorithm choices, making them more amenable to reuse and easy replacement. However, what one gains in flexibility and reusability, one may lose in simplicity. Such approaches require sophisticated users who understand the tools individually as well as their interactions. This dilemma is not unique to the CCA. Other component-like toolkits (e.g. AVS, Matlab, LabView) present similar issues.

In designing a component-based application, the interfaces between components become a key element of the design. Interface design is in many respects a social issue – a primary point of cooperation among developers. Depending on the requirements, interfaces may be designed with varying degrees of generality, ranging from being specific to a group of components or a particular application, to being useful across a range of applications developed by multiple research groups and possibly across multiple scientific domains. Clearly the level of effort required to develop an interface scales with its intended generality. However, as is the case with some widely used numerical libraries (i.e. BLAS, LAPACK, and others), the payoff can be high for a general interface capable of supporting many applications and a variety of implementations. In most cases, an interface to a CCA component can be a straightforward translation of the original software's interface in to SIDL. However, some researchers find that the componentization step is an opportunity to redesign the interface, based on experience with the original, and use a somewhat thicker wrapper to adapt the new interface to the existing code within the component. It is also important to realize that different aspects of a simulation may lend themselves to different ways of accessing the functionality of a component. This can be provided by a component having multiple ports implementing related interfaces. For example, a component that manages unstructured mesh data may have a port for entity-based access to the mesh and another for array-based access. The former would provide full capability but lower performance due to the generality of the data model used; the latter would

provide high-performance access to more limited functionality.

10.2 Componentizing Existing Software

Once the component architecture has been determined and the interfaces among components specified, it is necessary to implement the components themselves. The CCA specification is designed to make it easy to adapt existing code into components. For most scientific codes, the basic strategy is as follows:

1. Write the wrappers that adapt the source code generated from the SIDL interface definitions to the interfaces in the existing code. The complexity of this step depends on how close the SIDL interface is to the original.
2. Implement the `setServices()` method for each component. This tells the framework what ports the component provides and uses. If the component requires a signal from the framework when it is about to be destroyed, so that it can perform cleanup operations, the `ComponentRelease` interface must also be implemented.
3. Modify existing tightly coupled implementations containing direct method calls so as to access the same functionality through ports. This involves adding `getPort()` and `releasePort()` calls and modifying the original method call to use the port handle returned by `getPort()` and comply with the new interface. Some modification of internal data objects may be necessary in order to use them as arguments in calls on other ports (for example, a native array being adapted to a SIDL array via the `borrow()` call). As with Step 1, the extent of modification will depend on the granularity of componentization and on how close the new interfaces are to the existing ones.

11 Reusable Scientific Components and Interfaces

In addition to developing the CCA model and a suite of tools implementing it, one of the important goals of the CCA effort is to promote the development of domain-specific interface specifications and parallel scientific components which can be reused across multiple applications. The existence of reusable components and interfaces gives new CCA users a base of "tools" to draw upon in the development of their component-based applications.

In this section, we describe some of our current activities in these areas. This work is in collaboration with applications scientists and domain specialists, with whom we are also exploring research topics including quality of service issues related to robust, efficient, and scalable performance.

11.1 Common Interface Development

The development of common interfaces that support a large number of applications is critical to the notion of plug-and-play scientific computing. Common interfaces are especially important for scientific data components, as the interfaces for numerical tools and application components often include a certain degree of specificity of the data structures used as input and output arguments. We are working with domain experts in the Terascale Simulation Tools and Technologies (TSTT) (Glimm, Brown, and Freitag 2005) and APDEC (Colella 2005) groups to develop common interfaces for two broad areas of scientific data components, namely, interfaces for (structured, unstructured, and adaptive) mesh access and a descriptor for dense arrays distributed across the processes of a parallel computer. These data-centric components will define the layout of data across processes and form the *lingua franca* for other components that manipulate the data. We are also working with the Terascale Optimal PDE Simulations (TOPS) (Keyes 2005) group to define component interfaces for linear, nonlinear, and optimization solvers. In addition, we engage the high-performance scientific community at large to participate in similar dialogs in their particular areas of expertise.

11.2 High-Performance Components

We have developed (internally and through collaboration) high-performance production components that are used in scientific applications, as well as prototype components that aid in teaching CCA concepts. A common theme in this work is combining application-specific components, as introduced in the previous sections, with more general purpose ones that can be reused across a range of applications. These freely available components include various service capabilities, tools for mesh management, discretization, linear algebra, integration, optimization, parallel data description and redistribution, visualization, and performance evaluation. We have also developed a variety of component-based scientific applications that demonstrate component reusability and composability, including several that employ domain-specific interfaces defined by CCA working subgroups for scientific data and parallel data redistribution. Several of these applications implement PDE-based models using either adaptive structured meshes or unstructured meshes (McInnes et al. 2004), while others solve unconstrained minimization problems that arise in computational chemistry (Kenny et al. 2004).

These components and applications, many of which are available from `http://www.cca-forum.org/`

software.html, serve as part of CCA tutorial material and have provided a starting point for developing interfaces in several scientific applications. This software is based on a strong foundation in the form of rich parallel tools that already used abstractions in their design, including CUMULVS (Geist, Kohl, and Papadopoulos 1997; Kohl and Geist 1999), Global Arrays (Nieplocha, Harrison, and Littlefield 1996; Pacific Northwest National Laboratory 2004a), GrACE (Parashar et al. 2004), CVODES (Hindmarsh and Serban 2002), MPICH (Argonne National Laboratory 2003), PETSc (Balay et al. 1997, 2003), PVM (Geist et al. 1994), and TAO (Benson, Mclnnes, and More 2001; Benson et al. 2003). A partial list of components follows.

11.2.1 Utilities and services

- *Services in Ccaffeine.* B. Allan, R. Armstrong, S. Lefantzi, and E. Walsh (SNL), M. Govindaraju (Binghamton). The CCA specification treats framework services exactly like CCA components except that the port that embodies the service is *always* connected to the component. A number of such services are available in the Ccaffeine framework. The parameter port service is the most commonly used, and it allows a user to set parameters on a component interactively. Another service allows the connection of the original "classic" (C++ only) ports to Babel components, thereby accommodating legacy CCA software. Additional utility services include permitting a component to access MPI, to receive connection events, and to establish its own interactive window with a user.
- *Performance Observation.* S. Shende and A. Malony (University of Oregon), C. Rasmussen and M. Sottile (LANL), and J. Ray (SNL). The TAU (Tuning and Analysis Utilities) performance observation component (version 1.6) provides measurement capabilities to components, thereby aiding in the selection of components and in creating performance aware intelligent components; see Shende et al. (2003) and Ray et al. (2004) for further details. This component is currently used in combustion applications discussed in Section 13.1. Future plans include incorporation into a variety of other simulations, including a quantum chemistry application (see Section 13.3), in order to provide comprehensive inter- and intra-component performance instrumentation, measurement and analysis capabilities.

11.2.2 Data management, meshing, and discretization

- *Global Arrays.* M. Krishnan and J. Nieplocha (PNNL). Many scientific applications rely on dense distributed arrays. The Global Array (GA) library (Nieplocha, Harrison, and Littlefield 1996; Pacific Northwest National Laboratory 2004a) provides an extensive set of operations on multidimensional dense distributed arrays. A rather unique capability is the support in GA for the shared memory programming model, where arrays can be accessed as if they were located in shared memory. This is accomplished using one-sided communication operations that transfer data between the local memory of the calling process and the arbitrary sections of distributed/shared arrays.
  We developed a GlobalArray component that provides interfaces to the full capabilities of GA, including the data- and task- parallel operations. This component provides three ports: GlobalArrayPort, DADFPort and LinearAlgebraPort. GlobalArrayPort offers interfaces for creating and accessing distributed arrays. These interfaces are intended to support the collection of global information and creation of GlobalArray objects. All details of the data distribution, addressing, and data access are encapsulated in the GlobalArray objects. The LinearAlgebraPort provides core linear algebra support for manipulating vectors, matrices, and linear solvers. Some of the linear algebra operations are implemented internally, and others are provided through interfaces to third-party parallel linear algebra libraries such as ScaLAPACK (Blackford et al. 1997). DADFPort offers interfaces for defining and querying array distribution templates and distributed array descriptors following the API proposed by the CCA Scientific Data Components Working Group. The GlobalArray component is currently used in applications involving molecular dynamics and quantum chemistry, as discussed in Sections 6 and 13.3; further details are in (Benson et al. 2003).
- *TSTTMeshQuery.* L. F. Diachin (LLNL). This component is a prototype of an unstructured, triangular mesh component that supports the TSTT mesh query interface (Glimm, Brown, and Freitag 2005) for access to node and element geometry and topology information. It uses opaque tags to support user-defined data; further information is in (Norris et al. 2002). This interface is sufficient to implement linear, finite-element discretization for diffusion PDE operators. This approach will be extended to several other TSTT-compliant mesh components built from existing DOE software that support a wide range of two and three-dimensional meshes. Such components will be used to demonstrate the utility of interchangeable and interoperable meshing infrastructures in the solution of PDE-based applications.
- *FEMDiscretization.* L. F. Diachin (LLNL). This component provides linear, finite-element discretizations for commonly used PDE operators and boundary conditions. It currently employs unstructured triangular meshes, through the TSTTMeshQuery component, and provides matrix and vector assembly routines to create linear systems of equations in simple PDE-based applications; see (Norris el al. 2002) for details.

The FEMDiscretization component approximates advection and diffusion operators as well as Dirichlet and Neumann boundary conditions, with either exact or Gaussian quadrature. It uses the TSTTMeshQuery and LinearSolver ports. This interface is expected to evolve as the TSTT discretization library is developed, and this prototype component will be replaced with a more sophisticated variant that supports multiple discretization schemes and mesh types. Such components will be used to demonstrate the utility of interchangeable and interoperable discretization strategies in the solution of PDE-based applications.

- *GrACEComponent.* J. Ray (SNL). This component discretizes a domain with a SAMR mesh and implements regridding to preserve resolution and to load-balance the mesh. As a wrapper around the GrACE library (Parashar et al. 2004) developed by M. Parashar of Rutgers University, GrACEComponent takes care of all the geometric aspects of the mesh (size and location of patches, their resizing due to regridding, and load-balancing). This component also serves as a factory for a "Data Object" that contains data on all the patches. This data object takes care of message passing for ghost cell updates. This component is used in the combustion applications discussed in Section 13.1, and will continue to evolve to incorporate the latest GrACE features.
- *HODiffusion and SpatialInterpolations.* C. Kennedy and J. Ray (SNL). These components use an underlying FORTRAN 77 library developed by C. Kennedy that implements high-order (orders 2–8) finite difference stencils, including both first and second derivatives. HODiffusion supports both symmetric and skewed stencils for collocated and staggered output and calculates high order diffusion fluxes using these stencils. The SpatialInterpolations component supplies the prolongation and restriction operators between SAMR patches at two adjacent levels of refinement, where the order of interpolation has to be commensurate with the spatial discretization in HODiffusion. These components currently handle diffusion transport subassembly in the combustion applications discussed in Section 13.1; future plans include continued testing on hierarchical grids.

11.2.3 Integration, optimization, and linear algebra

- *CvodesComponent.* R. Serban (LLNL). This component includes ports both for a generic implicit ODE integrator (OdeSolverPort) and for an implicit ODE integrator with sensitivity capabilities (OdeSolverSPort). CvodesComponent is based on CVODES (Hindmarsh and Serban 2002) and is used for chemistry integration in the combustion applications discussed in Section 13.1.
- *TAOSolver.* S. Benson, L. C. McInnes, B. Norris, and J. Sarich (ANL). This component implements a simple OptimizationSolver interface for unconstrained and bound constrained optimization problems; see (Norris et al. 2002) and (Benson et al. 2003) for details. The underlying optimization solvers are provided by the Toolkit for Advanced Optimization (Benson, McInnes, and Moré 2001; Benson et al. 2003) and include Newton-based methods as well as limited-memory variable-metric algorithms that require only an objective value and first order derivative information. TaoSolver employs external components for parallel linear algebra, where current support includes both Global Arrays and PETSc. TaoSolver is used within applications involving molecular geometry optimization and molecular dynamics, which are further discussed in Sections 6 and 13.3. TaoSolver is the basis for an evolving optimization solver component that will employ linear algebra interfaces under development within the TOPS group.
- *LinearSolver.* B. Norris (ANL). This component provides a prototype port for the solution of linear systems. These interfaces will support common interfaces for linear algebra that are under development within the TOPS group. Future work will include transitioning this component, as well as others such as TaoSolver and FEMDiscretization, to use the new TOPS interfaces, so that they can utilize the full suite of linear algebra software available within the TOPS center.

