

# Software Challenges in Extreme Scale Systems

Vivek Sarkar<sup>1</sup>, William Harrod<sup>2</sup>, and Allan E Snavely<sup>3</sup>

<sup>1</sup> Department of Computer Science, Rice University

<sup>2</sup> Information Processing Technology Office, Defense Advanced Research Projects Agency

<sup>3</sup> San Diego Supercomputer Center, University of California, San Diego

E-mail: vsarkar@rice.edu, william.harrod@darpa.mil, asnavely@ucsd.edu

**Abstract.** Computer systems anticipated in the 2015 – 2020 timeframe are referred to as *Extreme Scale* because they will be built using *massive multi-core processors* with 100’s of cores per chip. The largest capability Extreme Scale system is expected to deliver *Exascale* performance of the order of  $10^{18}$  operations per second. These systems pose new critical challenges for software in the areas of *concurrency*, *energy efficiency* and *resiliency*. In this paper, we discuss the implications of the concurrency and energy efficiency challenges on future software for Extreme Scale Systems. From an application viewpoint, the concurrency and energy challenges boil down to the ability to express and manage parallelism and locality by exploring a range of *strong scaling* and *new-era weak scaling* techniques. For expressing parallelism and locality, the key challenges are the ability to expose all of the intrinsic parallelism and locality in a programming model, while ensuring that this expression of parallelism and locality is *portable* across a range of systems. For managing parallelism and locality, the OS-related challenges include *parallel scalability*, *spatial partitioning* of OS and application functionality, *direct hardware access* for inter-processor communication, and *asynchronous* rather than interrupt-driven events, which are accompanied by runtime system challenges for *scheduling*, *synchronization*, *memory management*, *communication*, *performance monitoring*, and *power management*. We conclude by discussing the importance of *software-hardware co-design* in addressing the fundamental challenges for application enablement on Extreme Scale systems.

## 1. Introduction

It is widely recognized that the computer systems anticipated in the 2015 – 2020 timeframe will be qualitatively different from current and past computer systems. Specifically, they will be built using *massive multi-core processors* with 100’s of cores per chip, their performance will be *driven by parallelism and constrained by energy*, and they will be subject to *frequent faults and failures*. We use the term, *Extreme Scale*, to refer to these systems. A characterization of Extreme Scale systems can be found in the recent report on “Technology Challenges in Achieving Exascale Systems” [21]. This characterization identifies three distinct classes of systems:

- **Data-center-sized Exascale systems**, capable of delivering 1 ExaFlops or 1 ExaOps *i.e.*,  $1,000\times$  the capability of currently emerging Petascale data-center-sized systems<sup>1</sup>.
- **Departmental-sized Petascale systems** that allow the capabilities of a Petascale system to be shrunk in size and power to fit within a few racks, allowing widespread deployment.

<sup>1</sup> Following common usage, “ops” refers to operations per second in this paper unless otherwise specified.

- **Embedded Terascale systems** that reduce Terascale capability to a few chips and a few ten's of watts, thereby enabling deployment in a range of embedded environments.

Since the first system class listed above achieves Exascale performance, the terms *Exascale* and *Extreme Scale* are often interchangeably in the community. However, in this paper, we prefer to use *Extreme Scale* to refer to systems across all three classes and *Exascale* specifically for the largest data-center sized system class.

The focus of this paper is on *software challenges* for Extreme Scale systems. The scope of software considered spans the spectrum of operating systems; runtimes for scheduling, memory management, communication, performance monitoring, power management, and resiliency; computational libraries; compilers; programming languages; and application frameworks. Though there are significant differences in the software environments and requirements for the three classes of Extreme Scale systems, all of them share some critical challenges namely, *concurrency*, *energy efficiency* and *resiliency*. The scope of this paper is restricted to concurrency and energy efficiency challenges related to software and identification of opportunities for software-hardware co-design, as well as potential new interfaces between applications and system software and between system software and hardware.

