A Component Architecture for High-Performance Computing*

David E. Bernholdt, Wael R. Elwasif, and James A. Kohl
{*bernholdtde,elwasifwr,kohlja*}*@ornl.gov*
Computer Science and Mathematics Division
Oak Ridge National Laboratory
P. O. Box 2008
Oak Ridge, TN 37831-6367 USA

Thomas G. W. Epperly
tepperly@llnl.gov
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P. O. Box 808
Livermore, CA 94551 USA

Abstract

The Common Component Architecture (CCA) provides a means for developers to manage the complexity of large-scale scientific software systems and to move toward a "plug and play" environment for high-performance computing. The CCA model allows for a direct connection between components within the same process to maintain performance on inter-component calls. It is neutral with respect to parallelism, allowing components to use whatever means they desire to communicate within their parallel "cohort." We will discuss in detail the importance of performance in the design of the CCA and will analyze the performance costs associated with features of the CCA.

1 Introduction

In some ways, high-performance scientific computing is a victim of its own success. The ability to simulate physical phenomena in a scientifically useful way leads to demands for more sophisticated simulations with greater fidelity and complexity. At the same time, the supercomputers on which such simulations are run grow ever more powerful, but simultaneously more complex. Obtaining maximum performance from modern supercomputers requires careful algorithm design, including management of multiple levels of the memory hierarchy. Combining the support of a range of modern supercomputer architectures with the increasing demands from the scientific side of the problem can lead to nearly unmanageable complexity in the software created for modern computational science.

The computer science community is exploring a variety of approaches to help alleviate some of the complexity of large-scale scientific software. Libraries or computational engines may be created with algorithms and/or algorithmic parameters optimized for the target computer system. While this approach focuses on performance issues, it may also provide some help with complexity, since in many cases algorithms are abstracted and parameterized to cover a range of computer systems. Domain-specific, high-level languages can greatly simplify the scientist's view of the scientific programming problem, but typically a scientist will rely on a large and complex infrastructure of libraries, computational engines, or generated code in more traditional programming languages. This approach shifts much of the complexity to the development of the libraries, engines, or tools, but once again to the extent these tools embody important (often domain-specific) abstractions and generalizations, they can further reduce the complexity of the overall software system.

*Research supported by the Mathematics, Information, and Computational Sciences Office, the Office of Advanced Scientific Computing Research, and the U. S. Department of Energy under contract no. DE-AC05-00OR22725 with UT-Battelle, LLC, and W-7405-Eng-48 with the University of California. LLNL report UCRL-JC-148723

Even with the benefit of such techniques, high-performance simulation software tends to be large and complex. Moreover, there is much "legacy" software in the scientific community which cannot, for technical or practical reasons (i.e., time or funding), be rewritten to the extent required to accommodate these high-level approaches. Consequently, it is valuable to look at other ways to help scientific programmers manage the complexity of their software systems.

One such approach which has become very popular and successful in other areas of computing, most notably the "business" and "internet" areas, is component-based programming. Components may be thought of as objects that encapsulate useful units of functionality and interact with other components only through well-defined interfaces. To the extent that these interfaces are specified in such a way as to be broadly useful to a community (i.e., scientific domain) rather than a specific program, the component approach can facilitate reuse and interoperability of code. Component-based applications are typically constructed by connecting the required components together in a software framework; creating a complex scientific application could become a matter of assembling components to express the "physics" of the problem and coupling them with numerical solvers and other components to form the complete software package. As component-based programming for scientific computing develops, we can anticipate that many of the components required for a given application would already have been created by experts in the relevant domains and made available through a component repository.

