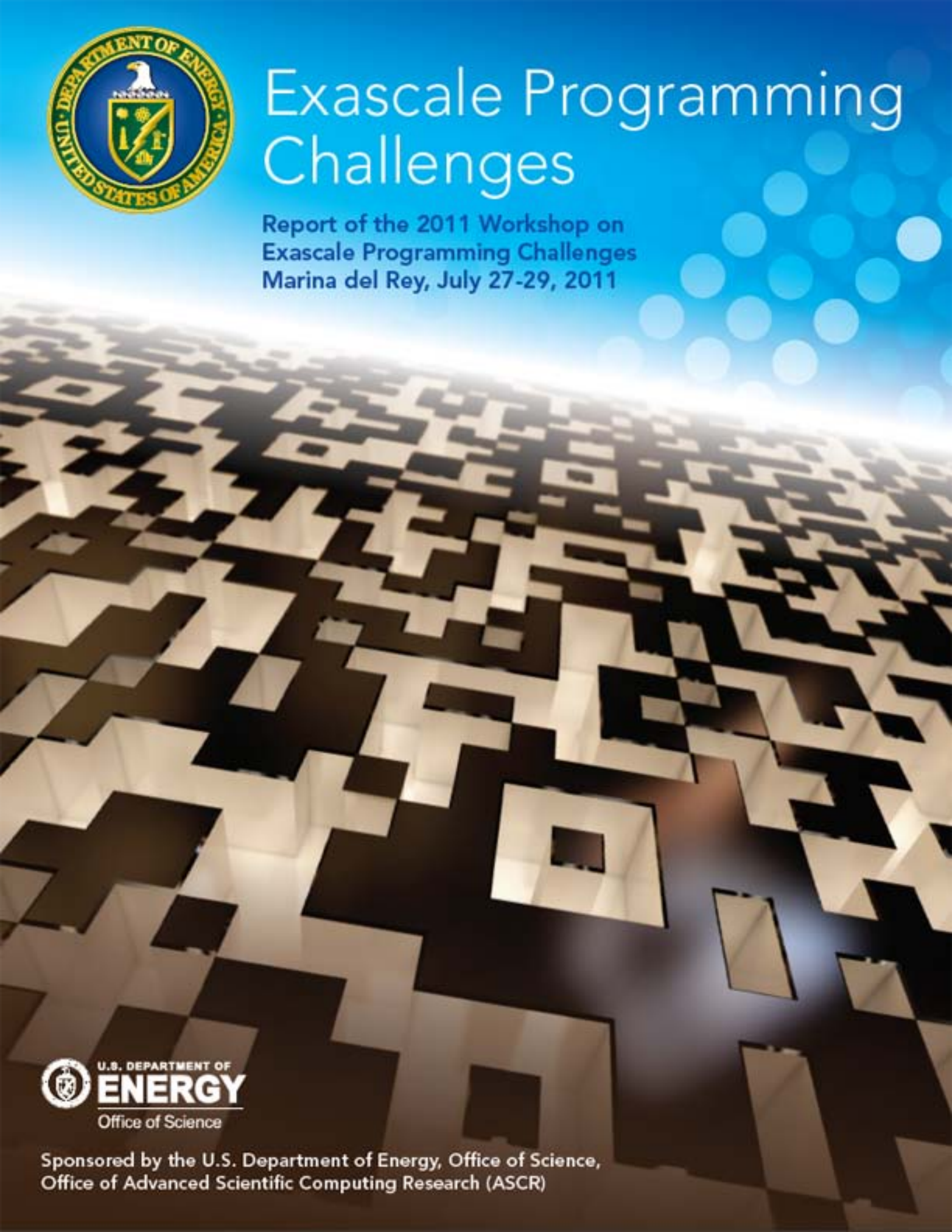

Exascale Programming Challenges
Report of the 2011 Workshop on Exascale Programming Challenges
Marina del Rey, July 27-29, 2011
U.S. DEPARTMENT OF ENERGY
Office of Science
Sponsored by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR)

ASCR Programming Challenges for Exascale Computing

July 27 – 29, 2011
University of Southern California / Information Sciences Institute (USC/ISI)
Marina del Rey, CA

Workshop Report Committee
Saman Amarasinghe (MIT)
Mary Hall (U. Utah)
Richard Lethin (Reservoir Labs)
Keshav Pingali (U. Texas-Austin)
Dan Quinlan (LLNL)
Vivek Sarkar (Rice U.)
John Shalf (LBNL)
Robert Lucas (USC/ISI)
Katherine Yelick (LBNL)

Additional Authors
Pavan Balaji (ANL)
Pedro C. Diniz (USC/ISI)
Alice Koniges (LBNL)
Marc Snir (ANL)

Report Editors
Sonia R. Sachs (ASCR)
Katherine Yelick (LBNL)

This paper reports on the results of a workshop on programming models, languages, compilers and runtime systems for exascale machines. The goal was to identify some of the challenges in each of these areas, the promising approaches that should be pursued, and measures to assess progress. The challenges derived from more complex systems with additional levels of memory, orders of magnitude more concurrency, and with less memory, memory bandwidth, and network bandwidth per core than today's petascale machines. Additional problems arise from heterogeneous processor architectures or accelerators, and the possible need for power and resilience management. The workshop participants considered a range of approaches, from evolutionary ones based on existing programming techniques to revolutionary ones that changed the basic concepts for creating and managing parallelism and locality. They also considered models that separated the intra and inter-node parallelization strategies and agreed that while there were challenges in both, the more daunting ones were within the node where most of the hardware disruptions would occur.

As described in this paper the participants in the workshop articulated the research challenges in programming support for anticipated exascale systems, including specifying what is known and what remains uncertain.

Contents

Executive Summary

For application programmers, meeting the exascale challenge will require rethinking the algorithms and software used in HPC applications. Because clock speeds are expected to remain relatively constant, exascale systems are expected to have approximately 3-5 orders of magnitude more concurrency than on current petascale platforms. Because the available memory will not scale by the same factor, machines cannot be exploited simply by weak scaling of applications that benefit from larger problem sizes as in the terascale and petascale eras. New algorithms will be required that can exploit vastly more parallelism than existing algorithms without requiring the same order of magnitude more memory. A further complication is that exascale systems will be energy constrained, so reducing power consumption will be a paramount concern. Much of the power required to execute an application will be consumed in data movement, so new parallel algorithms with improved locality of reference are required.

In the terascale to petascale transition, HPC machines used a consistent execution model as a network of serial processors, with most of the performance increases coming from single processor performance increases and larger system scale. While there was extensive performance tuning with some systems, they did not involve major algorithmic changes or code restructuring. With the significant increase in complexity of exascale platforms due to energy-constrained billion-way parallelism, and most of those changes happening within the node architecture, this approach is infeasible. We must investigate new abstractions for machines and their execution models. The workshop participants therefore considered many different approaches to exascale programming, from evolutionary ones based on the MPI+Fortran/C/C++ model to revolutionary ones that change the underlying execution model along with the programming interface.

For the evolutionary approaches, MPI can be used as the software substrate for inter-node communication, but will need to be extended to support resilience, increasing importance of topology-aware communication, and to take advantage of integration of networks onto processor chips. Even more significant programming challenges will focus on intra-node parallelism approaches and how to integrate that parallelism with MPI communication. Although MPI is still popular today within multicore compute nodes, the shrinking memory space per core and the likelihood of heterogeneous compute nodes with accelerators make MPI impractical as the intra-node programming model for exascale. There may be no evolutionary path to an intra-node programming model: popular models based on serial programming with OpenMP annotations will require significant breakthroughs in compilers, runtimes and the annotation languages to support mapping onto heterogeneous nodes with more explicit levels of memory; if entirely new models are developed or adopted from other fields such as graphics, this may greatly reduce the implementation challenges, but will shift cost and risk to application software. A more revolutionary approach takes advantage of the hardware disruptions to rethink the entire software substrate and has the potential to simplify programming by having a superior integration of the parallelism within and between nodes. This approach can take advantage of global addressability across the machine along with unified abstractions for handling hierarchies of parallelism and locality. There should be a natural migration path for applications from the evolutionary to revolutionary approaches, as inter-node communication becomes more tightly integrated and unified with intra-node parallelism.

Many scientific applications currently hide some of the parallelism within libraries and application-specific frameworks, so that application programmers do not use MPI directly. These software layers will continue to be important on exascale machines and may help ease the transition to systems that require resilience, increased network and memory hierarchies, or even new languages and new forms of parallelism. Expanding the historical notion of programming language implementation from statically compiled languages and libraries to more interactive and dynamic ones that use automatic

performance tuning, runtime optimization, and programmer feedback, will also be important to address the performance and productivity challenges. Other tools for correctness and performance analysis and debugging are also important part of the exascale programming environment, but were not addressed by this workshop.

