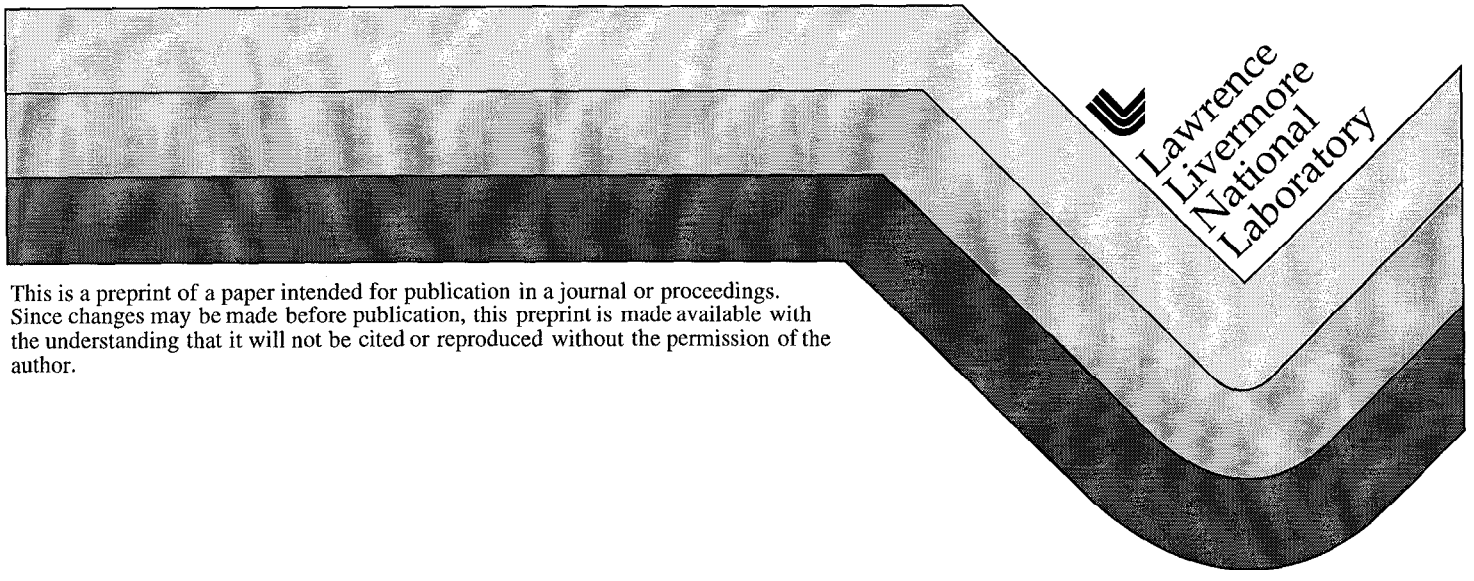
UCRL-JC-134475
PREPRINT

Toward a Common Component Architecture for High-Performance Scientific Computing

R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn,
L. McInnes, S. Parker and B. Smolinski

This paper was prepared for submittal to the
8th Institute for Electrical and Electronis Engineers
International Symposium on High Performance Distributed Computing
Redondo Beach, CA
August 3-6, 1999

June 9, 1999



Toward a Common Component Architecture for High-Performance Scientific Computing *

Rob Armstrong† Dennis Gannon‡ Al Geist§ Katarzyna Keahey¶ Scott Kohn‖
Lois McInnes** Steve Parker†† Brent Smolinski‡‡

Abstract

This paper describes work in progress to develop a standard for interoperability among high-performance scientific components. This research stems from growing recognition that the scientific community must better manage the complexity of multidisciplinary simulations and better address scalable performance issues on parallel and distributed architectures. Driving forces are the need for fast connections among components that perform numerically intensive work and parallel collective interactions among components that use multiple processes or threads. This paper focuses on the areas we believe are most crucial for such interactions, namely an interface definition language that supports scientific abstractions for specifying component interfaces and a ports connection model for specifying component interactions.

1 Introduction

The complexity and resource demands of present day software systems create the need for more flexible solutions than those offered by conventional programming styles based on a succession of subroutine calls. Component programming, based on encapsulating functionality units and providing a meta-language specification of their interfaces, enables us to address these issues. A clear specification of component outputs as well as inputs enables programmers to integrate multiple software libraries, which may be developed by various groups with differing expertise. These specifications enhance software reusability, since components can serve as interchangeable pieces that can be dynamically added, removed, or replaced in an ongoing simulation. Furthermore, meta-language specifications allow programmers to integrate components developed using different languages or libraries.

These advantages are especially appealing in high performance scientific computing, where high-fidelity, multiphysics simulations are increasingly complex and often require the combined expertise of multidisciplinary research teams working in areas such as mathematical modeling, adaptive mesh manipulations, numerical linear and nonlinear algebra, optimization, load balancing, computational steering, parallel I/O, sensitivity analysis, visualization, and data analysis. Consequently, the reusability and rapid application development afforded by component programming are of particular importance.

There are many differing opinions within the software community about component definitions [7, 48]. We present some working definitions as preliminaries for further discussion.

