

Simplifying Software Development and Increasing Software Productivity on High End Computers

a White Paper Submitted to the High End Computing Revitalization Task Force by

David E. Bernholdt*
Research Staff Member
Email: bernholdtde@ornl.gov
Phone: 865 574 3147
* Primary contact

Wael R. Elwasif
Research Staff Member
Email: elwasifwr@ornl.gov
Phone: 865 241 0002

Al Geist
Corporate Fellow
Email: gst@ornl.gov
Phone: 865 574 3153

James A. Kohl
Research Staff Member
Email: kohlja@ornl.gov
Phone: 865 574 3143

Stephen L. Scott
Research Staff Member
Email: scottsl@ornl.gov
Phone: 865 574 3144

Torsten Wilde
Research Staff Member
Email: wildet@ornl.gov
Phone: 865 241 5842

**Computer Science and Mathematics Division
Oak Ridge National Laboratory
P. O. Box 2008, MS 6367
Oak Ridge, TN 37831-6367**

Abstract

The complexity of modern scientific software is an important issue that needs to be addressed as part of an effort to revitalize high end computing (HEC). The complexity arises from both the scientific demands and the increasingly complex computers on which the software is run. We believe that HEC software developers need to be able to use a higher level of abstraction to express their computational problems. New programming models and technologies such as domain-specific languages coupled with automatic code generation, source-to-source translation tools, and component models are all sufficiently advanced that a concerted investment aimed at bringing them to HEC computational scientists will bring significant near-term payoffs.

Introduction

Through a combination of hardware and software improvement, the capabilities of high end computing (HEC) in scientific and engineering simulation have skyrocketed in recent years. As this capability has grown, so have our expectations of, and, ultimately, our reliance upon computational science performed on HECs. Indeed, computational simulation is now often described as the third leg of modern science, equal in importance to experiment and theory. However, these remarkable advances have often involved heroic efforts in software development. As the demands on computational science increase, the software driving the simulations becomes more complex due to increases in fidelity and problem size, as well as changes in solution methodology. Additional complexity arises from the need to extract high performance from parallel computer architectures that are increasingly complex and varied.

Although scientific simulation software itself has changed significantly with the rise of HEC, and the software environment in which it is developed has changed in outwardly improved a great deal, at the conceptual level the programming models and environments in which researchers develop software for modern machines are little different than when they were developed in the

early days of parallel computing. Consequently, the software developer is forced to deal directly with the increased hardware and scientific complexity, with obvious impacts on productivity, performance, portability, and scientific capability.

We believe that a program to revitalize HEC should include the development of tools and techniques to provide the software developer with a higher level of abstraction that simplifies software development and increases productivity. The computer science research community has pursued a variety of projects over the years with a direct or secondary focus on raising the level of abstraction in software development. Although few of these efforts have actually impacted the work of computational scientists to date, we believe that the seeds are present, and with the benefit of a focused research effort, relevant capabilities can be made available to computational scientists in the FY 2005-2009 time frame of interest to the Task Force.

Constraints

It is important to recognize that targeting computational scientists in the relatively near term imposes some constraints on the types of computer science efforts likely to be positively received by those actually using high end computers for scientific simulation. In many domains, simulation codes evolve and grow over the course of years, and sometimes decades. The level of effort embodied in these codes often makes it impractical to rewrite them from scratch in a new language or a new paradigm. Therefore, technologies that can accommodate existing code easily, approaches that can be applied incrementally, and those that work with widely used scientific programming languages, such as Fortran, are more likely to gain acceptance. Portability of both software and tools are important, and performance portability is often a significant concern – software developers are generally willing to spend effort tuning the performance of their code for a given platform only in proportion with the generality or portability of the result.

Vision

Raising the level of abstraction for HEC software involves programming models and environments that better support the range of architectures and diversity of implementations in high end computers with portable performance, and software development approaches that help manage overall software complexity.

