

Software Challenges at EXTREME Scale

Most experts expect that by 2015 or so, computer systems will be qualitatively different from current and past computer systems. Specifically, they will use massive multi-core processors with hundreds of cores per chip, and their performance will be driven by parallelism and constrained by energy. Consequently, they will be subject to frequent faults and failures.

The common term for such systems is *extreme scale*, as explained in Peter Kogge's recent report, "Technology Challenges in Achieving Exascale Systems." Kogge identifies three distinct classes of such systems:

- Data center-sized **exascale** systems, capable of delivering exaflop/s, or exa-ops (that is, one thousand times the capability of emerging petascale data center-sized systems; following common usage, "ops" means operations per second, unless otherwise specified)
- Departmental-sized **petascale** systems that allow the capabilities of a petascale system to be shrunk to fit within a few racks, allowing easier widespread deployment
- Embedded **terascale** systems that reduce terascale capability to a few chips and a few dozen watts, enabling deployment in a range of embedded environments.

The terms "exascale" and "extreme scale" are often used interchangeably. However, here extreme scale refers to systems of all three classes above, and exascale refers specifically to the largest, most powerful data center-sized system class.

Extreme scale systems have software challenges (figure 1). Although there are significant differences in the software environments and requirements for the three classes of extreme scale systems, they share at least two critical challenges: concurrency and energy efficiency. To address these, there are opportunities for both software-hardware co-design and new interfaces between applications, system software, and hardware.

The extreme scale concurrency challenge entails a need for at least a thousand times more concurrency (the simultaneous execution of instructions

or computations, potentially interacting with each other) than current software applications enjoy. The challenge is further exacerbated by the expected memory-computation imbalances in extreme scale systems, with bytes/ops ratios that may drop to values as low as 10^{-2} , where bytes and ops represent the main memory and computation capacities of the system, respectively. The effect of the reduced ratio is felt in a hundred-fold reduction in memory per core (relative to petascale systems) with an accompanying reduction in memory bandwidth per core. Therefore, a significant amount of extreme scale software concurrency must come from exploiting more parallelism within the computation performed on a single datum.

One path to more parallelism is strong scaling. However, strong scaling typically involves more frequent communication and synchronization than weak scaling (discussed below), reducing energy efficiency, due to losses in data movement and synchronization.

Another major obstacle to improving concurrency is the effect of communication and synchronization in software that contribute directly to the serial (Amdahl's law) fraction of the program's execution. A new software stack can reduce this overhead by orders of magnitude, especially with software-hardware co-design, thereby making it possible to achieve the necessary parallel efficiency. But the energy efficiency challenge is also critical because all three classes of extreme scale systems will be expected to deliver their thousand-fold improvements in computation capability without exceeding the power budgets of current systems. An aggressive hardware design for data center-sized systems will need at least 60 megawatts of power to achieve an exa-ops level of performance, under highly idealized zero-overhead assumptions for software. With current software overheads, exas-

A significant amount of extreme scale software concurrency must come from exploiting more parallelism within the computation performed on a single datum.