The Object Management Group's CORBA [19], Microsoft's COM and DCOM [29], and Sun's Enterprise JavaBeans [22] are examples of very popular component environments in the business and internet areas. Visualization systems such as Advanced Visualization System's AVS [31], OpenDX (derived from IBM's Data Explorer) [6], and VTK [10] also have a component flavor to them, with the connections between components typically representing data flow. Unfortunately, these environments do not address the requirements of high-performance scientific computing in various ways and have seen very limited use in scientific computing. Efforts by computational scientists to develop component environments [9, 13–16, 20, 26–28] have been mostly focused on specific problem domains, and tend to lack the generality and flexibility needed for use by a much broader user base.

In response to this situation, a grassroots effort was launched by researchers in several U.S. national laboratories and universities to create a component environment suited to the general needs of high-performance scientific computing. The resulting Common Component Architecture (CCA) [2] is now at the prototype stage and is being adopted by a wide variety of scientific computing projects.

2 An Overview of the Common Component Architecture

In the design of the CCA, a number of requirements were considered:

- **Performance:** It should impose a negligible performance penalty.
- **Portability:** It should support languages and platforms of interest in scientific computing.
- **Flexibility:** It should support a broad range of parallel programming paradigms, including SPMD, multithreaded, and distributed models.
- **Integration:** It should impose minimal requirements for existing software to be able to participate in the component environment.

The specification [1] developed by the CCA Forum defines:

- minimal required behavior for a CCA component,
- minimal required behavior for a CCA framework, and
- the interface between components and frameworks,

in such a way as to allow the above requirements to be satisfied, but, to the extent possible, without dictating specific solutions.

At the heart of the CCA is the concept of *ports*, through which components interact with each other and with the framework. Ports are merely interfaces that are completely separate from all implementation issues. They correspond to *interfaces* in Java, or *abstract virtual classes* in C++. CCA ports follow a uses-provides design pattern, so that each component must declare what ports it *uses* from others, and the ports for which it *provides* an implementation. This

typically occurs in the component's `setServices` method, which is invoked by the framework when the component is instantiated. `setServices` is the only method a CCA component is specifically required to implement.

A CCA framework is primarily a container for components being assembled into an application, which mediates the interconnection of ports. Its primary interaction with the component is through the `Services` interface, which allows components to register (used or provided) ports, and to get those ports for actual use. A design goal for the framework is to be able to cast as components even functionality that might be thought of as fundamental framework services. The details of such services are still evolving to some extent, but include things like event services, and "builder services," which provide the capabilities to load/unload components and to connect/disconnect ports. Currently both command-line and graphical means are provided to allow the user to assemble CCA applications.

During execution, when one component needs to use methods provided by a port on another component, it uses the `getPort` method of the `Services` interface to obtain a reference to the port, which can in turn be used to invoke the methods provided by the port. The using component calls `releasePort` when the port is no longer needed. The framework, through the `Services` object, mediates the invocation of methods provided by other components and is important in allowing the CCA to provide both high performance for "local" components and remote access in the case of distributed components; this will be explained in more detail below.

Another aspect of the CCA is the desire to make the use of components independent of their implementation language. To achieve this, we have adopted the Babel language interoperability tool [4]. A Scientific Interface Definition Language (SIDL) is used to generate the necessary glue code between languages. SIDL is also used by the CCA Forum to express the interfaces in the CCA standard. Babel currently supports C, C++, Fortran 77 (F77), Python, and client-side Java, with support for server-side Java and Fortran 90 planned.

The CCA specifications do not dictate implementation issues, such as exactly how calls are made from one component to another, but the model has been designed in such a way as to allow the implementation of highly efficient methods, preserving the innate performance of the environment. It is also worth noting that the specification says nothing specifically about parallelism. The basic philosophy in this matter is for the CCA environment to "stay out of the way" of parallelism. Performance matters are described in more detail in the next section.

3 Performance Considerations in the CCA

3.1 The CCA Framework

The CCA's uses-provides pattern for ports, in which the framework mediates the use of one component by another, is central to both the flexibility and performance of the model for inter-component calls. When a using component invokes `getPort`, the `Port` object returned by the framework might be a proxy for invocation of methods on a remote component in a distributed computing environment. In this case, it is up to the framework to marshal and unmarshal the arguments and make the remote invocation, and components on either end need not know they exist in a distributed environment. Of course, distributed computing is not usually considered to be a high-performance environment, and the CCA user creating the application is well advised to consider the frequency of use and the volume of data transfer in ports when setting up the application in a distributed environment.