- A *component* is an independent unit of software deployment. It satisfies a set of behavior rules and implements standard component interfaces that allow it to be easily composed with other components. These behavior rules are often specified as designed patterns that must be followed when writing the component.
- A *component framework* defines a set of interfaces and rules of interaction that govern the communication among the components connected to the framework [48]. Examples of component frameworks include JavaBeans [20] and DCOM [46].

*This work has been partially supported by the MICS Division of the U.S. Department of Energy through the DOE2000 Initiative. For further information on the Common Component Architecture Forum, see http://www.acl.lanl.gov/cca-forum or write to cca-forum@z.ca.sandia.gov.

†Sandia National Laboratory, rob@z.ca.sandia.gov.

‡Indiana University, gannon@cs.indiana.edu

§Oak Ridge National Laboratory, geist@msr.epm.ornl.gov.

¶Advanced Computing Laboratory, Los Alamos National Laboratory, kate@lanl.gov.

‖Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551, skohn@llnl.gov.

**Mathematics and Computer Science Division, Argonne National Laboratory, mcinnes@mcs.anl.gov.

††Department of Computer Science, University of Utah, sparker@taz.cs.utah.edu.

‡‡Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551, smolinski@llnl.gov.

- A *component architecture* encompasses the component framework along with other necessary tools, such as component repositories and composition tools.

In this context, component-based software development can be considered a evolutionary step beyond object-oriented design. Object-oriented techniques have been very successful in managing the complexity of modern software, but they have not resulted in significant amounts of cross-project code reuse. Sharing object-oriented code is difficult due to language incompatibilities, the lack of standardization for inter-object communication, and the need for compile-time coupling of interfaces. Component-based software development addresses issues of language independence—seamlessly combining components written in different programming languages—and component frameworks define standards for communication among components. Component-based programming supports incremental shifts in parallel algorithms and programming paradigms that inevitably occur during the lifetimes of scientific application codes.

In addition, since components can be configured to execute in remote locations, component programming can offer high-level abstractions facilitating the use of remote supercomputing resources. Systems such as Legion [30] and Globus [22] have shown that distributed high-performance programming can offer enormous potential as a gateway to the computational grid [23]. Through defining high-level abstractions, component programming brings us closer to reducing the programming overhead required to tap into that potential.

The mainstream computing community has defined component standards such as CORBA [41], COM [46], and Java Beans [20] to address similar complexities within their target applications (see Section 3 for a detailed discussion). Our approach leverages this work where appropriate, but addresses the distinctly different technical challenges of large-scale, high-performance scientific simulations. We have recently established the Common Component Architecture Forum (CCA) [16], a group whose current membership is drawn from various Department of Energy national laboratories and collaborating academic institutions. A number of different research high-performance component architectures have been developed or are currently under development by CCA participants [3, 45, 27, 32, 36, 37], and various research projects are considering related design issues [1, 28, 6]. Based on the lessons learned from these research efforts, we are developing a single high-performance component interface standard that will enable interactions among the scientific components that follow this standard. Additional related work on software design issues for high-performance scientific computing includes [10, 8, 24, 35, 39].

We recognize two levels of interoperability: *component-level* interoperability, for which all the vital functions of any *one* architecture are accessible to any compliant component through a standard interface (e.g., facilities available within a CORBA ORB), and *framework-level* interoperability, for which the frameworks themselves interoperate through a standardized interface (e.g., inter-ORB communication via CORBA IIOP). Providing component-level interoperability requires defining a data and interaction model common for all components and a small set of indispensable high-level framework services. In addition, framework-level interoperability requires the standardization of a number low-level services. Since providing a framework-level interoperability standard requires a superset of features that need to be defined for component-level interoperability, our current focus is on providing the latter and extending it in the future to include framework-level interoperability features. The scope of this paper is limited to component-level interoperability.

The remainder of this paper motivates and explains our approach, beginning in Section 2 with a discussion of some of the challenges in large-scale scientific computing. Section 3 compares our strategy with related work in the mainstream computing community. Section 4 presents a high-level view of the CCA standard and provides a roadmap outlining the relationships among its constituents. The succeeding two sections describe in detail the parts of the CCA standard that are most crucial for defining component interactions in high-performance scientific software, namely a scientific interface definition language and ports with direct-connect and collective capabilities. Finally, Section 7 outlines future directions of work.

2 Motivating Examples

Our work is motivated by collaborations with various computational science research teams, who are investigating areas such as combustion [15], materials science [49], climate, accelerator physics, and fusion [44], among many others. In conjunction with theoretical and experimental research, these simulations are playing increasingly important roles in overall scientific advances, particularly in fields where experimental models are prohibitively costly, time consuming, or in some cases impossible.

While each of these simulations requires different mathematical models, numerical methods, and data analysis techniques, they could all benefit from infrastructure that is more flexible and extensible and therefore better able to manage complexity and change. Designing such tools is difficult given our target architectures, which range from clusters of networked workstations to clusters of symmetric multiprocessors and possibly distributed resources. Additional challenges arise because of the diversity of our target software users and developers – scientific researchers,

who have different programming paradigm preferences and widely varying experience in translating scientific abstractions into large-scale codes.