11.2.4 Parallel data description, redistribution, and visualization

- *DistArrayDescriptorFactory.* D. Bernholdt and W. Elwasif (ORNL). This component provides a uniform means to describe dense multi-dimensional arrays and is based upon emerging interfaces from the CCA Scientific Data Components Working Group.
- *CumulvsMxN.* J. Kohl, D. Bernholdt, and T. Wilde (ORNL). Building on CUMULVS (Geist, Kohl and Papadopoulos 1997; Kohl and Geist 1999) technology, this component provides an initial implementation of the M × N parallel data redistribution interfaces that are under development by the CCA M × N Working Group. CumulvsMxN is designed to span multiple CCA frameworks and to pass data between two distinct parallel component applications. See Section 12 for further information.
- *ParticleCollectionFactory.* J. Ray (SNL) and J. Kohl (ORNL). This component is a prototype for M × N parallel data redistribution of combustion data for use in post-processing in the applications discussed in Section 13.1. The ParticleCollectionFactory component imitates a patch on a SAMR grid as a "particle," which can then be employed for data redistribu-

tion and post-processing. Future plans include increasing the robustness of the code and using it in off-machine, concurrent post-processing.

- *VizProxy* J. Kohl and T. Wilde (ORNL). This component provides a companion M × N endpoint for extracting parallel data from component-based applications and then passing this data to an external front-end viewer for interactive graphical rendering and exploration. Variants provide general-purpose components for interactive visualization of data based on structured meshes as well as unstructured triangular meshes. Support for particle-based data is under development. Using the CUMULVS viewer library and protocols, a variety of commercial and public domain visualization tools can be utilized as the front-end user interface (Wilde, Kohl, and Flanery 2002, 2003). Currently provided front-ends include a simple 2D "slicer" viewer and a 3D viewer for AVS 5; additional viewers are under development for VTK, AVS/Express and the CAVE.

12 M × N Parallel Data Redistribution

Because parallel simulations are increasingly being coupled to form complex multi-physics and multi-scale applications, developers face a growing need to efficiently manage the movement of distributed data from one simulation component to another. Typically, the sending and receiving sides in such couplings utilize different distributions of data across their parallel environments. When such coupled simulations are run in a multiple program/multiple data paradigm, even the number of parallel processes on each side may be different (M and N processes, respectively). The CCA refers to this as the "M × N parallel data redistribution" problem, illustrated in Figure 6.

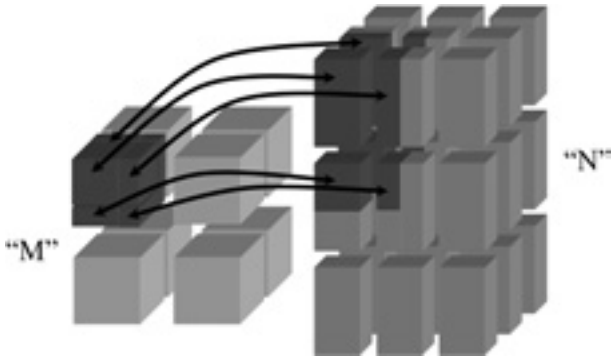

Fig. 6 A schematic of an M × N parallel data redistribution problem. In this case, M = 8 and N = 27, and the dark patches illustrate the fact that data originating on a single process may be redistributed across many processes on the receiving end.

Solving the M × N problem is a complex and tedious programming task, which involves determining the mapping of data between the two decompositions (often referred to as the communication schedule) and then efficiently implementing an all-to-all communication using this schedule. At present, such coupling in most such applications is done in an application-, or domain-specific fashion, limiting the type of data distributions supported. Often, separate M × N couplers are created for each pair of models in a multi-physics simulation, with no generalized or reusable solutions.

In so far as a component-based environment is intended to facilitate the development of large coupled simulations, the development of a more general approach to the M × N problem is an important research area within the CCA effort. Our long-range view of this research bridges applications, components, interfaces, and the core CCA environment. We are starting with the development of interfaces and component-based implementations that provide a generalized M × N capability, which applications can call to affect the transfer of data from one group of components to another. Another approach involves integrating the mechanics of this data redistribution directly into the CCA framework, thereby removing from the user the need to explicitly invoke the data transfers. This approach can also be generalized into "parallel remote method invocation" (PRMI), in which not only are data redistributed among disparate process groups, but method invocations themselves are coordinated between distinct groups of parallel processes. Finally, we also look beyond "simple" data redistribution toward ways of generalizing the additional interfacial processing, such as mesh interpolation, time or spatial averaging, and unit conversion. Here we provide an overview of the current status of progress toward general component- and framework-based M × N capabilities.

12.1 M × N Interfaces and Component Implementations

The CCA Forum's M × N Working Group has developed a prototype interface specification for an M × N component. This effort has drawn heavily on experiences with two particular tools that offer fairly general M × N-like capabilities, based on libraries rather than components: CUMULVS (Kohl and Papadopoulos 1995; Geist, Kohl, and Papadopoulos 1997; Kohl 1997; Kohl and Geist 1999) and PAWS (Beckman et al. I998; Keahey, Fasel, and Mniszewski 2001). These tools utilize complementary models of parallel data sharing and coupling. PAWS is built on a "point-to-point" model of parallel data coupling, with matching "send" and "receive" methods on corresponding sides of a data connection. CUMULVS is designed for interactive visualization and computational steering, and so provides

protocols for persistent parallel data channels with periodic transfers, using a variety of synchronization options.

The current M × N interface therefore allows the specification of both "one-shot" and "persistent" periodic data transfers and is structured to allow for "third-party" control of a data redistribution connection. This possibility minimizes the modifications necessary for the sending and receiving components to make them "M × N aware." The minimal changes involve only providing a descriptor of the parallel data distribution for each data object of interest, and then "instrumenting" the simulation code with calls to the M × N component's `dataReady()` method at points where the data is in a "consistent" state globally. Such points in the computation occur where it would be appropriate to transfer data to the recipient, or at points where it would be appropriate for the recipient to accept new data. Control over the details of the connection – how often data exchange occurs, or even whether it occurs at all – can all be placed into a third component if desired.

Implementations of the prototype M × N interface have been built on top of both CUMULVS and PAWS, and have been used in a number of models and actual scientific simulations. This experience, together with that from application- or domain-specific M × N implementations (e.g. the Distributed Data Broker and Model Coupling Toolkit discussed in Section 13.2) and other related efforts (i.e. Chaos (Ranganathan et al. 1996; Edjlali, Sussman, and Saltz 1997; Lee and Sussman 2004)), are now being assessed as part of an effort to refine the interface specification and create more general and efficient implementations.

12.2 Framework-Based M × N

Although the component-based M × N capability is powerful, it can also be somewhat cumbersome to use, since it requires users to modify their code and explicitly manage the transfers. A much more user-friendly approach, which is also being actively pursued in the CCA, is to subsume the M × N data transfer capabilities *into* the CCA framework (i.e. as an optional CCA service) and make the framework responsible for ensuring that data is appropriately redistributed when required by method calls.

The framework-based M × N option raises a number of issues. One is that the framework must somehow obtain data distribution information for the sending and receiving of data objects. A logical approach, which has been used in the past (see Keahey and Gannon 1997), is to introduce such information into the interface definition. Research using Utah's SCIRun2 CCA framework extends the SIDL language with primitives for parallel data redistributions and also parallel remote method invocation. A modified version of Babel parses the extended SIDL specifications and generates appropriate glue code to handle the data redistribution when method invocations are made (Damevski 2003).

A second issue, alluded to above, is the fact that the framework-based approach extends the problem from "simple" data redistribution to a parallel remote method invocation. An extremely complex research task is both the specification and implementation of an environment in which arbitrary subsets of M processes can, in parallel, invoke methods on arbitrary subsets of N processes, including the ability to designate the appropriate redistribution for all of the method arguments. In addition to the work using SCIRun2, with PRMI directives in the SIDL, researchers at Indiana University have developed DCA (Bertrand and Bramley 2004), an experimental distributed CCA framework based on MPI, as a tool to explore PRMI in greater detail.

13 Scientific Applications

Groups adopting the Common Component Architecture environment for the development of their applications do so for a variety of reasons. For example, they may be particularly interested in the black-box nature of components to facilitate the creation and evolution of a computational toolkit. They may need more efficient and flexible ways to assemble complex coupled simulations or want to refactor code to make it more modular. They may be interested in increasing interoperability with other similar applications in the same scientific domain. Alternatively, they may want to more easily leverage software developed by experts in other domains.

In this section, we present overviews of CCA-based application development activities in the areas of combustion simulation, global climate modeling, and quantum chemistry. These simulations illustrate the diversity of CCA applications and reasons for adopting CBSE for scientific computing.

13.1 Combustion Modeling

One of the most successful implementations of the CCA paradigm to date is in combustion modeling. The endeavor, which started in 2001 within the context of the SciDAC-funded (US Dept. of Energy 2003) Computational Facility for Reacting Flow Science (CFRFS) project (Najm et al. 2003), seeks to create a facility for the high fidelity simulation of flames, involving realistic physical models, nonlinear PDEs, and a spectrum of time and length scales. Given the complexity of the problem and the multiplicity of physical and mathematical models required for the task, a component based approach was clearly indicated. CCA was chosen primarily for its high performance and simplicity.

The target is the modeling of combustion problems defined in simple geometries, i.e. logically rectangular

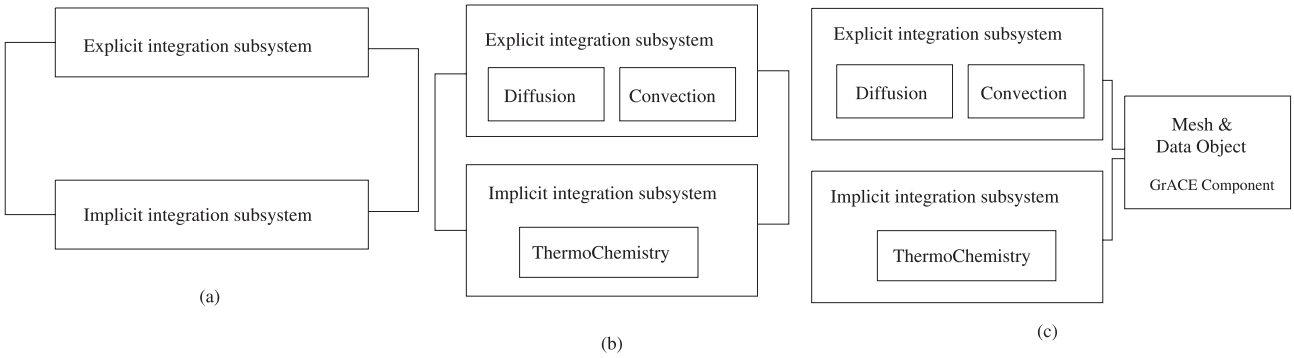

Fig. 7 Decomposition sequence for the combustion modeling software. (a) shows the coarse decomposition along numerical lines into explicit and implicit integration subsystems. In (b), the dictates of physics result in the separation of physical processes as separate subsystems, which are then populated by various elementary physical and mathematical models. In (c), the domain decomposition and domain discretization functionalities are formally removed to form new subsystems. The process follows closely the general approach described in Section 10.1.

domains, employing structured adaptively refined meshes (SAMR). Evolution of the flow quantities (density, momentum, temperature, species concentrations, etc.) is governed by a set of nonlinear PDEs, which have the general form:

$$\frac{\partial \mathbf{\Phi}}{\partial t} = \mathbf{F}(\mathbf{\Phi}, \nabla\mathbf{\Phi}, \nabla^2\mathbf{\Phi}, \ldots) + \mathbf{G}(\mathbf{\Phi}), \qquad (1)$$

where $\mathbf{\Phi}$ is the vector of flow variables at a given mesh point. Numerically, **G** involves the variables only at the same mesh point, while **F** involves spatial derivatives which are computed using finite difference or finite volume schemes and consequently depend upon the mesh point and its close neighbors. The dependence of **F** on neighboring points requires that steep variations of $\mathbf{\Phi}$ be fully resolved in scattered time-dependent regions of the domain, which is usually achieved by increasing the grid density locally. The SAMR technique employed overlays a uniform coarse mesh on the domain and dynamically generates a mesh hierarchy of rectangular patches with different grid densities based on an error threshold. Physically, for the reacting flow systems described by equation (1), **F** expresses diffusion and convection in the flow, and **G** expresses the chemically reacting source terms of the flow.