The concurrency challenge is manifest in the need for software to expose at least  $1000\times$  more concurrency in applications for Extreme Scale systems, relative to current systems. It is further exacerbated by the projected 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. These ratios will result in  $100\times$  reductions in memory per core relative to Petascale systems, with accompanying reductions in memory bandwidth per core. Thus, a significant fraction of software concurrency in Extreme Scale systems must come from exploiting more parallelism within the computation performed on a single datum *i.e.*, from strong scaling or from the “new-era” weak scaling discussed in Section 2. Strong scaling typically involves more frequent communication and synchronization than weak scaling, which in turn exacerbates the energy efficiency challenge since data movement and synchronization are major contributors to energy costs in Extreme Scale systems. Another major obstacle to achieving a large degree of concurrency arises from the communication and synchronization overheads in current system software that contribute directly to the serial (Amdahl's Law) fraction of the program's execution. A new software stack can reduce these overheads by orders of magnitude, especially with software-hardware co-design, thereby making it possible to achieve the parallel efficiency needed for Extreme Scale systems.

The energy efficiency challenge is also critical because all three classes of Extreme Scale systems will be expected to deliver their  $1000\times$  improvements in computation capability while essentially remaining within the power budgets of current systems. An aggressive hardware design for data-center-sized systems will need at least 60MW of power to achieve an Exa-op level of performance, under highly idealized zero-overhead assumptions for software [21]. When current software overheads are taken into account, it is clear that the Exascale capability cannot be achieved without a significant redesign of the system software stack.

As discussed in this paper, current software approaches will be inadequate in enabling future Grand Challenge applications (outlined in Section 2) on Extreme Scale systems. Instead, the potential for a  $1000\times$  increase in computation capability offered by each class of Extreme Scale system will only be achievable through radical re-design of the underlying system software and hardware. Current system designs won't work at Extreme Scale because of their sequential foundations and their inherent energy inefficiencies. Recent trends in High Productivity Computing Systems (HPCS) have demonstrated reductions in human effort involved in developing high-productivity software for current Petascale systems, but do not address the requirements of Extreme Scale architectures such as energy-constrained many-core parallelism and heterogeneous processors. Also, while there will be some overlap between system

software requirements for Extreme Scale and those for large scale commercial data centers, there are also significant differences. Commercial system software for cloud computing is primarily focused on optimizing throughput capacity of independent jobs, whereas system software for Extreme Scale must be capable of delivering  $1000\times$  (or more) increase in parallelism to a single job.

The rest of the paper is organized as follows. Section 2 discusses selected grand challenge applications that are being developed from scratch or are being scaled up from existing Petascale applications. Section 3 discusses challenges in expressing parallelism and locality at the finest granularities possible so as to support *forward scalability*, whereas Section 4 outlines the challenges in low overhead management of this fine-grained parallelism and locality. Finally, Section 5 discusses how future Extreme Scale software stacks must be tightly integrated with new Extreme Scale hardware via software-hardware co-design, and Section 6 contains a concluding summary.

## 2. Challenges in Application Scaling

Application scaling remains a major challenge in the utilization of high-end parallelism. Of applications that operate at sustained Terascale performance today, only a small fraction is expected to be successful at reaching Petascale and an even smaller fraction at Exascale. Further, even for applications that reach Petascale performance today, the nature of scaling necessary to obtain Petascale performance on an Extreme Scale departmental system will raise new challenges for rewriting the application to address the concurrency and locality requirements of such systems. In this paper, we argue that the existing software “stack” is a major contributor to these scalability limitations, and that the approaches discussed in the following sections could have a significant impact in removing obstacles to scaling. We start by examining two primary ways to scale applications.

*Strong scaling* refers to the concept of applying more resources to the same problem size to get results faster. Unfortunately, few applications are amenable to strong scaling. As you strongly scale an application, the work at a node/processor/core decreases and the relative overhead increases. Speedup may initially equal the number of processors, but eventually the amount of overhead causes the slope of the speedup curve to flatten. At this point, adding processors does not cause the application to run faster. Eventually, it is possible that overhead grows so rapidly that adding processors actually causes the time to solution to increase and speedup to decrease. An application that demonstrated reasonable scaling over three orders of magnitude increase in the number of processors is the first principles molecular dynamics “Qbox” code that won the 2006 ACM Gordon Bell Prize for “peak performance” with over 200 Tflop/s sustained performance (56% efficiency) on the LLNL BlueGene/L [14]. More recent winners of the 2008 ACM Gordon Bell Prize further underscored the importance of algorithmic innovations that attain very high levels of spatial and temporal locality.