The complexity of current programming models is easily seen. While extremely popular, traditional message passing paradigms require explicit coordination between sending and receiving processes. Important to HEC, increasing processor counts makes such coordination ever more challenging, and potentially performance-reducing. Shared-memory programming models, often thread-based, are often considered easier to use, but do not easily lend themselves to the development of performance-portable algorithms on distributed shared memory or other platforms. “Multi-level” parallel approaches, i.e., combining OpenMP and MPI are increasingly popular, reflecting the rise of HECs built as symmetric multiprocessors (SMPs) on high-speed interconnects. Experience to date suggests that while such approaches can sometimes achieve good results, they come at a significant cost in terms of code complexity and computational experimentation required to identify the best combinations of processes and threads (which typically cannot be intuited from simple information about the target computer). A higher level of abstraction in this area would provide a simplified programming model capable of expressing the desired parallel algorithms, while automating the “implementation” on a given platform based on performance models for the hardware and software. We believe a programming model

that provides a shared memory abstraction will likely form the basis for the most effective approach; however, it is also important to provide migration paths for codes based on other models, especially message passing.

The increasing levels of parallelism required to maintain efficiency in traditional single-program multiple data programs running on machines with ever-increasing processor counts is also a significant challenge, both in terms of scientific formulations that provide the required degree of parallelism and in terms of the management of so many processes. One way to finesse such issues is to move to an environment which facilitates the simultaneous use of coarse- and fine-grain parallelism by allowing the running parallel job to be partitioned into a number of parallel sub-tasks (of different sizes) running concurrently. Such an environment must be largely or completely automated, and ideally would be capable of making decisions about the optimal partition size for each task based on performance models, as well as inferring computational dependencies among tasks. Another important issue that arises with increasing parallelism and the linking of distributed HEC resources via the Grid is fault tolerance and recovery. Large machines are reaching the point where the time required to boot the system is roughly equal to the mean time between failures. The hardware, operating system, and programming environment must work together to provide users with simple, preferably transparent mechanisms to allow HEC software to tolerate faults.

Although the use of available programming environments with modern HECs frequently leads to complicated programming in order to extract the best possible performance, the nature of the science, the problems, and the methodology needed to solve them also contributed significantly to the complexity of modern HEC software. Raising the level of abstraction in this context entails being able to express the necessary computations at a higher level. For example, through the use of domain-specific high-level languages, or component models which facilitate the assembly of large, complex applications from smaller, more manageable software units. But to be truly effective, such approaches need to be combined with the capability to “reason” about the computational context in order to generate the most efficient code, or to assemble the most efficient set of components for the task and HEC platform at hand. With the use of tools to automate aspects of code generation and application composition also comes the need to be able to validate the results they produce and verify final results of the code. This will necessitate the increased use of formal specification languages and tools, and other approaches, which are currently little used in HEC.

We draw on two analogies to help solidify our vision. First is the modern approach to developing graphical user interface based applications from elementary widgets. Widgets of increasing complexity can be composed into skeleton applications by automatic code generation tools. Developers maintain the ability to alter “tunable” aspects of the widgets as they desire. In HEC, the widget concept could be similarly used to abstract away certain details of the underlying programming environment. The second comes from a recent DARPA High Productivity Computing Systems workshop, where the discussion repeatedly returned to MATLAB as the archetype of a high-level environment in which developers are extremely productive, but in which performance requirements cannot be easily satisfied. What is needed then, is something akin to MATLAB at the top level, but which allows the user to drill down through the tool chain and intervene in the processing or code generation in order to tune the results.

In the following section we briefly describe some of the technologies we see as being important to our vision of raising the level of abstraction for the HEC software developer.

Relevant Technologies

Domain-specific languages and automatic code generation tools provide an obvious means of raising the level of abstraction. Domain-specific languages (DSLs) allow software developers to express programs in a form that is tailored to the scientific domain of interest, typically closer to the way the scientist thinks about the problem than is possible in a traditional programming language. The DSL input can be processed in a variety of ways: interpreted, compiled into object code, or translated into a traditional programming language through the use of automatic code generation tools. Historically, DSLs have been viewed primarily as a convenience for the programmer – reducing the effort required to create software for experimentation, but generally not producing code with performance or capability suitable for production use. However, a number of recent projects have shown that much more is possible. In traditional software development, the developer must make many decisions regarding the implementation that are then fixed in the source code. A well-designed DSL can be viewed as a means of encapsulating the “science” before the implementation decisions are made. Code generation tools can then be designed to include extensive and rigorous optimizations in the process of generating the traditional-language source code in a way that is not possible in the traditional development approach, as well as introducing appropriate code for fault tolerance and recovery and other features. Performance models for both the (generated) code and the target hardware platform can be used as additional input to the optimizations in order to tailor the generated code. Such an approach can be used to address hardware differences, as well as simplifying the task of tailoring algorithms and code to different programming models, even on the same hardware (though clearly this requires a high level of abstraction in the DSL). DSLs coupled with optimizing code generation has the potential to bring tremendous benefits to HEC software development, but much work is needed to develop an infrastructure that allows DSLs and processors to be created quickly and easily.

