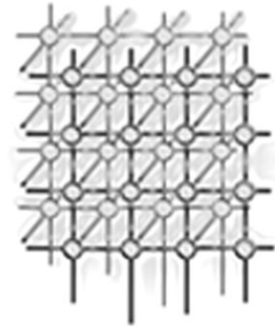
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; (in press)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.911



The CCA component model for high-performance scientific computing

Rob Armstrong[1,*,†], Gary Kumfert[2],
Lois Curfman McInnes[3], Steven Parker[4], Ben Allan[1],
Matt Sottile[5], Thomas Epperly[2] and Tamara Dahlgren[2]

[1]*Sandia National Laboratories, Livermore, CA 94551-9915, U.S.A.*
[2]*Lawrence Livermore National Laboratory, Livermore, CA 94550, U.S.A.*
[3]*Argonne National Laboratory, Argonne, IL 10439, U.S.A.*
[4]*University of Utah, Salt Lake City, UT 84112, U.S.A.*
[5]*Los Alamos National Laboratory, Los Alamos, NM 87545, U.S.A.*

SUMMARY

The Common Component Architecture (CCA) is a component model for high-performance computing, developed by a grass-roots effort of computational scientists. Although the CCA is usable with CORBA-like distributed-object components, its main purpose is to set forth a component model for high-performance, parallel computing. Traditional component models are not well suited for performance and massive parallelism. We outline the design pattern for the CCA component model, discuss our strategy for language interoperability, describe the development tools we provide, and walk through an illustrative example using these tools. Performance and scalability, which are distinguishing features of CCA components, affect choices throughout design and implementation. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: component; parallel computing; framework; design patterns

1. INTRODUCTION

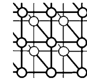
In the commercial software industry, the increased complexity of contemporary software has required a new mechanism that scales across people, geography, and time to achieve economics of scale. Component technology is the solution to this demand. It is component technology that enables

*Correspondence to: Rob Armstrong, Sandia National Laboratories, Livermore, CA 94551-9915, U.S.A.
†E-mail: rob@sandia.gov

Contract/grant sponsor: U.S. Department of Energy, Office of Science, Scientific Discovery through Advanced Computing (SciDAC) Program

Copyright © 2005 John Wiley & Sons, Ltd.

Received 19 June 2003
Revised 27 March 2004
Accepted 3 May 2004



MS Word™ documents to appear in MS Powerpoint™ slides and has led to the point-and-click graphical user interfaces (GUIs) that inhabit most desktops today. However, high-performance scientific computing has not benefited from these advancements.

Because the fast and scalable performance required for scientific computing puts most of the component tools developed for business applications out of reach for most scientists, a component model for parallel computing must be supported; this is now wholly absent from commercial offerings. A more detailed explanation of the inadequacies of commercial component models for high-performance computing can be found in a previous work [1]. In short, science needs a different set of component tools than the commercial sector supplies, and because of the scientific community's relatively small size, there is little financial incentive for mainstream industry to provide this infrastructure.

The Common Component Architecture (CCA) was conceived to fill this gap. Since 1998, the CCA has grown from a grass-roots effort among scientists (computer scientists and otherwise) to develop an architecture for high-performance component computing. The CCA has developed a component model that is concrete, but avoids fixed language bindings through a variety of mechanisms. Depending on how the language bindings are implemented, the performance is no different for applications composed of CCA components than if they were natively built in their own language. This is because of the CCA's 'direct connection' mechanism, which allows function calls to be connected directly from one module to another with nothing in the way to slow execution. The CCA design pattern accommodates interactive composition of components, although it allows a distinct separation of program execution from composition.

Because scientists use a variety of programming languages, and because it is crucial to build a comprehensive toolbox of components, there is no possibility for the CCA to force scientists to use a particular language. As a counter example, Java Beans requires the practitioner to write only in Java. For this reason, the Scientific Interface Definition Language (SIDL) was developed [2]. The CCA specification [3] is written solely in SIDL, and, when coupled with the appropriate language binding, specifies simultaneously what is to be a CCA component in Fortran 90, Fortran 77, C, C++, and Python. A component that conforms to the CCA specification and is written in any of these languages will work with any component written in another language.

2. THE CCA DESIGN PATTERN

This section introduces the two main facets of the CCA design pattern, namely the *provides/uses* and *single component multiple data* (SCMD) patterns, and discusses the separation of the design pattern from language bindings.

2.1. Provides/uses design pattern

The CCA model is based on the *provides/uses* pattern, similar to COM [4] and CORBA [5,6]. According to the CCA specification [3], a *component* must have a `setServices()` method to communicate with the framework, which aids in component composition. The specification also states that a *port* is a resource that can either be exported to or imported from a component. Most often this resource is a collection of subroutines implemented in some language (i.e. interface). In this way a *provided* port can be *used* by another component (see Figure 1).

Copyright © 2005 John Wiley & Sons, Ltd.