*Weak scaling* refers to the concept of adding work as an application is run on more processors. By adding work, it is possible to assure that overhead does not destroy performance. Traditionally, weak scaling has referred to adding work due to spatial scaling. Meanwhile, there are additional sources of scaling – referred here as “new-era” weak scaling — arising from new application trends in which additional work is done per datum *e.g.*, multi-scale, multi-physics, interaction analysis, and data mining. Weak scaling permits the user to look at larger or more complicated problems and use the 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 – *e.g.*, modeling the airflow around an entire airplane versus modeling the airflow over a section of the wing – results in a situation when memory scales nearly proportionally with work.

When the grid size is reduced (refined) in a 3-D mechanics simulation, the time step also needs to be reduced thereby increasing the amount of work relative to the amount of memory. When scaling in three dimensions, a  $3/4$  power rule applies to the amount of memory required whereas a  $2/3$  power rule applies when scaling in only two dimensions. Thus, for these applications, the required increase in memory size for a  $1000\times$  increase in work is  $180\times$  and  $100\times$  for 3-D and 2-D applications respectively.

In summary, we do not expect most applications to get to Extreme Scale by strong scaling. Instead, applications will be scaled by using algorithmic innovations with a combination of traditional and new-era weak scaling techniques.

### 3. Challenges in Expressing Parallelism and Locality

The focus of this section is on the challenges in expressing parallelism and locality that are encountered by application-level programmers across the three classes of Extreme Scale systems. The task of managing the parallelism and locality is relegated to the system software discussed in Section 4. It is likely that some heroic programmers, particularly for the data-center and embedded configurations, will wish to program directly at the system level using the interfaces in Section 4. However, for the remainder, it will be critical to address the challenges outlined in this section to enable them to use the capabilities of Extreme Scale systems.

#### 3.1. Portable Expression of Parallelism and Locality

The requirement of pervasive extreme-scale parallelism outlined earlier demands that all of the intrinsic parallelism be exposed at all levels in the application. This is a marked contrast to current practice where programmers repeatedly rewrite applications to expose incrementally more parallelism for the next generation of hardware. Instead, our 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 managing parallelism and locality (Section 4). While this is likely to be a more demanding task for current programmers who have been trained in sequential programming, it is expected that expression of parallelism and locality will be simplified in future programming models that *break sequential habits of thought* [38]. In this section, we briefly summarize some of the key points made in [38] and related work.

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 *i.e.*, *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 (say) perform redundant operations to reduce communication. As mentioned earlier, locality optimization will have a first-order impact on energy reduction for future Extreme Scale systems. Second, good sequential algorithms attempt to minimize space usage and often include clever tricks to reuse storage; however, parallel algorithms need to use extra space to permit temporal decoupling and to achieve larger scales of parallelism. Finally, sequential idioms often stress linear problem decomposition through sequential iteration and linear induction. On the other hand, good parallel code usually requires multi-way problem decomposition and multi-way aggregation of results. A simple example is the difference between specifying a summation as a sequential iteration vs. a Fortran 90 SUM intrinsic for arrays.

A fundamental issue in the portable expression of parallelism is the need for data structures that lend themselves naturally to *data-parallel* operations. Fortunately, the *array* or *vector* data structure, which is a cornerstone of traditional HPC applications, is very well suited to data parallelism as evidenced by programming languages such as APL [17], Fortran 90 [29], NESL [2] and Ct [11]. These languages are able to express both flat data parallelism on vectors (element-wise operations, reductions, constrained permutations) and nested data parallelism

on sparse or indexed vectors. *Streams* represent another data structure that is well suited for parallelism, as exemplified in data flow languages and programming models such as Sisal [27], Synchronous Data Flow [24], Brook [4] and StreamIt [12]. However, graph and other pointer-based data structures necessary for new Extreme Scale applications pose additional challenges for expression of parallelism and locality. The notion of *abstract collections* in modern object-oriented languages can help bring some of the benefits of data parallelism from arrays and streams to pointer-based data structures. In addition, *asynchronous dynamic parallelism*, as embodied in languages such as Cilk [3], Chapel [7], Fortress [1], and X10 [6], is necessary for operating on irregular data structures. Compared to current approaches, a key challenge for Extreme Scale is the ability to express this parallelism at the finest granularity possible, while delegating to the implementation the choice of what parallelism to exploit in a locality-sensitive manner.