The current model of MPI+FORTRAN/C/C++ that has sustained HPC application software development for the past decade is likely to be inadequate for exascale era machines, and it is important that new programming techniques are usable across machine scales and are portable across multiple machine generations. Research investments need to address these aspects of scalability and portability, as well as offering a transition path for applications through interoperability and multi-language support. Systems software designers need to rethink programming models, compilers and runtime systems for this new era of fine-grained parallelism, strong scaling, and resilience on heterogeneous and hierarchical hardware. They need to offer programming techniques that will be productive and performant across many machine generations, and across current and emerging application domains.

1 Introduction

This report summarizes discussions conducted at the DoE Workshop on Exascale Programming Challenges held during July 27-29, 2011 in Marina del Rey, California. The goals for the workshop were as follows [24]:

1. Define objective criteria for assessing programming models, language features, compilers, and runtime systems for exascale systems, and metrics for their success.
2. Prioritize programming model, language, and compiler challenges for exascale systems.
3. Prioritize options for (i) evolutionary solutions, (ii) revolutionary solutions, and (iii) bridging the gap between evolutionary and revolutionary solutions.
4. Lay out a roadmap, with options, timeline, and rough cost estimates for programming Exascale systems that are responsive to the needs of applications and future architectural constraints.

To that end, the workshop agenda included 24 presentations by leading experts in parallel computing from DoE laboratories, academia, and industry, six focused panel discussions in break-out groups, and two general panel discussions. A brief summary of the workshop presentations is included below, with a focus on past failures and successes highlighted in these talks as well as recommendations for future directions. The remainder of the report dives deeper into the technical details of these topics and also discusses their strategic implications in a broader context.

1.1 Application Requirements

Given the workshop's focus on programming challenges for exascale systems, it was appropriate to start with a session on application requirements. First, it was observed that past changes in parallel computing hardware have all relied on *programmer intervention* [36]. Examples of programmer intervention for parallel computing include the use of MPI for distributed-memory parallelism, OpenMP directives for SMP parallelism, SSE directives for SIMD parallelism[1], and new variants of C (notably, CUDA and OpenCL) for GPU parallelism. Severe energy constraints will require applications to exploit multiple degrees of heterogeneity while minimizing data movement, and current approaches to these requirements also require significant programmer intervention. One area in which it is hoped that programmer intervention will be minimal is resilience. With hardware approaches such as ECC available for memory and networks, standard checkpointing techniques may suffice for processor-level fault tolerance. However, given the tight energy budget for exascale systems, some application groups may be willing to trade-off resilience for energy/performance benefits by tolerating the presence of certain levels of error thresholds.

Second, a report on experiences with five applications highlighted promising approaches [32] on the path to exascale — GTS, PIR3D, fvCAM, S3D and Computational Screening of Carbon Capture Materials. The first three applications highlighted the importance of *hybridization* of inter-node MPI/CAF parallelism and intra-node OpenMP parallelism. For GTS, the classical approach of only performing MPI communications in OpenMP serial regions was shown to be inferior to approaches in which inter-node communications are performed by OpenMP threads in parallel with other computational/communicating OpenMP threads. The experience with PIR3D reinforced the benefits of hybridization, and demonstrated that the best choice of number of cores

[1] Given past successes in using vectorizing compilers for vector processors, it is reasonable to expect that compilers can help reduce the programmer intervention burden for SIMD parallelism (Section **??**).

per process to use for OpenMP parallelism can vary significantly depending on the application, platform, and the scale of the run. The experience with fvCAM showed that there may be cases when the best choice for performance is to just use one core per MPI process, but the overall memory requirement was significantly reduced when multiple cores were used within a single MPI process; these *memory benefits* of hybridization due to strong scaling in intra-process parallelism will become increasingly important as the imbalance between computation and memory continues to increase on the path to exascale. The experience with the Computational Screening of Carbon Capture Materials application underscored the fact that certain applications can get a very significant boost from the use of heterogeneous accelerators; for this application, the execution time was reduced from years to weeks through the use of GPUs. Further, experience with S3D showed that currently proposed OpenMP directives for GPUs can provide both portability and performance improvements relative to hand-coded CUDA kernel invocations.

Third, an opinion shared by multiple participants [7,64] is that application developers recognize the major disruptions expected in exascale systems, and are open to the possibility of *rewriting applications for exascale*. However, if a rewrite is undertaken to a new "revolutionary" programming model, application developers need to be assured that the return on their effort can be leveraged for multiple years thereafter with at most evolutionary rewrites thereafter (which could include changes in programming mechanisms and notations, while the overall programming model remains unchanged). This goal was referred to as requiring at most "one and a half" code re-writes, as opposed to a rewrite (say) each time a new hardware platform is introduced. Further, the importance of supporting sparse irregular applications in new programming models was also highlighted, which includes support for asynchronous data movement and effective movement of non-contiguous data [7]. Another shared opinion was support for *performance transparency* such as programmer-visible maps from logical to physical views of data for locality control [7], and the ability for programmers to tune for performance, power, and resilience [64]. In addition, the importance of mini-applications was cited in validating new programming models [64].

Finally, the case for *asynchrony* was made using a Hartree-Fock theory application as an exemplar [28]. In this application, the use of synchronous constructs such as barriers and collectives significantly limited the exposed parallelism compared with the intrinsic parallelism the exists if the application is represented as a directed acyclic graph (DAG). It was recommended that a hierarchical approach be pursued for exploiting this DAG parallelism, both in terms of computation and data hierarchies. A two-fold approach was proposed to support the introduction of new languages: 1) creation of embedded DSLs, libraries, and frameworks in ways that do not require changes to the base sequential language, and 2) introduction of new general-purpose languages when they are ready to be adopted by a wide community.

1.2 Advanced programming models and languages

This state-of-the-art session summarized examples of current advanced programming models and languages whose features could be indicative of constructs that may be expected in future exascale programming models.

The case for a systematic approach to expressing fine-grained nested parallelism was made in [10]. The NESL project has shown that it is possible to express large amounts of nested parallelism in a high-level declarative language in which programmers need not worry about pernicious parallel programming bugs such as data races and deadlock. Further, this approach to specifying parallelism provides great flexibility to the implementation in scheduling parallelism at both vector/SIMD and multicore levels while also using hierarchical approaches to optimizing locality using cache-oblivious algorithms. There is a natural synergy between fine-grained nested parallelism and the collective

operations that have been used for decades by DOE application developers when using MPI and OpenMP, suggesting that this form of parallelism is a likely candidate for implementation as a new language or as an embedded domain-specific language (DSL).

A synergistic opinion in favor of functional dataflow programming was offered in [53]. The justification for revisiting this model is that achieving asynchrony with imperative semantics can be very challenging, and usually leads to over-specified ordering constraints. As explained in this presentation, there are a wide range of options to express functional semantics e.g., use of current languages in a careful way, use of libraries and frameworks, use of directives/pragmas, use of an embedded DSL with C++ templates (as in Intel's Ct model), use of new type systems, and the use of fully functional languages. Further, a clear separation of concerns can boost the productivity of different classes of programmers (e.g., domain experts, CS experts, hardware experts) when working at different levels of an application (driver implementation, solver implementation, application assembly) as shown in figure 1. There was a general recognition across multiple talks that when talking about programming models it is important to identify, which kind of programmers we are referring to. The domain experts were loosely referred to as "Joe" programmers and the computer science experts were loosely referred to as "Stephanie" programmers. Their differing levels of expertise regarding computer hardware and the scientific domain demand a different set of criteria for the programming environment that they would use. So a Joe programmer would need a more declarative programming style that emphasizes the semantics of the domain whereas a Stephanie programmer would want a more imperative programming model that provides full control over mapping of the problem to hardware architecture. The declarative style of the Joe environment provides the Stephanie programmers more freedom to rearrange computation to better map onto hierarchical system architecture. One way to provide Stephanie with the constraints for a declarative style of programming is to have the modules of computation from Joe's programming environment obey three simple rule; be stateless, only operate on the data you were given (isolation), hand control of Main() over to Stephanie. These three principles constitute functional semantics, which can be implemented in many different ways – from careful programming (shown in Heroux's talk) all the way to functional programming languages. Functional semantics provide Stephanie with the freedom to reorganize computation that she needs for optimization combined with the constraints required to preserve correctness. However, it is also well know that functional programming creates implementation challenges in memory management and scheduling. As a result, a pragmatic suggestion is to build a language that allows incremental adoption of functional semantics rather than an all-or-nothing approach.

