PROGRAMMING MODELS AT EXASCALE: ADAPTIVE RUNTIME SYSTEMS, INCOMPLETE SIMPLE LANGUAGES, AND INTEROPERABILITY

Laxmikant Kale

PARALLEL PROGRAMMING LABORATORY, SIEBEL CENTER FOR COMPUTER SCIENCE, UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN, USA
(KALE@ILLINOIS.EDU)

Abstract

Applications running on exascale machines will be complex in many ways. They will involve dynamic and adaptive refinements, and will be composed of multiple, independently developed modules, often involving a multiphysics simulation. The programming models of this era must have several characteristics. First, they need to do away with the notion of processors, and automate resource management via adaptive runtime systems. Data structure-specific frameworks and domain-specific environments will be needed to further simplify programming. More importantly, parallel mini-languages need to be developed, such that each language captures only a restricted subset of possible parallel interactions, but allows for a simple expression of them. Coupled with interoperability and parallel composition, which must be supported in many ways, including message-driven runtime systems, this will create a productive ecosystem of parallel programming models for the exascale era.

Key words: parallel programming models, exascale, adaptive runtime systems, interoperability, compositionality

The International Journal of High Performance Computing Applications, Volume 23, No. 4, Winter 2009, pp. 344–346
DOI: 10.1177/1094342009347497

Programming models to be used on the exascale machines should be influenced by the needs of the classes of applications that will run on these machines. We know that many of these codes will rely on variable resolution techniques, such as dynamic and adaptive refinements, rather than uniform increases in resolution, to utilize the higher computational capacity of such machines. This will make programming more challenging, because of issues such as dynamic load balancing. Also, applications will need to integrate independently developed modules, including those arising from multiphysics computations.

From this, it follows that the programming models (at least a major category of them) need to free the programmer from worrying about what data is stored where, and which computation happens on what processor. This need to remove "processors" from the ontology of the application programmer should be met by future models. *Overdecomposition* (e.g. object-based overdecomposition), and *virtualization* are examples of such an approach. With such approaches, mapping of work-units and data-units to processors must be handled by an *adaptive runtime system*.

An adaptive runtime system mediates communication among work units – such as user-level threads, and message-drive-objects – as well as communication between work-units and data-units. Thus it can keep track of communication affinities and their weights. It also explicitly schedules execution of work units on processors, and thus can keep track of the computational needs of each work-unit. Based on such runtime monitoring, it can migrate the units across nodes of a system – either periodically or continuously – so as to handle dynamic load imbalances. The runtime system for exascale systems should also help effect proactive as well as post-fault fault-tolerance.

Because of the complexity of exascale applications, alluded to above, we need to make efforts to simplify parallel programming further. Admittedly, techniques such as overdecomposition and the consequent adaptive runtime systems simplify programming because they take away resource management concerns from the programmer. However, expressing parallel interactions will still remain substantially complex; in some cases, it can become more complex than before, because of the use of asynchrony to gain efficiency, and issues of expressing coordination among multiple collections of interacting objects. There are at least two complementary ways for further simplifying parallel programming.

1. **Frameworks**: This avenue is known to HPC researchers and there probably is a consensus that appropriate frameworks can enhance productivity.

In my view, *framework* includes both data-structure specific frameworks (DSSF) and higher level domain specific frameworks. We know that a small number of data structures underlie most parallel CSE codes: structured meshes including adaptively refined ones, unstructured meshes, particles distributed in uniform cells or via spatially decomposed trees. Each DSSF can embody common functionality for parallel use of one such data structure, so it does not have to be programmed repeatedly. In some cases, several frameworks have been proposed and built, but their utility and use has remained limited, probably because of their tendency to "take over" the entire data structure from the application developer, and/or their lack of interoperability (see below). More research in this area is therefore needed. Similarly, higher level domain-specific environments (e.g. computational-astronomy environment) can collect multiple techniques making it easy to put together a complete code with different combination of strategies.

2. **Simpler but incomplete languages**: Compared with frameworks this idea is probably more novel and controversial. The basic idea is that *restricting* modes of interactions among parallel entities leads to simpler languages. Such a restricted language may not be "complete" in that it cannot express all parallel interactions (and therefore cannot express all algorithms and application modules); yet, if it covers a significant class of modules, *and* substantially simplifies expression of programs that it does cover, it is a useful language. There are several candidates for such languages, including those that allow only static data flow patterns, and those that allow restricted access patterns in a global address space. We should identify and develop such languages. One way in which some languages may become "simpler" is by outlawing non-determinacy. Another desirable property worth exploring is languages that cleanly separate parallel code from sequential (typically "science") code. Note that I am not advocating that all languages should separate parallel and sequential codes, but rather that some of the languages in our future language ecosystem may attain simplicity in that fashion.
3. **Interoperability and parallel composition**: The above approaches will succeed (in the context of multiphysics, multiscale, and multimodule exascale applications) only if they can be made to interoperate efficiently without losing simplicity. This means entities from different modules must be able to interleave their execution on individual processors without them being aware of each other explicitly. The related concept of parallel composition can be illustrated with a simple example: consider parallel modules A, B, C, and D, which are be executed as: [A; (B ‖ C); D]; that is, work in B and C, each of which consists of multiple work and data units spread across all processors, is concurrent, and so should interleave in the interest of efficiency. Therefore, idle time in B can be overlapped with useful computation in C and vice versa. Spatial separation, where B and C execute on disjoint sets of processors, is not adequate in the context of dynamic and complex applications; nor is it enough to force a sequencing, e.g. by executing B before C on all processors, since that precludes the adaptive overlap mentioned above. The current MPI-based execution models do not support such concurrent composition. Message driven execution appears to be a pre-requisite for supporting such interoperability, although the research community may aim at finding multiple alternative methods for supporting interoperability.

Interoperability also avoids the need for declaring a single programming paradigm a "winner." I expect that multiple models will be developed and will survive in the parallel programming toolkit. Programmers will choose the model best suited for the particular module they are coding, possibly further influenced by their subjective tastes and backgrounds, in addition to the innate needs of the module being coded. Interoperability will allow programming models to be tested and evaluated (e.g. by coding a single module in a large application) with ease, leading to a rapid evolution of programming models. In biological evolution, the principle of "survival of the fittest" does not lead to a single species to the exclusion of others, but to a stable ecosystem. Similarly, I expect such an evolution of parallel programming models to lead to an ecosystem of multiple interdependent programming models, rather than a single model that is declared a winner and imposed on all applications.

Author Biography

Laxmikant Kale is a professor of computer science at the University of Illinois at Urbana-Champaign, where he has been a faculty member since 1985. He received a Ph.D. in computer science from State University of New York, Stony Brook, in 1985. Professor Kale's research has involved various aspects of parallel computing, with a focus on enhancing performance and productivity via adaptive runtime systems, and

designing programming abstractions based on use-cases from multiple applications. He led the development of Charm++ and AMPI programming systems that embody an adaptive runtime system. He has collaboratively developed well-known parallel applications in the areas including biophysics, astronomy, and quantum chemistry. He was co-recipient of a Gordon Bell award in 2002.