In the case that both components are local, the `getPort` call might return a reference to the actual implementation. This is done, for example, in the prototype Ccaffeine framework [11], which focuses on supporting high-performance parallel CCA applications and is written in C++. Components (in the form of shared object libraries) are loaded into distinct namespaces within a single address space (process). The use of different namespaces ensures that the components cannot interfere with each other, and the framework is the only part of the environment which can "see" all components. Since components are all in a single address space, the framework can easily return a direct reference to the port's implementation from `getPort`. This is referred to as a *direct connection* environment, and allows one component to call methods on another with a cost equivalent to a C++ virtual function call — essentially, a lookup of the method in the component's function table, followed by invocation of that function.

Since `getPort` calls occur infrequently (they are required only once per used port), their cost is negligible. The overhead of the CCA framework is almost entirely due to the cost of inter-component calls relative to the equivalent calls in a native language environment. Since this overhead is on the order of the cost of a native language function call, it will not play a significant role in a great many inter-component calls — most scientific software is designed to put a reasonable amount of work in each function — nor will it effect *intra*-component calls. For those rare cases where the overhead imposed by the CCA framework is an unavoidable concern, we characterize the costs below.

3.2 Language Interoperability via Babel

With the use of Babel for language interoperability, as is now being introduced into the CCA environment, some additional overhead is introduced. Babel uses a C-based internal object representation (IOR) to provide the glue between different languages. In general, the overhead is roughly two subroutine calls. The client calls a stub routine that translates the arguments into C. The stub routine calls the skeleton routine which translates the arguments into the implementation language, and the skeleton calls the implementation. In some cases there is an additional overhead due to data conversions between languages (especially with character strings), and with existing code, the developer might need to insert an additional layer to adapt from the object-based representation used by Babel to the style of the existing code. As might be expected, when reasonable amounts of computation take place in the methods called via Babel, the overhead of the Babel system is not noticeable [21]. Obviously somewhat more care is required in using Babel with methods that might be called a large number of times and involve little work.

Babel is distinctive from other language interoperability tools because it provides bi-directional function calls. For example, a Python program can call a F77 subroutine that calls a C++ function that calls something implemented in Python. This flexibility comes at a cost. The software developer must write a SIDL file to describe the interfaces that will be accessible from multiple languages. There is also a runtime overhead that will be quantified below.

Babel was designed with the CCA in mind, but it can also be used without the CCA framework. Babel can be used to wrap legacy applications and libraries to provide a high-level, language-independent interface. The code wrapped by Babel can use native function calls in whatever the implementation language happens to be. When Babel is integrated with a CCA framework, the overhead of the "full" CCA environment is equivalent to the overhead of Babel — the framework's virtual function call is simply carried out in the Babel environment.

3.3 Parallelism

The final performance issue, and perhaps the most important for modern scientific computing, is that of parallelism. As noted, the CCA's primary approach to parallelism is staying out of the way of the parallelism built into the components. In a parallel environment, the CCA framework mediates interactions between components in the same process, just as it does in the sequential case. Interactions among parallel instances of a component in different processes (referred to as a *cohort*) are up to the developer of that component. Components may use whatever parallel communication environment they prefer (i.e., MPI [5, 18, 30], PVM [7, 17], Global Arrays [12, 23, 24], shared memory), and different components may even use different systems. The framework itself essentially does not know it is running in parallel, apart from the need in some cases to initialize the communication system — a dependence we plan to shift into a separate component shortly. Because the framework is, in effect, embarrassingly parallel, we will not concern ourselves with scalability measurements in this paper.