A declarative approach to fine-grained parallelism, as in NESL or dataflow, allows programmers to "think parallel, write sequential" as advocated in [25]. Another approach to achieving this goal is to use frameworks such as Tramonto. Both approaches are well suited to the reality that the number of science experts (justifiably) far outweighs the number of parallel programming experts in DOE. However, the framework approach also simplifies integration with a cluster communication library like MPI, by supporting commonly used patterns such as halo exchanges and collectives without the application programmer having to worry about the details of their implementation in MPI. The opinion offered in [25] is that MPI can continue to suffice for inter-node communication, but significant changes are needed at the intra-node level. One suggestion in [25] to simplify intra-node parallelism is to use stateless vectorizable computational kernels as the basic building block for applications. This goal is synergistic with declarative approaches to vector parallelism and dag parallelism. Further, the approach suggested in [25] is to provide domain scientists with a rich "palette" of intra-node parallel primitives to choose from. Examples of these primitives include forall parallel loops, reductions, pipeline filters,thread teams, and new storage layouts. Metaprogramming and composability are viewed as two requirements for integrating these primitives in

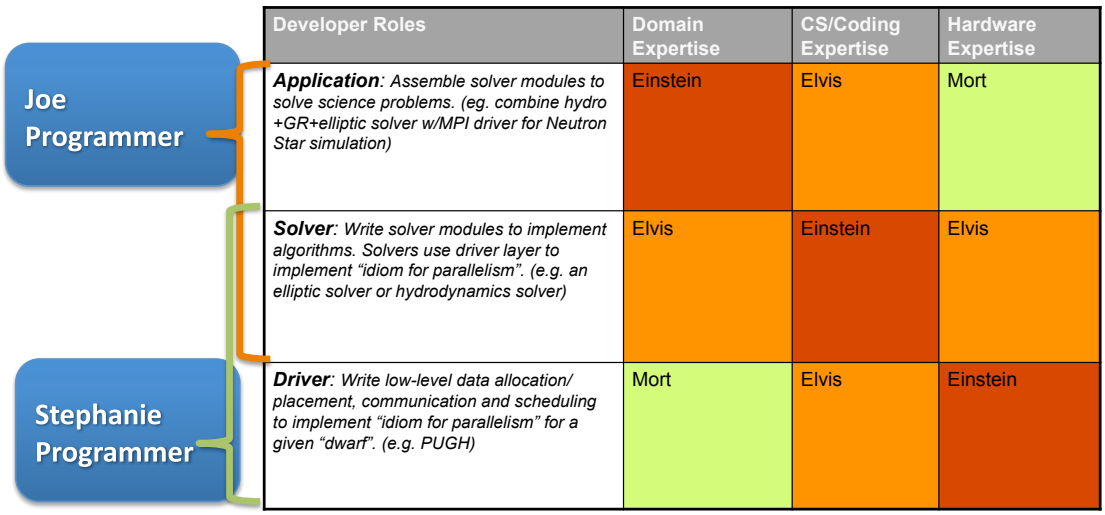


Figure 1: This diagram shows the stratification of developer roles and expertise on a complex code project. Domain experts were referred to as "Joe" programmers whereas the computer science experts were referred to as "Stephanie" programmers as shown on the left. The matrix on the right represents the expertise of these respective programmers in the areas of the science domain, numerical algorithms, and the low level details of computer architecture and tuning. A person is an "einstein" if they are an expert, an "elvis" who is conversant in the area, and an "mort" who is a non-expert. The programming environment offered to Joe may require a different set of criteria than the environment offered to Stephanie.

a standard language. In the opinion of [25], PGAS languages such as CAF and UPC are "too little, too late" at this stage, and that new programming environments have to be standardized to ensure that they get full vendor support. On a related note, [15] identified five "anti-patterns" in HPC programming models — exposing platform details in algorithms, lack of portability, lack of migration paths, lack of applicability to smaller scale problems and ignoring the development of tool components such as debuggers performance tools, and storage systems.

Vis-a-vis debugging, a case for higher levels of programming with logical invariants and algorithmic debugging was made in [51] and [42]. The concurrent constraint programming approach advocated in [51] supports deterministic "tell" and "ask" operations instead of standard imperative read/write operations on shared data. The Orc language advocated in [42] includes "powerlists" as a core data structure for synchronous parallelism enabling declarative specifications of algorithms such as FFT. Both approaches also offer the possibility of expressing formal specifications of algorithms.

Another trend in the talks presented at the workshop was the emphasis on different dimensions of *variability* that can be expected in future exascale systems and the challenges they impose on programming models. These include architectural variability, application execution variability, algorithmic variability, and application programmer variability [33]. Sources of architectural variability include multi-threading, memory hierarchies, heterogeneity, and power management. Sources of application execution variability include strong vs. weak scaling, stand-alone execution or execution in the context of another calculation, as well as the degree of coupling with other applications. Sources of algorithmic variability include new coupling of existing components (e.g.,

direct vs. iterative solvers), reformulation of existing algorithms (e.g., factorized representation of a specific input operator), and new algorithms (e.g., low-order methods with increased sparsity). Another source of application variability is related to earlier comments regarding the separation of roles and concerns. For example, different abstractions are needed by infrastructure developers and the implementers of computational methods. An evolutionary approach was advocated in light of these variabilities, in which programming models are designed to be inter-operable e.g., a phase-based execution that is capable of dynamically switching between an SPMD model and a PGAS model. A number of technologies can support this interoperability including the use of DSLs for key primitives, automatic generation of glue code, auto-tuning, and intelligent/adaptive runtime systems.

The variability theme was reinforced by the viewpoint of [37] that there will always be a degree of uncertainty in the global state of an exascale computer, so an "unintelligent design" approach should instead be pursued in which the exascale computer is a resilient self-aware system and global state emerges from local behavior. A proof-of-concept of such a self-organizing system that was cited was past work on "paintable computers" as motivation to apply the same principle to arrays of tiled processing elements. A key software implication is that the execution model must include dynamic distributed tasks that can be replicated for fault recovery. This in turn requires the programming model to support ensembles of tasks interacting through distributed data structures. While there is past work (such as global arrays and tuple spaces) on such programming model approaches, a big open problem is how to create new resilient algorithms for these forms of self-aware systems.

Finally, as a counterpoint, an opinion was offered from the hardware perspective that new programming models are a necessary part of the codesign needed to perform a system-wide power optimization for exascale [25]. Further, the two main tests of a programming model are: 1) how well does it match the application writer's needs?, and 2) can the hardware implement it in a performant, energy-efficient fashion? An additional challenge is that exascale hardware designers need to know what the key elements of the programming model are by 2015, so that the hardware can be designed accordingly. This talk highlighted key elements that need to be specified in an execution model (concurrency, coordination, movement of data/computation, naming, introspection) and summarized the ParalleX model as an exemplar. It emphasized that the current SPMD approach in MPI is not tenable for an exascale execution model because of its high energy requirements. Thus, MPI+X is not the right question to be asking. Instead, the focus needs to be on small data transfers for both asynchrony, energy efficiency, and high message rates. The talk also advocated data-oriented synchronization, which was synergistic with earlier discussions of dataflow computing.

1.3 Language constructs, compiler and runtime engines

Another state-of-the-art session summarized the past and future of implementations of programming models in both compilers and runtime systems.

Two presentations covered a similar theme of previous successes and failures of the compiler community's support of high-performance computing, and described a path forward. The first presentation counted instruction-level parallelism and memory hierarchy optimization as major compiler successes for the high-performance computing community, the former benefitting from speculation and branch prediction, and the latter from techniques for loop transformation and polyhedral frameworks [46]. A unifying aspect of these two optimization areas is that they rely on the compiler to lower abstraction of the application code to target low-level hardware features. The second presentation similarly included register allocation and instruction-level parallelism as key compiler successes for high-performance computing, and described compiler contributions to

programmer productivity in general-purpose computing, such as expression and construction of codes in higher-level languages, elimination of many classes of bugs and portability [1]. Both presentations declared that automatic parallelization of dusty decks has failed, and had similar arguments for why this is not a realistic goal. Identifying parallel constructs from sequential specification requires that the compiler raise the abstraction of the code from what the programmer has written, resulting in a heroic compiler that is complex and fragile. A proposed path forward is to rely on the programmer to specify information that is impossible for a compiler to automate, such as available parallelism, data partitioning, and algorithmic choices. The compiler can then automate the mapping of this specification to different architectures, perhaps by evaluating a space of different implementation possibilities proposed by the programmer. Runtime support must then work in collaboration with the compiler to finalize decisions based on runtime knowledge. Both talks recognized that a system must support multiple levels of parallel programming expertise – both the "Stephanie programmers" that are performance experts, and the "Joe programmers" that are not (such as, for example, the domain scientists). Both speakers also point out that an evolutionary approach that is based on existing programming models such as MPI and OpenMP will be insufficient, as they do not support well the proposed features.