In order to design a component-based software infrastructure to model reacting flow systems, one needs to consider and classify two aspects of the simulation: the physics of the system and the numerical scheme that will be employed. This classification determines the software subsystems of the infrastructure, that is, collections of components that embody specific physical or numerical functionality. For the class of reacting flow problems in which we are interested, the main physics sub-problems are: diffusion and convection (**F**), and chemical reactions (**G**). Numerically **G** (equation 1) is stiff, i.e. the ratio of the largest and the smallest eigenvalues of $\partial\mathbf{G}/\partial\mathbf{\Phi}$ is large, while **F** is non-stiff. Time integration is "decoupled" by employing an operator-splitting scheme (Strang 1968; Sportisse 2000). The stiffness problem (**G**) was addressed using an implicit backward difference formulation and the non-stiff problem (**F**) using an explicit integrator. Thus, the numerical scheme naturally divides into two separate integration software subsystems: the *Implicit Integration Subsystem* and the *Explicit Integration Subsystem* (Figure 7a).

The physics of the problem, on the other hand, dictate a finer decomposition: for the Explicit Integration Subsystem a *Diffusion Subsystem* and a *Convection Subsystem*, and for the Implicit Integration Subsystem a *Chemical Reactions Subsystem* (Figure 7b). Also, the code is fully parallel with structured adaptive mesh refinement; we need to employ a "data object" to account for efficient domain decomposition and a "mesh object" to perform the adaptive mesh regeneration. Both objects are accommodated by the componentized version of the GrACE library (Parashar et al. 2004) as a separate software subsystem (Figure 7c). This path of classification results in the first

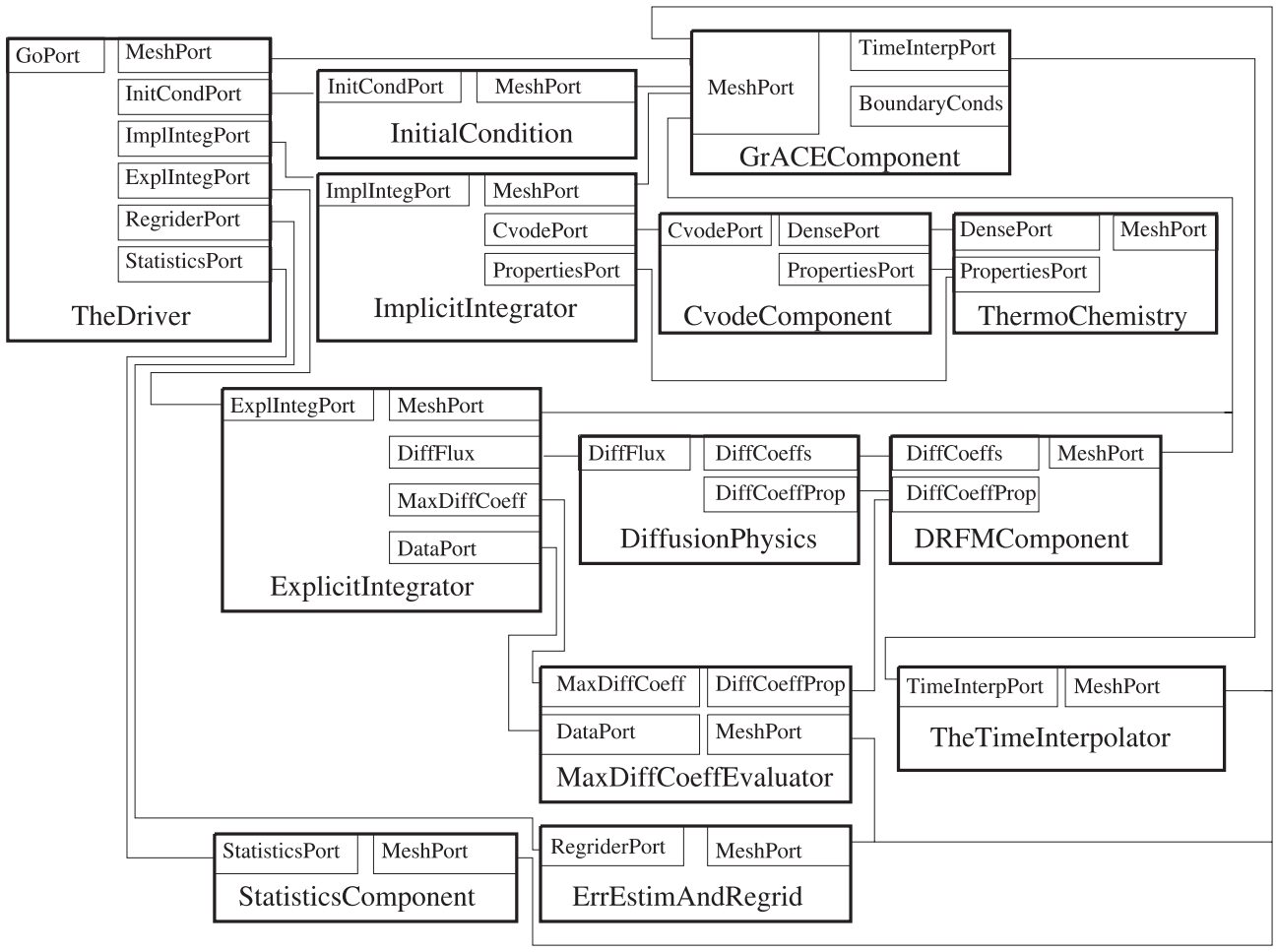

Fig. 8 The final results of the decomposition described in Figure 7 for a reaction-diffusion problem. Large boxes represent components, smaller rectangles represent *provides* ports (left justified within component box) and *uses* ports (right justified). Lines connect *uses* and *provides* ports. Details can be found in (Lefantzi, Ray, and Najm 2003).

design approximation for the reacting flow toolkit. The next step is a further decomposition of the software subsystems into components, resulting in a complex software infrastructure where each individual module is reusable and easy to maintain. Details are beyond the scope of this paper and can be found in (Lefantzi, Ray, and Najm 2003). A schematic reaction-diffusion code assembly is presented in Figure 8.

Our experience with developing CCA-based scientific components for combustion modeling has demonstrated the following:

1. General purpose components, implementing a particular numerical or physical functionality, are reused in various code assemblies. References (Lefantzi and Ray 2003; Lefantzi, Ray, and Najm 2003) document our experience with numerical integration components (implementing a number of explicit Runge-Kutta and backward difference algorithms), physical models (diffusion coefficients of gaseous mixtures calculated via a number of different approaches), data and mesh objects, along with the rationale for particular code assemblies.

2. Once a particular decomposition has materialized, the modular nature of component-based codes allows the easy replacement of an individual component by another that supplies the same functionality but may exhibit different characteristics, such as higher accuracy. References (Lefantzi, Ray, and Najm 2003; Lefantzi et al. 2003; Ray et al. 2003) document how we exploited this modular nature to experiment with multiple models of diffusion coefficients of gaseous mixtures (using constant Lewis number approximations, mixture averaged models based on Lennard-Jones potentials, etc.) and higher order spatial discretizations in the evaluation of the diffusion term (**F** in equation 1).
3. One of the most significant advantages offered by components is the ability to perform "unit" testing, i.e. each component is tested individually, in the absence of the final assembly where they will be typically used. This simple testing process with its quick turn-around time allows one to debug and incorporate a verified component in the final assembly easily. While this approach is not particular to components, their modular nature allows their incorporation into the final assembly *without any code modification*, thus drastically reducing the possibility of introducing errors.

13.2 Global Climate Modeling

Simulation using coupled climate models is the chief tool used by scientists to understand the Earth's climate system's past (i.e. paleoclimate), its internal variability and sensitivity to external forcings such as insolation and trace chemical concentrations (e.g. carbon dioxide), and to construct scenarios for its future change. The fact that climate scenarios are constructed from large history datasets, resulting from long model integrations that span the century-to-millennial timescale, makes climate modeling a "grand challenge" computational science problem. The high degree of complexity in the number of physical processes modeled and the numerous interactions among the system's components (atmosphere, ocean, sea-ice, river, biosphere, and cryosphere) result in a correspondingly high level of software complexity. For these reasons, the climate community is examining better methods to write and maintain their applications. Several of these efforts employ the CCA in various aspects of the problem (Larson et al. 2004).

13.2.1 The UCLA Coupled Climate Model and the Distributed Data Broker An ongoing project at JPL consists of a demonstration of CCA with three specific climate components: the UCLA Atmospheric General Circulation Model (AGCM) (Wehner el al. 1995), the LANL POP Ocean General Circulation Model (OGCM) (Smith, Dukowicz, and Malone 1992), and the UC Berkeley/UCLA Distributed Data Broker (DDB) (Drummond et al. 2001). Neither the AGCM nor OGCM will be described in detail in this paper. In brief, they are both explicit time-marching schemes that read in initial conditions, define boundary conditions (held fixed, except at the ocean-atmosphere interface), and then start marching forward in time. The atmosphere and ocean models need to exchange flux and state data at their interface. Flux data exchanged include radiative, fresh water, and momentum fluxes. State data include sea surface temperature and ocean albedo.

The DDB is not as well known as the AGCM and OGCM, so it is described here in a bit more detail. The DDB is used in two phases. First, components must register with it by signing up to produce and/or consume data. In this phase, they also describe the global view of the data that they have/want, and the mapping of this data onto the processes. The DDB then calculates what messages will have to be sent to transfer data. This calculation is done on a single process at the end of this first phase, but the information about the messages themselves is stored on the processes that will be sending or receiving data, and the actual data exchange is also distributed. In the second phase, the messages are actually sent when the components on a given process signal that they have produced or are ready to consume data. Also, any interpolation (needed if the grids are not coincident or if no data exists at some set of points) is done by the DDB.

The usual approach to running a simulation using this coupled climate model would be as a multiple program/multiple data (MPMD) application, with the AGCM and OGCM each running on subsets of the processes, and the DDB running on all processes to couple the AGCM and OGCM. In this case, using the Ccaffeine framework (Allan et al. 2002), the AGCM was cast as the overall driver of the simulation (providing a `Go` port and using ports provided by the DDB), and the AGCM determined the processes on which the OGCM should be run. An alternative approach would be to move the driver functionality and partitioning decisions from the AGCM to a separate driver component and use a full MCMD model for the application. Demonstrations of such capabilities were developed for the Ccaffeine framework, but have not yet been incorporated.

The DDB component is a domain-specific solution to what the CCA refers to as the "M × N parallel data redistribution" problem. In addition to being directly useful in other applications, both in climate and other fields, experience with this component also informs the CCA's efforts to develop a more general parallel redistribution capability (Section 12).

13.2.2 The Earth System Modeling Framework The Earth System Modeling Framework (ESMF) (Killeen, Marshall, and da Silva 2003) is a national effort to develop

common software utilities ("ESMF infrastructure") and coupling services ("ESMF superstructure") in the climate, weather, and data assimilation domains. The fifteen application testbeds for the ESMF include the Community Climate System Model (CCSM), the Weather Research and Forecast (WRF) model, and models from the NOAA Geophysical Fluid Dynamics Laboratory, the National Centers for Environmental Prediction, MIT, and the NASA Global Modeling and Assimilation Office. Each of these applications either is a multi-component, coupled modeling system or has the potential to be used in such a modeling system. As collaborators on the ESMF project, these groups will use the ESMF component constructs to represent their atmosphere, ocean, land, sea-ice, data assimilation, and coupler components and the ESMF regridding and communication utilities to connect these components into applications.

The objectives of ESMF are to facilitate:

- an extensible, hierarchical architecture for structuring complex climate, weather and data assimilation models;
- interoperability of modeling components among major centers;
- lower cost of entry into Earth system modeling for university and smaller groups; and
- sharing of common software utilities and services.

Next-generation Earth modeling systems require the incorporation and evaluation of new physical processes such as biogeochemistry and atmospheric chemistry, which may be represented as new sets of software components. Thus, the advances that the ESMF offers – the ability to organize large models, to easily incorporate new components, and to compare different component implementations (e.g. different representations of the ocean) – are critical capabilities for scientific progress.