4 Performance Measurement Techniques

To characterize the performance overheads in CCA-based environments, we measured a variety of simple subroutine and function calls in native C, C++, and F77, in the C++-based Ccaffeine CCA framework, the C++-based omniORB CORBA environment, and in nine language combinations using Babel (C, C++, and F77 as calling language and called language). The functions were intended to illustrate the cost of calls passing a single variable of various data types. The complete list of functions and the environments in which they were tested is shown in Table 1. The functions are divided into several groups to simplify presentation and analysis of the results:

A: those for which Babel types map directly to native language types,

B: additional "simple" functions which show significantly different costs from group A in certain languages,

C: those requiring some measure of adaptation between languages in Babel and therefore show greater variation in cost, and

D: remaining functions, mostly those which are relevant only to object-oriented environments, such as Babel and (in some cases) C++.

Some of the function or arguments require a brief explanation:

Table 1: Measurements of function call overheads were obtained with a variety of arguments, corresponding to basic datatypes of the languages as well as additional special types introduced by Babel. This table details the languages and environments (Native, Babel, Ccaffeine, omniORB) in which each type was tested. Group designations are used to simplify the analysis.

		C		C++				F77	
Group	Function/Argument	N	B	N	B	C	O	N	B
A	Double	Y	Y	Y	Y	Y	Y	Y	Y
	Float	Y	Y	Y	Y	Y	Y	Y	Y
	Int	Y	Y	Y	Y	Y	Y	Y	Y
	Long	Y	Y	Y	Y	Y	Y	Y	Y
B	no arguments	Y	Y	Y	Y	Y	Y	Y	Y
	no args., returns double	Y	Y	Y	Y	Y	Y	Y	Y
C	Array	Y	Y	Y	Y	Y	Y	Y	Y
	Bool	Y	Y	Y	Y	Y	Y	Y	Y
	Complex (by reference)	Y		Y		Y		Y	
	Complex (by value)	Y	Y	Y	Y	Y	Y		Y
	Double Complex (by reference)	Y	Y	Y	Y	Y		Y	Y
	Double Complex (by value)	Y	Y	Y	Y	Y	Y		Y
	OrderedArray	Y	Y	Y	Y	Y	Y	Y	Y
	String (by reference)		Y	Y	Y	Y	Y		Y
	String (by value)	Y	Y	Y	Y	Y	Y	Y	Y
D	Char		Y		Y		Y		Y
	Interface		Y	Y	Y	Y	Y		Y
	no args. (static call)		Y	Y	Y				Y
	Double (static call)		Y		Y				Y
	createReference/deleteReference		Y		Y		Y		Y

- Array and OrderedArray: Babel's array object allows arrays to be declared specifically as row major or column major ("ordered array") or to be defined implicitly by the strides through memory. Ordered arrays require additional checking as they are passed, and may require translation from the ordering in which they are presented into the ordering requested by the callee. In our tests, no translation was required.
- Static calls: Tests the difference between invoking functions in their static forms, i.e., `Class::function()`, rather than via an object pointer, i.e., `object_ptr->function()`.
- createReference/deleteReference: Tests the cost of Babel's reference counting mechanisms.

In the case of native C++ and Babel, both concrete and virtual function calls were tested.

Because the total duration of an empty function call (where the function merely returns, doing no work) is so short, our approach was to measure the cost of repeatedly calling the function within a loop relative to the cost of a matching empty loop. We used a range of iteration counts (1,000 to 8,192,000 by factors of 2) to ensure that our measurements scaled appropriately, and at each iteration count we took the minimum time from ten consecutive trials. While efforts were made to minimize the interference with these timing runs by running in single-user mode with a minimum of operating system services active, some interference is inevitable. The per-call overheads we report represent an average over the 14 different iteration counts and we estimate that they are generally reliable to $\pm 10\%$. Because the overhead of any specific application function call will depend on the function's arguments, and our primary interest in these timings is the costs of the CCA environment *relative to* the native language costs, we do not consider the observed variability to be a serious issue.