To enable more concrete discussion of the CCA approach, we briefly review some challenges arising in chemically reacting flow simulations, which are demanding due to the requirements for high resolution and complex physical submodels for turbulence, chemistry, and multiphase flows. Section 2.1 presents current functionality of a particular application, while Section 2.2 describes potential enhancements that component-based technology could help to support in interoperable tools for numerics, data analysis, and domain-specific problem solving infrastructure.

2.1 Computational Hydrodynamics

We consider the CHAD (Computational Hydrodynamics for Advanced Design) application [15, 43], which has been developed for fluids simulations in the automotive industry under the Supercomputing Automotive Applications Partnership CRADA with the United States Council for Automotive Research and five Department of Energy national laboratories (Argonne, Los Alamos, Lawrence Livermore, Oak Ridge, and Sandia). CHAD is the successor of KIVA [2], which has become a standard tool for device-level modeling of internal combustion engines. CHAD is intended for automotive design applications such as combustion, interior airflow (HVAC), under-hood cooling, and exterior flows; the application was designed from its inception as parallel code using FORTRAN90 and encapsulation of nonlocal communication in gather/scatter routines using the Message Passing Interface (MPI) standard [40].

CHAD computes three-dimensional fluid flows with chemical reactions and fuel sprays. The current code solves the single-phase, compressible, Navier-Stokes equations using an arbitrary Lagrangian-Eulerian (ALE) formulation to allow a moving mesh; for turbulent flows a standard $K-\epsilon$ turbulence model is employed. The resulting nonlinear system may be expressed as

$$R_l(\dot{q}_m, q_m, t) = 0,$$

where t indicates time, and q_m represent the independent field variables (pressure, velocity, etc.). The indices l, m range over $1, 2, \ldots N$, where N is the number of independent field variables. Hybrid unstructured meshes are used to construct vertex-centric control volumes for use in an edge-based finite volume discretization scheme.

2.2 Component Challenges and Opportunities

Continuing a trend toward more implicit formulations, CHAD researchers are experimenting with numerical strategies ranging from explicit through semi-implicit and even more fully implicit schemes using Newton-type methods. Increased implicitness helps to overcome stability and accuracy restrictions on computational timesteps, and thereby can often help to reduce overall time to solution.

Figure 1 demonstrates some typical interactions among components for a semi-implicit solution procedure within a simplified PDE-based numerical model. Parallel numerical components that use distributed data structures and require interconnections with low latency and high bandwidth are represented by the overlayed boxes in the right-hand-side of the figure. Components for visualization of field variables, which can often be loosely coupled and differently distributed than the numerical components, are shown in the left-hand-side box. While a single diagram cannot express the richness of interactions within CHAD, nor the range of functionality needed by the various scientific applications that motivate this work, this picture does convey key themes that motivate the CCA approach: (1) *fast interactions* between components that perform numerically intensive work, as shown by the directly connected ports in the right-hand-side boxes (see Section 6.2); (2) *collective interactions* among components that use multiple processes or threads; such interactions are needed for both tightly coupled tasks such as solving algebraic systems and loosely coupled tasks such as visualizing field variables (see Section 6.3); and (3) the use of possibly *distributed resources* for phases like data analysis and visualization.

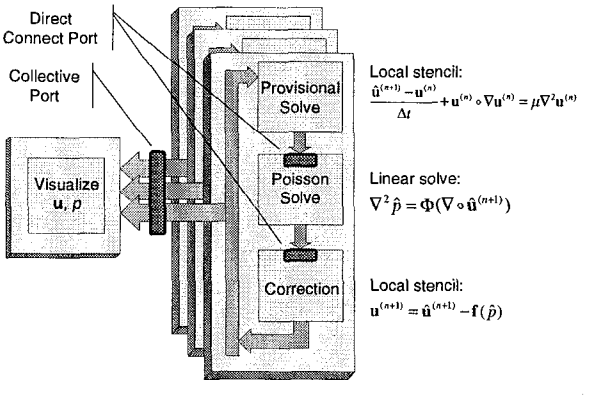


Figure 1. Schematic Diagram of Component Interactions

The goals of the CCA forum are to simplify the infusion of new techniques within the lifetimes of existing applications such as CHAD as well as to facilitate the construction of new models. Interactions among multiple tools that use current-generation infrastructure typically

require labor-intensive translations between interfaces and data structures. We aim to simplify this process and also to enable dynamic interactions, since researchers may wish to introduce new components during the course of ongoing simulations. For example, a researcher may wish to visualize flow fields on a local workstation by dynamically attaching a visualization tool to an ongoing simulation that is running on a remote parallel machine. Upon observing that the flow fields are not converging as expected, he may wish to introduce a new scheme for hierarchical mesh refinement.

The CCA assumes component granularity at the procedural level. This can range from relatively fine granularity when the procedure is a call to a subroutine implementing a subtask, to very coarse granularity when the procedure is associated with an entire parallel application. We note that developers must carefully consider a computation's granularity when determining whether to use abstraction via component-based (or even simply object-oriented) design. The overhead of indirection is usually considered too high for very fine-grained tasks such as a single scalar computation, but can be adequately amortized for aggregate operations on a collection of data.

