IESP Challenges: Application Development, Translation and Execution Environments

Barbara Chapman, Richard Graham, Barney Maccabe, Oscar Hernandez, Bernd Mohr, Wolfgang E. Nagel

The complexity of high performance computing (HPC) architectures is increasing dramatically as we seek to make exascale computing a reality. Yet unlike technological developments of the recent past, emerging architectures will not be based on increased clock speeds. Instead, they will most likely consist of many hundreds of thousands of processor cores, organized as networked collections of multi- or many-core processors with attached, fundamentally distinct special-purpose processors, spanning increasingly complex memory hierarchies. Future architectures are projected to continue, and magnify, this trend toward interconnected asymmetric, heterogeneous processors and fragmented memory. The sheer number of cores, the different kinds of parallelism exhibited within a single machine, and the widespread use of specialized components in conjunction with general-purpose compute cores all will necessitate **fresh thinking about how to provide system software support for productive application creation, translation and execution.**

We Need New Translation Technology

If exascale performance and productivity goals are to be met, our current compiler technology must undergo significant change. We need to create a new application program translation infrastructure that enables compilers, users and tools to collaborate, exploiting synergies between them in order to achieve exascale levels of application performance. Compilers must no longer be viewed as a black box but rather as open translation infrastructures that must be capable of interoperating with all elements of the development and execution environment. Access to the entire translation process will enhance the functionality and quality of both compilers and tools, enabling a variety of interoperations and the sharing of information, and will moreover provide an open infrastructure for research on programming languages, their implementation, and scalable runtime libraries. Such a complete, open compiler infrastructure will promote collaborations across systems and among application and system software communities around the world, as users and tools will be able to interoperate with this infrastructure and control the translation strategies as well as benefit from compiler functionality.

A full end-to-end open source translation infrastructure should provide a framework for translating HPC languages including state-of-the-art optimizations, and the target exascale runtime system software with fault tolerance support, power management, scalable runtime measurements and more. The optimization infrastructure should include parallel data and control flow analysis; such analyses will be needed to support the translation of parallel constructs to lower-level forms of representations including current mainstream languages and their runtime support. This infrastructure should support the maximal use of high-level language constructs for both intra- and inter-node programming to ensure portability while facilitating efficient code generation for the different devices on exascale platforms, allowing the non-expert user to rely on the compiler and tools to make many of the implementation decisions, while expert users may explicitly direct the compiler's low-level optimizations. High-level compiler cost models may be used to drive non-linear optimizations that occur within loop nest and interprocedural optimization phases, while the selection of architecture-specific intrinsics or other optimizations that attempt to make the best possible use of architectural resources can be carried out during intermediate translation steps or subsequent code generation phases.

Such an infrastructure should also help to break existing rigid boundaries between different kinds of system software by making a broad range of program analyses and optimizations accessible to tools. As programs are translated to object code, compilers generate different intermediate program representations that encapsulate information about the code and its data objects. Each representation is utilized for a limited time, exploited for certain kinds of optimizations, and then discarded. Tools should be able to access these program representations, and to provide information that may be critical to improve the quality of the information (analysis) and translation decisions; conversely, tools may exploit the information encoded in or associated with the intermediate representations for a variety of purposes, such as giving an explanation of the runtime behavior of the important portions of an application, understanding what assumptions and reasoning lay behind a choice of compiler optimizations, and what program features have prevented important optimizations, or parallel execution.
.
Appropriate interfaces must be crafted so that tools can access the information generated by the compiler and so they can assume full control of all the stages of lowering code, from source via the different intermediate representations, to object code. Tools may then not only be able to influence the different optimization phases in the compiler but also apply new translation strategies. For instance, the placement of optimizations relative to the lowering of parallel constructs is an important decision that may significantly affect the overall outcome of program translation, and a tool might determine the best approach in a given context. Transformation tools that do not cover the entire translation process are unable to influence any subsequent translation steps performed by an external tool such as a compiler, nor can they enforce the effective application of their transformations during the object code generation. The additional program analysis results and information needed to close this gap could be supplied by such an infrastructure.