A similar set of themes was the topic of a talk on programming model requirements for exascale, with a cautionary look backward at High Performance Fortran, which engaged the best efforts of the community during its time but was never able to garner widespread acceptance [38]. From HPF, we learned that proper data partitioning and optimizing communication were critical to scalability, and that single-processor efficiency must be competitive with serial implementations. A challenge of HPF for savvy developers was that it tried to do too much behind the scenes, and did not give programmers enough control to hide or avoid latency. This talk proposed a unified programming model for exascale, rather than a hybrid programming model that combines MPI for domain decomposition and a threading language for each socket (often called "MPI + X"). The talk proposed partitioned global address space (PGAS) languages as the unified programming model. Compilers for PGAS, and particularly for CoArray Fortan, can provide the support to communicate across address spaces, and compiler support to optimize these operations and interface to runtime layer. The Cilk runtime was also proposed as an exemplar of how to manage plentiful fine-grain parallelism on a socket. This presentation concluded with an argument that the HPC world moves slowly, so a new programming model involves working with language standards committees to make changes to existing languages.

Another presentation echoed the theme of the need to extend existing languages rather than invent new ones [48]. The reasons included user training, existing code base and tool availability. In this talk, the proposed solution for exascale was to provide a source-to-source compiler toolkit for building domain-specific programming models. In this way, a set of users (perhaps "Stephanie programmers") could encapsulate abstractions with well-defined semantics into implementations in the compiler and runtime to support domain scientists ("Joe programmers"). Such support could also address new exascale challenges such as offloading work to accelerators, improving application resilience, and energy management.

Another presentation expanded on the theme of new compiler technology needed to address exascale challenges [34]. The talk pointed out that we should not expect the numerous new challenges of exascale to be addressed by programmers or libraries, and that compiler support was essential to managing complexity. The presentation advocated the use of high-level, semantics-rich programming approaches that are separated from architecture tuning languages such as is done, for example, in Concurrent Collections (CnC). Compiler support then automates transformations of code from their semantic description to their implementation on a specific architecture. As a driver for this sort of approach, could the system automatically check a proof that an application's

implementation is correct with respect to dynamism, faults, precision? This resilience challenge could be used as a litmus test for the programming language, annotation system, tools, runtime, and hardware.

The remainder of the presentations in this session addressed runtime mechanisms and their support in the programming models and compilers. One runtime presentation advocated virtualization of processor resources in the application program, creating abundant work, dynamic parallelism and association of data with an affinity to the virtualized parallel constructs [29]. This concept was called "work-unit, data-unit" or WUDU. Scheduling would be performed dynamically, and may involve reducing parallelism and migrating data at runtime to map more efficiently to hardware.

On the topic of migrating to new programming models, one presentation argued that it was essential for new models to interoperate with existing programming models and runtime systems to offer a smooth migration path [5]. Interoperability was defined as being able to use data from one model in a different model, with safe use of data buffers and the ability to use data objects on other models as examples. Resource management and tool availability were cited as other barriers to interoperability.

Related to interoperability, another presentation described four key intermediate-level parallel programming constructs that would require support in programming models, or at least extensions to evolve current runtime systems: (1) asynchronous tasks and data transfers; (2) collective and point-to-point synchronization and reductions; (3) mutual exclusion; and, (4) locality control for task and data distribution [52]. The talk described language support for these concepts in X10 and Habanero. The compiler should be involved in supporting these constructs for a scalable implementation, while evolutionary solutions would focus on runtime support.

2 Hardware Challenges

Over the past forty years, progress in supercomputing has benefitted from improvements in integrated circuit scaling according to Moores law, which has yielded exponential improvements in peak system-level floating-point performance. HPC system peak floating point performance (an indirect measure of system capability) has consistently increased by a factor of 1000x every 11 years. Moore's law has supplied 100x of that improvement, but the extra 10x has been delivered through innovations that are specific to the leading-edge HPC architectural space.

Unfortunately, for the first time in decades, the advances in computing technology are now threatened, because while transistor density on silicon is projected to increase with Moores Law, the energy efficiency of CMOS logic in silicon is not. Power has rapidly become the leading design constraint for future HPC systems systems. Numerous studies conducted by DOE-ASCR [58], DOE-NNSA [59] and DARPA [19] have concluded that given these new constraints, the current approach to designing leading-edge HPC systems is unsustainable as it would lead to machines consuming hundreds of megawatts. New approaches will not emerge from evolutionary changes in processor speed and scale from todays petascale systems, but will require fundamental breakthroughs in hardware technology, programming models, algorithms, and software at both the system and application level.

The primary design constraint for future HPC systems will be power consumption, and the cost of data movement has overtaken the energy cost of a floating point operation. Consequently, management of data locality is key to future scaling of application performance and energy efficiency. Programming models, algorithms, and applications must evolve to overcome these disruptive changes in hardware characteristics. This section provides a short overview of how the emerging design constraints motivate changes in the construction of future programming models.

2.1 Power as a Leading Design Constraint

Since 1986 performance has improved by 52 percent per year with remarkable consistency. During that period, as process geometries scaled according to Moore's law, the active capacitance of circuits scaled down accordingly. This effect is referred to as Dennard scaling after the theory advanced by Robert Dennard of IBM Research in 1974 [50]. As a consequence of Dennard scaling, supply voltages could be kept constant or even dropped modestly in order to allow manufacturers to increase clock-speeds. This application of the Dennard scaling parameters, known as *constant electric field frequency scaling* [13] fed the relentless increases in CMOS CPU clock-rates over the past decade and a half. However, below the 90nm scale for silicon lithography, this technique began to hit its limits. With the end of Dennard scaling, power density has now become the dominant constraint in the design of new processing elements, and ultimately limits clock-frequency growth for future microprocessors. Energy efficiency of all design choices has become the leading order constraint on the future system architectures.

The direct result of power constraints has been a stall in clock frequency that is reflected in the flattening of the performance growth rates starting in 2002 as shown in Figure 2. In 2011, individual processor cores are nearly a factor of ten times slower than if progress had continued at the historical rate of the preceding decade. Other approaches for extracting more performance such as Instruction Level Parallelism (ILP) and out-of-order instruction processing have also delivered diminishing returns as shown in Figure 2.

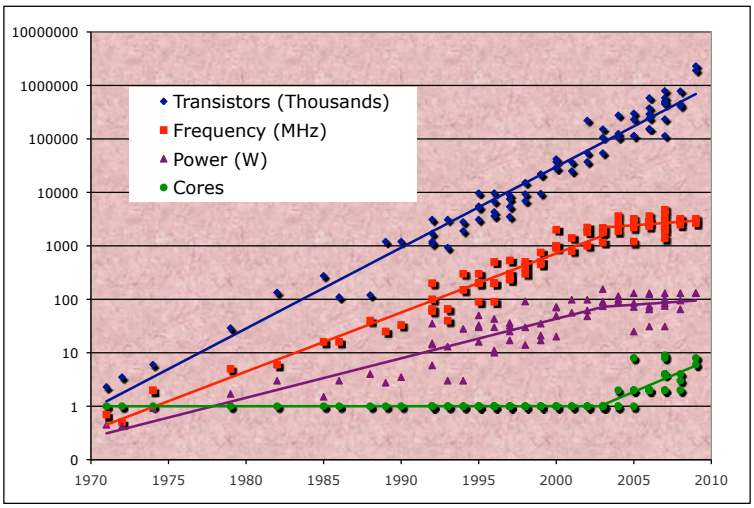


Figure 2: This graph shows that Moores law is alive and well, but the traditional sources of performance improvements (ILP and clock frequencies) have all been flattening. The performance of processors as measured by SpecINT has grown 52% per year with remarkable consistency, but improvements tapered off around 2003 due to the end of Dennard scaling rules. (*This figure is based on original data from Kunle Olukotun and Herb Sutter, but updated with more recent data.*)

2.2 Parallelism

With growth in clock frequency stalled, performance growth is new being achieved by an exponential growth in the number of processing elements per chip and growing hardware threading per core. Having exhausted other well-understood avenues to extract more performance from a uniprocessor, the mainstream microprocessor industry has responded by halting further improvements in clock frequency and increasing the number of cores on the chip. The number of "cores" or explicitly parallel computational elements per chip is likely to double every 18-24 months henceforth. These trends motivate new programming abstractions that virtualize the notion of a core (implicit parallelism) and threading APIs with expanded semantics for thread control, placement, launching, and synchronization as well as scalable runtimes to manage massive numbers of threads.