Like the CCA, the ESMF provides a generic component abstraction that connects user-defined components to the framework through a `SetServices()` method. Unlike the CCA, the ESMF customizes methods and data structures for Earth system models, and it provides utilities and services specific to the climate/weather/data assimilation domain. For example, the data exchanged among ESMF components is packed into a data structure called an `ESMF_State`, which is a container for data types that represent geophysical fields. Since ESMF applications typically perform a setup, time step through numerical solution of PDEs, and then shut down, ESMF codes are required to have `Initialize()`, `Run()`, and `Finalize()` methods.

The ESMF and CCA groups have worked closely together to ensure interoperability between the ESMF and CCA frameworks, i.e. to guarantee that ESMF components may be used within the CCA, and that the ESMF will be able to utilize the wide variety of CCA-compliant components that exist or are under development. An ESMF-CCA prototype, described in more detail later in this section, has been developed that uses the component registration methodology and GUI from the CCA. ESMF components do not need to be modified, but can simply be wrapped to become CCA-compatible. This prototype demonstrates that interoperability between the CCA and the ESMF is quite easy, as similar architectural principles underlie both frameworks. Interoperability between these frameworks opens the exciting possibility of bringing Earth system modeling components into the CCA arena, and of making CCA tools available to a host of Earth system applications.

ESMF-CCA Interoperability Several notable distinctions exist between the CCA and the ESMF component frameworks. First and foremost, while the CCA provides a general component model, the ESMF provides a specialized component model tailored to a specific application domain. Additionally, the CCA enables dynamic composition of component-based applications at run time, while the ESMF dictates static linkage of a component hierarchy. On the other hand, the CCA, by its nature, does not provide concrete classes with which to build components, while the ESMF does (the ESMF infrastructure). In this sense, the CCA is component framework, while the ESMF is an application framework.

These differences suggest the intriguing possibility of using the CCA component model and the ESMF application framework to build climate-related applications, thereby bringing the dynamism of the CCA component model to this community. This would also allow coupling to CCA components from within the ESMF framework, so that ESMF components could step outside of bounds of the `ESMF_Initialize()`/`ESMF_Run()`/`ESMF_Finalize()` paradigm if a richer component interface is required. Furthermore, the union of these two frameworks would provide the CCA community with access to ESMF components.

Bridging the CCA and ESMF to create a joint ESMF-CCA component model is fairly straightforward. ESMF components are wrapped with a thin layer of C++ code to provide the CCA component veneer. This wrapper allows an ESMF-CCA application to be composed by selecting from a palette of ESMF-CCA components using the Ccaffeine CCA framework. A special version of the ESMF component registration function is provided for all ESMF-CCA components, to allow them to register their ESMF interface functions (initialize, run, finalize). Once all components have been created and connected, the CCA framework passes control flow over to the ESMF framework.

A major advantage of this dynamic approach is that components can be easily substituted for one another without modifying user code (as long as the swapped components

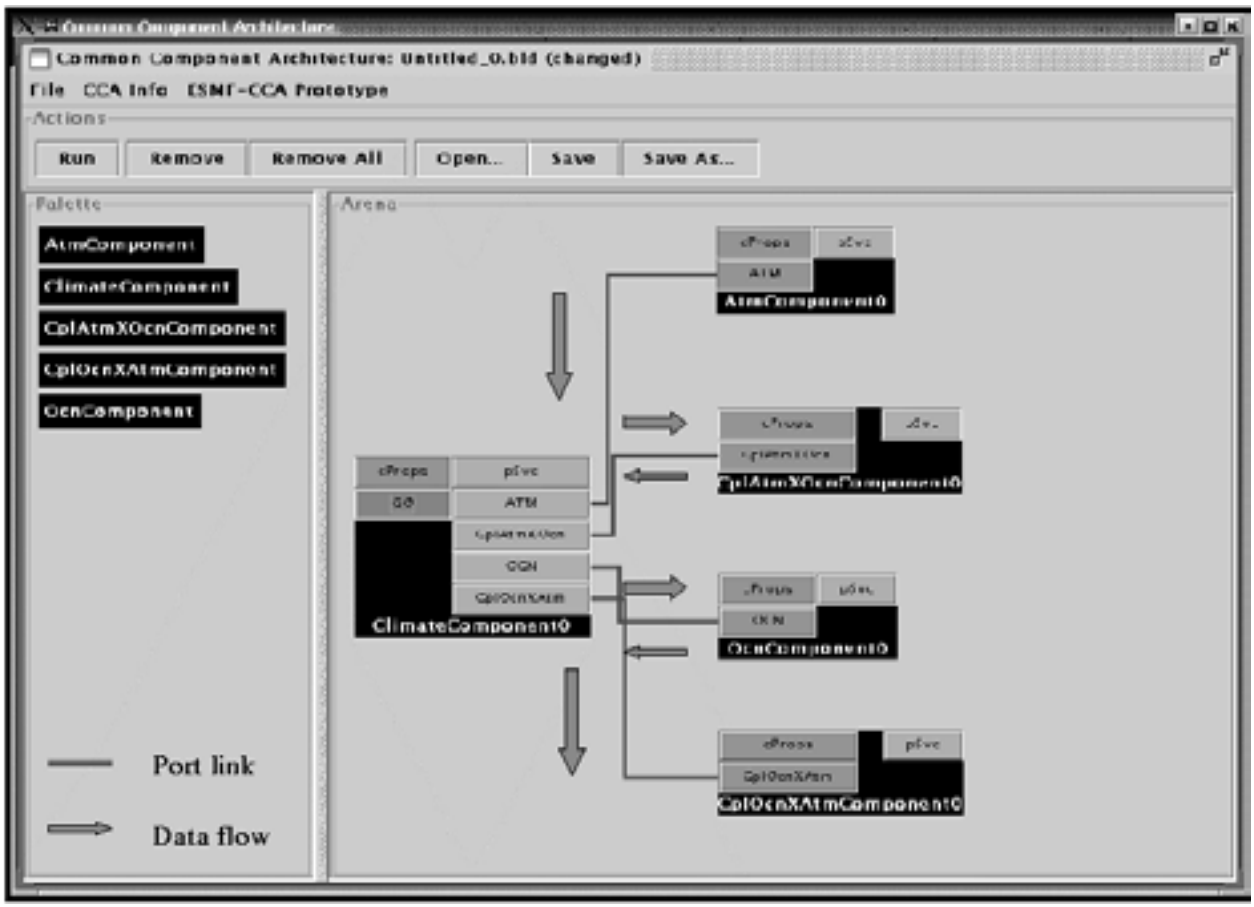

Fig. 9 CCA wiring diagram showing component relationship in a simulation of coupled atmosphere and ocean model components of the ESMF-CCA prototype.

conform to the same interface). As it now stands, one must modify ESMF superstructure components to replace a component, as the complete component hierarchy is coded into an ESMF application. We recognize that climate and weather models do not typically require run-time swapping of geophysical components, since it takes on the order of months to tune and validate these applications, and months to run long climate simulations. However, run-time swapping may be useful for communication and input/output components.

ESMF Prototypes Using the CCA To demonstrate possibilities for interoperation between the ESMF and the CCA, we have developed a prototype ESMF implementation on top of CCA tools and CCA-compliant components (Zhou 2003; Zhou et al. 2003). The CCA components additionally provide the basic methods required by the ESMF (initialize, run, finalize). Data exchange occurs through a self-describing data type similar to `ESMF_State`. We are currently working on a C++ version of the ESMF component interface so that ESMF components can be used with the CCA in a straightforward way. A model ESMF-CCA application has been implemented using the Ccaffeine framework; this project illustrates the case of sequential coupling of an atmosphere model and an ocean model (Figure 9). At the beginning of a simulation, five components are created: a driver (Climate), the atmosphere (Atm), the ocean (Ocn), a coupler from atmos-

phere to ocean (CplAtmXOcn), and a coupler from ocean to atmosphere (CplOcnXAtm). The component execution sequence of events is as follows:

1. The atmosphere component provides data at its boundary, exportAtm, to the coupler, CplAtmXOcn.
2. CplAtmXOcn uses exportAtm, transforms it into importOcn with interpolation routines, and then provides importOcn to the ocean component.
3. With importOcn, the ocean component runs its solver for the evolution equations and then provides its data at the boundary, exportOcn, to the coupler, CplOcnXAtm.
4. The coupler, CplOcnXAtm, uses exportOcn, transforms it into importAtm, and provides importAtm to the atmosphere component.
5. The atmosphere component uses importAtm, runs its solver for evolution equations, and then provides its data at the boundary, exportAtm, to the coupler, CplAtmXOcn, as in Step 1.

13.2.3 The Community Climate System Model The Community Climate System Model (CCSM) (Zhou 2003; Zhou et al. 2003) couples several mutually interacting models developed through an interagency collaboration between the U.S. Department of Energy and the National Science Foundation's National Center for Atmospheric Research. Key modules of the CCSM include models for atmosphere, ocean, sea ice, land surface, river routing, and a flux coupler. The flux coupler provides the overall coordination of the coupled model's execution and a variety of services related to the simulation data: intermodel state and flux data transfer, interpolation between model grids, calculation of fluxes and other diagnostic quantities, time averaging and accumulation of flux and state data, and merging of flux and state data from multiple model sources for use by another model.

Collaboration between the CCSM and the CCA is focused on three areas, prototyping use of CCA at (1) the system integration level, (2) the model subcomponent level, and (3) the algorithmic level. We are also exploring the use of CCA to package portions of the Model Coupling Toolkit (the foundation code on which CCSM's flux coupler is built; see (Larson et al. 2001; Ong, Larson, and Jacob 2002; Larson, Jacob, and Ong 2004)) as CCA components. Since CCSM plans to adopt the standard interfaces being developed by the ESMF project (see Section 13.2.2), this work is also being performed in collaboration with the ESMF group. The next three sections describe individual projects in this area.

Model Coupling Toolkit The current CCSM flux coupler was implemented using the Model Coupling Toolkit (MCT) (Larson et al. 2001; Ong, Larson, and Jacob 2002; Larson, Jacob, and Ong 2004). MCT is a software package for constructing parallel couplings between MPI-based distributed-memory parallel applications which supports both sequential and concurrent couplings. It can support multiple executable images if the implementation of `mpirun` used supports this feature. MCT is implemented in Fortran 90, and its programming model is friendly to scientific programmers, being based on a Fortran 90 module that declares MCT-type variables and on the invocation of MCT routines to create couplings.

The first major focus of the MCT-CCA collaboration has been the creation of a migration path between MCT-based coupling and a CCA-compliant component-based approach. The approach we are taking involves the use of the ESMF specification to bridge between the MCT and the CCA's more generic component environment. Low-level MCT functionality is being wrapped to provide an implementation of the relevant ESMF-defined interfaces and adapters to ESMF-defined data objects, such as `ESMF_State`. Currently, we support SCMD component scheduling and coupling only, but support will soon be extended to MCMD coupling as well.

We have begun the task of describing ESMF-compliant MCT interfaces using SIDL, with the goal of repackaging MCT as a collection of CCA-compliant components. Prospective MCT-based components include distributed multifield data storage, parallel sparse matrix-vector interpolation, global integrals, and averages. In addition to their use in the flux coupler context, such components can be used *within* individual models as well, as described below. Another application of the SIDL interface description is the extension of the MCT's programming model to languages other than Fortran 90. This is an example of how the Babel interlanguage interoperability tool can be used in a context outside of the CCA. Finally, like the Distributed Data Broker, the MCT is clearly another domain-specific solution to the M × N parallel data redistribution problem. The experience gained from the development and use of the MCT, as well as comparisons with the DDB, will inform the CCA's efforts to develop general M × N capabilities. In addition, once the general M × N interface is more mature, we anticipate using elements of the MCT as the basis for an implementation of a more general data redistribution component.