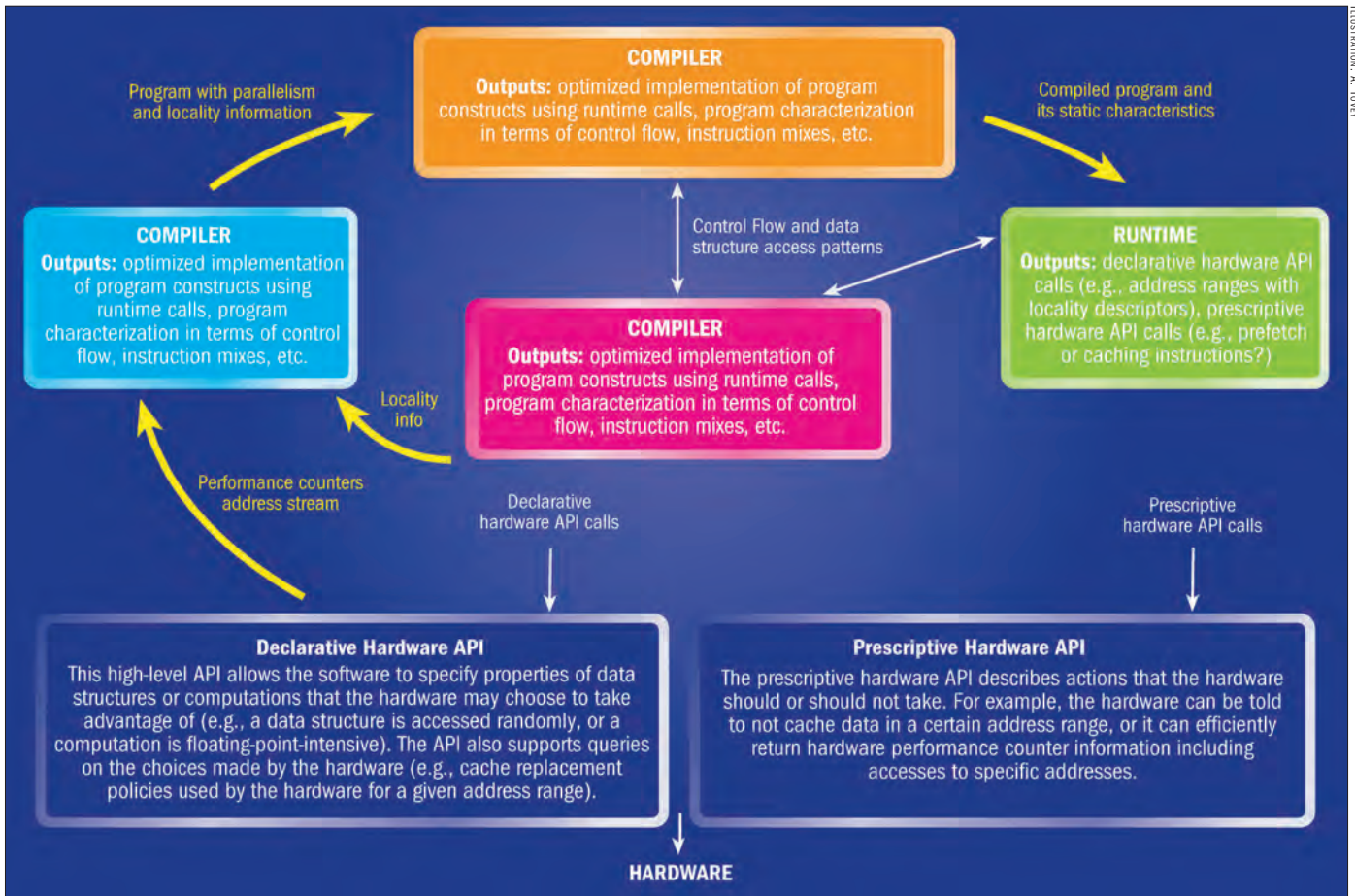


Figure 1. A diagram illustrating possible software flows for the parallel concepts discussed in the text.

cale capability cannot be achieved without a significant redesign of the system software stack.

Challenges in Application Scaling

Strong scaling refers to the concept of *applying more resources* to the same problem size to get results faster. Unfortunately, this approach is limited. As one strongly scales an application, the work at a node/processor/core decreases as the relative overhead increases. Eventually the amount of overhead eliminates any hope of running the application any faster by adding any more processors. Eventually, overhead grows so rapidly that adding processors actually causes the time to solution to increase.

Weak scaling refers to the concept of *adding work* as an application is run on more processors. By adding work, overhead does not destroy performance. Weak scaling permits the user to look at larger or more complicated problems and use the freed, additional processors to solve larger problems, obtain better resolution, or learn more about the phenomenon being examined.

Traditional weak scaling occurs in classical mechanics simulations, where either (1) larger problems are examined, or (2) the grid size and time-step interval are reduced. Solving larger problems – for

example, modeling the airflow around an entire airplane versus modeling the airflow over a section of the wing – results in a situation when memory requirements scale nearly proportionally with the extra work. When the grid size is reduced (refined) in a 3D mechanics simulation, the time step also needs to be reduced, increasing the amount of work relative to the amount of memory.