Source-to-source transformation tools (S2S) share with DSLs and automatic code generation tools many similar capabilities with respect to software development, but also provide a means of working with existing software. S2S tools will facilitate the evolution of code that researchers can’t afford to rewrite from scratch. An important example would be the use of an S2S tool to transform a generically written code into one specialized to a particular programming model or hardware platform. Simple text tools (e.g., the unix sed command) are not sufficient to the task. Rather, tools are required that parse the source language to an abstract syntax tree (AST) and allow user-defined manipulations on the AST prior to writing code back out. As with DSL/automatic generation tools, some initial work has been done in this area, but efforts are needed to broaden the available languages, generalize the capabilities, and make it easier for software developers to express the desired transformations.

S2S tools are obviously closely related to compilers, and can be derived from compiler tool suites. We note in general the need for high-quality, freely available and distributable (i.e., open source) compiler tools to support this and other HEC computer science R&D activities. An important gap in this area is the lack of production-quality **open source compiler suites for Fortran 90/95/2000**, an important language to HEC scientific software. Preferably, such a tool suite should be integrated with or interoperable with compilers for a wide range of other languages.

Programming models, of course, have a tremendous influence on the level of abstraction and complexity of software. Unfortunately, message passing, presently the most widely used parallel programming model, can be likened to assembly language because the programmer must manage nearly every detail of the communication. Shared-memory models offer much greater ease of use, but the memory space is often treated as “flat”, promoting the development of algorithms that work well only on uniform memory systems (of which there are very few in HEC). However, experience has shown that a shared memory programming model which exposes the locality of data and promotes consideration of the non-uniform memory access hierarchy, is very effective in encouraging the development of parallel algorithms which are efficient across both shared- and distributed-memory platforms. Approaches focusing on shared data rather than explicit links between processes also have a significant advantage when incorporating fault tolerance. In message passing and similar approaches, access to the data manipulated by the parallel algorithm is a “second order” operation, which the programmer must translate into explicit instructions for process-to-process communications. Failure of a process requires the programmer to find a new mapping between the data of interest and the processes they have to talk to. In a data-centric model, on the other hand, the programmer indicates directly the data they want to access, paving the way for the underlying environment to transparently relocate the data. Data-centric models are also more amenable to techniques such as redundant storage (for example, using an in-memory RAID-like approach to distributed data structures) to facilitate recovery from a fault. Programming models designed from the start with fault tolerance capabilities will also be more amenable than when it is grafted on to existing environments. Data-centric approaches will also work better in environments that provide workflow/scheduling capabilities to provide increased parallelism, as mentioned in the Vision section.

With mixed-language programming on the rise, is it important that programming models be available across the major HEC languages in an interoperable form. The question of whether programming models are implemented as libraries or as compilers is also relevant. The use of source-to-source transformation tools can help to blur the distinction, and to make compiler (-like) solutions more universally available.

Component models are emerging as an important tool for managing the complexity of large-scale software systems in the business and internet communities. While domain-specific computational frameworks have been used in HEC for some time, only recently have efforts begun to develop the more flexible and extensible component approach for HEC. Components raise the level of abstraction by treating functional units of software as building blocks for large-scale applications. These units (components) are defined by the interface they present while their internal implementation remains opaque. The component approach promotes the creation of reusable, interoperable software with a potential user base much larger than if the code were embedded in a monolithic application. With a large suite of components available for numerical solvers, data management, and other common needs, many researchers would be able to assemble applications from a large proportion of “off the shelf” components, thus increasing productivity. Tools to facilitate selection of components, automatic composition, and performance tuning will be very useful in reducing software complexity.

Finally, most of the technologies cited above involve some degree of automation of the process of code generation or application composition. To have confidence that such tools are doing what they claim, and that the resulting applications will run reliably and produce correct results, **specification languages and verification technologies** will also become increasingly important.