Measurements were carried out on a 500 MHz Pentium III (Coppermine) Dell Latitude CSx laptop running Debian's "unstable" GNU/Linux distribution and 2.4.18 Linux kernel. Version 2.95.4-15 of the GNU compiler toolchain was used, along with Ccaffeine version 0.3, Babel version 0.7.1 (a prerelease of 0.7.2), and omniORB version 3.0.4. The `-O2` flag was used to optimize all compiled code. The `gettimeofday` system function was used for the timing. This function returns wall clock time rather than CPU utilization, as `getrusage` does, but tests showed that despite

Table 2: Actual timings for F77 function calls and relative costs for other environments. Results represent the average across the group or the range (minimum–maximum) where there is significant variation ($>\sim 10\%$) within the group.

Function Group	F77 Time (ns)	C Rel. F77	C++ Rel. F77	Babel C to C Rel. F77	Ccaffeine Rel. F77	OmniORB Rel. F77
A	18	1.0	1.2	2.6	2.4	91.1
B	10–16	1.0–2.2	2.4–3.8	3.2–3.9	3.5	130.8
C	18	1.1	1.1–3.7	2.1–14.4	2.2–4.3	90.8
Overall Average	17	1.1	1.8	3.8	2.8	97.6

Table 3: Timings for Babel interlanguage function calls, relative to the Babel C to C and native F77 timings, according to the function groupings in Table 1. Results represent the average across the group or the range (minimum–maximum) where there is significant variation ($>\sim 10\%$) within the group.

Calling	Called	Timing Rel. Babel C to C					Timing Rel. F77			
Lang.	Lang.	A	B	C	D	Avg.	A	B	C	Avg.
C	C	1.0	1.0	1.0	1.0	1.0	2.6	3.2–3.9	2.1–14.4	3.8
C++	C	1.3	1.3–1.5	1.5–45.3	1.3–7.2	4.9	3.5	4.0–5.8	4.0–21.6	6.3
F77	C	1.0	1.1	0.94–33.6	0.91–1.1	4.3	2.7	3.4–4.1	2.5–41.2	7.3
C	C++	1.5	1.5	1.7–41.1	1.5–13.7	6.9	3.9	4.7–5.8	4.6–57.5	12.2
C++	C++	1.9	1.8	2.2–84.3	1.9–20.4	10.8	4.9	5.4–7.6	6.2–56.5	14.5
F77	C++	1.6	1.5–1.8	1.7–70.9	1.8–14.5	10.0	4.1	5.9	5.2–91.8	15.3
C	F77	1.6	1.7	1.0–90.1	1.8	8.9	4.1	6.1	3.9–61.1	10.3
C++	F77	1.8	2.2	1.5–132	2.2–8.1	12.8	4.9	7.1–8.3	5.8–65.6	13.3
F77	F77	1.7	1.6–2.1	1.0–121	1.8–2.2	12.2	4.4	6.5	4.4–103	14.2

reporting times down to the microsecond, the Linux implementation of `getrusage` had a resolution of only 10 ms, whereas testing indicated `gettimeofday` provides 2 μs resolution.

5 Results and Discussion

5.1 Native Language Results

Table 2 displays the per-call function costs for calls in the C, C++, and F77 native language environments. Within a language, variation among the single-argument functions is generally small. In C, we represented complex values by structures, and when passed by value, these cost roughly 1.2$\times$ most of the other C or F77 function calls. Also, the C function with no arguments that returned a double cost 2.2$\times$ the Group A result. In C++, the function calls with no arguments (with and without a return value), and those with boolean and string arguments, were relatively expensive, from 2.4$\times$–3.8$\times$ the corresponding F77 timings. The C++ results shown are for concrete function calls; virtual calls are uniformly twice as expensive, and because of its implementation, are represented by the Ccaffeine results.