In summary, we do not expect most applications to get to extreme scale by strong scaling. Instead, developers will scale applications by using algorithmic innovations, with a combination of traditional and innovative weak scaling techniques.

Challenges in Expressing Parallelism and Locality

Programmers encounter significant challenges in expressing parallelism and locality across the three classes of extreme scale systems.

Portable Expression of Parallelism and Locality

Modern programmers repeatedly rewrite applications to expose incrementally more parallelism for each next generation of hardware. Instead, their goal should be to express all opportunities for parallelism, leaving the choice of what to exploit to the layers of the software stack responsible for manag-

We do not expect most applications to get to extreme scale by strong scaling. Instead, developers will scale applications by using algorithmic innovations, with a combination of traditional and innovative weak scaling techniques.

The general structure of the classical software stack will be highly mismatched to the requirements of all three classes of extreme scale systems of the future.

ing parallelism and locality. Admittedly, this is a demanding task for current programmers, because they have been trained in sequential programming.

First, a major focus in writing efficient sequential code is to minimize the total number of operations. In contrast, efficient parallel code needs to focus on *maximizing parallelism*, or minimizing the number of operations *on the critical path*. With modern memory hierarchies, both sequential and parallel code must also focus on improved locality, but parallel code offers more opportunities than sequential code to perform redundant operations, which reduces communication.

Second, good sequential algorithms attempt to minimize space usage and often include clever tricks to reuse storage. Parallel algorithms, however, need to use extra space to eliminate timing constraints to achieve larger scales of parallelism.

Finally, sequential programming techniques stress linear problem decomposition through sequential iteration and linear induction. But good parallel code usually requires multi-path problem decomposition and aggregation of results.

Portable Expression of Synchronization with Dynamic Parallelism

Writing programs using today's state-of-the-art synchronization primitives is akin to using assembly language for programming. All the burden of performance, scalability, and correctness falls squarely on the shoulders of the programmer with minimal support from the programming languages, development tools, runtime systems, or hardware. So, to make parallel programs robust, portable, and scalable, while reducing the burden on the programmers, we must innovate in synchronization and communication.

One of the biggest challenges with synchronization for a programmer is the difficulty in avoiding deadlock and data races. Of the two, data race avoidance is more challenging than deadlock avoidance. Removing non-determinism from the programming model (as in declarative and functional programming approaches) can greatly simplify the testing and debugging of parallel programs, but the key challenge there is to ensure the resulting model is sufficiently expressive for extreme scale software while still being efficient enough for execution on extreme scale hardware.

Expressing Heterogeneity in a Portable Manner

With the increasing popularity of hybrid architectures, programming systems must exploit multiple levels of parallelism by working with different memory systems, instruction sets, and even numerics. Today's systems, such as the Los Alamos National Laboratory Roadrunner system, contain as many as three distinct types of processors with distinct mem-

ory, messaging, and performance characteristics. This variety of hardware is managed explicitly by the application programmers. Unfortunately, if such characteristics of hardware heterogeneity are explicitly addressed by the programmer, not only is the programming effort increased, it is likely that the software will not be portable to other systems.

Extreme scale systems of the future may have both *designed heterogeneity* (for elements like architecture, organization, and instruction set), as well as heterogeneity in manufacturing variability, configuration, or aging differences. It should be possible with proper design to move applications from one machine to another – with different heterogeneous characteristics – without significant change in the application source code. This requirement imposes major challenges for parallelism, locality, and computation, such that both the compiler and runtime must deliver performance on one exascale system, while remaining portable and flexible enough to execute on other heterogeneous exascale systems. This quality is fundamental for a technology landscape that supports exascale computing.

Challenges in Managing Parallelism and Locality

The hardware characteristics of future extreme scale systems were summarized above, as well as the challenges involved in developing applications and expressing parallelism and locality for such systems. Here, we focus on the challenges and implications in software management of parallelism and locality for extreme scale. Current software for high-end data center, departmental, and embedded systems all build on a classical software stack, consisting of operating systems, parallel runtimes, static compilers, and libraries. However, the general structure of the classical software stack will be highly mismatched to the requirements of all three classes of extreme scale systems of the future.