### 3.2. 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. In order to make parallel programs robust, portable, and scalable and reduce the burden on the programmers, many innovations are required in synchronization and communication. As an example, the *phasers* construct [34, 35] extends X10's clocks [6] so as to integrate collective and point-to-point synchronization with fine-grained dynamic parallelism, while providing a *next* statement that guarantees that all synchronizations will be performed in a deadlock-free manner. Each fine-grained task has the option of registering with a phaser in *signal-only/wait-only* mode for producer/consumer synchronization or *signal-wait* mode for barrier synchronization. Support for dynamic parallelism dictates that it should be possible for new tasks to be dynamically added and dropped from phaser registrations, which creates a potential challenge to avoid race conditions between synchronization operations and registration add/drop requests. The fine-grain synchronization that accompanies fine-grain parallelism also presents the challenge of *phaser contraction* [36] to reduce the synchronization overhead when reducing the actual parallelism that is exploited on a given system.

One of the biggest challenges with synchronization for a programmer is the difficulty in avoiding deadlock and data races, both of which can appear non-deterministically in current programming models. Of the two, data race avoidance is more challenging than deadlock avoidance, since deadlock freedom can be enforced by well-defined programming practices and the use of deadlock-free programming constructs such as phasers. 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 that the resulting model is sufficiently expressive for Extreme Scale software while still being efficient enough for execution on Extreme Scale hardware.

### 3.3. Expressing Heterogeneity in a Portable Manner

With the advent and increasing popularity of hybrid architectures, programming systems face the challenge of how to efficiently exploit multiple levels of parallelism, often coupled with different memory systems, instruction sets, or even numerics. In current-day systems, such as the LANL Roadrunner system [20], there may be as many as three distinct types of processors with distinct memory, messaging, and performance characteristics. These elements of heterogeneity are managed explicitly by the application programmers — through the use of coroutine-style models (one for each type of heterogeneity), explicit message passing for data movement, and distinct address spaces. Other examples of heterogeneous systems might include instruction

set and performance heterogeneity or simply differences in memory structure such as cache coherent shared-memory, partitioned global address space, or shared-nothing. 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 functionally portable, much less performance portable to other systems.

Extreme Scale systems of the future may have both designed heterogeneity (in dimensions such as architecture, organization, instruction set that are exhibited today in hybrid systems), as well as heterogeneity that arises from manufacturing variability, configuration, or aging differences. It is critical the software built for such large-scale parallel systems address the heterogeneity of the system in a fashion that supports portability of the applications. That is, it should be possible to move applications from one machine to another — with different heterogeneous characteristics — without significant change at the application source code level. This imposes major challenges in expression of parallelism, locality, and computation so as to both enable the compiler and runtime to deliver performance on one Exascale system, but also in a form portable and flexible enough that it can enable the compiler and runtime to deliver performance on other heterogeneous Exascale systems. This is a daunting challenge, but is in our view a fundamental requirement for a technology landscape that supports Exascale computing.

#### 4. Challenges in Managing Parallelism and Locality

Earlier in this paper, we summarized the hardware characteristics of future Extreme Scale systems as well as the challenges involved in developing applications (Section 2) and expressing parallelism and locality (Section 3) for such systems. In this section, 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 build on a *classical software stack* which primarily consists of operating systems, parallel runtimes, static compilers, and libraries. However, as described in the following sections, the general structure of the classical software stack has remained largely unchanged for decades, and will be highly mismatched to the requirements of all three classes of future Extreme Scale systems.

##### 4.1. Operating System Challenges

Extreme Scale processors containing hundreds or even thousands of cores will challenge current operating system (OS) practices. Many of the fundamental assumptions that underlie current OS technology are based on design assumptions that are no longer valid for a Extreme Scale processor containing thousands of cores. In the context of Exascale system requirements, as machines grow in scale and complexity, techniques to make the most effective use of network, memory, processor, and energy resources are becoming increasingly important. A baseline challenge for the Exascale software stack is how to get the OS out of the way without compromising the need to protect 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 attributes required by the execution model are nearly impossible to achieve when the OS intervenes for every operation that touches its privileged domain – it must intervene for inter-processor communication operations, has exclusive and privileged control of scheduling policy, and exclusive ownership of resource management policies. Over time, operating systems have evolved into multifaceted and hugely complex software implementations that have accreted a broad range of capabilities. We refer to the challenge of breaking the OS apart based on separation of concerns as “deconstructing the OS”.