One of the most computationally intensive phases within the semi-implicit and implicit strategies under consideration within CHAD is the solution of discretized linear systems of the form $Ax = b$, which are very large ($O(1,000,000)$ or orders of magnitude higher, even for relatively simple geometric domains) and have sparse coefficient matrices A. Preconditioned Krylov methods [25], which can be represented as $M_{L^{-1}}Ax = M_{L^{-1}}b$, are among the most effective solution strategies for such systems. The Equation Solver Interface Forum [13], is exploring interoperability issues for algebraic solvers, with a goal of enabling applications such as CHAD to experiment more easily with multiple solution strategies and to upgrade as new algorithms with better latency tolerance or more efficient cache utilization are discovered and encapsulated within toolkits. This area is one of many (e.g., partitioning, mesh management, discretization, optimization, visualization) that could benefit from component-based infrastructure to facilitate the use different tools.

3 Relationship to Existing Standards

Component architecture standards such as CORBA [41], COM [46], and Java Beans [20] have been defined by industrial corporations and consortia and are used by millions of users. Unfortunately, these standards do not address the problems of high-performance scientific computing. None of the industry standards supports efficient parallel communication channels between components. What is needed are abstractions suitable for high-performance computing. The existence of many successful high-performance languages and libraries—such as HPC++ [26], POOMA [4], ISIS++ [12], SAMRAI [31], and PETSc [5]—testifies to the fact that such abstractions enable the user to develop more efficient programs faster. Similarly, we need abstractions capturing high-performance concepts in component architectures. For example, PARDIS [37] and PAWS [6] successfully showed that introducing abstractions for single program multiple data (SPMD) computation leads to enabling more efficient interactions between SPMD programs. In this section, we briefly review these industry standards and evaluate their limitations for high-performance scientific computing.

3.1 Microsoft COM and ActiveX

COM (Component Object Model) is Microsoft's component standard that forms the basis for interoperability among all Window-based applications. ActiveX [11] defines standard COM interfaces for compound documents. Microsoft has developed a distributed version of COM called DCOM (Distributed COM) that targets networked Windows workstations.

COM is targeted towards business objects and does not include abstractions for parallel data layout or basic scientific computing data types, like complex numbers, and Fortran style dynamic multi-dimensional arrays. Also, COM does not easily support implementation inheritance and multiple inheritance (which are implemented through aggregation or containment). Scientific libraries such as the ESI [21] require multiple inheritance and a simple model for polymorphism, which are not provided by COM.

3.2 Sun JavaBeans and Enterprise JavaBeans

JavaBeans and Enterprise JavaBeans (EJB) are component architectures developed by Sun and its partners. They are based on Sun's Java Programming language and are cross-platform competitors to Microsoft's COM.

Neither JavaBeans nor EJB directly address the issue of language interoperability and therefore are inappropriate for the scientific computing environment. Both JavaBeans and EJB assume that all components are written in the Java language. Although the Java Native Interface (JNI) [34] library supports interoperability with C and C++, using the Java virtual machine to mediate communication between components would incur an intolerable performance penalty on every inter-component function call.

3.3 OMG CORBA

CORBA (Common Object Request Broker Architecture) is a distributed object specification supported by the OMG (Object Management Group), a consortium of over eight

hundred partners. CORBA supports the interaction of complex objects written in different languages distributed across a network of computers running different operating systems.

CORBA focuses on distributed computing, not high-performance parallel computing, and therefore existing CORBA implementations incur unacceptable overheads in the packing and unpacking of arguments during a function call. The current CORBA specification does not define a component model, although a CORBA Beans component specification is currently under review by the OMB. Like COM, CORBA does not provide abstractions necessary for high-performance scientific, such as Fortran style dynamic multi-dimensional arrays and complex numbers. CORBA also has a limited object model in that the semantics of multiple implementation inheritance can lead to ambiguities and method overriding is not supported.

4 Overview of the CCA Standard

We define the Common Component Architecture (CCA) as a set of standards and their relationships as depicted in Figure 2. The elements with gray background pertain to specific implementations of a component architecture, while the elements with white background depict parts of the CCA standards necessary for component-level interoperability.

As shown in the picture, components interact with each other and with a specific framework implementation through standard Application Programming Interfaces (APIs). Each component defines its inputs and outputs using a Scientific Interface Definition Language (SIDL); these definitions can be deposited in and retrieved from a repository by using the CCA Repository API. In addition, these definitions can serve as input to a proxy generator that generates component stubs, which form the component-specific part of the CCA Ports. Components can use framework services directly through the CCA Services Interface. The CCA Configuration API ensures that the components can collaborate with different builders associated with different frameworks.

A framework that conforms to these standards—that is, provides the required CCA services, can express component functionality using a SIDL, and implements the required CCA interfaces—is CCA compliant. Different components require different sets of services to interoperate. For example, some may require remote communication while others may communicate only in the same address space. Therefore, the CCA standard will define different flavors of compliance; each component will specify a minimum flavor of compliance required of a framework within which it can interact.