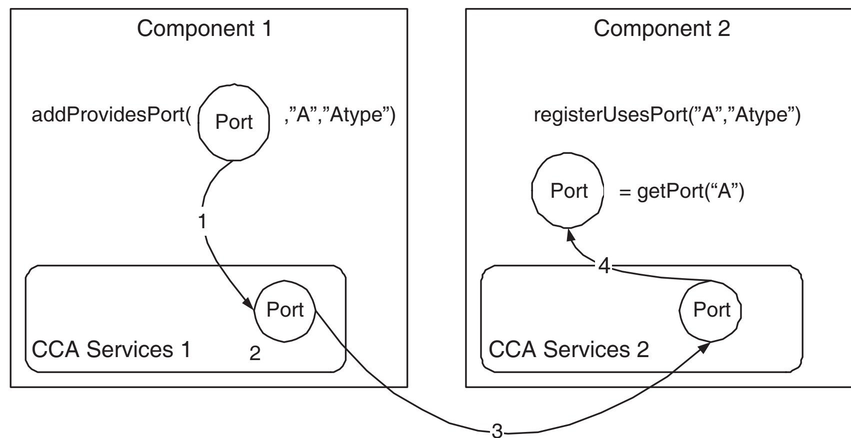
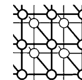


Figure 1. Diagram of the CCA *provides/uses* design pattern: (1) `Component1` donates a port by calling `addProvidesPort()`; (2) the Services object for `Component1` holds the port; (3) the framework connects it to another component, `Component2`; and (4) and `Component2` obtains the port with a call to `getPort()`.

This article emphasizes key features that permit the use of scientific components while retaining efficient and scalable performance and makes no attempt for exhaustive treatment of the entire CCA specification [3]. A few examples will be given in SIDL and C++ as an illustration of the mechanism[‡]. To better facilitate importing existing code into the CCA world, a CCA component must inherit the `gov.cca.Component` interface and implement its one method, `setServices()`, through which the component will communicate with its containing framework:

```
// sample SIDL file

package gov {

    package cca {

        /**
         * All components must implement this interface.
         */
        interface Component {

            /**
             * Obtain Services handle, through which the
             * component communicates with the framework.
             * This is the one method that every CCA Component
             * must implement.
```

[‡]All examples will be in the SIDL C++ binding to the CCA design pattern.



```
             */
            void setServices(in Services svcs);

        } // end interface

    } // end package cca

} // end package gov
```

In the following example a component provides a port of type `name.space.myProvidedPort`:

```
// sample C++ implementation of setServices() for providing component

void setServices(gov::cca::Services svcs) {
    /* ... */

    /* myImplementation is an interface this component wishes to
     * export, it is a kind of gov::cca::Port.
     */
    gov::cca::Port thePort = myImplementation;
    svcs.addProvidesPort(thePort, "name.space.myProvidedPort", "myPort", 0);

    /* ... */
}
```

and here a component registers to use `name.space.myPortToUse`:

```
// sample C++ implementation of setServices() for using component

void setServices(gov::cca::Services svcs) {
    /* ... */

    /* Register for the port we wish to use. */
    svcs.registerUsesPort("name.space.myPortToUse", "usePort", 0);

    /* ... */
}
```

and later uses it:

```
void some_function() {

    /* Later the port interface can be obtained ... */
    myUsesPort = svc.getPort("usePort");

    /* and used in some fashion ... */
    myUsesPort.some_method();

    /* and then released. */
    svc.releasePort("usePort");
}
```

Copyright © 2005 John Wiley & Sons, Ltd.

The above examples are meant to illustrate concretely the core of the CCA design pattern; naturally there is much more to the CCA specification than this. Further details can be found on the Web and in other publications [3,7].

A CCA framework is a generic term for a program that creates and connects CCA components, forming them into applications. The fact that the CCA is a *component* specification means that we specify only the behavior of components and facilities provided to components. Although the CCA specification admits and encourages the use of various frameworks supporting the specification, nothing whatsoever is stipulated regarding how a CCA framework is designed. Several CCA-compliant frameworks have been developed, including SCIRun [8,9], Ccaffeine [7], and XCAT [10]. Of these, SCIRun and Ccaffeine support high-performance parallel components and are featured in this paper. The CCA specification also supports distributed object components with the same design pattern and API; the XCAT framework deals exclusively with distributed object components.

In a nutshell, the CCA design pattern relies on moving a port interface from one component to another, thereby defining a connection between the two components. This pattern is quite powerful and is isomorphic to data-flow component models (e.g. SCIRun and AVS) and even event-driven Java Beans. Concretely, this means that Java Beans and the Java Bean component model could be bound to the CCA by a facade, with component creation, connection, and destruction handled by a framework that recognizes them only as CCA components. Although Java Beans is an unlikely candidate for subsuming into the CCA design pattern, this separation of design pattern and concrete implementation permits an open-ended view of what can be called a component and thus admits a wider circle of legacy software and performance options to the high-performance scientific component world.

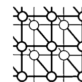
2.2. SCMD design pattern for SPMD computing

The most common identifiable pattern for parallel computing is single program multiple data (SPMD), where an identical program is run on every participating process, and the working data are decomposed among processes. Often parallel simulations have a strictly SPMD form (e.g. a combustion simulation done with the CCA [11]). However, sometimes they consist of multiple SPMD programs, partitioned on the same or different parallel machines, which are coupled with a complex data transfer (e.g. climate applications [12,13]), or a degenerate case where multiple processes are in communication with a single process (e.g. for visualization or data logging [14]).