CCSM System Integration At the system integration level, we are prototyping the use of the CCA to cast the CCSM's component models as CCA-compliant components. We have developed skeleton components for the atmosphere, ocean, sea ice, land surface, river routing, and flux coupler. These components can then be instantiated and connected by using the Ccaffeine GUI (Figure 10); their subsequent execution is similar to the ESMF-CCA

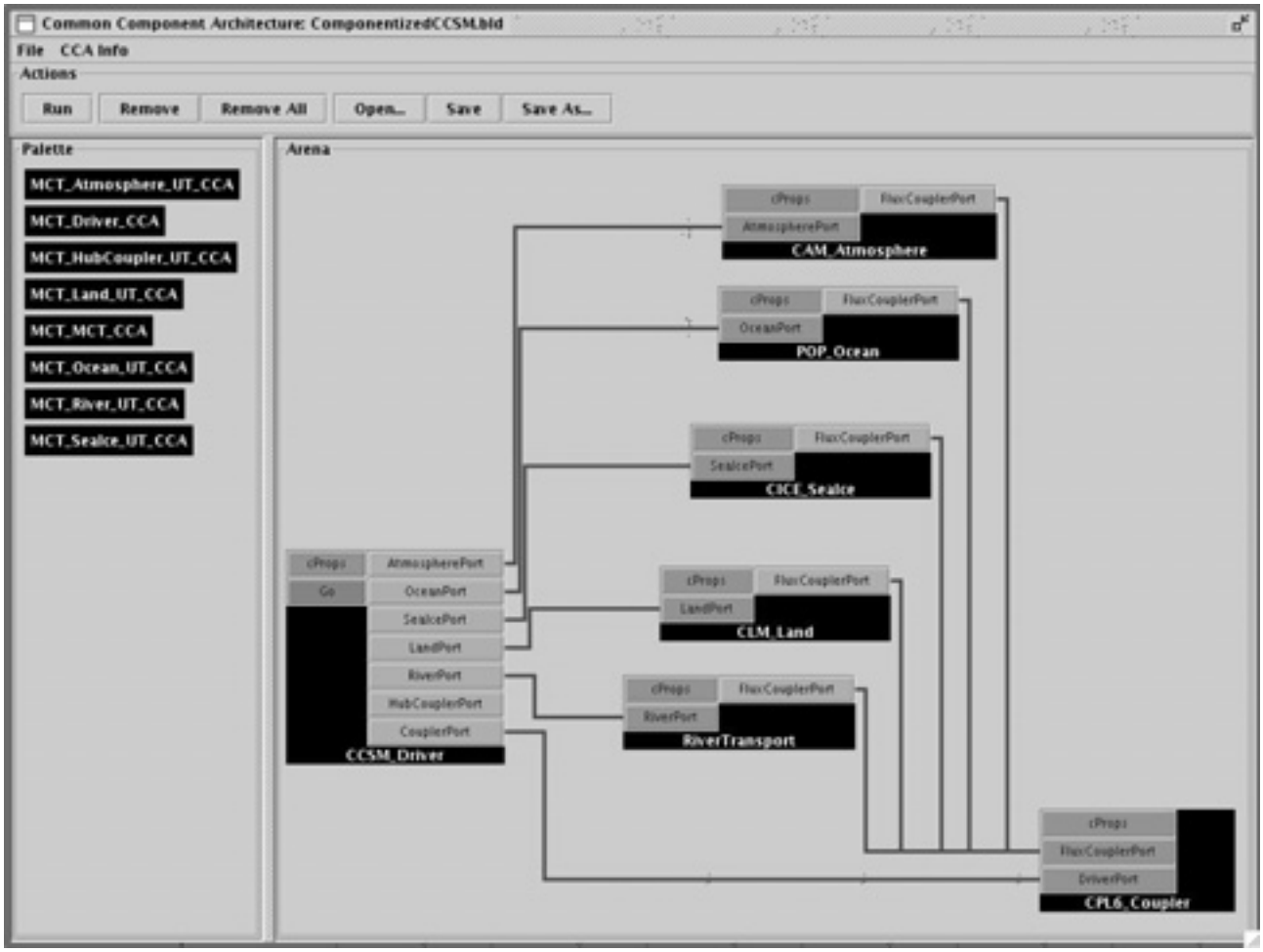

Fig. 10 Prototype of a CCA component-based system integration of the CCSM.

prototype described above. These skeleton components are being expanded to have the capabilities of the CCSM data models that are used to perform system integration tests on the CCSM coupler. The components will then entail sufficient computational complexity that we can begin to assess in the coupling context any performance costs associated with CCA components versus the current coupler.

Community Atmosphere Model An example of a CCA application targeted at the model subcomponent level is the refactoring of the CCSM's atmosphere model, called the Community Atmosphere Model (CAM) (University Corporation for Atmospheric Research 2004). CAM has been refactored to disentangle its dynamical core from its subgrid-scale physics parameterizations. This split has allowed a proliferation of dynamical cores, and CAM currently has three: a pseudo-spectral method, the Rasch-Williamson semi-Lagrangian method, and the finite-volume Lin-Rood scheme. Standardization of the interfaces to these dynamical cores and the overall subgridscale physics package is under way. Once complete, these software modules can be packaged as CCA components. This will allow one to use the CCA to

compose and invoke a stand-alone version of the CAM in which the user may plug in the dynamical core of choice. Furthermore, the developer of a new dynamical core will need only code it to the interface standard, and it will be easily integrated into the model for validation.

River Transport Model The river transport model (RTM) in the CCSM (version 2) computes river flow on a 0.5° latitude-longitude grid and uses no parallelism in its calculations. Future science goals for the RTM include support for significantly higher-resolution (up to 100 × the number of catchments) and catchment-based unstructured grids, support for transport of dissolved chemical tracers, and inclusion of the effects of water storage in the form of reservoirs. At the current resolution, the lack of parallelism in the RTM calculation does not retard the overall execution of CCSM, but the process of coupling to the RTM, which requires gathering (scattering) data to (from) a single process when sending (receiving) data to (from) the RTM, does impose a bottleneck. This communications cost exceeds the computation cost of the RTM itself.

To incorporate distributed-memory parallelism in the RTM and meet its science goals, we are developing a completely new implementation of the RTM using the MCT. In this approach, we view the problem of river transport as a directed graph with the nodes representing catchments and the directed edges representing the dominant inter-catchment flow paths. The river transport calculation is then implemented as a parallel sparse matrix-vector multiply, and the load balance for this operation can be computed by using graph partitioning. Mass conservation is ensured through the calculation of global mass integrals before and after the matrix-vector multiply. This problem is easily solved using the MCT. The ESMF-compliant MCT multifield data storage object is used to store runoff and tracer concentration data; the MCT's domain decomposition descriptors are used to describe the load balance; MCT's parallel sparse matrix-vector multiply is used to perform the water and chemical tracer transport; and the MCT's paired global spatial integral facility is used to carry out the mass balance and mass conservation calculations.

The component-based approach used for key MCT capabilities makes the path to a component-based RTM clear. Four MCT-based components will be used to implement the RTM: a distributed multifield array to store water and chemical tracer concentrations; a parallel sparse-matrix vector multiply to compute water and chemical tracer transport; a component describing the physical grid (i.e. the locations and sizes of the catchments); and a paired spatial integral component to enforce mass conservation.

13.3 Quantum Chemistry

The quantum chemistry community has an extended history of developing large, monolithic codes in which essentially all of the code, including the mathematical routines, are written by chemists. This practice has been changing, however. It is becoming much more difficult for any group to maintain expertise in all aspects of the chemistry, mathematics, and computer science. For example, it is now common for chemistry programs to use basic linear algebra subroutines (BLAS), which are bundled in libraries that have been developed by mathematicians and computer scientists to perform well. By using these libraries, chemists can concentrate on the chemical/physical problem at hand instead of worrying about the performance of the BLAS.

The developers of NWChem (Kendall et al. 2000; Pacific Northwest National Laboratory 2004b) and MPQC (Janssen, Nielsen and Colvin 1998; Sandia National Laboratories 2004) have joined with others in the CCA effort to examine the use of components in quantum chemistry. The first challenge chosen was that of optimizing a molecular geometry using NWChem and MPQC to calculate the energy, gradient, and Hessian (function, first derivative, and second derivative, respectively), TAO (Benson, McInnes, and Moré 2001; Benson et al. 2003) for the optimization capability, and PETSc (Balay et al. 1997, 2003) or Global Arrays (Nieplocha, Harrison, and Littlefield 1996; Pacific Northwest National Laboratory 2004a) for the linear algebra and management of distributed data structures. The basic component diagram is shown in Figure 11 and discussed in detail in (Kenny et al. 2004).

Molecular geometry optimizations are a very common part of quantum chemistry applications, and each code has its own special methods to reach a structure in as few function evaluations as possible. This approach makes it very difficult to reuse code since the optimization routines can be very tightly tied into the particular program. Our initial goal in this work was to decouple the optimization problem from the function evaluations so that multiple chemistry codes could be used with multiple optimization methods. Therefore, when a new optimization method is developed, it will be available to a wider chemistry community. This approach also allows users to have multiple chemistry programs available to them through a common interface, thereby allowing users to take advantage of the different algorithms available for solving a particular chemical problem.

In order to accomplish this goal, of course, interfaces needed to be standardized. In this case, a decision was made to develop a general quantum chemistry interface that satisfied the NWChem and MPQC developers' needs as opposed to developing a broader interface with the whole chemistry community. This decision was made so

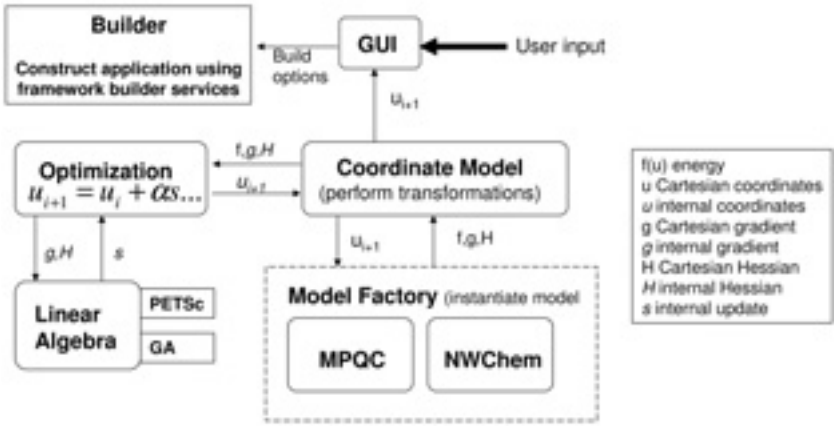

Fig. 11 A schematic representation of the CCA chemistry optimization component architecture. Image courtesy of Joseph P. Kenny, Sandia National Laboratory.

that initial progress could be made and experience could be gained. In the future, we anticipate that other chemists will be involved, at which time the interface will need to be revisited. These interfaces use SIDL specifications and Babel to create server implementations for both NWChem, which uses Fortran, and MPQC, which uses C++. Other interfaces associated with the optimization model and with the linear algebra had already been created before the chemistry developers became involved. These interfaces were used, with a few modifications, for the chemistry problem.

In this approach, initially a very coarse level of componentization was chosen. Very high level capability, such as setting molecular geometries, wavefunction type, and basis sets and then computing energies, gradients, and Hessians is made available through the quantum chemistry interface. Now that the developers have experience with the CCA infrastructure, we will be developing lower level components, such as those for building Fock matrices, calculating properties, and other mathematical constructs such as eigensolvers and diagonalizers.

We are also currently exploring the use of the MCMD execution model for carrying out independent function evaluations simultaneously and for increasing flexibility in composing programs. This type of capability will be critical to applications that rely on methods such as Monte Carlo, combined quantum chemical methods (those that use varying levels of theory to model different parts of the chemical system), and dynamics simulations where multiple, relatively independent function evaluations are required.

14 The People Behind the CCA

The small grass-roots effort that launched the CCA has grown into a sizable community of computational science researchers developing and using CCA technology, represented by the CCA Forum. Members of this community have also successfully launched a variety of funded research projects that focus on different aspects of the development and use of the CCA.

The CCA Forum can be thought of as a combination of a standards body and a user group for the CCA. The Forum has met quarterly since 1998 and is open to all interested parties. (Meeting information is available on the Forum's web site, `http://www.cca-forum.org`, and on its primary mailing list (CCA Forum 2004).) Meetings, as well as the mailing list, serve as an opportunity for discussion of issues and experiences relating to the development and use of the Common Component Architecture. The Forum also defines the CCA specification. Voting privileges are conferred on all who have attended two of the last three CCA Forum meetings as of the date of the proposal, and voting takes place via the web over a two week period. At the moment, the Forum has 24 voting members representing 11 different institutions.

The core group that launched the CCA Forum (Argonne, Lawrence Livermore, Los Alamos, Oak Ridge, Pacific Northwest, and Sandia National Laboratories, Indiana University, and the University of Utah) in 2001 created the Center for Component Technology for Terascale Simulation Software (CCTTSS) as part of the U. S. Depart-

ment of Energy's (DOE) Scientific Discovery through Advanced Computing (SciDAC) program (US Dept. of Energy 2003). The mission of the CCTTSS is to develop a better understanding of the most appropriate ways to use CBSE in scientific computing, as well as to develop the fledgling CCA ideas and prototypes into a production-quality environment for use in large-scale applications. Thus, the CCTTSS acts as the anchor for CCA development.