Compilers need to take advantage of the increased compute power within supercomputer nodes, to account for disruptive technologies such as (but not limited to) accelerators, and perform more aggressive optimizations to increase compile-time efficiency. Based on the node characteristics, they should be able to generate multiple code versions (e.g. to target different heterogeneous components) that can be additionally auto-tuned or specialized during runtime. Moreover, we can no longer afford to do without the benefit of compilers at runtime. Exascale compilers could benefit from recent experiences with just-in-time compilation and perform online feedback optimizations, try out different optimizations or perform more aggressive speculative optimizations. They will need to incorporate a variety of light-weight strategies for modifying code on the fly. Such techniques could also be used to support tools' interoperability, since an exascale compiler could take advantage of dynamic techniques to adaptively gather runtime information that might potentially help debuggers/performance tools diagnose problems during production, minimizing the level of intrusion or the need to rerun long applications.

We Need New Tools

Studies have shown that programmer productivity is tightly related to the functionality provided by the available development tools. It is critical therefore to develop the tools infrastructure that will enable the application developer to deal with the complexity and scale of exascale computing systems, and to manage the complexity with the tremendous amounts of data that will be consumed and generated. We need new tools to model complex architectures and applications to reduce an application's power consumption, detect potential programming errors, help in the process of auto-tuning, performance measurements and program optimization. Tools need to be designed and built that deliver information that can give the programmer deep insight into the program and its data structures and how they can be optimized for specific architectures, or to evaluate a variety of programming alternatives. For example, software partitioning tools may help the users determine the suitability of code regions for execution on special-purpose processors, help them modify such regions to meet the processors' requirements, or to rearrange their data to map to the different memories and achieve good locality, thereby minimizing data transfers across hardware components. Support for finding the different levels and kinds of parallelism and for determining appropriate optimizations is needed to help port codes to take advantage of specific characteristics of the various configured components.

Tools might help the application developer to produce information about the ordering of optimizations, identification and expression of concurrency, locality, or identify code regions that satisfy certain properties to facilitate debugging or overall program improvement. A variety of tools will need to be available dynamically to identify the need for tuning or adaptation of code, and to respond to it.

Application development and deployment will also require a raft of tools that support inter-node programming, including communication and synchronization analyzers, data distribution optimizers and load balance evaluators. Tools are also needed to summarize and interpret the information collected across the system at runtime and to support the dynamic tuning of long-running codes.

We Need New Techniques

We need new program and data analysis and interpretation techniques that scale to deliver the existing functionality and enhanced insights. For instance, some strategies used by performance tools today to gather and process program execution information clearly do not scale. Selective instrumentation can reduce the high cost of tracing; yet it implies knowledge of a program's hotspots and/or existing performance problems. Static selective instrumentation based on modeling has been shown to drastically reduce overheads at the cost of some loss of information. However, better analytical models with runtime information might yield more accurate results. An application developer, a compiler and runtime tools may all potentially collaborate to choose areas for focused evaluation, giving feedback to each other. Strategies involving the user, compiler *and* tools must be found to focus the performance evaluation, to reduce the associated overheads and the amount of data gathered.

Work is needed on runtime libraries to transport, deliver, and provide a reduction of runtime information. Standard APIs for performance measurements will promote tools' interoperability, and accesses to all parts of the system and programming interfaces. Moreover, the runtime environment should be tunable by steering methods which allow the dynamic specification of the amount of information extracted to address specific performance issues. These steering techniques – accompanied by selective methods – may also be the right answer to address debugging challenges on systems with millions of cores.

We Need Better Tools Integration

Current tools usage is labor-intensive and fragmentary. For example, porting a code to a given platform often begins with a study of benchmarks in order to learn from relevant, but simplified, problems. Strategies that work well are then typically manually implemented in the real-world application, which is further evaluated and tuned in an iterative cycle of improvements, potentially involving multiple tools, each of which provide different kinds of insight. We need to overcome this fragmentary approach to application development by providing the user with an integrated environment. This necessitates the construction of interfaces that enable separately developed and maintained software tools to interoperate and share information pertaining to a specific application or execution instance. As part of the solution we need to consider how the compiler can interact with other parts of the environment to improve functionality and quality, reduce tools' overheads and increase accuracy, or enable improvement in the interpretation of information.