2.3 Locality

Systems today, and those anticipated in the future, are increasingly bound by their communications infrastructure and the power dissipation associated with high-bandwidth information exchange across the rapidly growing number of computing nodes. According to DOE and DARPA projections, the energy efficiency of a FLOP in 2018 improves by a factor of $5x$ to $10x$ from an equivalent 2008 FPU [19, 54], but the energy required to move data is NOT improving at a corresponding rate as shown in figure 5. The cost of moving data over conventional copper wires is proportional to the product of the bitrate and the distance traversed, which leads to highly localized bandwidth [6, 40, 41]. Whereas our existing programming models embody a notion that communication costs between cores on-node are equal and that MPI communication between nodes is equal cost, the emerging machine architectures are increasingly non-uniform. Management of data locality has become a first order concern. This motivates adding features to the hardware design and to the programming

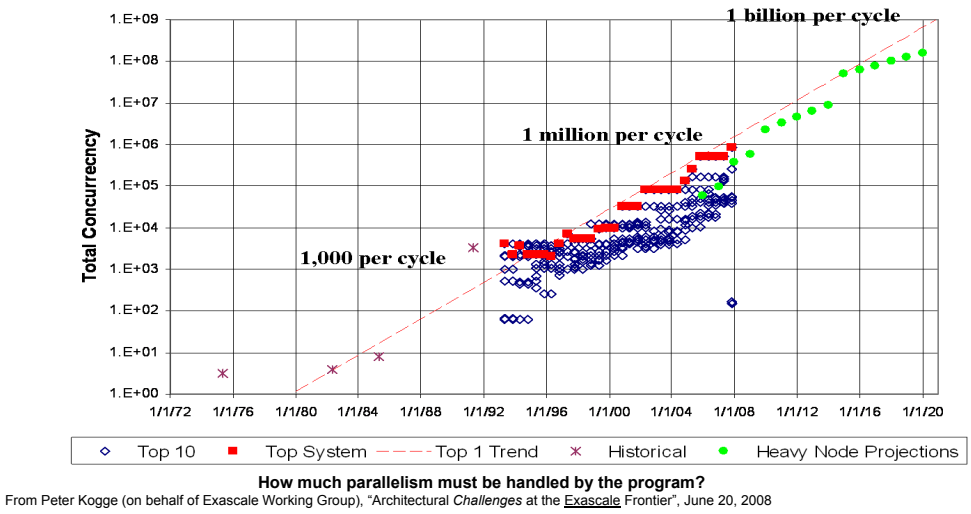


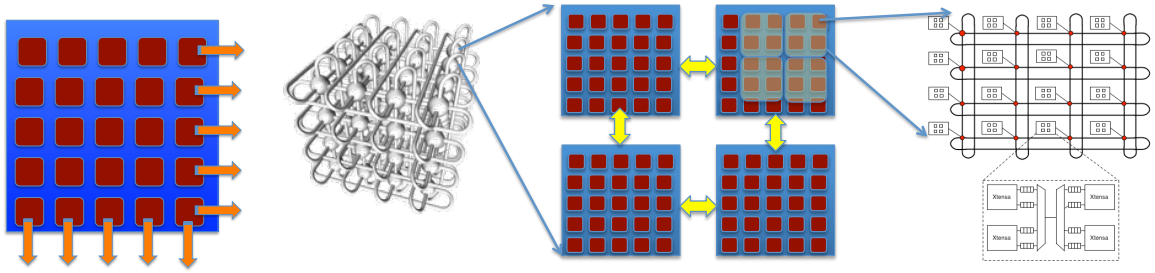
Figure 3: With flat clock rates, all performance improvements must come from parallelism. Exascale computing systems are anticipated to contain millions or even billions of FPUs (*source DARPA Exascale Report [19]*)

environment that conserve bandwidth (both on-chip and off chip) by carefully managing data movement to stay within practical thermal limits. Programming tools, execution models, and their requisite runtime systems must become aware of data locality to conserve available bandwidth and reduce power consumption. Diminished off-chip bandwidth (both memory bandwidth and interconnect bandwidth) motivates execution model that offers more explicit management of both vertical and horizontal data locality to conserve bandwidth. This also motivates an execution model that allows computation migration, i.e., migrating an executing thread from one core to another; in many cases execution migration can reduce the number of bits that need to be moved across the chip.

Vertical data movement is defined as the communication of data through the memory hierarchy (from memory to processing element and back). Existing automatically managed caches virtualize the notion of vertical data locality because it is convenient for our current programming models, but this approach will be very difficult to scale efficiently because it restricts the ability of programmers to make energy efficient or bandwidth conservative choices for data locality. Software controlled memories (such as STI Cell Local Stores) are emerging as an effective approach to explicitly manage data movement. The STI Cell has demonstrated huge benefits of using software-managed memory to explicitly control data movement [66,67], but it has also demonstrated the pitfalls of the approach because the local-stores are difficult to program.

Horizontal data movement refers to communication between peer computing elements in a system. Whereas existing programming models have evolved with an assumption of uniformity of data access performance between peer computing elements within a node or between nodes in a system, the locality-dependence of data communication is breaking that assumption. By 2018 chips are anticipated to have hundreds or even thousands of processing elements per chip. However, even with aggressive chip stacking (8-layers), the communication fabric that connects these processing elements together will need to scale out in a primarily 2-D planar geometry as shown in Figure 4 from the 2009 DOE Exascale Hardware Workshop [54, 61]. It is increasingly impractical to offer an interconnect topology with uniform bandwidth and latency between all cores and memories

within a node due to the wiring constraints, especially the superlinear wiring complexity and power consumption for a crossbar. By the end of the decade, the cost of moving data a mere 20mm across the surface of a chip will exceed that of performing a floating point operation (FLOP) as shown in Figure 5.



(a) 2Dplanar (b) HierarchicalMachine

Figure 4: As the number of computational elements per chip scale to hundreds of endpoints, Networks-on-chip are expected to adopt constrained topologies (meshes or torus) as they scale out a 2D planar geometry with hundreds of processing elements [54, 61]. The abstract machine model may need to include notions of hierarchy or even topology for effective management of horizontal data locality.

The topological constraints for horizontal data locality management together with the bandwidth locality constraints currently manifest themselves as NUMA (non-uniform memory access) behavior. We have witnessed a exponential growth rate in the quantity and performance disparity of NUMA domains for existing petascale node architectures. This trend is expected to continue at an exponential pace through the decade, with the Cray/AMD systems going from 1 NUMA domain in the first Jaguar to 2 NUMA domains for JaguarPF, and 4 NUMA domains in the latest NERSC-6 XE6 nodes (shown in figure 4 on the right). The on-chip fabric will increasingly resemble the topology of past system-wide networks. As the number of NUMA domains scales up, the topological connectivity between processing elements using Networks-on-chip (NoCs) will become increasingly important.

There are a number of new considerations that must be managed by programming systems as a result of locality management concerns.

System-Scale Topology Constrained interconnect topologies such as torus and dragonfly, require topology aware communication. The need to optimize the mapping of the application communication pattern onto diverse interconnection network topologies (e.g. toruii or dragonflies) motivates the need for topology-aware communication mechanisms in an execution model and for automated methods supported by runtime communication topology monitoring.

Chip-Scale Topology Increased localization of bandwidth due to cost of data movement results in exponential growth in the number of NUMA domains/chip. Networks-on-chip will require topology aware communication mechanisms. This motivates use of similar mechanisms for communication topology introspection and optimization that are required for the system-scale interconnect.

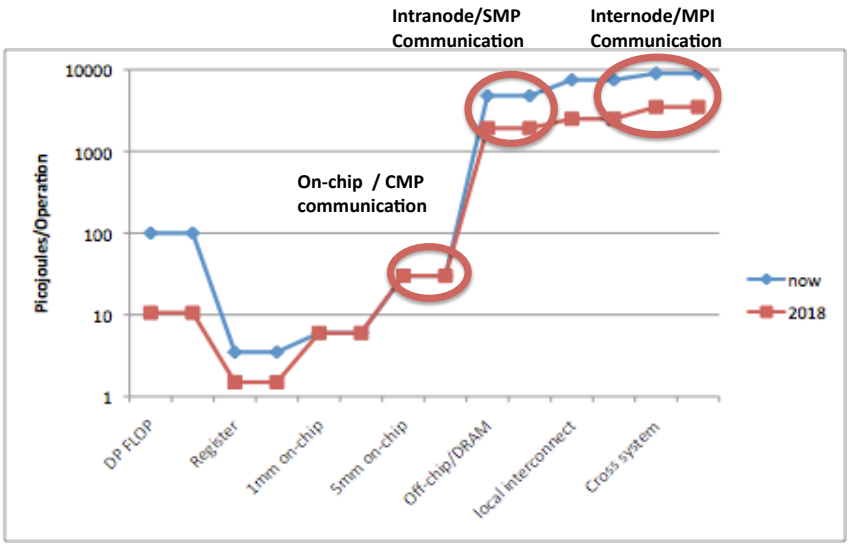


Figure 5: With new scaling rules and massive growth in parallelism, data locality is increasingly important. This diagram shows the cost of a double-precision multiply-add at different levels of the memory hierarchy (from the cost of performing the flop to the movement of data operands from registers, different distances on-chip, and distances off-chip.) *The model or FPU and register access is based on the Tensilica LX2 core energy model, the energy consumed for cross-chip wires uses the Orion2 power model which is based on Balfour's model [6], memory access is based on the JEDEC DDRx memory roadmap, and cross-system is based on projections for VCSEL-based optical transceivers.*

2.4 New Communication and Synchronization Mechanisms

Increased use of non-conventional (non-cache-coherent) methods for interprocessor communication on chip (e.g. GPU Warps and Intel SCC inter-processor hardware message queues) suggests new approaches are needed for expressing data locality so that runtime systems can manage data movement instead of increasing the burden on the application programmer.