Operating System Challenges

Many assumptions about current operating system design are no longer valid when considering an extreme scale processor containing thousands of cores. For example, a baseline challenge for the exascale software stack is how to get the operating system out of the way without compromising the need to protect the hardware state from errant (or malicious) software. Execution models that support more asynchrony will be necessary to hide latency. Such execution models will also require more carefully coordinated scheduling to balance resource utilization and minimize work starvation or resource contention. These execution models will also require extraordinarily low-overhead, fine-grained messaging. However, the required execution model attributes are nearly impossible to achieve when the operating system intervenes for

every operation that touches its privileged domain. Over time, operating systems have evolved into multifaceted and hugely complex software implementations with a broad range of capabilities. We refer to the challenge of breaking the operating system apart based on separation of concerns as “deconstructing the operating system.”

In their role as the gate-keeper to shared resources, operating systems have traditionally been a major bottleneck in achieving scalability on symmetric multiprocessing systems, where two or more identical processors often connect to a single shared main memory. This is especially true for the open source Linux operating system, which has historically lagged behind commercial Unix operating systems such as AIX and Solaris in scalability but has now become the system of choice for high-end systems. The Linux community has worked to bridge the scalability gap with commercial operating systems, starting with efforts such as improvements to the Linux scheduler in 2001. More recent examples of scalability efforts include large-page support, NUMA support, and the read-copy update (RCU) API. Yet, the scalability requirements for even a single socket of an extreme scale system will be two orders of magnitude higher than what can be supported by Linux today. It is clear that future scalability improvements in Linux are expected to be harder to achieve, as evidenced by the RCU experience and the complexities uncovered by ongoing efforts to reduce the scope of the Linux big kernel lock.

Runtime Challenges

Runtime support for parallel programming requires key innovations in lightweight mechanisms for communication and memory hierarchy management, and user-controllable policies for managing the system resources. Expected contributions to this area of research include:

- Lightweight runtime mechanisms to exploit the novel features of interconnection networks, including topology queries, atomic operations, remote procedure invocation, and fast one-sided transfer notification for synchronization
- Extensions of the execution models to handle fast and slow memory associated with a single thread, and demonstration of that model on a single-chip system with software-managed local memory that augments the traditional hardware-managed cache hierarchy
- Runtime support to virtualize the set of processors through the use of multi-threading and dynamic task migration, and programming model extensions that allow for such virtualization when needed, without enforcing it for all applications

- Runtime support for memory system virtualization, including object caching and migration. As with processor resources, the programming model will be extended to permit runtime-managed data placement in addition to the user-managed placement already available

- Support for multiple runtime systems for different execution models and soft real-time applications

Compiler Challenges

The crucial role of compilers at the extreme scale is mapping from language constructs that express a very high-level decomposition of an application, to highly power-efficient and memory-efficient machine code. Completely automatic compiler optimization from high-level code will not meet the performance requirements at extreme scale. Further, compiler-based approaches have generally focused on regular, static parallelism. But as we expand the applications for exascale platforms to encompass irregular, unstructured, and dynamic algorithms, so must the compiler technology support these challenging application domains.

Remember that the compiler requirements for exascale are similar to those for compilers at all scales. Mary Hall, David Padua, and Keshav Pingali wrote about a recent National Science Foundation-sponsored workshop on the future of compiler research. Demanding a broader collaboration between industry and academic institutions, and support from government agencies, some challenges on their agenda include:

- Make parallel programming mainstream
- Write compilers capable of self-improvement (that is, auto-tuning)
- Develop performance models to support optimizations for parallel code
- Enable development of software as reliable as an airplane
- Enable system software that is secure at all levels
- Verify the entire software stack.

Therefore, the compilers to be developed for exascale must be very different. Compilers at the extreme scale must collaborate closely with the application programmer to derive an architecture-independent algorithm description that can be mapped to high-quality code. Further, the compiler must incorporate lightweight mechanisms that interface with the runtime layer and

Software–hardware co-design will be another critical necessity for extreme scale systems.

