

High End Computing Component Technology

A Response to the HECRTF Call for White Papers

May 2003

Rob Armstrong

Distributed Systems Research
Sandia National Laboratory
7011 East Avenue
Livermore, CA 94551
(925) 294-2470
rob@ca.sandia.gov

David E. Bernholdt

Computer Science and Math Division
Oak Ridge National Laboratory
P. O. Box 2008, MS 6367
Oak Ridge, TN 37831-6367
(865) 574-3147
bernholdtde@ornl.gov

Tamara Dahlgren

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P. O. Box 808, L-365
Livermore, CA 94551
(925) 423-2685
dahlgren1@llnl.gov

Wael R. Elwasif

Computer Science and Math Division
Oak Ridge National Laboratory
P. O. Box 2008, MS 6367
Oak Ridge, TN 37831-6367
(865) 241-0002
elwasifwr@ornl.gov

Gary Kumfert

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P. O. Box 808, L-365
Livermore, CA 94551
(925) 424-2580
kumfert@llnl.gov

Lois Curfman McInnes

Math and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439-4844
(630) 252-5170
mcinnes@mcs.anl.gov

Jarek Nieplocha

Computational Sciences & Mathematics
Pacific Northwest National Laboratory
Battelle Blvd, MSIN: K1-85
Richland, WA 99352
(509) 372-4469
jarek.nieplocha@pnl.gov

Boyana Norris

Math and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439-4844
(630) 252-7908
norris@mcs.anl.gov

Abstract

Component technology promises to reduce “time to solution” for end-users of High End Computing (HEC) systems by reducing their software burden. Components also make the next level of software complexity tractable, improving software reuse, vertical integration, specialization, and ultimately return on investment. While these capabilities have already been demonstrated in the commercial software industry, they have been slower to affect the specialized segment of HEC systems because of unique performance constraints, massive parallelism, and widely varying — often one of a kind — architectures. We thus propose new directions of research to address outstanding issues in HEC components and their use in large-scale scientific simulations.

1 Motivation for Components in HEC

In the commercial software industry, the increasing complexity of contemporary software development has demanded a new mechanism that scales across people, geography, and time to achieve economics of scale. Component technology is the solution to that demand. It is component technology that enables MS Word™ documents to appear in MS Powerpoint™ slides and has led to the point-and-click graphical user interfaces that inhabit most desktops today.

High End Computing (HEC) software is also continuously increasing in complexity and in need of technology that scales across people, geography, and time, but HEC cannot effectively use component technologies provided by the commercial sector. HEC demands for performance and massive parallelism simply exceed the scope of the commercial market. Similar to data storage, networking, and visualization, component technology is a special case that requires HEC-specific solutions. Component models for parallelism, parallel data model coupling, and remote method invocation between asynchronous parallel components are concepts wholly absent from commercial offerings.

Components help to manage complexity. For end-users in HEC, components will reduce the burden of software details, freeing up more energy to focus on the application science. Components are especially well suited for the inherent complexity of exchanging and combining software from multiple sources, thereby increasing accessibility of software. Finally, components afford developers a whole new level of sophistication, opening possibilities for future capabilities that would have been intractable otherwise.

As members of the Common Component Architecture Forum (CCA Forum)¹ and researchers in the Center for Component Technology for Terascale Simulation Software (CCTSS)², we have implemented multiple component frameworks, dozens of components, and many full components-based scientific applications; investigated issues of automation, performance, and robustness; and paid careful attention to the issue of supporting legacy codes in the component paradigm. Yet there is so much more to do. Components are no less than the next evolutionary step in software technology, beyond object-oriented programming which itself superseded structured programming.

¹A grassroots organization dedicated to the benefits of component technology to bear in the simulation science, see www.cca-forum.org.

²An Integrated Software Infrastructure Center funded by the DOE Office of Science SciDAC Program, see www.osti.gov/scidac.

We recommend components to the High End Computing Revitalization Task Force (HECRTF) as a strategic technology that improves the capability and accessibility of software while minimizing “time to solution” for end-users. In Section 2 we discuss the particular challenge of creating and sustaining a HEC component economy, from a policy standpoint as well as Research and Development. Opportunities to migrate legacy software to the component paradigm are discussed in Section 3. Promising new capabilities afforded by components for automated composition and tuning of applications are presented in Section 4. In an even farther reaching vision, we highlight in Section 5 how Interface Definition Languages, which even commercial components use, can be improved by encoding not just calling interfaces but semantic context. In Section 6, we discuss how the emerging high-performance scientific component programming environment challenges the Single Program Multiple Data (SPMD) model and its underlying assumption of a static, homogeneous environment.