However, the CCTTSS is far from the only funded project relating to the CCA. Other more focused efforts investigate performance monitoring and tuning and various aspects of distributed computing with sponsorship from the DOE, NASA, and the National Science Foundation. In almost all cases, CCA users have separate funding through grant proposals that may or may not have explicitly mentioned CCA.

15 Future Work

The Common Component Architecture is a long-term research and development effort with contributions from numerous separate projects, anchored by the Center for Component Technology for Terascale Simulation Software. It is impossible to capture in this document the CCA-related research plans of all these projects and individuals. However, we can point to a number of enduring themes and highlight several specific initiatives that we expect to yield results in the near term.

15.1 Understanding of the Role of CBSE in HPC

One of the key initial motivations for the CCA was the fact that CBSE had made few inroads into HPC scientific computing with existing component environments. As a consequence, an important part of the CCA effort is to gain a better understanding of how CBSE can be best used in this area. As part of this effort, CCA researchers work closely with component and application developers not only on the software itself, but also to gain information about the use of CBSE is effecting the process. The results of this work feed back into the design of the CCA and future component technologies. They also contribute to the development of "best practices" and other guidance on how to use CBSE in scientific computing.

15.2 Development of the CCA Specification and Environment

As mentioned previously, the CCA specification is not a static document, but rather evolves as researchers find that additions or changes are needed. Likewise, the tools that implement the CCA environment are also still very much under development. New tools to improve or extend the CCA environment are also under development by numerous researchers.

15.3 Development of Interfaces and Components

Another of the primary motivations for the development of the CCA is to increase software productivity by facilitating the reuse of components. In this context, a rich library of components that can be used to simplify the creation of new applications is an attractive feature. Section 11 is a good start, but it is also just a tiny fraction of what *could* be made available as reusable components. CCA researchers will continue to engage domain experts to design common interfaces to promote interoperability and to encourage the development of components implementing these interfaces.

15.4 Parallel Data Redistribution and Parallel Remote Method Invocation

Section 12 described in some detail our ongoing effort to develop generalized tools and capabilities for model coupling and related problems. Work on parallel data redistribution has already been used in several demonstrations and applications, and the research is progressing to incorporate the initial lessons learned into an improved interface design and software implementations. Work on parallel RMI is much newer, but we expect it to produce useful results in the near term.

15.5 Richer Interface Descriptions

At present, the CCA works with a very basic syntactic definition of component interfaces in the form of method signatures, which consist of method names, argument types, and some information about memory management responsibilities. However, the full realization of the potential of plug-and-play components hinges on richer interface descriptions that enable the explicit specification of semantics and behavior in a form amenable to automated processing (Beugnard et al. 1999; Baudry, Traon, and Jezequel 2001). Such specifications would enable the exploration and development of advanced technologies and tools to address CBSE issues such as software correctness. From a component perspective, software correctness encompasses both the component implementation and its usage. This section describes the focus of current and long term research on the expressiveness and performance implications of richer interface descriptions.

The definition of the semantics and behavior of components can currently be expressed only through comments. However, we anticipate the release in mid-2005 of an alpha version of Babel (Dahlgren et al. 2004) that supports the specification of simple *Design by Contract* (Meyer 1997)

assertions suitable for interfaces; namely, preconditions, postconditions, and class invariants. A *precondition* is a condition, or property, that must be true prior to method execution. *Postconditions* are properties that must be true after execution. Finally, *class invariants* are properties that must be true before and after each method invocation once a class has been instantiated. The immediate benefit for component and application developers is enhanced debugging and testing through automated assertion enforcement (Dahlgren and Devanbu 2005).

Furthermore, in order to increase application developer confidence that plug-and-play scientific components obtained from a repository are implemented and being used correctly it is also necessary to retain a certain degree of assertion checking during deployment. However, the performance overhead under these circumstances can become a significant issue (Dahlgren and Devanbu 2004b). Consequently, preliminary work addressing this concern using basic sampling techniques has already been performed (Dahlgren and Devanbu 2004a,b). More sophisticated enforcement mechanisms are the focus of current research.

Additional challenges associated with richer descriptions include expressiveness and extension. For example, a common issue for any complex process or computation is determining whether methods are being invoked in the correct order (Bronsard et al. 1997; Hamie 1999). For instance, a matrix must be initialized and assembled before it can be used in computations. Unfortunately, it is easy for developers to forget or misplace a step, making debugging large or complex applications more difficult. In addition, an extending class may need to expand sequencing constraints of its ancestors through the insertion of additional steps in the middle of an inherited step. Suitable specification, efficient enforcement, and extension of such constraints will be a focus area of long term research on richer interface descriptions.

15.6 Dynamic Code Insertion and Remote Method Invocation in Babel

The fact that Babel sits between the user and provider of a port puts it in a very useful position to allow it to perform a wide range of operations in conjunction with method calls. The verification of assertions described above is one example. But one can use Babel's unique position to dynamically interpose code between caller and callee in a more general fashion. Hooks are being added to Babel to allow this to be done under user control. A prototype version of this capability has been used to support integration of the Tuning and Analysis Utilities (TAU) (Malony and Shende 2000; University of Oregon 2003) into the CCA environment, thereby "automatically" instrumenting all inter-port method calls with calls to the TAU routines to start and stop gathering of performance information. Other applications of this capability are under investigation, including limited forms of aspect-oriented programming (Aspect-Oriented Software Association 2004).

Similarly, the boundary between languages, and the boundary between components, is a logical place to implement remote method invocation (RMI) to provide more direct support for distributed computing. From Babel's perspective, RMI is just like any other language backend. Because Babel is between caller and callee, it can serialize the method invocation over a wire protocol and invoke the method in another process or on another machine.

SIDL already supports method arguments as `in`, `out` or `inout`, and these designations allow the in-process and RMI layer to optimize the method invocation. Arguments can also be designated as `copy` to indicate that Babel should serialize the object for a distributed call rather than only passing a remote reference to the object. Finally, methods can be designated as `oneway` so the caller can dispatch the call over the wire without waiting for a return code indicating completion of the method.

We are currently integrating an RMI backend using the Proteus multiprotocol message library (Chiu, Govindaraju, and Gannon 2002), which is implemented in C++. Once Proteus has been integrated into Babel, it will enable interoperability between the CCaffeine and XCAT frameworks.

15.7 Computational Quality of Service

The relative maturity of component-based software infrastructures encourages users to look beyond syntactically connecting components to using higher-level information about component properties to compose applications. Some of these properties pertain to the *quality* of the services used and provided by components. As more scientific components are developed, many applications are presented with a pool of algorithms with similar functionality but different resource requirements, performance, accuracy, and stability. For example, at the heart of many PDE-based simulations, a set of nonlinear equations must be solved, usually requiring the solution of many linear systems of equations, whose characteristics may vary throughout the simulation. By formally specifying the *quality* requirements of the nonlinear solution method, as well as the capabilities of different linear solution methods, one can at least partially automate the complex task of assembling components into applications and adaptively switching them at runtime.

Some nascent efforts on defining a model for the QoS-based composition of high-performance numerical components address various aspects of a possible architecture design that enables automated application composition and run-time application adaptivity (Hovland et al.

2003; Norris et al. 2004). The draft QoS architecture defines specifications and services for component characterization, component proxy services, component replacement, decision making functionality, and archival and retrieval of execution information. The quality metrics in this architecture include traditional ones, such as reliability and computational cost, as well as metrics that are uniquely important to scientific computations, such as accuracy and rates of convergence. The long term goal of this research is to guide the design of a methodical and automated approach to component application composition and runtime adaptation.

16 Conclusions

Component-based software engineering is a natural approach to facilitate the development of complex, large-scale, high-performance scientific simulations. Its adoption in this area has been limited by the lack of a component model that meets the special needs of the HPC community; the Common Component Architecture is being developed specifically to address this issue. The CCA supports sequential, parallel, and distributed computing paradigms with minimal performance overhead and with a special emphasis on facilitating the incorporation of existing code.

Although the project is far from complete, the CCA specification and associated tools have reached a level of maturity required for adoption by a number of projects across a range of scientific disciplines. We have provided a brief overview of several such efforts in combustion, climate modeling, and quantum chemistry, as well as offered a glimpse of the wide range of components currently available or under development.

Our understanding of how best to utilize CBSE in scientific computing advances continuously through our interactions with the various projects utilizing the CCA. However, the field is broad and highly varied, and it will take some time to gain the necessary breadth and depth of experience. There are also many opportunities on the research front of CBSE itself, and we outlined a number of areas that CCA researchers are exploring. Nevertheless, clearly much research and development remain to be done to bring the full benefits of CBSE to scientific computing and ultimately to make it part of the mainstream for computational science.

Acknowledgments

The CCA has been under development since 1998 by the CCA Forum and represents the contributions of many people, all of whom we gratefully acknowledge. We further acknowledge our collaborators outside the CCA Forum and the early adopters of the CCA for the important contributions they have made both to our understanding of CBSE in the high-performance scientific computing context and to making the CCA a practical and usable environment.

This work has been supported in part by the U. S. Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) initiative, through the Center for Component Technology for Terascale Simulation Software, of which Argonne, Lawrence Livermore, Los Alamos, Oak Ridge, Pacific Northwest, and Sandia National Laboratories, Indiana University, and the University of Utah are members. Members of the SciDAC Center Computational Facility for Reacting Flow Research have also contributed to this paper.

Research at Argonne National Laboratory was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W–11-109-ENG-38.

Some of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Oak Ridge National Laboratory is managed by UT-Battelle, LLC for the US Dept. of Energy under contract DE-AC–05-00OR22725.

This research was performed in part using the Molecular Science Computing Facility (MSCF) in the William R. Wiley Environmental Laboratory at the Pacific Northwest National Laboratory (PNNL). The MSCF is funded by the Office of Biological and Environmental Research in the U.S. Department of Energy. PNNL is operated by Battelle for the U.S. Department of Energy under contract DE-AC06–76RLO 1830.

Research at Binghamton University is sponsored by grant DE-FG02–02ER25526 from the MICS program of the U.S. Dept. of Energy, Office of Science.

Some of research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology and by Northrop Grumman with funding provided by NASA's Computation Technologies (CT) Project, part of the Earth Science Technology Office (ESTO), under a contract with the National Aeronautics and Space Administration.

Research at the University of Oregon is sponsored by contracts (DE-FG03–01ER25501 and DE-FG02–03ER25561) from the MICS program of the U.S. Dept. of Energy, Office of Science.

Research at the University of Utah is also sponsored by the National Science Foundation under contract ACI0113829.

Finally, we wish to thank Thomas O'Brien of the National Energy Technology Laboratory and the anony-

mous reviewers for their critical reading of the manuscript. Their suggestions have significantly improved the paper.

References

Allan, B., Armstrong, R., Lefantzi, S., Ray, J., Walsh, E., and Wolfe, P. 2003. Ccaffeine – a CCA component framework for parallel computing. http://www.cca-forum.org/ccafe/.

Allan, B. A., Armstrong, R. C., Wolfe, A. P., Ray, J., Bernholdt, D. E., and Kohl, J. A. 2002. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience* 14(5):1–23.

Allen, G., Benger, W., Goodale, T., Hege, H., Lanfermann, G., Merzky, A., Radke, T., Seidel, E., and Shalf, J. 2000. The Cactus code: A problem solving environment for the Grid. In: *High Performance Distributed Computing (HPDC)*, pp. 253–260. IEEE Computer Society.

Anonymous. 2003. Jython. http://www.jython.org/.

Anonymous. 2004. AVS kernel overview. http://www.agocg.ac.uk/reports/visual/vissyst/dogbo8.htm.

Argonne National Laboratory. 2003. MPICH homepage. http://www.mcs.anl.gov/mpi/mpich/.

Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., and Smolinski, B. 1999. Toward a Common Component Architecture for high-performance scientific computing. *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*.

Aspect-Oriented Software Association. 2004. Aspect-oriented software development. http://www.aosd.net/.

Balay, S., Buschelman, K., Gropp, W., Kaushik, D., Knepley, M., McInnes, L., Smith, B. F., and Zhang, H. 2003. PETSc users manual. Technical Report ANL-95/11 – Revision 2.1.5, Argonne National Laboratory. http://www.mcs.anl.gov/petsc.

Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. 1997. Efficient management of parallelism in object oriented numerical software libraries. In: *Modern Software Tools in Scientific Computing*, (eds E. Arge, A. M. Bruaset, and H. P. Langtangen), pp. 163–202. Birkhauser Press. URL ftp://info.mcs.anl.gov/pub/techeports/reports/P634.ps.Z, also available as Argonne preprint ANL/MCS-P634–0197.