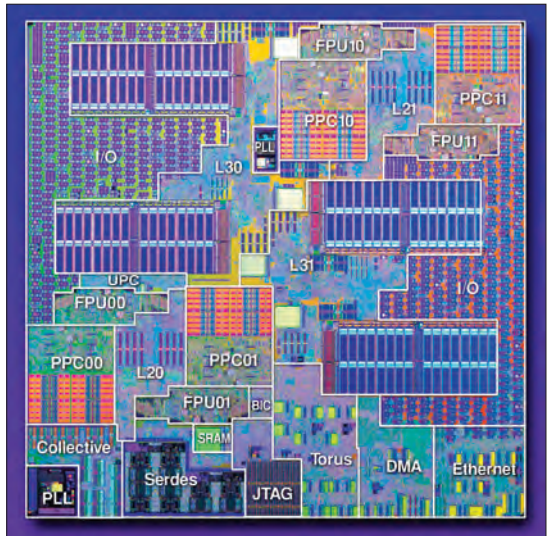


Figure 2. “System-on-a-chip” integration of components is one method being used to reduce hardware faults in extreme scale systems. This extremely low-power Blue Gene/P compute ASIC contains four PowerPC 450 cores with dual SIMD floating point units, two memory controllers, a 10 Gb Ethernet controller, a 3D adaptive router with DMA, a collective network, 8 MB of embedded DRAM cache, and other logic. There are 208 million transistors.

architecture to dynamically map this code for a specific execution context.

Software–Hardware Co-Design

Software–hardware co-design will be another critical necessity for extreme scale systems. Such co-design has been essential for vector parallelism in current and past systems and is also being explored for scalable approaches to mutual exclusion using transactional memory. Here are a few additional examples of software runtime capabilities that will be necessary for future extreme scale systems.

Scheduling Dynamic Parallelism with Fine-Grained Tasks

The intrinsic parallelism in a program must be expressed at the finest level possible – at the statement or expression level. Then, the compiler and runtime system can exploit the subset of parallelism that is useful for a given target machine. There have been multiple proposals for expressing fine-grained parallelism, but these operations are in stark contrast with bulk-synchronous parallel models. These approaches have shown increased scalability that is orders-of-magnitude superior to the scalability achieved if each task were to be created as a thread at the operating system level.

Yet, there remain significant limitations in a software-only approach that prevent it from being usable at extreme scale. These issues involve locking operations, and in the case of non-blocking algorithms,

involve spin loops on shared-memory locations with their accompanying cache consistency overheads. Such issues are especially problematic because they occur on critical paths in parallel programs.

Hardware support for shared queue data structures can result in orders-of-magnitude reductions in scheduling overheads and scalability bottlenecks, while retaining the flexibility of task scheduling policies in software.

Another source of overhead in task scheduling lies in the operations performed on the fast path to save local variables. A software-only approach introduces word-at-a-time instructions to save the local variables, and some of these saves are often redundant. In contrast, hardware support for saving and restoring local variables (as in calling conventions) can help reduce this fast path overhead.

Distribution and Co-Location of Tasks and Data

Another candidate for software–hardware co-design pertains to distribution and co-location of tasks and data. Runtime systems for programming languages such as UPC and Co-Array Fortran that are based on a partitioned global address space model include the notion of virtual home location for each shared datum. The more recent HPCS languages extend this notion of home locations to computational tasks, as in Chapel’s locales and X10’s places, so as to enable tasks to be shipped to data, data to be shipped to tasks, or any meet-in-the-middle combination thereof. Remaining opportunities include the use of translation buffers to accelerate virtual-to-physical address translations and DMA-like hardware support to reduce the processor overhead of data transfers.

Collective and Point-to-Point

Synchronization with Dynamic Parallelism

Remember that the fine-grained parallelism intrinsic to a program may need to be accompanied by fine-grained collective and point-to-point synchronization among dynamically varying sets of fine-grained tasks. These synchronization structures may be irregular, and tasks are permitted to dynamically join or leave these structures as in the phasers construct. Further, it is usually desirable to augment the synchronization structures with communication for reductions, collectives, and systolic computations. Hardware support can be used to reduce the overhead of inter-core synchronization, and extensions can reduce the overhead of communication. The use of a single master task to perform a reduction in software can be a scalability bottleneck, and a software-only approach to creating combining trees incurs high setup and tear-down overhead. Instead, hardware support for combining synchronization and reductions will greatly reduce

the overhead of collective and point-to-point synchronization with dynamic parallelism.