5.2 Babel

Table 3 shows the costs of various interlanguage calls within the Babel environment. Because there are a number of function calls possible in the Babel environment that are not possible in the native environment, we present results relative to both native F77 and the Babel C to C timings. All results are for concrete function calls. Virtual function calls in Babel have essentially the same cost, 1.02$\times$ the concrete call, averaged over all functions and all language combinations. Given the significant variations seen in some of the timings, the overall averages presented can be considered only as a very rough guide for comparisons — it is important to consider both the languages involved and the function arguments when comparing Babel results.

Table 4: Costs for individual Group C functions (see Table 1) for various language combinations in the Babel environment. Costs are relative to the Babel C to C results. For each column, the top label is the *calling* language and the bottom label is the *called* language.

	C	C++	F77	C	C++	F77	C	C++	F77
	C	C	C	C++	C++	C++	F77	F77	F77
Argument	Time (ns)	Time Rel. Babel C to C							
Array	44.3	3.8	1.0	8.7	11.4	8.7	1.6	4.6	1.8
Bool	43.7	2.5	1.9	2.2	3.3	2.7	2.4	3.5	3.0
Complex (by value)	49.8	1.5	1.2	1.7	2.3	1.9	1.6	4.0	1.5
Double Complex (by reference)	45.0	3.0	1.1	2.3	5.8	3.1	1.6	4.0	1.5
Double Complex (by value)	57.3	2.2	0.94	1.9	3.5	1.7	1.6	3.3	1.5
OrderedArray	255	1.5	1.0	1.7	2.2	1.7	1.0	1.5	1.0
String (by reference)	43.8	45.3	33.6	41.1	84.3	70.9	90.1	132	121
String (by value)	39.0	1.9	19.5	27.3	26.8	43.5	29.0	31.5	49.2

We can see that calls involving C tend to be the least expensive. This is not surprising, given Babel's interal object representation is implemented in C. Calls involving C++ tend to be the most expensive, because Babel tries to provide arguments as close as possible to the native language form, and C++ requires the most adaptation. We see that Group A and B functions are generally fairly consistent in cost across the various language combinations, at most 2.2× the C to C cost. Groups C and D show rather large variations in timing and are largely responsibly for driving up the overall average figures. In analyzing the relative costs of functions in Group C especially, it is important to consider them individually — the overall averages, or even the Group C ranges given, can be no more than a very rough guide.

Table 4 shows details of the costs of Group C functions. The most striking feature of these results is tremendous variation in the cost of passing strings, either by value or by reference. This is because in most cases, Babel must allocate new space (via `malloc`) and copy the string as part of adapting it from one language to the other. Most other functions show trends much more in line with the results for Groups A and B, though there are certain cases where the required adaptations are somewhat more expensive.

It is also worth noting that, thus far, development of Babel has focused almost entirely on correctness of the implementation and on expanding the base of languages supported — little effort has gone into optimization. Therefore, we can anticipate improvements in some of these results.

5.3 Native Languages, CCA, and CORBA

In addition to the native language results previously discussed, Table 2 shows the cost of various function calls in the Ccaffeine CCA framework, the Babel environment calling from C to C, and the omniORB CORBA framework relative to the native F77 timings.

As previously noted, an inter-component function call in a direct connect CCA framework, such as Ccaffeine, is equivalent to a C++ virtual function call. The cost is roughly 2.8× the cost of a native F77 function call.

Taking full advantage of the CCA environment — using Babel integrated into a CCA Framework — calls would incur the cost of a virtual function call in the Babel environment, which is practically identical to the cost of a concrete function call.

To gauge the cost of the CCA environment relative to a typical CORBA environment, we also present timings for same-process calls using omniORB. Timings were quite consistent within each group, and the overall average is that the CORBA calls take 97.6× a native F77 call. This is 34.9× the cost of calls in the Ccaffeine CCA framework, and roughly 25.7× the cost of the stand-alone Babel or full CCA (Babel integrated into a framework) environments.