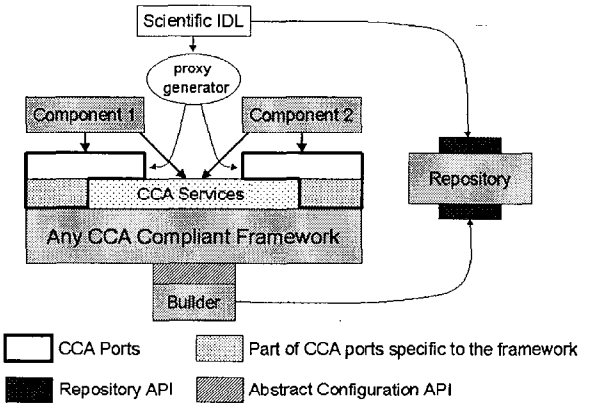


Figure 2. Common Component Architecture

We will now describe the elements of the CCA standard in some detail:

- The *Scientific IDL* (SIDL) is a programming-language-neutral interface definition language used to describe component interfaces. The SIDL provides a method for describing component and framework interfaces that is independent of the underlying implementation programming languages. Component descriptions in SIDL can be deposited into and retrieved from repositories through the Repository Interface. Component descriptions are used by the proxy generator to provide the "component stubs" element of communication Ports.

- CCA *Ports* define the communication model for all component interactions. Each component defines one or more ports using the SIDL to describe the calling interface. Communication links between components are implemented by connecting compatible ports, where port compatibility is defined as object-oriented type compatibility of the port interfaces as described in the SIDL. As shown in Figure 2, each port has two parts. The first part is a library of framework-specific but component-independent functionality pertaining to component interaction (e.g., adding a listener to an object) and has the same interface for every component. The second part implements component-specific but framework-independent functionality; this part can be automatically generated by a proxy generator based on the component definition expressed in SIDL, and is referred to as a component stub. For example, a component stub may contain marshaling functions in a distributed environment.

- CCA *Services* present a framework abstraction that can be used in the component stub implementation as well

as by the components themselves; this CCA element provides a clear definition of the *minimal* services a CCA framework must implement in order to be CCA compliant. Two critical concerns guiding this design are that the services enable high-performance interactions and are sufficiently compact and user friendly to enable a rapid learning curve for component writers, many of whom will not be computer scientists. As such, we have identified that the key CCA services are creation of CCA Ports and access to CCA Ports, which in turn enable connections between components.

Additional common facility services to handle naming, relationship management, error handling, querying, etc., are of course also important, since in practice many components would need and could share these facilities. However, because the particular needs of different components and frameworks vary considerably depending on usage environment, we view these areas within a secondary service category that is beyond the scope of this paper.

- The *Configuration API* encompasses the functionality necessary to support interaction between components and a builder. This includes functions such as as notifying components that they have been added to a scenario or deleted from it, redirecting interactions between components, or notifying a builder of a component failure.

- The *Repository API* defines the functionality necessary to search a framework repository for components (as defined by the SIDL), as well as to manipulate components within the repository.

A reference implementation is tracking the evolution of the common component architecture. Likewise, several ongoing computational science projects are experimenting with the CCA to manage interoperability among components developed by different research groups; these experiences will motivate further extensions and refinements to design.

The following sections discuss features of this architecture that we believe are most critical for high-performance scientific computing, namely the SIDL and the ports model. Work on the other portions of the CCA standard is also in progress, but details are beyond the scope of this paper.

5 The Scientific IDL

The Scientific Interface Definition Language (SIDL) is a high-level description language used to specify the calling interfaces of software components and framework APIs in the component architecture. SIDL provides language interoperability that hides language dependencies to simplify the interoperability of components written in different programming languages. With the proliferation of languages used for numerical simulation—such as `C`, `C++`, `Fortran 77`, `Fortran 90`, `Java`, and `Python`—the lack of seamless language interoperability can be a significant barrier to developing reusable scientific components.

For the purposes of our high-performance scientific component architecture, SIDL must be sufficiently expressive to represent the abstractions and data types common in scientific computing, such as dynamically dimensioned multidimensional arrays and complex numbers. Unfortunately, no such IDL currently exists, since most IDLs have been designed for operating systems [18, 19] or for distributed client-server computing in the business domain [33, 41, 47].

The basic design of our scientific IDL borrows many concepts from current standards, such as the CORBA IDL [41] and the `Java` programming language [29]. This approach allows us to leverage existing IDL technology and language mappings. For example, CORBA already defines language mappings to `C`, `C++`, and `Java`, and ILU [33] (which supports the CORBA IDL) defines language mappings to `Python`.

The scientific IDL provides additional capabilities necessary for scientific computing [14, 38]. It supports object-oriented semantics with an inheritance model similar to that of `Java` with multiple inheritance of abstract interfaces but single inheritance of implementations. IDL support for multiple inheritance with method over-riding is essential for scientific libraries that exploit polymorphism through multiple inheritance, such as some of solvers under development by the Equation Solver Interface [21] group. The IDL and associated run-time system provide facilities for cross-language error reporting. We have also added IDL primitive data types for complex numbers and multidimensional arrays for expressibility and efficiency when mapping to implementation languages.

SIDL also supports reflection and dynamic method invocation, which are important capabilities for a component architecture. Interface information for dynamically loaded components is often unavailable at compile-time; thus, components and the associated composition tools and frameworks must discover, query, and execute methods at run-time. The SIDL reflection and dynamic method invocation mechanisms are based on the design of the `Java` library classes in `java.lang` and `java.lang.reflect`. The SIDL compiler automatically generates reflection information for every interface and class based on its IDL description.