The CCA takes the next logical extension from the SPMD paradigm to the SCMD paradigm. The SCMD mode of programming treats each component as a separate black-box entity that is required to be identically instantiated and configured on every participating process. In order to proceed, we must first define some nomenclature. A *cohort* of components is a set of components that are identically instantiated on separate processes. Each member of the cohort is identically connected to different *peer* instances residing in the same process. Peers are connected using the provides/uses design pattern of the CCA and reside together in the same process address space. A cohort spans all participating processes, and together its members encapsulate the parallel algorithm that the component represents (see Figure 2).

It is required of each cohort that all message passing and computation be conducted and self-organized among the cohort. Similarly, if each process contains an identical graph of connections among SCMD components, then the resulting composition is itself a SPMD program.

Copyright © 2005 John Wiley & Sons, Ltd.



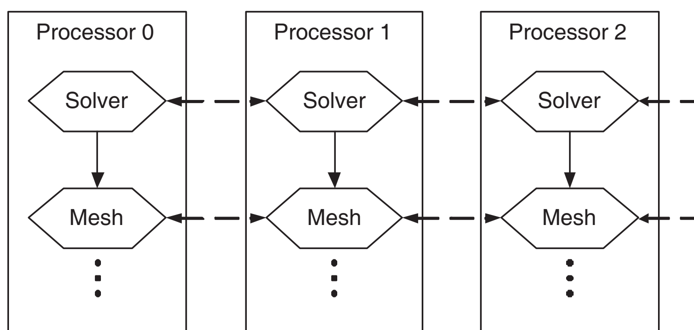


Figure 2. Diagram of the SCMD pattern. From the standpoint of nomenclature: the *cohort* of a component (such as Solver) is the set of instances connected by dashed horizontal arrows representing interprocess communication within a single distributed-memory component. The *peers* (distinct components) are connected by the vertical arrows representing in-process function calls. Peers are grouped in the same process address spaces, denoted by the rectangular boxes.

The SCMD programming paradigm, coupled with a uniform set of inter-component connections, will produce a SPMD program for the overall application. However, SCMD can also describe more general (non-SPMD) application structures, such as a SCMD simulation connected to a serial visualization tool, or multiple coupled parallel simulation components operating together to simulate a complex multi-physics problem.

Various CCA frameworks designed for parallel computing enable this SCMD style of execution. Ccaffeine [7], a CCA framework designed solely for the SCMD paradigm, enforces this style by providing only a single arena for component instantiation and connection. The user creates a single application description, and Ccaffeine replicates it on every participating processor. There is no facility within Ccaffeine for differentiating the component connections among participating processes, and therefore everything is forced to be identically configured on every processor. SCIRun [8,9] implements a more generic interconnection mechanism, thereby allowing a non-SPMD application to be built entirely from SCMD components.

In the SCMD model, every participating process has the same components connected together in the same way. Components are free to differentiate instances within their cohort to do whatever their algorithm demands. Any method invocations on an imported interface (obtained through `getPort()`) must be performed in every process in the same logical sequence. Note that the CCA specifies nothing to enforce this behavior, nor would that be practical. Note also that through its documentation, a particular port interface may specify that some methods are side-effect free and are safe to call individually and out of sequence. The CCA has considered specifying a key word, called ‘collective’ in SIDL, but it would be at best advisory and, presently, the CCA leaves this to the interface implementor’s documentation. Other CCA-related research is experimenting with using collective calls described by SIDL to couple two or more SCMD components with different numbers of processors.

Copyright © 2005 John Wiley & Sons, Ltd.

2.3. Separating the design pattern from language bindings

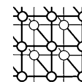
Although the CCA has a SIDL specification [3], the component design pattern is separable from any particular language binding of it. In fact, there are several bindings to languages in use by implementers of CCA applications. Historically, the first CCA binding, now called Classic [15], was implemented entirely in C++ and served as a test ground for the nascent component model. From the beginning, however, the SIDL [2] was conceived to bring language independence to the CCA component model. When Babel [16,17], an implementation of SIDL, became a viable tool, the official CCA specification [3] was written using that tool. The Babel binding brings object-oriented, transparent language interoperability to the component model for C, C++, Python, and Fortran. Recently, an additional binding based on Chasm [18] has been developed. Chasm [18] generates much of the support code necessary to be a CCA Fortran component, creating a programming paradigm that is more comfortable for Fortran programmers who prefer a non-object-oriented, imperative style. Work is currently underway to create an automatic translation from Chasm CCA components into Babel-bound CCA components, so that they would be compliant with the CCA SIDL specification.

At first these wholly or partly incompatible bindings seem inconsistent with the CCA vision of interoperability, and they would be if that were the only consideration. Two other considerations are relevant: performance and flexibility. Application developers resist a one-size-fits-all strategy, and may decide that they wish to use solely one language and not pay the modest performance penalty (around 1% of compute time, see Section 4.2) for SIDL. There is also the historical need to support previous bindings such as Classic, which has users who have legacy code written to it. Currently SCIRun [8,9] supports two bindings following the CCA design pattern, and Ccaffeine [7] supports three. In addition, there are currently three CCA-based applications in the U.S. Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) initiative [19] and two CCA-based applications in the U.S. Department of Energy's ASCI program [20]. Each of the bindings listed here is used by at least one of these applications. The key point is that separating the design pattern from the language bindings makes such choices possible.