Configurable Caches: Increased use of configurable caches (automatic vs. user-managed) to explicitly manage vertical data movement. Motivates new APIs for making effective use of these new hardware locality management features in a cross-platform manner.

Coordinated Power Management: Local power management decisions (such as Intel/Sandybridge thermal-throttling of FPU clock rates) will result in system-wide performance inhomogeneity. This motivates systemwide coordinated power management.

Energy Aware Data Movement/Load-Balancing: New sources of transient load-imbalances are increasing (such as software-based fault resilience mechanisms, and localized power management), but rebalancing the load may be more costly than tolerating it. Balancing the cost of fixing a load-imbalance vs. tolerating its continuation requires an API for expressing the energy-costs of data movement to the application and runtime.

2.5 Heterogeneity

Accelerators and heterogeneous processing offer an opportunity to greatly increase computational performance within a fixed power budget, while still retaining conventional processors to manage more general-purpose components of the computation such as operating system (OS) services. Currently, such accelerators have disjoint memory spaces that are at the other end of a constrained PCIe interface. That architecture makes programming very difficult due to the lack of support for implicit data movement. There is a move to integrate scalar cores with accelerators. These projects include AMD's Fusion, Intels MIC, and NVIDIA's recently announced ARM-64 IP agreement. We need to understand what the minimum necessary application-level information is required to manage computation for both categories of heterogeneous processing systems.

2.6 Asynchrony

The SPMD style of programming dominates the DOE workload. This approach has a lower cognitive load for programmers (it is easy to reason about), but exposes programs to severe performance losses due to sources of load imbalance. With exponential increases in system parallelism the consequences of such load-imbalances are much greater, and are further compounded by the fact that the sources of load imbalances are increasing.

SPMD/bulk-synchronous programming models presume that massively parallel systems offer homogeneous performance properties across their computing elements. However, numerous sources of performance inhomogeneity are emerging that may challenge our ability to maintain this illusion of uniformity. It remains to be seen whether SMPD-oriented programming models can be preserved, or if a more asynchronous execution paradigm must be considered to overcome the increasingly heterogeneous nature of future machines.

Hardware and software overheads have stymied previous attempts to introduce alternative, naturally asynchronous programming paradigms such as dataflow. It will require a quantitative assessment of programming model alternatives and realizable implementations on exascale hardware to determine a path for future programming models that can benefit from asynchrony.

Sources of non-uniform execution rates that contribute to load-imbalances include the following;

Adaptive Algorithms: The most commonly cited sources of load-imbalance are irregular and adaptive applications. In the context of SciDAC, adaptive applications will be critically important to tackling future multi-scale multi-physics computational problems. Adaptive Mesh Refinement (AMR) approaches can also dramatically reduce memory footprints and power consumption by applying increased resolution and computation only where it is needed.

The fastest FLOP is the one you dont have to execute.

The consequent benefits of reduced memory utilization through selective refinement are also important since the DARPA and DOE reports predict future machines will have severely constrained memory capacity because industry is no longer able to improve DRAM density as fast as logic density.

Power Management: From an applications perspective, active power management techniques improve application performance on systems with a limited power budget by dynamically directing power usage only to the portions of the system that require it. However, current power management features are primarily derived from consumer technology, where the power savings decisions are all made locally, which is tremendously non-optimal for large-scale systems. Performance heterogeneity due to thermal throttling of CPU clock rates (local decisions) will be increasingly prevalent. In particular, Intel has introduced Turbo Boost technology into its Nehalem line of chips that can temporarily boost the clock rate of the FPU if the chip is under-utilized. Whereas Nehalem will

use turbo-boost if only one core is active, Sandybridge expands this capability to temporarily boost clock rates until the chip gets too hot to operate, at which point it backs off the clock rate incrementally in response to temperature. However, non-uniformities in the chip fabrication process leads to chips that heat up at different rates, and consequently the thermal throttling will behave non-uniformly across the system. This is not unique to Intel, performance variations due to such process non-uniformities are anticipated to increase in the future leading to massive imbalances.

Fault Resilience: Another source of non-uniformity is fault detection and recovery mechanisms. There are examples of codes that saw load-imbalances that were traced back to correctable ECC errors in the memory system [62] causing irregular execution times. With increased soft error rates, such sources of irregularity will increase even for non-adaptive codes despite the fast response rate of hardware recovery mechanisms. This will get *dramatically* worse with the move towards software mechanisms for fault resilience, which take much longer to recover from transient errors. Whereas application load-imbalances tend to be persistent, so that the cost of correcting the imbalance in one application timestep can be amortized in the improved efficiency of subsequent timesteps, correcting transient load imbalances caused by fault recovery require new runtime mechanisms to both detect and proactively react to load imbalances.

Current SPMD programming models are ill-prepared to accommodate these emerging sources of performance heterogeneity. New mechanisms are needed to alert applications and system administrators that performance is being impacted by thermal variations or by hardware detection and correction of errors. It would likely simplify exascale system designs if these mechanisms could report such errors and leave it to the application to respond rather than having to transparently correct them.

2.7 Fault Tolerance

As with system power usage, we are faced with a looming crisis in system reliability. Current projections [27, 31] anticipate exascale systems in 2020 with hundreds of millions of cores. Circuits will have feature sizes as small as 7 nm, and they will operate at lower voltages than today. These very small feature sizes and the growing total feature count in a system will significantly increase the variability in the behaviors of different circuits and thus the probability that some circuit will occasionally produce the wrong result. In addition, these circuits will be implemented in deep submicron technologies with unknown aging issues. Today's petascale systems are already vulnerable to such *soft faults*. The 108k-node Blue GeneL at the Lawrence Livermore National Laboratory (LLNL) suffered one L1-cache bit flip every 4-6 hours and similarly, the ASCI Q machine experienced 26.1 cache errors per week [39]. More general failures are widely reported in a variety of clusters [3, 4].

The semiconductor manufacturing industry cannot be expected to expend the money to harden circuitry to the required levels for exascale systems since high-volume commodity systems in 2020 will have many fewer components. While data paths and memories can be protected with error correcting codes, logic cannot. Traditional triple modular redundancy (TMR) techniques require at least three times the hardware, consuming three times the energy, and hence are not an option. Therefore, we will need to invent techniques that can asymptotically provide the reliability of TMR, yet at a minimal hardware and energy cost (perhaps 10%).

Soft faults can affect applications in three major ways. First, they can reduce performance, leading not only to longer execution times but also higher energy consumption. Second, they can cause unexpected application aborts, which can cost large amounts of debugging effort looking for phantom coding errors and force applications to re-execute, wasting significantly more time and energy. Finally, and most important, they can corrupt the application's results, possibly leading

to invalid conclusions derived from the results of the computation. These outcomes underscore the need to study the resilience of scientific applications, identify their most vulnerable components, and design techniques to address these challenges. The technology that enables such resiliency will have to span the entire system stack, from circuit design to applications. It will include new mechanisms for detecting errors in hardware. There will had to be system software to isolate and contain faults, remap applications, and resume them from a correct state.

2.8 Disruptive Changes are Already Underway

The following architectural features are already present in existing petascale systems, and the technology trends are already apparent from examining 3 generations of petaflop-scale leadership computing systems in the DOE. Table 1 shows that the progression towards the disruptive changes predicted for exascale systems in 2018 are already underway today. The target for exascale programming models may well be the first-derivative of the exponential changes that are anticipated rather than the specific design of a single exascale platform.

System Architecture	**Cray XT3**	**Cray XT4**	**Cray XE6**	**Change**
Clock (GHz)	2.80	2.3	2.2	*flat*
Cores/node	1	4	24	*exponential growth*
FLOPs/core	5.6	9.2	8.4	*flat*
Memory/node	4GB	8GB	32GB	*modest increases*
Memory/core	4GB	2GB	1.3GB	*decreasing*
NUMA Domains/node	1	2	4	*exponential growth*
STREAM mem BW/node	4.5GB/s	14 GB/s	51 GB/s	*modest increases*
STREAM bandwidth/core	0.8	0.4	.25	*decreasing*

Table 1: **Summary of emerging hardware trends that are already evident in the last three generations of Cray hardware.. The disruptive changes to hardware characteristics are already well underway.**

3 Application Requirements

Exascale-class computing presents application developers with several unique challenges beyond those posed by current HPC systems. Some of these issues are driven directly by the hardware challenges outlined in Section 2 and present themselves to the application developer collectively as the exascale execution model (Section 3.2), while additional challenges flow from the extreme-fidelity science enabled by exascale machines (Section 3.1). This section will conclude with a discussion of programmatic challenges in Section 3.3.