Baudry, B., Traon, Y. L., and Jezequel, J. 2001. Robustness and diagnosability of OO systems designed by contracts. *Proceedings of the 7th International Software Metrics Symposium*, pp. 272–284.

Beazley, D. 2003. SWIG homepage. http://www.swig.org/.

Beckman, P., Fasel, P., Humphrey, W., and Mniszewski, S. 1998. Efficient coupling of parallel applications using PAWS. *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computation*.

Benson, S., Krishnan, M., McInnes, L., Nieplocha, J., and Sarich, J. 2003. Using the GA and TAO toolkits for solving large-scale optimization problems on parallel computers. Technical Report ANL/MCS-P1084–0903, Argonne National Laboratory. Submitted to ACM-TOMS, ftp://info.mcs.anl.gov/pub/techeports/reports/P1084.pdf.

Benson, S., McInnes, L. C., and Moré, J. 2001. A case study in the performance and scalability of optimization algorithms. *ACM Transactions on Mathematical Software* 27:361–376.

Benson, S., McInnes, L. C., Moré, J., and Sarich, J. 2003. TAO users manual. Technical Report ANL/MCS-TM-242 – Revision 1.5, Argonne National Laboratory. http://www.mcs.anl.gov/tao/.

Bernholdt, D. E., Armstrong, R. C., and Allan, B. A. 2004. Managing complexity in modern high end scientific computing through component-based software engineering. *Proceedings of HPCA Workshop on Productivity and Performance in High-End Computing (P-PHEC 2004)*, Madrid, Spain.

Bernholdt, D. E., Elwasif, W. R., and Kohl, J. A. 2002. Communication infrastructure in high-performance component-based scientific computing. *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 9th European PVM/MPI User's Group Meeting*, Linz, Austria, September/October 2002, (eds D. Kranzlmüller, P. Kacsuk, J. Dongarra, and J. Volkert), volume 2474 of Lecture Notes in Computer Science, pp. 260–270. Springer.

Bernholdt, D. E., Elwasif, W. R., Kohl, J. A., and Epperly, T. G. W. 2002. A component architecture for high-performance computing. *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02)*.

Bertrand, F. and Bramley, R. 2004. DCA: A distributed CCA framework based on MPI. *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pp. 80–89. IEEE Computer Society. URL http://csdl.computer.org/comp/proceedings/hips/2004/2151/00/2151toc.htm.

Beugnard, A., Jezequel, J., Plouzeau, N., and Watkins, D. 1999. Making components contract aware. *IEEE Computer* 32(7):38–45.

Birrell, A. D. and Nelson, B. J. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Science* 2(1):39–59.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. 1997. *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics.

Box, D. 1997. *Essential COM*. Addison-Wesley Pub Co.

Bronsard, F., Bryan, D., Kozaczynski, W. V., Liongosari, E., Ning, J. Q., Olafsson, A., and Wetterstrand, J. W. 1997. Toward software plug-and-play. *Proceedings of the 1997 Symposium on Software Reusability*, pp. 19–29.

Brown, D. L., Henshaw, W. D., and Quinlan, D. J. 1999. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. *Proceedings of the SIAM Workshop on Object Oriented Methods for*

Inter-operable Scientific and Engineering Computing, (eds M. E. Henderson, C. R. Anderson, and S. L. Lyons), pp. 58–67. SIAM.

CCA Forum. 2003. CCA specification. http://cca-forum.org/specification/.

CCA Forum. 2004. *cca-forum@cca-forum.org* mailing list. http://www.cca-forum.org/mailman/list-info/cca-forum/.

Chiu, K., Govindaraju, M., and Gannon, D. 2002. The Proteus multiprotocol message library. *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp. 1–9. IEEE Computer Society Press.

Chow, E., Cleary, A., and Falgout, R. 1999. Design of the hypre preconditioner library. *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, (eds M. E. Henderson, C. R. Anderson, and S. L. Lyons), pp. 106–116. SIAM.

Christensen, E. et al. 2001. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl/.

Colella, P. 2005. An Algorithmic and Software Framework for Applied Partial Differential Equations Center (APDEC). http://davis.lbl.gov/APDEC/.

Dahlgren, T., Epperly, T., Kumfert, G., and Leek, J. 2004. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, Livermore, CA, babel-0.9.4 edition. URL http://www.llnl.gov/CASC/components/docs/usersuide.pdf.

Dahlgren, T. L. and Devanbu, P. T. 2004a. Adaptable assertion checking for scientific software components. *Proceedings of the Workshop on Software Engineering for High Performance Computing System Applications*, pp. 64–69. Edinburgh, Scotland. Also available as Lawrence Livermore National Laboratory Technical Report UCRL-CONF-202898, Livermore, CA, 2004.

Dahlgren, T. L. and Devanbu, P. T. 2004b. An empirical comparison of adaptive assertion enforcement perform ance. Technical Report UCRL-CONF-206305, Lawrence Livermore National Laboratory, Livermore, California.

Dahlgren, T. L. and Devanbu, P. T. 2005. Improving scientific software component quality through assertions. *Second International Workshop on Software Engineering for High Performance Computing System Applications*. St. Louis, Missouri. Also available as Lawrence Livermore National Laboratory Technical Report UCRL-ABS-209858, Livermore, CA, 2005.

Damevski, K. 2003. *Parallel RMI and M-by-N Data Redistribution using an IDL Compiler*. Master's thesis, The University of Utah.

Denis, A., Pérez, C., and Priol, T. 2001. Towards high performance CORBA and MPI middlewares for Grid computing. *Proceedings of the 2nd International Workshop on Grid Computing*, (ed. C. A. Lee), volume 2242 of Lecture Notes in Computer Science, pp. 14–25. Berlin: Springer-Verlag.

Denis, A., Pérez, C., Priol, T., and Ribes, A. 2003. Parallel CORBA objects for programming computational grids. *Distributed Systems Online* 4(2).

Drummond, L. A., Demmel, J., Mechoso, C. R., Robinson, H., Sklower, K., and Spahr, J. A. 2001. A data broker for distributed computing environments. *Proceedings of the International Conference on Computational Science*, pp. 31–40.

Edjlali, G., Sussman, A., and Saltz, J. 1997. Interoperability of data-parallel runtime libraries. *International Parallel Processing Symposium. Geneva, Switzerland*: IEEE Computer Society Press.

Englander, R. 1997. *Developing Java Beans*. O'Reilly and Associates.

Falgout, R. et al. 2003. hypre. http://www.llnl.gov/CASC/hypre/.

Foster, I. and Kesselman, C. 1998. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann.

Foster, I., Kesselman, C., Nick, J., and Tuecke, S. 2002. Grid services for distributed system integration. *Computer* 35(6):37–46.

Geist, G. A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. 1994. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: MIT Press.

Geist, G. A., Kohl, J. A., and Papadopoulos, P. M. 1997. CUMULVS: Providing fault tolerance, visualization and steering of parallel applications. *The International Journal of High Performance Computing Applications* 11(3):224–236.

Glimm, J., Brown, D., and Freitag, L. 2005. Terascale Simulation Tools and Technologies (TSTT) Center. http://www.tstt-scidac.org/.

Gosling, J., Joy, B., and Steele, G. 1996. The Java Language Specification. Available at http://java.sun.com/.

Govindaraju, M., Krishnan, S., Chiu, K., Slominski, A., Gannon, D., and Bramley, R. 2003. Merging the CCA component model with the OGSI framework. *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*.

Grimshaw, A., Ferrari, A., Knabe, F., and Humphrey, M. 1999. Legion: An operating system for wide-area computing. *IEEE Computer* 32(5).

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., and Nielsen, H. F. 2003. SOAP version 1.2. http://www.w3.org/TR/soap12-part1/.

Guilyardi, E., Budich, R. G., and Valcke, S. 2002. PRISM and ENES: European approaches to Earth System Modelling. *Proceedings of Realizing TeraComputing – Tenth Workshop on the Use of High Performance Computing in Meteorology*.

Hamie, A. 1999. Enhancing the object constraint language for more expressive specifications. *Proceedings of 6th Asia-Pacific Software Engineering Conference (APSEC '99)*, pp. 376–383.

Harper, L. and Kauffman, B. 2004. Community Climate System Model. http://www.ccsm.ucar.edu/.

Henshaw, W. et al. 2002. Overture. http://www.llnl.gov/CASC/Overture/.

Heroux, M., Bartlett, R., Hoekstra, V. H. R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., and

Williams, A. 2003. An overview of Trilinos. Technical Report SAND2003–2927, Sandia National Laboratories.

Heroux, M. et al. 2004. Trilinos. http://software.sandia.gov/Trilinos.

Hindmarsh, A. and Serban, R. 2002. User documentation for CVODES, an ODE solver with sensitivity analysis capabilities. Technical Report UCRL-MA-148813, Lawrence Livermore National Laboratory. http://www.llnl.gov/CASC/sundials/.

Hoare, M. R. 1979. Structure and dynamics of simple microclusters. *Advances in Chemical Physics* 40:49–135.

Hovland, P., Keahey, K., McInnes, L. C., Norris, B., Diachin, L. F., and Raghavan, P. 2003. A quality of service approach for high-performance numerical components. *Proceedings of Workshop on QoS in Component-Based Software Engineering, Software Technologies Conference*. Toulouse, France.

Janssen, C. L., Nielsen, I. M. B., and Colvin, M. E. 1998. Parallel Processing for Ab Initio Quantum Mechanical Methods In: *Encyclopedia of Computational Chemistry*. John Wiley & Sons, Chichester, UK.

Johnson, C. and Parker, S. 1995. Applications in computational medicine using SCIRun: A computational steering programming environment. In: *Supercomputer `95*, (ed H. Meuer), pp. 2–19. Springer-Verlag.

Katz, D. S., Tisdale, E. R., and Norton, C. D. 2002. The Common Component Architecture (CCA) applied to sequential and parallel computational electromagnetic applications. *Recent Advances in Computational Science & Engineering: Proceedings of the International Conference on Scientific & Engineering Computation (IC-SEC)* 2002, pp. 353–356. Imperial College Press.

Keahey, K. 1996. *Architecture for Application-Level Parallel Distributed Computing*. PhD thesis, Indiana University – Bloomington.

Keahey, K., Fasel, P., and Mniszewski, S. 2001. PAWS: Collective interactions and data transfers. *Proceedings of the High Performance Distributed Computing Conference*. San Francisco, CA.

Keahey, K. and Gannon, D. 1997. PARDIS: A parallel approach to CORBA. *Proceedings of the High Performance Distributed Computing Conference*, pp. 31–39.

Kendall, R. A., Apra, E., Bernholdt, D. E., Bylaska, E. J., Dupuis, M., Fann, G. I., Harrison, R. J., Ju, J. L., Nichols, J. A., Nieplocha, J., Straatsma, T. P., Windus, T. L., and Wong, A. T. 2000. High performance computational chemistry: An overview of NWChem, a distributed parallel application. *Computational Physics Communication* 128:260–270.

Kenny, J. P., Benson, S. J., Alexeev, Y., Sarich, J., Janssen, C. L., McInnes, L. C., Krishnan, M., Nieplocha, J., Jurrus, E., Fahlstrom, C., and Windus, T. L. 2004. Component-based integration of chemistry and optimization software. *J. of Computational Chemistry* 25(14):1717–1725.

Keyes, D. 2005. Terascale Optimal PDE Simulations (TOPS) Center. http://tops-scidac.org/.

Killeen, T., Marshall, J., and da Silva, A. 2003. Earth System Modeling Framework homepage. http://www.esmf.ucar.edu/.

Kohl, J. A. 1997. High performance computers: Innovative assistants to science. *ORNL Review, Special Issue on Advanced Computing* 30(3/4):224–236.

Kohl, J. A. and Geist, G. A. 1999. Monitoring and steering of large-scale distributed simulations. *IASTED International Conference on Applied Modeling and Simulation*. Cairns, Queensland, Australia.

Kohl, J. A. and Papadopoulos, P. M. 1995. A library for visualization and steering of distributed simulations using PVM and AVS. *High Performance Computing Symposium*. Montreal, CA.

Krishnan, S. and Gannon, D. 2004. XCAT3: A framework for CCA components as OGSA services. *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pp. 90–97. IEEE Computer Society. URL http://csdl.computer.org/comp/proceedings/hips/2004/2151/00/2151toc.htm.