This separation has certain advantages for frameworks as well. Independence of the design pattern from the language binding allows reuse of code in the portion dedicated to the design pattern and more innovation in the design of the underlying language bindings. The mechanics of creating, connecting, running, disconnecting, and destroying components can be separated from the actual implementation of the components or the application. Any software dedicated to these framing tasks can be reused regardless of its implementation.

As an illustration of the power of this approach, refer to Figure 3. The top portion of the figure shows what the user is trying to perform—to connect three different components that perform some task. Two of these components are written using the SIDL-based CCA specification, but the rightmost uses a different component model, such as the Classic binding described here, or perhaps even something more foreign such as CORBA, COM, or SCIRun dataflow components. The bottom portion of the figure, below the dotted line, shows how the framework might actually deploy those components. Each component is connected to the framework using adapter classes that conform to the requirements of the concrete component model, i.e. Classic, SIDL, etc., and provides the core of the framework with a consistent interface for managing those components. Sometimes additional *bridge* components may be inserted (as shown) to mediate communication between the disparate models. The CCA builder

Copyright © 2005 John Wiley & Sons, Ltd.



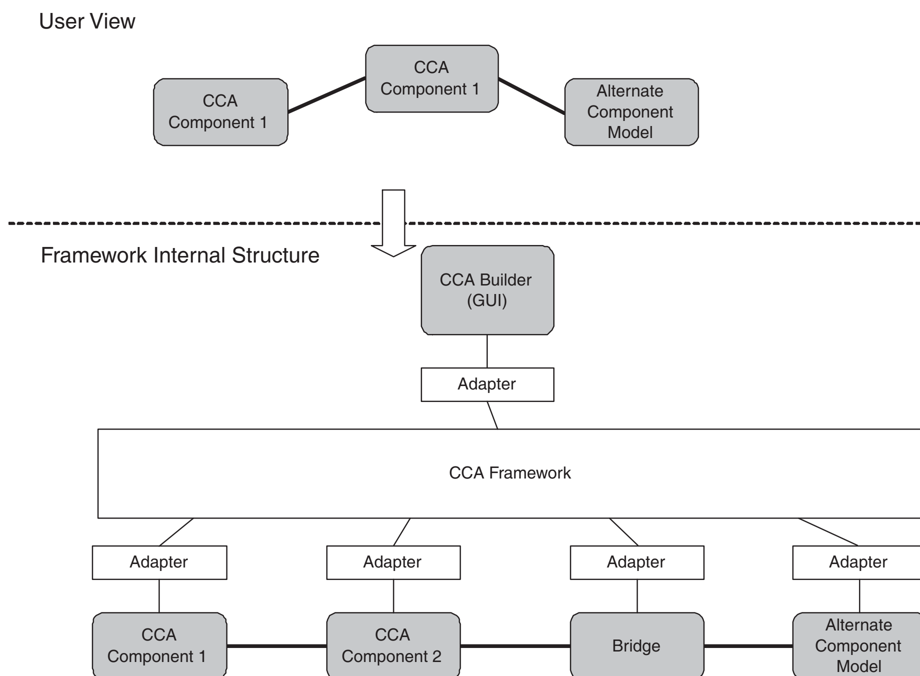
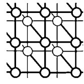


Figure 3. An example of integrating multiple component models using the CCA design pattern.

shown in the diagram can use the CCA-specified protocol to manipulate any of the components in the diagram, even if those components are not true SIDL-based CCA components. This adapter-based approach provides a flexible model that can be used to combine a diverse range of tools for a single scientific program.

Separating the language implementation from the component design pattern has important implications for performance. By separating the language implementation from the design pattern, the user can write all components in a single homogeneous language (e.g. C or Fortran) and not pay any performance penalty at all for introducing components into the application: the language calls are made natively. If the user wants to create components of broader appeal and wishes to pay the modest performance penalty for SIDL interoperability, then they should use the official CCA specification in SIDL. Experience has shown that many users will first create their components in a language they already know, for example, using the Classic C++ binding to the CCA. After they become comfortable with component concepts, they will often move their components into SIDL and the official CCA specification.

Copyright © 2005 John Wiley & Sons, Ltd.



3. CCA LANGUAGE BINDINGS

Language bindings are where the CCA design pattern becomes working software. Although the component architecture has great flexibility, it also presents confusing choices for newcomers. This section is aimed at providing insight into the choices available.

The official CCA specification is expressed only in SIDL. Language bindings from SIDL to C, C++, Fortran 77, Fortran 90, Python, and Java are generated by the Babel tool (see Section 3.2). SIDL is supported by the Ccaffeine framework and an experimental version of SCIRun. There is a third framework, XCAT, that only supports distributed, Grid-enabled components not under consideration here.

Alternative bindings add additional flexibility for two distinct purposes: backwards compatibility and ongoing independent research and development. Before Babel became generally available, Ccaffeine used a C++-only language binding now referred to as *Classic* (see Section 3.1), which is still in use by early adopters. SCIRun—which predates the CCA itself—has a dataflow model and a parallel interface description language (PIDL) with its own C++ bindings (see Section 3.3). Chasm (see Section 3.4) is the most recent entry into the CCA language binding portfolio, supported only in the Ccaffeine framework, but strongly feeding technology into Babel's own Fortran 90 bindings.