3.1 The Extreme-fidelity Science Challenge

From the perspective of memory volume or arithmetic throughput, the needs of computational scientists are, for all practical purposes, unbounded. As computing power grows, resolution increases, time scales are extended, and the number of phenomena that can be studied together, in context, increases. Application codes have moved from one, to two, to three-dimensional geometries, often on unstructured and dynamically changing grids. Growth in computational capability has also enabled the expansion of simulations using discrete particles, including highly accurate many-body quantum theory predictions, molecular dynamics simulations, and Monte Carlo applications. In the next generation of application codes there will be a continued drive towards higher fidelity through full multiphysics simulations of real-world phenomena. Examples include full nuclear reactor simulations from the fuel and chemistry to the containment vessel, plasma physics simulations that model the extremely small scale of the fusion reaction all the way to the fusion device's heat exchange mechanisms, quantitative theoretical understanding of the multiphase interfacial dynamics of excitons and molecules, and complicated combustion engine models that capture detailed chemistry and fuel combustion in addition to the turbulent dynamics. Figure 6 shows a typical multiphysics simulation which captures different regimes of material state (solid fragments, radiating plasma, and vapor) all on an irregular moving adaptive mesh.

The various physics models employed must be designed to inter-operate in a multi-scale, multiphysics environment, both in terms of data layout and in terms of employing the appropriate boundary or scale-coupling physics. Additionally, challenges arise when considering the large scale of the systems that can be studied with an exascale machine. Convergence of numerical methods often slows as the system size increases and conditioning becomes worse. Development of new preconditioners and judicious use of variable precision (both lower and higher than double) can deal with some of these issues. However, in some cases, the best approach may be to rethink the theories and methods employed to find better-suited approaches in the context of extreme computational power and the concomitant large physical system sizes.

3.2 The Exascale Execution Model Challenge

The manner in which scientific software is developed is both influenced and constrained by foundational factors, ranging from the computing platforms available, to programming languages, software libraries, mathematical algorithms, and even the precision of the arithmetic. This ensemble has been referred to as the *execution model* [60]. An execution model is not simply a programming model, it provides the conceptual scaffolding for deriving system elements in the context of and consistent with all of the others. It is a coherent abstract schema that permits co-design and operation of such multiple layers.

The parallel scientific and engineering applications written for today's execution model tend to be large codes, often with many years of effort behind them. The computational scientists who

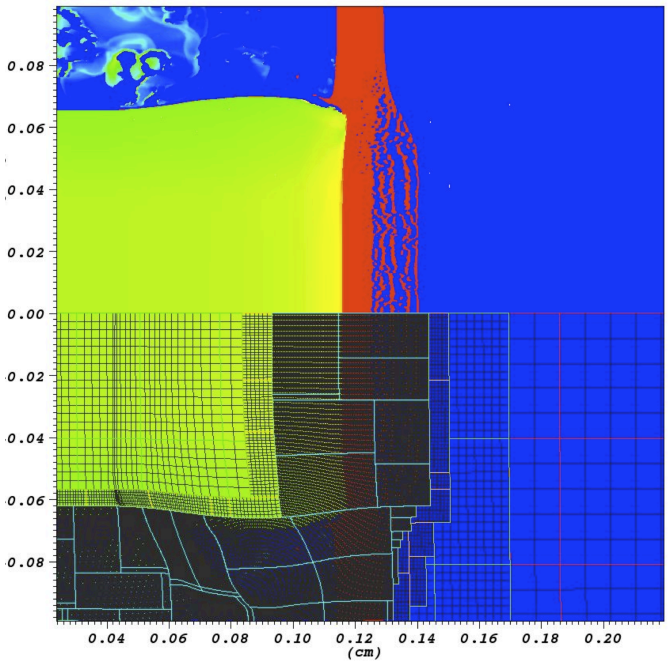


Figure 6: A typical current-day multiphysics simulation including vastly different physical regimes from hot vaporizing plasma to cold fragmented solid, all on a moving AMR mesh.

have written them enjoy an execution model that has been slowly evolving for half a century. This model has largely assumed a von Neumann node architecture, programmed in Fortran and more recently in mainstream languages such as C++. Leadership class systems for the last two decades have been large ensembles leveraging components developed for commodity markets, utilizing the Message Passing Interface to realize Hoare's communicating sequential processes model (CSP). The end of Dennard scaling and the advent of multicore microprocessors has reintroduced shared memory to the individual nodes of the model. Hardware for improving graphics performance has reintroduced data parallel computing to the model for individual nodes (SSE and/or GPUs).

The prospect of moving today's codes to a new model is daunting to the application scientist. However, if a new programming paradigm allows a significant performance improvement and/or the ability to add significant new physics, application scientists will adopt the new model. Such transitions have happened before, first with the emergence of vector mainframes and later, with distributed memory CSP systems. The execution model must change to allow for code developers to fully exploit such systems by offering appropriate architectural features, programming models, and APIs to effectively utilize memory hierarchies, manage faults, and minimize power consumption. The process to make this adjustment will be highly application-dependent. Some applications may be able to achieve sufficient efficiency on an exascale machine using an evolutionary approach, where gradual software changes result in an exascale-ready application. Other applications will require a revolutionary approach necessitating an extensive rethinking and rewrite of the code. Indeed, this has already happened in some application subdomains to deal with recent increases in both architectural and application complexity, a case in point being the Tensor Contraction Engine

(TCE) Domain-Specific Language (DSL) used in many-body theory as illustrated in Figure 7.

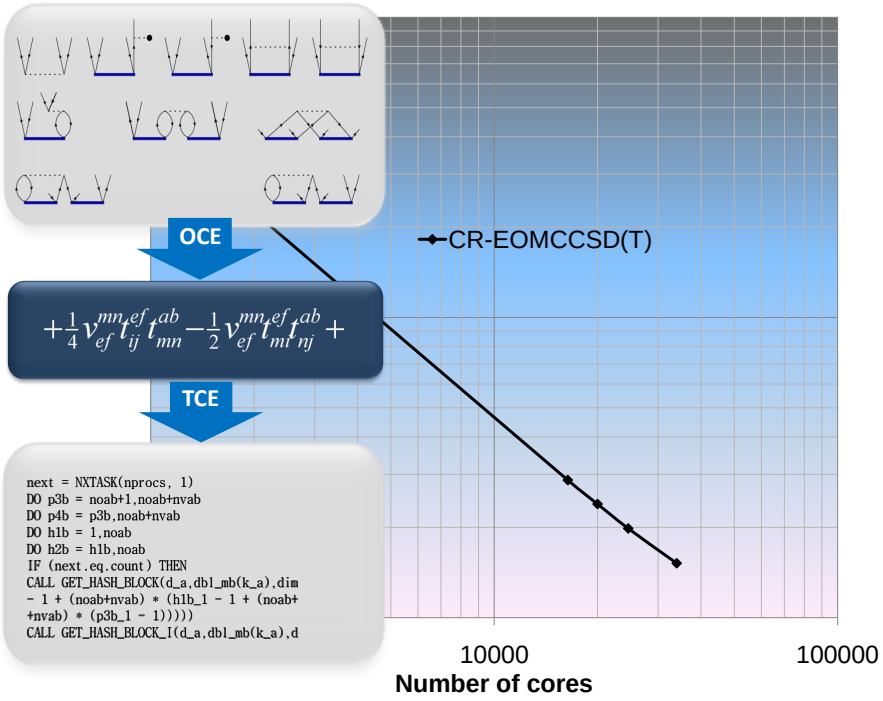


Figure 7: The Tensor Contraction Engine generates high-performance code from a high-level domain-specific language, thus reducing the coupling between application and architecture complexity into a relatively small translation layer. The resulting code has demonstrated excellent scaling to today's largest machines.