Producer–Consumer Parallelism

Another common idiom in fine-grained parallel programs is that of producer–consumer parallelism. In this model, a single-writer task serves as the producer of a datum for multiple readers. To accomplish this, the writer task typically stores its result in a designated location, and the reader tasks must wait when they request the result if the result is not ready. In the case of futures, the execution of the writer task may (optionally) be deferred till the datum’s value is requested by one of the readers. Once again, we observe that a software-only approach suffers cache consistency and serialization bottlenecks, and hardware support can be used to reduce these bottlenecks.

Summary

There are several reasons for paying attention to software in the development of extreme scale systems. First, the exascale systems of 2015–2020 will be dramatically different from today’s petascale systems and will require correspondingly fundamental changes in the execution model and structure of system software. Second, while there has been significant innovation at the hardware and system level for today’s petascale systems, previous approaches have not paid much attention to the co-design of multiple levels in the system software stack (operating system, runtime, compiler, libraries, application frameworks) that is needed for exascale systems. Third, while certain execution models such as Map-Reduce in cloud computing and CUDA in GPGPU data parallelism have demonstrated large degrees of concurrency, they haven’t demonstrated the ability to deliver a thousand-fold increase in parallelism to a single job with the energy efficiency and strong scaling fraction necessary for extreme scale systems (see under Further Reading, “The International Exascale Software Project”). To understand the software challenges for extreme scale systems, we must examine multiple application classes. The concurrency and energy challenges boil down to the ability to express and manage parallelism and locality in the applications. Applications can be enabled for exploiting extreme scale hardware by exploring a range of strong scaling and innovative weak scaling techniques, but only with attention to efficient parallelism and locality.

There are significant challenges in expressing parallelism and locality in extreme scale software. One of them is the ability to expose all of the intrinsic parallelism and locality in an application, so as to make the application forward scalable. Another is to ensure that this expression of parallelism and locality is portable across vertical and horizontal dimensions.

Operating system-related challenges include parallel scalability, spatial partitioning of operating system and application functionality, direct hardware access for inter-processor communication, and asynchronous rather than interrupt-driven events. There are additional challenges in runtime systems for scheduling, memory management, communication, performance monitoring, power management, and resiliency, all of which will be built atop future extreme scale operating systems. Yet, there are wonderful opportunities for addressing the concurrency and energy challenges through software–hardware co-design. ●

Contributors Vivek Sarkar, Department of Computer Science, Rice University; William Harrod, Information Processing Technology Office, Defense Advanced Research Projects Agency; and Allan E. Snaveley, San Diego Supercomputer Center, University of California–San Diego

Acknowledgments This paper was derived in large part from an exascale software study conducted over a series of seven meetings held from June 2008 to February 2009. The goal was to examine the state of the art, identify key challenges, and outline elements of a technical approach that can address the challenges without prescribing specific solutions. We would like to thank the study members and guests for their dedication to the field of parallel software and systems, and for all their hard work in contributing to the study. A detailed report on the study is now available (see Further Reading, below).

The applications discussion also benefited from numerous meetings held in 2007 to investigate future exascale computing applications and hardware/software requirements. These included three DOE Exascale Town Hall Meetings held at LBNL, ORNL and ANL, the Council on Competitiveness meeting on Exascale Applications, and the Frontiers of Extreme Computing 2007/Zettaflops workshop.

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under AFRL contract FA8650-07-C-7724. The views, opinions, and/or findings contained in this presentation are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the Department of Defense.

Further Reading

ExaScale Computing Software Study report: Software Challenges in Extreme Scale Systems, September 14, 2009. Editor, Vivek Sarkar. http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/ECS_reports.htm

J. Dongarra et al. 2009. The International Exascale Software Project: a Call To Cooperative Action By the Global High-Performance Community. *The International Journal of High Performance Computing Applications* 23: 309–322.

The exascale systems of 2015–2020 will be dramatically different from today’s petascale systems and will require correspondingly fundamental changes in the execution model and structure of system software.