3.1. Classic

As the CCA was being defined before SIDL became a usable reality, a C++ binding [15] was initially developed for the CCA. In this binding, a port is a C++ pure virtual class, implemented by the component that provides it. The performance of an application composed of Classic components is exactly that of C++. The one advantage of this binding is that methods invoked on ports have no overhead beyond that which C++ itself incurs. The obvious disadvantage to this approach is that every component in an application composed of these components must be written in C or C++. SIDL provides more interoperability with a usually acceptable performance penalty (see Section 4.2).

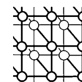
3.2. Babel and SIDL

Babel [16] is an IDL-based language interoperability tool that currently supports C, C++, Fortran 77, Fortran 90, Python, and Java. Any software in any of these languages, after being wrapped in Babel, is equally accessible from all other languages. To developers programming in a particular supported language, Babel provides the illusion that all 'Babelized' software is in their native language, when in fact, it could be any arbitrary combination of Babel-supported languages. In short, Babel removes 'implementation language' as a barrier to reusing software.

SIDL [17] is the input language for Babel. IDLs are used in the commercial sector, where the two most common are Microsoft's IDL for COM and the CORBA IDL. SIDL distinguishes itself from commercial IDLs by its intrinsic support for complex numbers and dynamically allocated, multi-dimensional, arbitrarily strided arrays. SIDL has a clean, simple grammar designed to be intuitive for computational physicists, biologists, chemists, etc.

SIDL provides a common set of object-oriented features in all supported languages—even those that lack intrinsic support for object-oriented features. Contemporary software engineering practices such as encapsulation, inheritance, polymorphism, exception handling, and reference counting are

Copyright © 2005 John Wiley & Sons, Ltd.



available using SIDL/Babel. When languages have built-in support for these features (e.g. reference counting in Python or object-oriented programming in C++), we use the native language capabilities. For procedural languages such as C and Fortran, we provide the missing capabilities. For example, SIDL exceptions are exposed as native exceptions in C++ and Java. Since Fortran has no native concept of catching and throwing exceptions, an extra argument is appended to the argument list. The value of the extra argument must be explicitly tested to check whether an exception has occurred.

In the context of CCA, using the SIDL mode of Ccaffeine allows a component to be implemented in any Babel-supported language and used with total disregard to that language. In the reverse sense, components do not know or care about Ccaffeine's implementation language either. In classic mode, Ccaffeine components rely on the fact that Ccaffeine is implemented in C++.

Performance is a critical issue for Babel because its target customers, people in scientific computing, care deeply about performance. With any inter-language situation and any object-oriented system supporting polymorphism (also known as virtual function dispatch or runtime binding), some overhead is technically unavoidable. With Babel these costs are compulsory, but often negligible. For large parallel runs with our alpha testers in the numerical library, *hypre* [21], the Babel overhead was so small that it could not be measured when compared with a raw C-only implementation [22]. In more detailed studies, we measured the function call overhead to be 2.7-times a raw Fortran 77 function call, which is on par with a C++ virtual function call and hundreds of times faster than using a popular, open source CORBA implementation [23].

3.3. SCIRun dataflow and PIDL bindings

An experimental version of the SCIRun framework provides bindings to CCA compliant SIDL/Babel-based components as described above and also currently provides support for two other component models: SCIRun dataflow and PIDL. The parallel dataflow model is ideal for visualization, and the provides/uses design pattern can be adapted to it perfectly. The SCIRun dataflow model provides support for the components that have been built in SCIRun over the last nine years and has a large base of users. In addition to simply reusing the GUI application builder and other software for manipulating components, a set of bridge components have been created. This approach allows SCIRun dataflow components and CCA components to be composed together in a single application.

Also supporting the same CCA is a binding called PIDL. This binding is an experimental extension of the SIDL language described above that can describe the interaction between two distributed parallel components [24]. Support for other component models is planned in the future.

3.4. Chasm and Fortran 90

Chasm is a research effort and a set of extensible tools that can be used to automatically create components out of existing Fortran modules. In order to adapt a Fortran module to the Ccaffeine framework, Chasm automatically generates a C++ adapter class to mediate between a Fortran module and the C++ framework. A Fortran component obtains a port from the framework via the adapter and then makes calls on the port, again through the same adapter. The `this` pointer for the port is contained in a Fortran derived type that is explicitly passed as the first argument in a port call. This structure contains C pointers and other state variables required to identify the associated port instance for a given subroutine invocation. While a Fortran component cannot be 'directly connected'