In addition to existing CORBA language mappings, we are developing both `Fortran 77` and `Fortran 90` mappings of our scientific IDL to enable scientific programmers both to call and to write components in `Fortran`.

The `Fortran 77` language mapping is similar to the `C` language mapping defined by CORBA except that SIDL interfaces and classes are mapped to `Fortran` integers instead of opaque data types. The SIDL run-time environment automatically manages the translation between the `Fortran` integer representation and the actual object reference. The `Fortran 90` language mapping is still under development. `Fortran 90` is a particular challenge for scientific language interoperability, since `Fortran 90` calling conventions and array descriptors vary widely from compiler to compiler.

6 Component Interaction through Ports

Every component architecture is characterized by the way in which components are composed together into applications. As introduced in Section 4, *CCA Ports* can be considered communication end points that define the connection model for component interactions. Within Figure 1, ports define the interactions between relatively tightly coupled parallel numerical components, which typically require very fast communication for scalable performance; ports also define loosely coupled interactions with possibly remote components that monitor, analyze, and visualize data.

To address this range of latency requirements, we adopt a *provides/uses* interface exchange mechanism, similar to that within the CORBA 3.0 proposal [42]. This approach enables connections that do not impede inter-component performance, yet allows a framework to create distributed connections when desired. In the ideal case, an attached component would react as quickly as an inline function call. We refer to this situation as *direct connection*, which is further discussed in Section 6.2. Loosely coupled distributed connections should be available through the very same interface as the tightly coupled direct connections, and without the components needing to be aware of the connection type. This need arises from the fact that high-performance components will often be parallel programs themselves. A parallel component may reside inside a single multiprocessor or it may be distributed across many different hosts. Existing component models have no concept of attaching two parallel components together, and existing research systems, such as Cumulvs [28], PAWS [6], and PARDIS [37], approach this problem in different ways. We therefore introduce a *collective port* model to enable interoperability between parallel components, as discussed in Section 6.3.

We briefly survey approaches used for component interactions in other systems. In the Java Beans model [20], components notify other "listener" components by generating events. Components that wish to be notified of events register themselves as "listeners" with the target components. In the DCOM model [46], one component calls the interface functions exported by another. In the proposed CORBA 3 component model, both events and a *provides/uses* interface model[42] are used.

6.1 The Basics of CCA Ports

The concept of *CCA Ports* arises from the data-flow world, where component interactions are limited to pipelining data from one component to the next. CCA Ports are generalized to admit method calls and return values along the pipeline, allowing for a richer variety of component interactions. Links between components are implemented by a *provides/uses* (i.e., input/output) interface design pattern. The CCA Port concept is flexible enough to allow direct component interface connections for high-performance, or connections through proxy intermediaries enabling distributed object interactions. Significantly, in the CCA model, connecting ports is the responsibility of the framework; therefore, a particular CCA component may find itself connected in a variety of different ways depending on its environment of use.

In the CCA architecture components are linked together by connecting a "port" interface from one component to a "port" interface on another. There are two types of ports:

- *Provides* (or input) port. A *Provides* port is an interface that a component provides to others.
- *Uses* (or output) port. A *Uses* port interface has methods that one component wants to call on another component.

Provides ports are "listeners" in the sense that they listen to *Uses* interfaces (i.e., calls of their functions by another component). Each *Uses* port maintains a list of listeners. To connect one component to another, one simply adds a *Provides* (input) port of one component to another's *Uses* (output) port. This approach follows many facets of the proposed CORBA 3.0 design.

When a component calls a member function on one of its *Uses* ports, the same member function on each "listening" *Provides* port is called. Note that this means one call may correspond to zero or more invocations on provider components. If a value is returned by any of the interface's methods, then the user-provider link must be one-to-one.

As introduced in Section 4, all interaction between the component and its containing framework will occur through the component's `CCAServices` object, which is set by the containing framework. The component creates and adds *Provides* ports to the `CCAServices`, and registers and retrieves *Uses* ports from the `CCAServices`. The CCAServices object enables access to the list of *Provides* and *Uses* Ports and can access an individual port by its instance name. It also implements a method for obtaining the various ports and registering them with the framework.

We next consider an example of using *CCA Ports* for the simulation introduced within Section 2.2. This example demonstrates the use of the SIDL to express scientific abstractions in component interfaces and illustrates the one possible approach to using the ports concepts introduced above.

[Note from Lois: Add example of using ports for one interaction introduced in Section 2, e.g., solving Ax=b, or visualizing field variables ... We need to work out the details of this at the CCA meeting!]

6.2 Direct-Connect Ports

Much of the reason for adopting the CORBA3-like *uses-provides* interface exchange mechanism for connecting CCA components is to enable high-performance computing. Absent the SIDL bindings to `UsesPort` and `ProvidesPort` interfaces, the overhead for the privilege of becoming a CCA component is nothing over a direct function call to the connected object. That is, there is no penalty for using the *uses-provides* component connection mechanism proposed in the CCA specification. The cost for the intervening SIDL binding for language independence is estimated to be around a 2-3 function call overhead.