As discussed earlier, these overheads will noticeably effect only the small fraction of functions which are called many times and contain very little work. These results can be used by software architects and component developers to help gauge which functions, if any, are likely to require special consideration in the design of their interfaces. A variety of options are available, depending on the specific situation. For example, if performance is more important than language interoperability, it may be desirable to eliminate the Babel layer for selected component interfaces. If there is flexibility in the overall architecture, it might be modified to make the sensitive function calls intra-component rather than inter-component, thus eliminating the framework overhead. It is worth noting that CORBA does not

provide this kind of flexibility to developers.

6 The Future of the CCA

The CCA is currently at the stage of a highly functional prototype environment. The specification is nearly complete and is more than adequate to enable serious scientific simulations to be developed. A number of prototype frameworks exist, each focusing on different environments (i.e., parallel, distributed, etc.). A variety of mini-applications have been demonstrated using these frameworks, abstracted from real scientific simulations [25], and more than 15 groups have already adopted the CCA as the basis for new terascale scientific simulations which are now under development.

Development of the CCA continues and is accelerating, thanks especially to the formation of the Center for Component Technology for Terascale Simulation Software (CCTTSS) [3] with funding from the U. S. Dept. of Energy's Scientific Discovery through Advanced Computing (SciDAC) initiative [8]. The CCTTSS, a subset of the CCA Forum that includes participants from six national laboratories and two universities, carries out research in component technology for high-performance computing and will develop the CCA into a full production-quality environment.

One focus area of the Center is the development of a suite of components based on popular numerical and other libraries in order to "seed" the development of a component-rich environment. Related to this is the the development of domain-specific "standard" interfaces to facilitate the creation of reusable and interoperable components. As such activities are best undertaken by experts in the relevant domain, the role of CCTTSS and the CCA Forum is primarily to encourage and promote the formation of communities around such efforts. Efforts are already underway to develop interfaces for basic scientific data objects, such as distributed arrays, structured and unstructured meshes, and adaptive mesh refinement.

Another focus of the CCTTSS, and one with more performance considerations, is the development of general interfaces and tools for parallel data redistribution, especially for the case of coupling parallel models running on differing numbers of processors. While the initial implementation of this capability will be based on components, there is longer-term interest in doing this at the framework level through more expressive interfaces able to capture the desired parallel semantics, leading to what might be termed *parallel remote method invocation*.

7 Conclusions

The CCA provides a means for developers to manage the complexity of large-scale scientific software systems, and to move toward a "plug and play" environment for high-performance computing. The CCA model allows for a *direct connection* between components within the same process, maintaining performance on inter-component calls. It is neutral with respect to parallelism, allowing components to use whatever means they desire to communicate within their parallel *cohort*. The current prototype CCA environment is being used to create serious scientific simulations, and is also being refined toward production quality. Performance concerns will continue to be central to the development of the CCA.

8 Acknowledgments

The CCA has been under development since 1998 by the CCA Forum and represents the contributions of many people, all of whom are gratefully acknowledged.

This work has been supported by the U. S. Dept. of Energy's Scientific Discovery through Advanced Computing initiative, through the Center for Component Technology for Terascale Simulation Software, of which LLNL and ORNL are members.

References

[1] CCA Documents. `http://www.cca-forum.org/documents/`.

[2] CCA Forum home page. `http://www.cca-forum.org`.

[3] Center for Component Technology for Terascale Simulation Software (CCTTSS) home page. http://www.cca-forum.org/ccttss/.

[4] Components @ LLNL: Babel. http://www.llnl.gov/CASC/components/babel.html.

[5] Message Passing Interface (MPI) Forum home page. http://www.mpi-forum.org.

[6] Open Visualization Data Explorer. http://www.opendx.org.

[7] PVM: Parallel Virtual Machine. http://www.csm.ornl.gov/pvm/.

[8] Scientific Discovery through Advanced Computing (SciDAC) home page. http://www.science.doe.gov/scidac/.