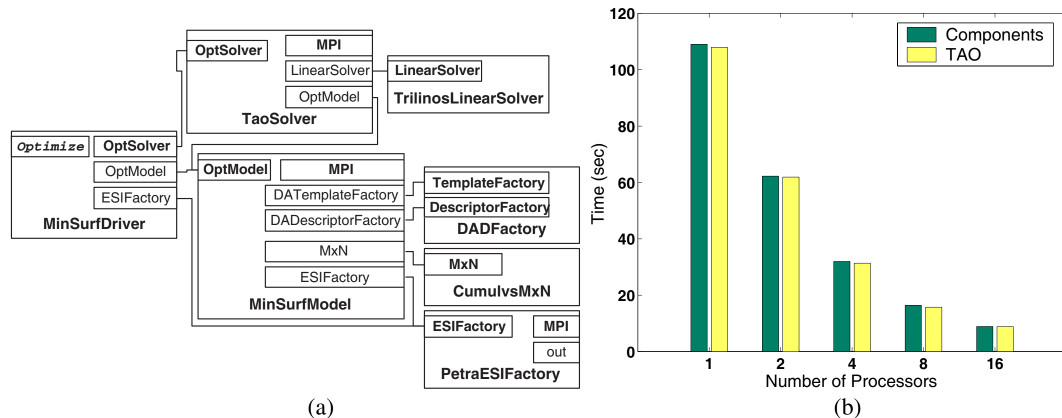


Figure 4. (a) The component wiring diagram for the unconstrained minimization example; and (b) the component overhead for finding the solution of this model on a 250×250 mesh.

to a C++ component, the overhead is small because there is only one additional function call through the adapter.

4. SCIENTIFIC COMPONENTS

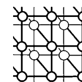
Members of the CCA Forum have developed a range of production components that are used in scientific applications as well as prototype components that aid in teaching CCA concepts. These freely available components include various service capabilities, tools for mesh management, discretization, linear algebra, integration, optimization, parallel data description and data redistribution, visualization, and performance evaluation (see [25] for details). The following section introduces an example CCA application that has the same form as some of our production applications, except that the overall run times are manageably shorter.

4.1. A numerical component example

To demonstrate some sample components that use the Ccaffeine framework [7], we consider a model problem for unconstrained minimization. The model is sufficiently simple to enable a succinct description, yet analogous in form to several computational chemistry applications [26–28] that motivate this work. These examples are also currently under development using CCA components. Given a rectangular two-dimensional domain and boundary values along the edges of the domain, the objective is to find the surface with minimal area that satisfies the boundary conditions, that is, to compute $\min f(x)$, where $f : \mathbb{R}^n \to \mathbb{R}$ (see [29]).

Figure 4(a) illustrates the composition of this application using Ccaffeine's GUI. The optimization solver component in this snapshot has been configured to use an inexact Newton method, which

Copyright © 2005 John Wiley & Sons, Ltd.



requires an application-specific component to evaluate the model's function, gradient, and Hessian, as well as a linear algebra component to manipulate vectors, matrices, and linear solvers. The application uses the SPMD pattern and is composed of SCMD components, as discussed in Section 2. The large boxes in the wiring diagram show components for the optimization solver, linear algebra, and application-specific driver and model. Inside each large component box on the left are smaller boxes representing *provides* ports. Similarly, inside each large box on the right are the *uses* ports. Some components provide ports but do not use any, such as CumulvsMxN. The lines in the diagram represent connections between *uses* and *provides* ports. For example, the `TaoSolver` optimization component's `OptModel` *uses* port is connected to the `MinsurfModel`'s `OptModel` *provides* port; hence, the optimization solver component can invoke the interface methods for function, gradient, and Hessian computation that the MinsurfModel component has implemented. The special `GoPort` (named 'Optimize' in this application) is used to start the execution of the application. The wiring diagram also shows components for parallel data description and parallel data redistribution, which can be used for dynamic visualization of the solution as the simulation progresses; these components have been developed by our CCA collaborators D. Bernholdt and J. Kohl of Oak Ridge National Laboratory (see [25] for further information).

We briefly describe two of the optimization ports; additional details can be found in [30]. The `OptimizationSolver` port defines a prototype high-level interface to optimization solvers; this interface is expected to evolve as common interface definition efforts for optimization software progress. The `TaoSolver` component implementation is based on the Toolkit for Advanced Optimization (TAO) [31,32], which provides algorithms for constrained and unconstrained optimization. The single abstract interface for the `TaoSolver` component enables the user to employ a variety of solution techniques for unconstrained minimization, including Newton-based line search and trust region strategies, a limited-memory variable metric method, and a nonlinear conjugate gradient method. The CCA component implementation is a thin wrapper over existing TAO interfaces, and thus took minimal effort to develop.

The `OptimizationModel` port includes methods that define the optimization problem via methods for function, gradient, Hessian, and constraint evaluation. The `MinsurfModel` component implements the minimum surface area model described above. The development of this component required moderate effort and involved the conversion of an existing code to use new abstractions for vectors and matrices and to implement the component-specific functionality. Many different optimization problems can be implemented by providing the `OptimizationModel` port, which is then used by solver components such as `TaoSolver`.

4.2. Achieving high performance with the CCA

One of the main goals of the CCA specification is to help manage the complexity of scientific simulations while achieving efficient and scalable performance; however, a common concern about the use of components is the effect that overhead may have. We conducted experiments to evaluate the performance differences between the CCA component version and the traditional library-based implementation of this optimization problem. Our results were obtained on a Linux cluster of dual 550 MHz Pentium-III nodes with 1 GB of RAM each, connected via Myrinet.