Direct connections between components can be accomplished in a variety of ways, with probably the simplest being to create an object that exports a `DirectConnectPort` interface subclasses both the `UsesPort` and `ProvidesPort` interfaces. This way the framework gets a *Provides* interface from one component and gives that same interface directly to a connecting component as a *Uses* interface. Note that with this approach the framework still retains full control over the connection between components. At the framework's option the provided `DirectConnectPort` can be made a proxy through a separate `UsesPort` provided by the framework, without the components on either end of the connection needing to know. (See [9] for additional information and an applet demonstration.)

6.3 Collective Ports

Collective ports are an extension of the basic CCA ports that have been designed to handle interactions among parallel components and thereby to free programmers from focusing on the often intricate implementation-level details of parallel computations. Classes such as `PortInfo`, `ProvidesPort` and `UsesPort` are accessible from every thread or process in a parallel component. The CCA standard does not place any restrictions on how this is enabled by a particular implementation. For example, in a distributed memory model a copy of these classes could be maintained by every process participating in computation, whereas in shared memory a class could be represented only once. However, the CCA standard does require that as one of the CCA services the implementation maintain consistency between those classes.

Collective ports are designed to implement interactions among parallel components. The creation of a collective port requires that the programmer specify the mapping of data (or processes participating) in the operations on this port. In the most common case the mappings of the input and output ports match each other. For example, n processes or threads in one component are mapped to n processes or threads in the other, and in this case data would not need redistribution between the parallel components. In the second most common case, a serial component interacts with a parallel component. The semantics of this interaction are very similar to broadcast, gather, and scatter semantics used in collective communication. By having the mapping be a property of the `CCA-port`, we are not restricted to these common cases. Collective ports are defined generally enough to allow data to be distributed arbitrarily in the connected components.

7 Future Directions

This discussion has introduced the foundation for research by the CCA Forum in defining a common component architecture that supports the particular needs of the high-performance scientific computing community that have not been and will not likely be addressed in existing component standards. Key facets of this work are development of an IDL that supports scientific abstractions for component interface specification and definition of a ports connection model that supports collective interactions. This architecture enables connections that do not impede inter-component performance, yet allows a framework to create distributed connections when desired. We will enhance the CCA design and the corresponding reference implementation based on feedback from experiments of its use within several ongoing multi-institution computational science research projects.

Future plans include incorporating support for different computational models (e.g., SPMD and threaded models) and extending the definition of CCA ports to accommodate dynamic interfaces. We are also investigating repository and services issues, where we plan to leverage capabilities of the mainstream computing community as much as possible. Once component-level interoperability has been achieved, we plan to compile a database of actual components based on existing applications; such a database will not only validate our work, but will also provide valuable feedback on how to improve the standard. Finally, we plan to address framework-level interoperability.

Acknowledgments

The Common Component Architecture (CCA) Forum was initially inspired by the DOE2000 Initiative [17] and is motivated by ongoing collaborations with various scientific research groups. We especially thank Tom Canfield for conveying some of the challenges in device-scale combustion modeling, as discussed in Section 2.

The CCA Forum comprises researchers from national laboratories within the Department of Energy and collaborating academic institutions; current participants are Argonne National Laboratory, Indiana University, Lawrence Berkley National Laboratory, Lawrence Livermore National Laboratory, Los Alamos National Laboratory, Oak Ridge National Laboratory, Sandia National Laboratory and the University of Utah. The results presented here were developed with the participation of various individuals at these institutions, including Rob Armstrong, Pete Beckman, Randal Bramley, Robert Clay, Andrew Cleary, Al Geist, Paul Hovland, Bill Humphery, Kate Keahey, Scott Kohn, Lois McInnes, Bill Mason, Carl Melius, Brent Milne, Noël Nachtigal, Steve Parker, Barry Smith, Steve Smith, Brent Smolinski, and John Wu. These ideas have been influenced by laboratory and university software development teams, some of whose members are represented above.

References

[1] ALICE Web page. http://www.mcs.anl.gov/alice, Mathematics and Computer Science Division, Argonne National Laboratory.

[2] A. A. Amsden, P. J. O'Rourke, and T. D. Butler. KIVA-II: A computer program for chemically reactive flows with sprays. Technical Report LA-11560-MS, Los Alamos National Laboratory, May 1989.

[3] R. C. Armstrong and A. Chung. POET (parallel object-oriented environment and toolkit) and frameworks for scientific distributed computing. In *Hawaii International Conf. on System Sci.*, 1997.

[4] S. Atlas, S. Banerjee, J. Cummings, P. J. Hinker, M. Srikant, J. V. W. Reynders, and M. Tholburn. POOMA: A High Performance Distributed Simulation Environment for Scientific Applications. In *Supercomputing '95 Proceedings*, December 1995.

[5] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[6] P. H. Beckman, P. K. Fasel, W. F. Humphrey, and S. M. Mniszewski. Efficient Coupling of Parallel Applications Using PAWS. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computation*, July 1998.

[7] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plášil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski. What Characterizes a (Software) Component? *Software – Concepts and Tools*, 19:49–56, 1998.

[8] H. Casanova, J. Dongarra, C. Johnson, and M. Miller. Tools for Building Distributed Scientific Applications and Network Enabled Servers, 1998. In *Computational Grids*.

[9] CCA Ports Web page. http://z.ca.sandia.gov/~cca-forum/gport-spec.