[9] Sierra home page. http://www.cfd.sandia.gov/sierra.html.

[10] VTK home page. http://public.kitware.com/VTK/.

[11] Benjamin A. Allan, Robert C. Armstrong, Alicia P. Wolfe, Jaideep Ray, David E. Bernholdt, and James A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, in press.

[12] Global Array Toolkit home page. http://www.emsl.pnl.gov:2080/docs/global/.

[13] 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.

[14] David L. Brown, Geoffrey S. Chesshire, William D. Henshaw, and Daniel J. Quinlan. OVERTURE: An object-oriented software systems for solving partial differential equations in serial and parallel environments. In Michael Heath, Virginia Torczon, Greg Astfalk, Petter E. Bjørstad, Alan H. Karp, Charles H. Koebel, Vipin Kumar, Robert F. Lucas, Layne T. Watson, and David E. Womble, editors, *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, 1997. (Published only on CD-ROM).

[15] Edmond Chow, Andrew J. Cleary, and Robert D. Falgout. Design of the HYPRE preconditioner library. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 106–116. Society for Industrial and Applied Mathematics, 1999.

[16] John de St. Germain, Steve Parker, John McCorquodale, and Chris Johnson. Uintah: A massively parallel problem solving environment. In *9th IEEE international Symposium on High Performance Distributed Computing (HPDC-9, 2000)*, pages 33–42. IEEE Computer Society, 2000.

[17] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, November 1994.

[18] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference*, volume 2 – The MPI-2 Extensions. MIT Press, September 1998.

[19] Object Management Group. OMG's CORBA website. http://www.corba.org.

[20] S. Karamesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. Williams. Array design and expression evaluation in POOMA II. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, number 1505 in Lecture Notes in Computer Science. Springer, 1998.

[21] Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, 2001. Also available at http://www.llnl.gov/CASC/components/publications.html.

[22] V. Matena, M. Hapner, and B. Stearns. *Applying Enterprive JavaBeans: Component-Based Development for the J2EE Platform*. The Java Series. Addison-Wesley, 2000.

[23] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing'94*, pages 340–349, Los Alamitos, California, USA, 1994. Institute of Electrical and Electronics Engineers and Association for Computing Machinery, IEEE Computer Society Press.

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

[25] Boyana Norris, Satish Balay, Steve Benson, Lori Freitag, Paul Hovland, Lois McInnes, and Barry Smith. Parallel components for PDEs and optimization: Some issues and experiences. Technical Report ANL/MCS-P932-0202, Argonne National Laboratory, February 2002. Available via `http://www.mcs.anl.gov/cca/papers/p932.pdf`; under review as an invited paper in a special issue of *Parallel Computing* on Advanced Programming Environments for Parallel and Distributed Computing.

[26] Manish Parashar and James C. Browne. Systems engineering for high performance computing software: The HDDA/DAGH infrastructure for implementations of parallel structured adaptive mesh refinement. In S. B. Baden, N. P. Chrisochoides, D. B. Gannon, and M. L. Norman, editors, *Structured Adaptive Mesh Refinement (SAMR) Grid Methods*, volume 117 of *The IMA Volumes in Mathematics and its Applications*, pages 1–18. Springer, 2000.

[27] Manish Parasher, James C. Browne, Carter Edwards, and Kenneth Klimkowski. A common data management infrastructure for adaptive algorithms for PDE solutions. In *SC97 Conference Proceedings*. Association for Computer Machinery and IEEE Computer Society, November 1997. `http://www.supercomp.org/sc97/proceedings/`.

[28] John V. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Kataryzna Keahey, M. Srikant, and MaryDell Tholburn. POOMA: A framework for scientific simulations on parallel architectures. In Gregory Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 553–594. MIT Press, 1996.

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

[30] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*, volume 1 – The MPI Core. MIT Press, 2nd edition, September 1998.

[31] Advanced Visualization Systems. `http://www.avs.com`.