Kumfert, G. 2003. Understanding the CCA specification using Decaf. Technical Report UCRL-MA-145991, Lawrence Livermore National Laboratory. http://www.llnl.gov/CASC/components/docs.html.

Kumfert, G. and Epperly, T. 2002. Software in the DOE: The hidden overhead of "the build". Technical Report UCRL-ID-147343, Lawrence Livermore National Laboratory.

Larson, J., Jacob, R., and Ong, E. 2004. Model Coupling Toolkit. http://www.mcs.anl.gov/mct/.

Larson, J. W., Jacob, R. L., Foster, I. T., and Guo, J. 2001. The Model Coupling Toolkit. *Proceedings of the International Conference on Computational Science (ICCS) 2001*, (eds V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. J. K. Tan), volume 2073 of Lecture Notes in Computer Science, pp. 185–194. Berlin: Springer-Verlag.

Larson, J. W., Norris, B., Ong, E. T., Bernholdt, D. E., Drake, J. B., Elwasif, W. R., Ham, M. W., Rasmussen, C. E., Kumfert, G., Katz, D. S., Zhou, S., DeLuca, C., and Collins, N. S. 2004. Components, the Common Component Architecture, and the climate/weather/ocean community. *84th American Meteorological Society Annual Meeting*. Seattle, Washington: American Meteorological Society.

von Laszewski, G., Foster, I., Gawor, J., and Lane, P. 2001. A Java commodity Grid kit. *Concurrency and Computation: Practice and Experience* 13(8–9):643–662.

Lawrence Livermore National Laboratory. 2004. Babel homepage. http://www.llnl.gov/CASC/components/babel.html.

Lee, J. and Sussman, A. 2004. Efficient communication between parallel programs with InterComm. Technical Report CS-TR-4557 and UMIACS-TR-2004–04, University of Maryland, Department of Computer Science and UMIACS.

Lefantzi, S., Kennedy, C., Ray, J., and Najm, H. 2003. A study of the effect of higher order spatial discretizations in SAMR (Structured Adaptive Mesh Refinement) simulations. *Proceedings of the Fall Meeting of the Western States Section of the The Combustion Institute*. Los Angeles, California. Distributed via CD-ROM.

Lefantzi, S. and Ray, J. 2003. A component-based scientific toolkit for reacting flows. *Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics*,

June 17–20, 2003, Cambridge, MA, volume 2, pp. 1401–1405. Elsevier.

Lefantzi, S., Ray, J., and Najm, H. N. 2003. Using the Common Component Architecture to design high performance scientific simulation codes. *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, 22–26 April 2003, Nice, France. IEEE Computer Society. Distributed via CD-ROM.

Lewis, M., Ferrari, A., Humphrey, M., Karpovich, J., Morgan, M., Natrajan, A., Nguyen-Tuong, A., Wasson, G., and Grimshaw, A. 2003. Support for extensibility and site autonomy in the Legion grid system object model. *Journal of Parallel and Distributed Computing* 63:525–538.

Liberty, J. 2003. *Learning Visual Basic .NET*. O'Reilly and Associates.

Malony, A. D. and Shende, S. 2000. Performance technology for complex parallel and distributed sys tems. In: *Distributed and Parallel Systems: From Concepts to Applications*, pp. 37–46. Norwell, MA: Kluwer.

McInnes, L. C., Allan, B. A., Armstrong, R., Benson, S. J., Bernholdt, D. E., Dahlgren, T. L., Diachin, L. F., Krishnan, M., Kohl, J. A., Larson, J. W., Lefantzi, S., Nieplocha, J., Norris, B., Parker, S. G., Ray, J., and Zhou, S. 2006. Parallel PDE-based simulations using the Common Component Architecture. In: *Numerical Solution of PDEs on Parallel Computers* (eds A. Magnus Bruaset, A. Tveito), volume 51 of *Lecture Notes in Computational Science and Engineering (LNCSE)*, pages 327–384, Berlin: Springer-Verlag, 2006, invited chapter, also Argonne National Laboratory technical report ANL/MCS-P1179-0704.

Meyer, B. 1997. *Object-Oriented Software Construction*. Upper Saddle River, New Jersey 07458: Prentice Hall.

Microsoft Corporation 1999. Component Object Model specification. `http://www.microsoft.com/com/resources/comdocs.asp`.

MPI Forum 1994. MPI: a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing* 8(3/4):159–416.

Najm, H. N. et al. 2003. CFRFS homepage. `http://cfrfs.ca.sandia.gov/`.

Nieplocha, J., Harrison, R. J., and Littlefield, R. J. 1996. Global arrays: A non-uniform-memory-access programming model for high-performance computers. *J. Supercomputing* 10(2): 169.

Norris, B., Balay, S., Benson, S., Freitag, L., Hovland, P., McInnes, L., and Smith, B. 2002. Parallel components for PDEs and optimization: Some issues and experiences. *Parallel Computing* 28(12):1811–1831.

Norris, B., Ray, J., Armstrong, R., McInnes, L. C., Bernholdt, D. E., Elwasif, W. R., Malony, A. D., and Shende, S. 2004. Computational quality of service for scientific components. *Proceedings of the International Symposium on Component-Based Software Engineering (CBSE7)*, (eds I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. Wallnau), volume 3054 of Lecture Notes in Computer Science, pp. 264–271. Edinburgh, Scotland: Springer. URL `http://springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3054&spage=264`, (also available as Argonne preprint ANL/MCS-P1131–0304).

Object Management Group. 2002. CORBA component model. `http://www.omg.org/technology/documents/formal/components.htm`.

Ong, E., Larson, J., and Jacob, R. 2002. A real application of the Model Coupling Toolkit. *Proceedings of the 2002 International Conference on Computational Science*, (eds C. J. K. Tan, J. J. Dongarra, A. G. Hoekstra, and P. M. A. Sloot), volume 2330 of Lecture Notes in Computer Science, pp. 748–757. Berlin: Springer-Verlag.

OpenLDAP Foundation. 2003. Lightweight directory access protocol. `http://www.openldap.org/`.

Pacific Northwest National Laboratory. 2004a. Global Array Toolkit homepage. `http://www.emsl.pnl.gov:2080/docs/global/`.

Pacific Northwest National Laboratory. 2004b. NWChem homepage. `http://www.emsl.pnl.gov/docs/nwchem/`.

Parashar, M. et al. 2004. GrACE homepage. `http://www.caip.rutgers.edu/~parashar/TASSL/Projects/GrACE/`.

Parker, S. G. and Johnson, C. R. 1995. SCIRun: A scientific programming environment for computational steering. *Proceedings of the IEEE/ACM SC95 Conference*. IEEE Computer Society.

Parker, S. G., Weinstein, D. M., and Johnson, C. R. 1997. The SCIRun computational steering software system. In: *Modern Software Tools in Scientific Computing*, (eds E. Arge, A. Bruaset, and H. Langtangen), pp. 1–44. Birkhauser Press.

Ranganathan, M., Acharya, A., Edjlali, G., Sussman, A., and Saltz, J. 1996. A runtime coupling of data-parallel programs. *Proceedings of the 1996 International Conference on Supercomputing*. Philadelphia, PA.

Rasmussen, C. E., Lindlan, K. A., Mohr, B., and Striegnitz, J. 2001. CHASM: Static analysis and automatic code generation for improved Fortran 90 and C++ interoperability. *2001 LACSI Symposium*.

Rasmussen, C. E., Sottile, M. J., Shende, S. S., and Malony, A. D. 2003. Bridging the language gap in scientific computing: The CHASM approach. Technical Report LA-UR–33-3057, Advanced Computing Laboratory, Los Alamos National Laboratory.

Ray, J., Kennedy, C., Lefantzi, S., and Najm, H. 2003. High-order spatial discretizations and extended stability methods for reacting flows on structured adaptively refined meshes. *Proceedings of the Third Joint Meeting of the U.S. Sections of The Combustion Institute*, March 16–19, 2003, Chicago, Illinois. Distributed via CD-ROM.

Ray, J., Trebon, N., Shende, S., Armstrong, R. C., and Malony, A. 2004. Performance measurement and modeling of component applications in a high performance computing environment: A case study. *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. Los Alamitos, California, USA: IEEE Computer Society. Also Sandia National Laboratories Technical Report SAND2003–8631, November 2003.

Reynders, J. et al. 2001. POOMA. `http://www.codesourcery.com/pooma/pooma`.

Reynders, J. V. W., Cummings, J. C., Hinker, P. J., Tholburn, M., S. Banerjee, M. S., Karmesin, S., Atlas, S., Keahey, K., and Humphrey, W. F. 1996. In: *POOMA: A Frame-*

Work for Scientific Computing Applications on Parallel Architectures, chapter 14. MIT Press.

Roman, E. 1997. *Mastering Enterprise JavaBeans*. O'Reilly and Associates.

Sandia National Laboratories. 2004. MPQC homepage. http://aros.ca.sandia.gov/~cljanss/mpqc/.

Schmidt, D. C., Pyarali, I., and Harrison, T. 1996. Design and performance of an object-oriented framework for high-speed electronic medical imaging. *USENIX Computing Systems* 9(4).

Schmidt, D. C., Vinoski, S., and Wang, N. 1999. Collocation optimizations for CORBA. C++ Report 11(9).

Shende, S., Malony, A. D., Rasmussen, C., and Sottile, M. 2003. A performance interface for component-based applications. *Proceedings of International Workshop on Performance Modeling, Evaluation and Optimization, International Parallel and Distributed Processing Symposium*. URL http://www.cs.uoregon.edu/research/paracomp/publ/htbin/bibify.cgi?cmd=show&coll=CONF&id=ipdps03&dataresent=no.

Slominski, A., Govindaraju, M., Gannon, D., and Bramley, R. 2001. Design of an XML based interoperable RMI system: SoapRMI C++/Java 1.1. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 1661–1667.

Smith, R. D., Dukowicz, J. K., and Malone, R. C. 1992. Parallel ocean general circulation modeling. *Physica D* 60(38):38–61.

Sportisse, B. 2000. An analysis of operator splitting techniques in the stiff case. *J. Comp. Phys.* 161:140–168.

Strang, G. 1968. On the construction and comparison of difference schemes. *SIAM J. Numer. Anal.* 5(3):506–517.

Sun Microsystems. 2004a. Enterprise JavaBeans downloads and specifications. http://java.sun.com/products/ejb/docs.html.

Sun Microsystems. 2004b. Java Remote Method Invocation (Java RMI). http://java.sun.com/products/jdk/rmi/.

Szyperski, C. 1999. *Component Software: Beyond Object-Oriented Programming*. New York: ACM Press.

UDDI.org. 2003. Universal Description Discovery and Integration of business for the Web (UDDI). http://www.uddi.org/specification.html.

University Corporation for Atmospheric Research. 2004. The Community Atmosphere Model (CAM) homepage. http://www.ccsm.ucar.edu/models/atm-cam/.

University of Oregon. 2003. TAU: Tuning and analysis utilities. http://www.cs.uoregon.edu/research/paracomp/tau/.

US Dept. of Energy. 2003. SciDAC Initiative homepage. http://www.osti.gov/scidac/.

Wehner, M. F., Mirin, A. A., Eltgroth, P. G., Dannevik, W. P., Mechoso, C. R., Farrara, J., and Spahr, J. A. 1995. Performance of a distributed memory finite-difference atmospheric general circulation model. *J. Parallel Comp.* 21: 1655–1675.

Wilde, T., Kohl, J. A., and Flanery, Jr., R. E. 2002. Integrating CUMULVS into AVS/Express. *International Conference on Computational Science (2)*, pp. 864–873. Amsterdam, The Netherlands.

Wilde, T., Kohl, J. A., and Flanery, Jr., R. E. 2003. Immersive and 3D viewers for CUMULVS: VTK/CAVETM and AVS/Express. *Future Gener. Comput. Syst.* 19(5):701–719.

Zhang, K., Damevski, K., Venkatachalapathy, V., and Parker, S. 2004. Scirun2: A cca framework for high performance computing. *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pp. 72–79. IEEE Computer Society. URL http://csdl.computer.org/comp/proceedings/hips/2004/2151/00/2151toc.htm.

Zhou, S. 2003. Coupling earth system models: An ESMF-CCA prototype. http://webserv.gsfc.nasa.gov/ESS/esmfasc/.

Zhou, S., da Silva, A., Womack, B., and Higgins, G. 2003. Prototyping the ESMF using DOE's CCA. *NASA Earth Science Technology Conference 2003*. College Park, MD. http://esto.nasa.gov/conferences/estc2003/papers/A4P3(Zhou).pdf.