These particular experiments used the inexact Newton method, described above, which at each nonlinear iteration requires a function, gradient, and Hessian evaluation, as well as an approximate

Copyright © 2005 John Wiley & Sons, Ltd.

solution of a linear system of equations. The component wiring diagram in Figure 4(a) illustrates these interactions via port connections; the library-based version of code is organized similarly, although all software interactions occur via traditional routine calls within application and library code instead of employing component ports. The results presented in Figure 4(b) used as a linear solver the conjugate gradient method and block Jacobi preconditioner with no-fill incomplete factorization as the solver for each subdomain.

By employing abstract interfaces at several levels of the implementation, the component version introduces a number of virtual function calls, including linear solver methods and function, gradient, and Hessian evaluation routines used by the Newton solver. Figure 4(b) gives the difference in total execution time between the component implementation and the original application on one, two, four, eight, and 16 processors for a fixed-size problem of dimension 250×250. The differences between the CCA Classic configuration and original library-based code are less than 2% of total execution time for this application; the differences between the Classic and SIDL versions of the component-based simulation are less than 1%. Such performance results indicate that the CCA component overhead is negligible when reasonable levels of granularity for component interactions are employed. Given the production chemistry applications that motivate this work utilize even larger granularity components, the only conclusion is that the performance cost is immeasurable and insignificant. This figure also illustrates that good scaling behavior is not lost by the use of CCA components: the time for the total component-based computation on one processor is 109 seconds, while the corresponding time using a 16-processor linux cluster is 9 seconds. More detailed analysis of the overhead of individual component invocations, including the use of SIDL in various programming languages, can be found in [23].

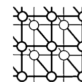
5. CONCLUSIONS AND FUTURE WORK

The purpose of this paper is to expose the CCA design pattern specifically for high-performance component computing. Several observations were made as follows.

1. The *provides/uses* component model or design pattern preserves performance by allowing direct method invocations between components without an intervening proxy (see Section 2).
2. The component model can be considered separable from the concrete language binding, causing the trade-off between performance and interoperability to be made at the language level. Given that the components are connected by an exchange of interfaces, the language binding determines the speed of a method invocation (see Section 3).
3. The CCA Forum decided that the SIDL is the concrete specification for CCA, opting for the promise of language interoperability—important for building a library of interoperable components—over the slight performance decrease. Finally, it is demonstrated that this tradeoff is justified in actual practice (see Section 4).

Confusion between the SIDL-based CCA specification and other language bindings that also follow the CCA design pattern is possible, even likely. The specification, if followed, will yield components that are interoperable regardless of their language of implementation. In practice, however, there are two major reasons for entertaining bindings outside of the specification: (1) compatibility with existing code; and (2) experimentation in other designs for bindings. Several applications currently are written outside of the specification but inside the design pattern, using CCA tools to accomplish their work.

Copyright © 2005 John Wiley & Sons, Ltd.



Regardless of their current compliance with the concrete CCA specification, such components have a clear and relatively simple path towards compliance by writing adapters to the most convenient SIDL representation. Applications written outside the design pattern are essentially without hope of interoperability.

The CCA Forum is an active and ongoing collaboration among numerous researchers at various U. S. Department of Energy laboratories and collaborating universities. The current CCA specification [3] (version 0.6) enables a powerful array of realistic scientific applications, including combustion, chemistry, and climate simulations. Ongoing work will further enhance the capabilities of the architecture and will assist development of abstract interfaces for common application domains.

ACKNOWLEDGEMENTS

We gratefully acknowledge all participants within the CCA Forum; the development of the CCA specification, infrastructure, and components is an ongoing collaboration among many individuals from various institutions. We also thank Steve Benson, Boyana Norris, and Jason Sarich for generating the performance results discussed in Section 4.

This work has been funded by the Scientific Discovery through Advanced Computing (SciDAC) Program of the U.S. Department of Energy, Office of Science.