In its role as the gate-keeper to shared resources, operating systems have traditionally been a major bottleneck in achieving scalability on SMP's. This is especially true for the open source Linux operating system, which has historically lagged behind commercial Unix OS's such as AIX and Solaris in scalability but has now become the dominant OS of choice for high-end systems. Significant attention has been devoted by the Linux community over multiple years to bridge the scalability gap with commercial OS's, starting with efforts such as improvements to the Linux scheduler in 2001 [22]. More recent examples of scalability efforts explored and undertaken by the Linux community include large-page support, NUMA support [26], and the Read-Copy Update (RCU) API [28]. While these Linux enhancements have resulted in improvements for commercial workloads with independent requests and flow-level parallelism [40] on small-scale SMP's, 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 this gap cannot be bridged by business-as-usual efforts; in fact, future scalability improvements in Linux are expected to be harder rather than easier to achieve, as evidenced by the RCU experience [28] and the complexities uncovered by ongoing efforts to reduce the scope of the Linux Big Kernel Lock (BKL) *e.g.*, see [19].

#### 4.2. 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, fast one-sided transfer notification used in 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 replaces or 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. 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.

#### 4.3. Compiler Challenges

The crucial role of compilers at the extreme scale is to map from language constructs that express a very high-level decomposition of an application to highly power-efficient and memory-efficient architecture-specific code and runtime layer calls. As has been proven historically, completely automatic compiler optimization from high level code will not meet the performance requirements at Extreme Scale, and in this regime, we will also encounter memory and power constraints that programmers and tools could previously ignore. Further, compiler-based approaches, such as, for example, compilers for PGAS languages, have generally focused on regular, static parallelism. 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. It is important to note that the compiler requirements for

Exascale are similar to those for compilers at all scales. An article reporting on a recent NSF-sponsored workshop on the future of compiler research listed the following 6 research challenges, in addition to other guidelines on enhancing the research approach and enriching education [15].

Research challenges in optimization:

- Make parallel programming mainstream;
- Write compilers capable of self improvement (i.e., auto-tuning); and
- Develop performance models to support optimizations for parallel code.

Research challenges in correctness:

- Enable development of software as reliable as an airplane;
- Enable system software that is secure at all levels; and
- Verify the entire software stack.

Therefore, the compilers we must develop for Exascale as discussed further in this section will necessarily 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 architecture to dynamically map this code for a specific execution context to be both high performing and power efficient.

## 5. Software-Hardware Co-Design

We believe that software-hardware co-design will be a critical necessity for Extreme Scale systems, in addition to the interfaces outlined in the previous section. This form of co-design has been essential for *vector parallelism* [18] in current and past systems, and is also being explored for scalable approaches to mutual exclusion using *transactional memory* [23]. In this section, we discuss a few additional examples of software runtime capabilities that will be necessary for future Extreme Scale systems, and examine how they can be made more effective with software-hardware co-design.

### 5.1. Scheduling dynamic parallelism with fine-grained tasks

As discussed in Section 3, it is important to ensure that the intrinsic parallelism in a program can be expressed at the finest level possible *e.g.*, at the statement or expression level, and that the compiler and runtime system can then exploit the subset of parallelism that is useful for a given target machine. There have been multiple proposals for expressing fine-grained parallelism *e.g.*, statement-level *spawn* [3] or *async* [6] operations, expression-level *future* [16] operations, and operator-level data flow graphs [8, 37]. These operations for fine-grained parallelism are in stark contrast with the *bulk-synchronous parallel model* [39]. While profile-directed compile-time partitioning can be used to optimize the granularity of fine-grained tasks in certain cases [32, 33], in general the runtime system also needs to participate in the partitioning so as to best adapt to unpredictable execution times. A classic approach to runtime partitioning is *lazy task creation* [30], which has been extended into *work-stealing runtimes* for fine-grained tasks [10, 13]. A work-stealing runtime system creates a fixed number of worker threads, with one local double-ended queue (deque) per worker. Each worker repeatedly picks up work from a deque of lightweight tasks using scheduling policies that are designed to achieve good load balance while bounding the size of the deques. This approach has been shown to yield scalability that is orders-of-magnitude superior to the scalability achieved if each task were to be created as a thread at the OS level.

However, there are still significant overheads that remain in a software-only approach, that will likely prevent it from being usable at Extreme Scale. These overheads involve locking

operations, and in the case of nonblocking algorithms involve spin loops on shared-memory locations with their accompanying cache consistency overheads. As mentioned earlier, these overheads are especially important 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 still retaining the flexibility of task scheduling policies in software. Section 5.4 identifies other uses for hardware support for shared queues in Extreme Scale systems.

