Software Challenges for Extreme Scale Computing: Going from Petascale to Exascale Systems

Michael A. Heroux, Sandia National Laboratories

1. Introduction

Preparing applications for a transition from petascale to exascale systems will require a very large investment in several areas of software research and development. The introduction of manycore nodes, the abundance of parallelism, an increase in system faults (including soft errors) and a complicated, multi-component software environment are some of the most challenging issues we face. In this paper we address four topics we believe to be most the challenging issues and therefore the greatest opportunities for making effective next-generation scalable applications.

2. Parallel Programming Transformation

The first and foremost barrier to optimal use of extreme scale computers is the required transformation of parallel programming strategies. There is mounting evidence that optimal parallel applications for scalable manycore computer systems will rely on MPI for inter-node parallelism, but will need to introduce large-volume functional parallelism and SIMD vectorization. Vectorization is the job of the compiler, with a little help from the programmer via pragmas and directives. The real issue is that presently there is no obvious parallel programming model for implementing the middle layer of parallelism. Current standards such as OpenMP, Pthreads and UPC are not designed for manycore nodes. CUDA, RapidMind and related products target manycore but are proprietary. OpenCL is an emerging standard but is not really a user-oriented interface, and will likely not provide optimal performance (e.g., in comparison to CUDA on GPUs).

However, even without an emerging programming model for manycore, there is a vast amount of work required to prepare existing applications for manycore nodes. Two major tasks are (i) reducing bandwidth requirements as much as possible, primarily by introducing the use of mixed precision, storing data in 32-bit arrays wherever possible, and (ii) rewriting low-level kernels as stateless functions with large enough granularity to keep a SIMD core busy, and small enough that there is a large volume of simultaneous function calls to execute.

Application developers can immediately begin refactoring software in anticipation of manycore nodes, but a manycore programming model will need to emerge in the near future.

3. Beyond the Forward Problem

In many areas of science and engineering, solving a single problem with given input conditions, the *forward problem*, is sufficiently challenging, and higher forward problem fidelity is the highest priority for scalable computing. However, as the fidelity of the forward problem becomes sufficiently good, it becomes possible and imperative to study parameter sensitivities, quantify uncertainties and automatically compute an optimal solution over a range of parameter values.

All of these advanced modeling and simulation techniques quickly increase problem size and parallelism—often by orders of magnitude—and large problems can easily exceed the computing capacity of our largest systems. The simplest of these approaches are "black box" in nature and do not require a true peta/exascale system (instead requiring a cluster of tera/petascale systems). However, more advanced methods (often called embedded methods) rely on a tightly coupled aggregation of forward problems and require a true peta/exascale system. The challenge with embedded methods is that they require the transformation of an application into a "subroutine" because embedded methods need to call the forward solve as a function. Most applications were not designed with this mindset, so this transformation will be challenging.

4. A Fault-resilient Application Environment

If hardware fault predictions are accurate, exascale systems will have very high fault rates and will in fact be in a constant state of decay. "All nodes up and running," our current sense of a well-functioning scalable system, will not be feasible. Instead we will always have a portion of the machine that is dead, a portion that is dying and perhaps producing faulty results, another that is coming back to life and a final, hopefully large, portion that is computing fast and accurate results.

Our current hardware and software environments are not well prepared for this kind of "stable" system. In fact, the only reliable, portable resilience mechanism we have is checkpoint-restart. Although there have been many research efforts in fault tolerance, much of this work has been focused on a single layer in the hardware and software stack, without sufficient consideration of the whole set of requirements. One of the biggest needs we have in resilient computing research is an increased effort to include the full vertical scope of the software and hardware stack into our design discussions. Furthermore we need a full-featured environment to probe the system, make decisions based on system state and recover from system faults, both hard and soft. Without a dramatic improvement in this environment, we face the very real risk that application developers will reject exascale systems in favor of smaller, more reliable systems that provide a better overall throughput.

Regardless of how unreliable a system is, from an application developer's perspective there has to be some way to perform reliable computations. This does not mean that every computation must be reliable, but that certain, perhaps higher cost, computations and their input and resulting data are highly reliable. Without

this kind of capability, it becomes extremely difficult to provide any kind of verifiable result. An application needs the ability to declare certain ranges of data as highly reliable. Furthermore, it needs to know that certain computations have completed correctly or, if not, have the ability to react to faulty or interrupted computations. If the runtime environment can provide these two features, we can develop algorithms that will be reliable on exascale systems.

5. Hierarchical Software Engineering and Development

The CSE software community, by most accounts, has been slow to adopt formal software engineering practices. Although a lot of high quality software has been developed without formal practices, the demands of collaborative development, multi-code environments and large collective teams require more attention to the benefits that formal practices can provide.

Typically, single-physics CSE application and library software efforts naturally involve a small team of researchers who work closely with each other on a daily basis. However, advanced CSE projects require a coordinated effort of dozens or more researchers who, although contributing to a larger effort, continue to work in small teams on their portion of the project. The Trilinos project, as one example of a "project of projects," has used a kind of "federalist" approach to addressing these competing realities. We have formally defined a "package" to be a collection of related functionality developed by a small team with certain rights and responsibilities in the larger Trilinos framework.

This basic approach has enabled a great deal of local autonomy in decision-making, allowing us to tolerate and appreciate a variety software research and development styles, and team cultures. We can handle modest redundancy in software functionality and adapt to change in many ways. At the same time, this approach also provides a global interaction that promotes a variety of desirable outcomes: (i) cross-fertilization of ideas, techniques and tools across package teams, (ii) adoption of "best practices" from one package across other packages, (iii) fostering of trust among disparate groups (iv) software modularity that is naturally enforce by package and team boundaries and (v) well-defined interfaces between packages for interoperability.

One important factor that improves the effectiveness of the Trilinos architecture is the constant focus on improving software engineering practices and processes. The philosophy we promote is that we spend time on improving software engineering so that we can spend less time on software development and maintenance and more time on science and engineering. This emphasis has two major impacts on our efforts: (i) better software engineering in the project makes for better software so that package teams are willing to use each other's software and (ii) discussions of incompatibilities in practices and processes across packages can focus on the goal of determining best practices and not decay into expressions of personal preference that can be contentious and counter-productive.

The net result of this approach to software research and development is a large and growing collection of inter-related tools where Trilinos as a whole has an identity but, even more importantly, each package has its own identity within its community of interest. It is worth noting that this kind of approach is also operative within the TOPS-2 SciDAC project. The climate community uses the CCSM in a similar way, but we are unfamiliar with its internal dynamics.

We believe an international effort to coordinate the efforts of many groups can benefit from the kind of model the Trilinos project is using. This type of approach will allow individual teams to simultaneously continue with their current efforts, practices and culture while at the same time start contributing to a larger whole.

6. Conclusion

There are many challenges facing application development in the transition from petascale to exascale. We believe the four issues above have the highest priority and, if addressed, will greatly improve exascale computing capabilities.