[10] K. M. Chandy, A. Rifkin, P. A. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman. A World-Wide Distributed System Using Java and the Internet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, August 1996.

[11] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1997.

[12] R. L. Clay, K. Mish, and A. B. Williams. ISIS++ Web page. http://ca.sandia.gov/isis.

[13] R. Clay et al. Equation Solver Interface Working Group Web page. http://z.ca.sandia.gov/esi.

[14] A. Cleary, S. Kohn, S. Smith, and B. Smolinski. Language interoperability mechanisms for high-performance scientific computing. In *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, October 1998.

[15] J. P. Collins, P. Colella, and H. M. Glaz. Implicit-explicit eulerian godunov scheme for compressible flows. *J. Comp. Phys.*, 116:195–211, 1995.

[16] Common Component Architecture Forum. See `http://www.acl.lanl.gov/cca-forum`.

[17] DOE2000 Initiative Web page. http://www.mcs.anl.gov/DOE2000.

[18] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

[19] E. Eide, J. Lepreau, and J. L. Simister. Flexible and optimized IDL compilation for distributed applications. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 1998.

[20] R. Englander. *Developing Java Beans*. O'Reilly, June 1997.

[21] Equations Solver Interface Forum. See `http://z.ca.sandia.gov/esi/`.

[22] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[23] I. Foster and C. Kesselman. *Computational Grids: State of the Art and Future Directions in High-Performance Distributed Computing*. Morgan-Kaufman, 1998.

[24] G. Fox and W. Furmanski. Web Technologies in High Performance Distributed Computing, 1998. In *Computational Grids*.

[25] R. Freund, G. H. Golub, and N. Nachtigal. *Iterative Solution of Linear Systems*, pages 57–100. Acta Numerica. Cambridge University Press, 1992.

[26] D. Gannon, P. Beckman, E. Johnson, T. Green, and M. Levine. HPC++ and the HPC++Lib Toolkit. *Languages, Compilation Techniques and Run Time Systems (Recent Advances and Future Perspectives)*, To appear.

[27] D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. "component architectures for distributed scientific problem solving". *IEEE Computational Science and Engineering*, 5(2):50–63, 1998.

[28] A. Geist, J. Kohl, and P. Papadopoulos. CUMULVS: Providing Fault Tolerance, Visualization and Steering of Parallel Applications. *The International Journal of Supercomputer Applications and High Performance Computing*, (11):224–235, 1997.

[29] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*, July 1996. Available at `http://java.sun.com`.

[30] A. S. Grimshaw and W. A. Wulf. Legion — A View From 50,000 Feet. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computation*, August 1996.

[31] R. Hornung and S. Kohn. The use of object-oriented design patterns in the SAMRAI structured AMR framework. In *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, October 1998. See `http://www.llnl.gov/CASC/SAMRAI`.

[32] InDEPS Web page. http://z.ca.sandia.gov/~indeps/, Sandia National Laboratory.

[33] B. Janssen, M. Spreitzer, D. Larner, and C. Jacobi. *ILU Reference Manual*. Xerox Corporation, November 1997. Available at `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

[34] JavaSoft. *Java Native Interface Specification*, May 1997.

[35] A. Joshi, T. Drashansky, J. R. R. andS. Weerawarana, and E. Houstis. Multiagent simulatin of complex heterogeneous models in scientific computing. *Math. Comput. Simul.*, 44:43–59, 1997.

[36] K. Keahey, P. Beckman, and J. Ahrens. Ligature: Component architecture for high-performance applications. *presented at the 1st NASA Workshop on Performance-Engineered Information Systems; to appear in the International Journal of High-Performance and Scientific Applications*, 1998.

[37] K. Keahey and D. Gannon. PARDIS: A Parallel Approach to CORBA. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computation*, pages 31–39, August 1997.

[38] S. Kohn and B. Smolinski. Component interoperability architecture: A proposal to thc common componcnt architccture forum. In preparation, 1999.

[39] D. Kotz and R. Gray. D'Agents: Mobile Computing at Dartmouth College, visited May 7, 1999. http://agent.cs.dartmouth.edu.

[40] MPI: A message-passing interface standard. *International J. Supercomputing Applications*, 8(3/4), 1994.

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

[42] OMG. *Corba Components. Revision 3.0.* OMG TC Document orbos/99-02-05, March 1999.

[43] P. J. O'Rourke and M. S. Sahota. A variable explicit/implicit numerical method for calculating advection on unstructured meshes. *J. Comp. Phys.*, 143:312–345, 1998.

[44] W. Park, E. V. Belova, G. Y. Fu, X. Z. Tang, H. R. Strauss, and L. E. Sugiyama. Plasma simulation studies using multilevel physics models. *Physics of Plasmas*, 6:1796–1803, 1999.

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

[46] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.

[47] J. Shirley, W. Hu, and D. Magid. *Guide to Writing DCE Applications*. O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

[48] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, New York, 1997.

[49] G. von Laszewski, M.-H. Su, J. A. Insley, I. Foster, C. K. M. T. John Bresnahan, M. L. Rivers, S. Wang, B. Tieman, and I. McNulty. Real-time analysis, visualization, and steering of microtomography experiments at photon sources. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

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