Another source of overhead in task scheduling lies in the operations that need to be performed on the fast path to save local variables, so as to ensure that the task can be resumed on a separate worker from the one that it started on (if needed). A software-only approach introduces word-at-a-time store instructions to save the local variables, and some of these stores are often redundant. In contrast, hardware support for saving and restoring local variables (as in calling conventions) can help reduce this overhead that occurs on the fast path.

### 5.2. 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, which is one mechanism that can be used in support of locality optimization. As observed throughout this report, it will be critical to optimize vertical locality so as to satisfy the energy constraints of Extreme Scale systems. Runtime systems for programming languages such as UPC [9] and Co-Array Fortran [31] that are based on a *Partitioned Global Address Space* (PGAS) 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* [7] and X10's *places* [6], so as to enable tasks to be shipped to data, data to be shipped to tasks, or any meet-in-the-middle combination thereof. The translation from global to local addresses is a major source of overhead in a software-only approach to implementing such languages, along with the communications that accompany non-local accesses. Thus, it becomes important for a compiler for such languages to perform redundancy elimination on address computations, to coalesce contiguous accesses into a single communication operation, and to overlap communication with computation [41]. Opportunities for software-hardware co-design 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.

### 5.3. Collective and point-to-point synchronization with dynamic parallelism

As discussed in Section 3.2, 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 [34]. Further, it is usually desirable to augment the synchronization structures with communication for reductions [35], collectives, and systolic computations. As mentioned in Section 5.1, synchronization structures represent good candidates for software-hardware co-design since a software-only approaches for synchronization suffer from unnecessary cache consistency and serialization bottlenecks. Hardware support (*e.g.*, in the form of counting semaphores) can be used to reduce the overhead of inter-core synchronization, and extensions in the form of register-level inter-core communication (*e.g.*, as in the Raw project [25]) can reduce the overhead of communication. Further, 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.

#### 5.4. 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 block when they request the result (if the result is not ready). In the case of *futures* [16], 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. A classical example of hardware support in this area is the *full-empty bit*, but there may be many other variations. Also, in many cases, the location on which the producers and consumers wish to synchronize may be designated by a tag rather than an address. Hardware support to accelerate the translation of tags to addresses can be very useful. Intel's Concurrent Collections (CnC) [5] is an example of a high-level programming model that relies heavily on producer-consumer parallelism and that would benefit greatly from any hardware support.

## 6. Summary

There are several reasons for paying attention to software in the development of Extreme Scale systems. First, the Exascale systems that are projected for the 2015 – 2020 timeframe are dramatically different from today's Petascale systems and will require correspondingly fundamental changes in the execution model and structure of system software (both of which have remained relatively stagnant during the last two decades). 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 (OS, 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 1000× increase in parallelism to a single job with the energy efficiency and strong scaling fraction necessary for Extreme Scale systems.

To better understand the software challenges for Extreme Scale systems, we studied the challenges and implications in developing applications for Extreme Scale computing by examining multiple application classes (Section 2). From an application viewpoint, the concurrency and energy challenges boil down to the ability to express and manage parallelism and locality in the applications. This section concluded that applications can be enabled for exploiting extreme scale hardware by exploring a range of *strong scaling* and *new-era weak scaling* techniques, but only with suitable attention to efficient parallelism and locality.

Given this context, Section 3 summarized the 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. The challenges in *managing* parallelism and locality were discussed next in Section 4. OS-related challenges include *parallel scalability*, *spatial partitioning* of OS 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. The section also outlined challenges in *compilers* for Extreme Scale systems. Finally, Section 5 identified opportunities for addressing the concurrency and energy challenges through *software-hardware co-design*.

## 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 of the study 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 will be forthcoming in the near future.

The applications discussion in this paper also benefited from numerous meetings held in 2007 to investigate future Exascale computing applications and hardware/software requirements. These included three DOE Exascale Townhall Meetings held at Lawrence Berkeley National Laboratory, Oak Ridge National Laboratory and Argonne National Laboratory, 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 the Defense Advanced Research Projects Agency or the Department of Defense.