2 Creating and Sustaining a HEC Component Economy

Component technology allows the productive exchange of software at a finer granularity, greater frequency, and broader range than traditional software libraries. Widely-used components will have extended lifetimes, usually longer than the project or application for which they were originally developed. These are some of the important long-term benefits of component technology, but they also highlight a number of challenges to the traditional “economy” for HEC software.

The component approach addresses technical barriers to effective software reuse, but might aggravate the non-technical barriers. Components are generally harder to create than regular software; their benefit does not come with the first use, but repeated reuse. Such repeated reuse implies sustained maintenance. Unfortunately, obtaining funding specifically for software maintenance is problematic. Emphasis on “software development cost” over “total cost of ownership,” or “sustained value to the community” has the unfortunate side-effect of providing a financial disincentive to effective software reuse.

Strategies for positive reinforcement of sharing software can readily be borrowed from the Open Source Software (OSS) movement. Scientists can be rewarded by publishing components that are featured in important applications. Scientists’ reputations can be built on the usability and fidelity of their components. Although nothing precludes other approaches to handling the intellectual property embodied in a component, the adoption of open source licenses and approaches would facilitate precisely the kind of reuse and longevity that components are intended to provide.

It is important to recognize that not all useful components will gather a sufficiently large community to become self-sustaining. Even if a set of useful components could be self-sustaining eventually, it takes time for communities to form and even then often requires a core band of “benevolent dictators” to direct the enterprise. In the HEC context, the OSS model is vital for achieving the economy of scale in users, but cannot completely negate all long-term maintenance costs for components. In order to create and sustain a component economy, and reap its financial rewards, funding agencies will most likely need to “prime the pump,” and make on-going investments in targeted cases.

The central marketplace for this economy will be component repositories. A cross between SourceForge.org (where software is developed and maintained), Amazon.com (where merchan-

dise is distributed and consumers publish their recommendations), and an Object Request Broker (which actually launches components); these repositories will serve as a virtual center of exchange of ideas, technology, and software. In the context of HEC, special considerations such as licensing, IP, export controls, etc. should also be taken into account. This is an important area of research, as it will be the main portal into the HEC component world. Component repositories could also be used as resources to develop metrics for “total cost of ownership” for software, or “sustained value to the community.”

3 Migrating Legacy Software to Components

HEC components must provide a migration path for legacy software, or they will fail under the sheer inertia of the existing code base. We identify two techniques to address this issue, one well established, and the other an area of active research.

The current technique exploits the fact that components are programming language neutral entities. Whenever two components are connected, each has no knowledge of the programming language used for the other component. This also means that legacy codes can be componentized in their native programming language, without requiring a rewrite in a different language. It does often require wrappers be partially written by hand, which can vary from simple connections, to extensive code analysis, depending on the quality and understanding of the legacy code’s design and organization.

A second technique that deserves significant attention is automated componentization. This technique actually parses existing source code and generates an intermediate form. Then, through a series of transformations, it generates output source code that effectively wraps the legacy code in a component wrapper. Reason would dictate that this automated solution would lack the cleaner design of hand-wrapped legacy code, but as a practical matter some codes are simply too large and arcane to handle any way but in an automated fashion.

Source-to-source transformation tools open up other interesting possibilities, such as ensuring that specifications are kept consistent with evolving implementations, or generating components that are semantically different from the original implementation. For example, automatic differentiation could be used to generate new components that compute the derivatives of a component implementing a problem-dependent function evaluation. Such components would then enable the use of sophisticated algorithms that require derivatives, without requiring low-accuracy approximations or the burden of hand coding.

4 Automated Composition and Tuning of Applications

Software components are a plug-and-play technology; their very name comes from the fundamental concept of being composable. Component technology is also exceptionally well suited to supporting domain-specific interface standardization efforts. Using components, the interfaces can be defined in a platform and language neutral manner. Differing projects can then write their software to implement that standard in the language of their choosing. Then, customers of the standard can then write their application to the standard and arbitrarily swap between various implementations at runtime, without a single line of source code modification.