REFERENCES

1. Armstrong R, Gannon D, Geist A, Keahey K, Kohn S, McInnes LC, Parker S, Smolinski B. Toward a common component architecture for high-performance scientific computing. *Proceedings of High Performance Distributed Computing*. IEEE Computer Society Press: Los Alamitos, CA, 1999; 115–124.
2. Epperly T, Kohn S, Kumfert G. Component technology for high-performance scientific simulation. *Working Conference on Software Architectures for Scientific Computing Applications*, Ottawa, Ontario, Canada, October 2000. International Federation for Information Processing, 2000.
3. CCA specification. http://www.cca-forum.org/specification [12 September 2005].
4. Box D. *Essential COM*. Addison-Wesley: Reading, MA, 1997.
5. Object Management Group. CORBA components. *OMG TC Document orbos/99-02-05*, March 1999.
6. Siegel J. OMG overview: CORBA and the OMG in enterprise computing. *Communications of the ACM* 1998; **41**(10):37–43.
7. Allan BA, Armstrong RC, Wolfe AP, Ray J, Bernholdt DE, Kohl JA. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience* 2002; **14**:1–23.
8. Johnson CR, Parker S, Weinstein D, Heffernan S. Component-based problem solving environments for large-scale scientific computing. *Journal on Concurrency and Computation: Practice and Experience* 2002; **14**:1337–1349.
9. Parker SG, Beazley DM, Johnson CR. Computational steering software systems and strategies. *IEEE Computational Science and Engineering* 1997; **4**(4):50–59.
10. Govindaraju M, Krishnan S, Chiu K, Slominski A, Gannon D, Bramley R. Merging the CCA component model with the OGSI framework. *Proceedings of the 3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 12–15 May 2003, Tokyo, Japan. IEEE Computer Society Press: Los Alamitos, CA; 2003.
11. Lefantzi S, Ray J, Najm HN. Using the common component architecture to design high performance scientific simulation codes. *Proceedings of the International Parallel and Distributed Processing Symposium*, Nice, France, 2003.
12. Jacob R, Schafer C, Foster I, Tobis M, Anderson J. Conceptual design and performance of the fast ocean atmosphere model, version one. *Proceedings of the International Conference on Computational Science (ICCS)* (*Lecture Notes in Computer Science*, vol. 2073), Alexandrov VN, Dongarra JJ, Juliano BA, Renner RS, Tan CJK (eds.). Springer: Berlin, 2001; 175–184.
13. Larson J, Jacob R, Ong E. Model Coupling Toolkit Web site. http://www.mcs.anl.gov/mct [12 September 2005].
14. Geist A, Kohl J, Papadopoulos P. CUMULVS: Providing fault tolerance, visualization and steering of parallel applications. *The International Journal of High Performance Computing Applications* 1997; **11**(3):224–236.
15. Classic interface definition. http://www.cca-forum.org/bindings/classic [12 September 2005].

Copyright © 2005 John Wiley & Sons, Ltd.

16. Babel Web site. http://www.llnl.gov/CASC/components [12 September 2005].
17. Dahlgren T, Epperly T, Kumfert G. *Babel User's Guide* (0.8 edn). CASC, Lawrence Livermore National Laboratory, Livermore, CA, January 2003.
18. Rasmussen C, Lindlan K, Mohr B, Striegnitz J. Chasm: Static analysis and automatic code generation for improved Fortran 90 and C++ interoperability. *Proceedings of the Los Alamos Computer Science Symposium*. Los Alamos National Laboratory: Los Alamos, NM, 2001.
19. Scientific Discovery through Advanced Computing (SciDAC) Initiative Web site. http://www.scidac.org [12 September 2005].
20. ASCI Web site. http://www.sandia.gov/NNSA/ASC [12 September 2005].
21. Falgout RD, Yang UM. *hypre*: a library of high performance preconditioners. *Computational Science—ICCS 2002 Part III* (*Lecture Notes in Computer Science*, vol. 2331), Sloot PMA, Tan CJK, Dongarra JJ, Hoekstra AG (eds.). Springer: Berlin, 2002; 632–661.
22. Kohn S, Kumfert G, Painter J, Ribbens C. Divorcing language dependencies from a scientific software library. *Proceedings of the 10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 2001.
23. Bernholdt DE, Elwasif WR, Kohl JA, Epperly TGW. A component architecture for high-performance computing. *Proceedings of the Workshop on Performance Optimization via High-Level Languages (POHLL-02)*, New York, June 2002. Available at: http://www.ece.Isu.edu/jxr/ics02workshop.html [12 September 2005].
24. Damevski K, Parker S. Parallel remote method invocation and m-by-n data redistribution. *Proceedings of the 4th Los Alamos Computer Science Symposium*, October 2003. Los Alamos National Laboratory: Los Alamos, NM, 2003.
25. Common Component Architecture Forum. See http://www.cca-forum.org [12 September 2005].
26. Harrison R, *et al*. NWChem: A computational chemistry package for parallel computers, version 4.0.1. *Technical Report*, Pacific Northwest National Laboratory, 2001. Available at: http://www.emsl.pnl.gov/docs/nwchem/nwchem.html [12 September 2005].
27. Janssen C, Seidl E, Colvin M. Object-oriented implementation of *ab initio* programs. *Parallel Computers in Computational Chemistry* (*ACS Symposium Series*, vol. 592). ACS: Washington, DC, 1995.
28. Benson S, Krishnan M, McInnes L, Nieplocha J, Sarich J. Using the GA and TAO toolkits for solving large-scale optimization problems on parallel computers. *Technical Report ANL/MCS-P1084-0903*, Argonne National Laboratory, 2003.
29. Averick BM, Carter RG, Moré JJ. The MINPACK-2 test problem collection. *Technical Report ANL/MCS-TM-150*, Argonne National Laboratory, 1991.
30. Norris B, Balay S, Benson S, Freitag L, Hovland P, McInnes L, Smith B. Parallel components for PDEs and optimization: Some issues and experiences. *Parallel Computing* 2002; **28**:1811–1831.
31. Benson S, McInnes LC, Moré J. A case study in the performance and scalability of optimization algorithms. *ACM Transactions on Mathematical Software* 2001; **27**:361–376.
32. Benson S, McInnes LC, Moré J, Sarich J. TAO users manual. *Technical Report ANL/MCS-TM-242—Revision 1.5*, Argonne National Laboratory, 2003. Available at: http://www.mcs.anl.gov/tao [12 September 2005].

Copyright © 2005 John Wiley & Sons, Ltd.