## References

- [1] Eric Allen et al. Project Fortress: A multicore language for multicore processors. *Linux Magazine*, pages 38–43, 2007.
- [2] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ICFP '96: Proceedings of the first ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, New York, NY, USA, 1996. ACM.
- [3] R. D. Blumofe et al. CILK: An efficient multithreaded runtime system. *Proceedings of Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 207–216, July 1995.
- [4] Ian Buck et al. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.
- [5] Zoran Budimlić et al. Multi-core Implementations of the Concurrent Collections Programming Model. In *CPC '09: 14th International Workshop on Compilers for Parallel Computers*. Springer, January 2009.
- [6] Philippe Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005.
- [7] Cray Inc. The Chapel language specification version 0.4. Technical report, Cray Inc., February 2005.
- [8] Jack B. Dennis. Data Flow Supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.
- [9] Tarek El-Ghazawi et al. UPC Language Specification v1.1.1, October 2003.
- [10] Matteo Frigo et al. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.
- [11] Anwar Ghuloum. Ct: channelling NeSL and SISAL in C++. In *CUFP '07: Proceedings of the 4th ACM SIGPLAN workshop on Commercial users of functional programming*, pages 1–3, New York, NY, USA, 2007. ACM.
- [12] Michael I. Gordon et al. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, New York, NY, USA, 2002. ACM.
- [13] Yi Guo et al. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, May 2009.
- [14] Francois Gygi et al. Large-scale electronic structure calculations of high-z metals on the bluegene/l platform. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 45, New York, NY, USA, 2006. ACM.
- [15] Mary Hall et al. Compiler research: the next 50 years. *Commun. ACM*, 52(2):60–67, 2009.
- [16] Robert Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions of Programming Languages and Systems*, 7(4):501–538, October 1985.

- [17] Kenneth Iverson. *A Programming Language*. John Wiley and Sons, New York, NY, 1962.
- [18] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [19] Removing the Big Kernel Lock. <http://kerneltrap.org/Linux/Removing.the.Big.Kernel.Lock>, May 2008. Viewed on September 27, 2008.
- [20] Michael Kistler et al. Petascale computing with accelerators. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 241–250, New York, NY, USA, 2009. ACM.
- [21] Peter Kogge et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. [http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/exascale\\_final\\_report.100208.pdf](http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/exascale_final_report.100208.pdf), 2008.
- [22] Mike Kravetz and Hubertus Franke. Implementation of a multi-queue scheduler for linux. <http://lse.sourceforge.net/scheduling/mq1.html>, 2001.
- [23] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.
- [24] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
- [25] Walter Lee et al. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, October 1998.
- [26] Tong Li et al. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.
- [27] J. McGraw et al. SISAL: Streams and Iteration in a Single Assignment Language Reference Manual Version 1.2. Technical report, Lawrence Livermore National Laboratory, 1985. No. M-146, Rev. 1.
- [28] Paul E. McKenney and Jonathan Walpole. Introducing technology into the Linux kernel: a case study. *SIGOPS Oper. Syst. Rev.*, 42(5):4–17, 2008.
- [29] M. Metcalf and J. Reid. *Fortran 90 Explained*. Oxford Science Publications, Oxford, England, 1990.
- [30] E. Mohr et al. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.
- [31] Robert W. Numrich and John Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum Archive*, 17:1–31, August 1998.
- [32] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.
- [33] Vivek Sarkar. Automatic Partitioning of a Program Dependence Graph into Parallel Tasks. *IBM Journal of Research and Development*, 35(5/6), 1991.
- [34] Jun Shirako et al. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.
- [35] Jun Shirako et al. Phaser accumulators: a new reduction construct for dynamic parallelism. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, May 2009.
- [36] Jun Shirako et al. Chunking parallel loops in the presence of synchronization. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*. ACM, June 2009.
- [37] S. Skedzielewski and J. Glauert. IF1 – An Intermediate Form for Applicative Languages. Technical report, Lawrence Livermore National Laboratory, 1985. No. M-170.
- [38] Guy Steele. The Future Is Parallel: What’s a Programmer to Do? Breaking Sequential Habits of Thought. One-hour talk presented at the 5 March 2009 meeting of the New England Programming Languages and Systems (NEPLS) Symposium at Mitre. <http://research.sun.com/projects/plrg/Publications/NEPLSMarch2009Steele.pdf>, 2009.
- [39] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
- [40] Bryan Veal and Annie Foong. Performance scalability of a multi-core web server. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 57–66, New York, NY, USA, 2007. ACM.
- [41] Katherine Yelick et al. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA, 2007. ACM.