With the possibility of hundreds, even thousands, of components executing on a peta-scale platform, research is needed in tools to help automate selection and composition. A typical strategy to mitigate the growing complexity of software is to compose pieces hierarchically into growing levels of complexity, and expose the top of the hierarchy to users. This works well when the developer properly anticipates how the hierarchy will be used, but when third party modifications of middle or lower levels is attempted, mileage will vary. Richer interfaces presented by components can make automated composition a new and intriguing alternative to static hierarchies. Tools could automate the construction of a hierarchy on demand, responding to changes in inherited and synthesized attributes of individual components in the hierarchy. For example, if one replaces an existing component in the middle of a hierarchy, it may necessitate additional adjustments downstream in the calculation, but it may also impose new requirements that must be anticipated upstream.

By varying composition rules and further augmenting the interfaces, new techniques for automated tuning, adaptive algorithms, and fault-tolerance become available. Automated composition can utilize available information about performance requirements and capabilities of individual components as well as information on the underlying execution architecture to tune component compositions. Even if an application is automatically generated a near-optimal initial composition of components, it is unlikely that the same composition will remain optimal in terms of achieving the best time to solution throughout a long-running application's execution. In addition to changing application requirements, large-scale simulations would likely be subject to hardware or software failures. One cannot rely entirely on the fault-tolerance of hardware or low-level libraries to ensure an application completes successfully and efficiently. To address the dynamic application requirements and recover from hardware fluctuation, applications must be adaptable. Component technology provides new opportunity to implement and hide the software complexities so the applications can simply exploit the added capabilities.

Although the plug-and-play capabilities and amenability to supporting standards is a reality today, automated composition and tuning is still open to extensive research. How to scale performance models and monitoring infrastructure to petaflop machine sizes is an open question. Investigating how to effectively augment component interfaces with required information is part of a larger issue in the following section.

5 Specification Technologies

Contemporary component interface specifications are generally limited to the calling interface (also called API); which is insufficient for automated selection, composition, integration, and application of components in scientific computing. In particular, they lack the information needed to convey the purpose, context, performance, and proper usage of components in a machine-friendly format. Currently such information must be gleaned from documentation or code inspection. This places an unrealistic burden on computational scientists to understand the software and ultimately limits reuse.

Specifications of semantic information and their automated discovery will enable a greater variety of tools to truly reduce "time to solution" for component and application programmers. Context information can be used to specify the types of problems a component has been designed to address. Performance annotations have already been discussed in light of composition and tuning. Annotations regarding the proper use of components should indicate how the supplied

software was designed to be employed. For example, a time integration component may have means to advance time when given a floating-point number. By also annotating the specification to indicate that the input argument must be a positive number in microseconds, tools can ensure that the interface is being invoked properly and in the correct context.

In addition to providing explicit documentation, this kind of meta-information has the added benefit of enhancing debugging. Specification technologies can also be used as a basis for tools that compare specifications to their implementations. This can be beneficial for checking that the specifications accurately reflect their implementations.

6 Programming Models beyond SPMD

The current programming methodologies, typically falling into the broad Single- Program Multiple Data (SPMD) category, were primarily designed for static execution environments (e.g., a fixed set of homogeneous processors fully dedicated and tightly coupled to work on a single application). However, even today, applications on large systems experience a substantial variability in their execution environment, which challenges the basic assumptions of the SPMD model. In complex HEC applications, software components themselves can represent parallel tasks of variable length and can span disjoint sets of processors. We expect that in the future these applications will increasingly be more asynchronous, employ different mechanisms for fault tolerance, use different mechanisms for latency tolerance such as multithreading, and even execute on variable sets of processors, e.g., by employing MPI-2 dynamic process creation.

The efficient use of HEC components in such dynamic environments requires significant inroads into the areas of runtime support and execution environment abstraction. The runtime environment in which such components execute would need to provide better abstractions to shield component developers from the need to manage complex runtime scenarios and configurations. A delicate balance needs to be struck between facilitating ever more dynamic and complex component interactions, and reducing the effect that such a runtime environment would have on application performance.

For example, by hiding the gory details of remote method invocation (RMI), components in industry have been very successful in making distributed programming look like serial programming to the developer. In the context of HEC components, one can generalize to what looks like a SPMD code actually being a collection of SPMD codes communicating via parallel RMI (PRMI). This model has multiple programs, not just a single one, and they can work collaboratively and asynchronously instead of following the traditional synchronous manner of SPMD processor coupling.

7 Closing

Components pose a fundamental shift in HEC software methodology, capabilities, and accessibility. While the reasons to pursue it are legion, the full implications are still not broadly understood throughout the community. Our list of issues and ideas regarding HEC components is only a sampling of new opportunities that come with this emerging technology. Although components are generally useful in many contexts, the specialized regime of high end computing requires specialized R&D to realize the performance and productivity gains that this technology has to offer.