The dominant driving force that will cause applications to undergo a revolutionary change is the need for higher performance in the solution of new science problems and the underlying architectural changes described in the Section 2. Concurrency will increase exponentially, and neither memory bandwidth nor volume will keep up with the growth in arithmetic performance. Locality must be exploited in a way that allows the memory hierarchy to be used with good power and time efficiency. Soft error rates will increase, and it will no longer be enough to simply drop an occasional checkpoint to achieve resilience. Finally, to attain orders-of-magnitude more concurrency than today, it is likely that we will need to rearchitect our application code base, and perhaps even our methodologies, in order to exploit finer-grained parallelism. However, a revolutionary change for each future generation of hardware is unacceptable. It is necessary to design an execution model that can be applied to exascale machines and their descendants. Essentially the same application code must be portable from developers' laptops to the exascale and beyond. They must present abstractions to applications programmers which map into multiple architectures. Once applications have made a revolutionary change, then future adaptations, if necessary, must be evolutionary.

3.3 Programmatic Challenges

The current approach typically taken towards application development is to adapt an application to new hardware once the characteristics of the hardware design have been fixed. Exascale machines

will not have enough margin in feeds and speeds for applications to perform well using this decoupled design approach, motivating the concept of hardware/software co-design for exascale systems. This requires the diversion of application developers' attention from the math, physics, and programming issues at hand to interactions with vendors and computer scientists. In order to get sufficient mind share from developers for co-design to succeed, we must rethink how projects are structured and funded, and the ASCR Exascale Co-design Centers are a small step in this direction.

An additional burden on application developers is the anticipated need to perform a revolutionary rewrite of their codes. Even though this may be justified strictly in terms of the new capabilities permitted in such a rewrite, this is still a psychological hurdle which many developers still find difficult to accept, particularly given the current uncertainty as to how the exascale execution model will manifest itself. Computer scientists, with input from application developers, must soon begin charting a clear path that developers can confidently follow to obtain exascale-ready applications.

Finally, only one or two of exascale machine designs are likely to be built, at least initially, and these must support the needs of multiple applications. Either the current applications being targeted to exascale will end up bearing enough similarity to each other that co-design will converge onto a single platform with sufficient performance for all, or it will be necessary to prioritize application requirements in consideration of realizable hardware designs.

4 Programming Models: Challenges, and Strategies

DOE codes have largely evolved from being directly written in C++/Fortran with MPI into being layered above sophisticated frameworks and making heavy use of libraries. The complexities of node-level parallelism are forcing another form of layering, with two levels of parallelism. As systems become more complex, it usually becomes necessary to add more layers of abstraction between the application programmer and the hardware. Each layer is associated at least implicitly with an *abstract machine* and a related *programming model*. An abstract machine exposes some but not all features of the platform, and the programming model permits the specification and optimization of how those features are used by the program without having to deal with the complexity of the full machine. It is useful to distinguish between a programming model and a *programming notation*, which refers to the actual syntax and semantics of the language used to express programs in that model; in general, a given programming model can be implemented in many programming notations.

The production of high-performance code for an application therefore involves multiple levels of transformation that map a high-level specification of a problem into an executable that is specific to a particular platform. Each transformation maps a higher level programming model into a lower level one. The higher-level models are more problem-domain specific, while the lower level models are more machine specific. Transformations at the top level currently tend to be more manual, while the transformations at the bottom levels tend to be more automated and are usually performed by compilers.

4.1 Programming Models Stack

Figure 8 shows one possible stack of programming models. Although the figure is very general, we will use examples from the digital signal processing (DSP) domain to illustrate various points. At the highest level, we might have a very abstract specification of the computation; for example, a linear transform like an FFT of a certain size might be specified by writing down the matrix representing that transform. In general, there are many algorithms for implementing a given specification, and it may be advantageous to use different algorithms on different platforms. Therefore, the first transformation determines which algorithm is best suited for the machine of interest, and produces a representation of that algorithm. For the FFT example, this step involves determining the best factorization of the transform matrix. In most domains, this step is currently done manually, but in DSP, this step has been automated by systems like SPIRAL, using very sophisticated search techniques. The next step in lowering the abstraction level might map the computations and data of the selected algorithm to a virtual topology of abstract processors, and select a particular communication framework such as MPI or active messages. At this point, the computations mapped to a single node are still represented abstractly. The next step might choose library routines for implementing some of these computations; for the remaining computations, it might generate OpenMP or CUDA code. The resulting node program might be optimized further by a compiler using transformations like loop blocking and loop invariant removal, and then converted into executable code. At runtime, this executable might be optimized using just-in-time techniques if this is useful for the particular problem domain.

A variety of technologies exist for implementing these programming models: frameworks embedded into general-purpose languages, domain-specific languages, pragmas and annotations, libraries, meta-programming, etc. What is important is that the existence of these well-defined layers enables distinct programmers with specialized skills to work at different levels – with "Joe" programmers that are highly skilled in the application domain working at the top level(s) and "Stephanie" pro-

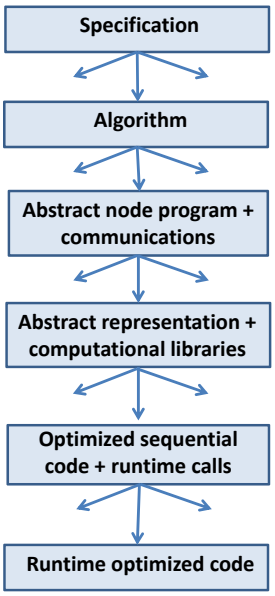


Figure 8: Possible Stack of Programming Models

grammers that are highly skilled in the computing domain working at the lower level(s).

With respect to such a stack of programming models, the parallel programming community has learned a number of important lessons over the decades, summarized below.

1. In environments where performance is paramount, full automation of transformations may be infeasible: even for simple tasks such as matrix multiplication, a carefully hand-coded library usually achieves better performance than code produced by a compiler from a triple-nested loop. Therefore, a properly designed hierarchical environment should support the intervention of the "human in the loop" at each level.

2. In environments where performance matters, the successive mappings should not only be (mostly) "semantic-preserving", but also (approximately) "performance-preserving". Otherwise, one cannot manage performance at the higher levels of programming through the design of efficient algorithms. This means that one should be able to define performance metrics at each level of the hierarchy and define how performance metrics at a higher level map into performance metrics at the lower level. The metric of operation counts, which is useful in the context of sequential programming, does not capture parallelism, and does not capture communication/memory accesses, which is the dominant performance bottleneck today as well as tomorrow in the exascale era.

3. While automated translation tools are good at preserving semantics, they are not as effective at preserving performance: Programmers are repeatedly surprised by programs that "do not behave as expected". Even with a perfect translation, programmers have wrong expectations about their programs; their programs have semantic bugs and performance bugs. These

are discovered through a debugging and tuning process that tests the software. Bugs introduced at some programming level must be fixed at that programming level. Therefore, a good hierarchical environment has to support accurate "reverse mapping" where semantic or performance information captured at execution can be translated into semantic or performance information at the level of a programming model, and information expressed in a lower programming model can be translated back into concepts of the higher programming model.

4. There is an inherent conflict between good isolation between layers and performance. Isolation between layers restricts the amount of information passed across layers and hides the details of lower layers. This facilitates programming at a given level. However, good performance may require passing more information across levels.

5. While the current conventional view is that the high level stages require more manual effort, increasingly, the application of automation for higher levels of programming are increasingly the norm. Tools like the aforementioned SPIRAL and FFTW perform algorithm exploration automatically, and increasingly, can exceed the performance of that attainable by human programmers through the sheer range of alternative mappings that can explored and the target complexities that can be managed. As evidence, it is the case that the major vendors of high performance math libraries are moving away from doing this coding manually with outsourced mathematician teams in low-wage countries and toward moving to auto-generation of the libraries from tools such as SPIRAL. This is increasingly the trend for linear algebra and spectral transform (e.g., FFT) libraries. The reason for this is that the portability of middle and lower levels of code, especially across different hardware targets and through hardware generations, is increasingly low. It is not just the particular blocking of loops or unrolling, or register use, but the specific algorithms themselves need to change from machine generation to generation. Thus any investment in coding at low levels of the programming stack for libraries or application code, will have very short time for payoff, as it will rapidly become obsolete with hardware architecture changes. In contrast, investment in programming at higher levels, with richer levels of semantics, and the embodiment of tools for automating the exploration of algorithms, is the trend for its better long term payoff. Furthermore, it is likely to be the route to the absolute best performance.

These considerations lead to a view of the stack of programming models illustrated in Figure 9: The successive mappings from level to level are "semantics and performance-preserving"; reverse mappings translate information captured during execution to the constructs of each programming model, to support debugging and tuning (system in the loop); and the Joe and Stephanie programmers, while preferring to program at higher levels, do occasionally program directly at lower levels (human in the loop).

Programming models that permit this kind of user intervention and guidance at all levels of abstraction are called *multi-resolution* programming models. It is important to note that each programming model will be embodied in a particular programming technology (language, framework, library) and will come with a suitable *programming environment*. The programming environment, which consists of the set of tools available for creating, generating, debugging, testing and tuning codes is a major contributor to productivity, often more important than the programming model itself.

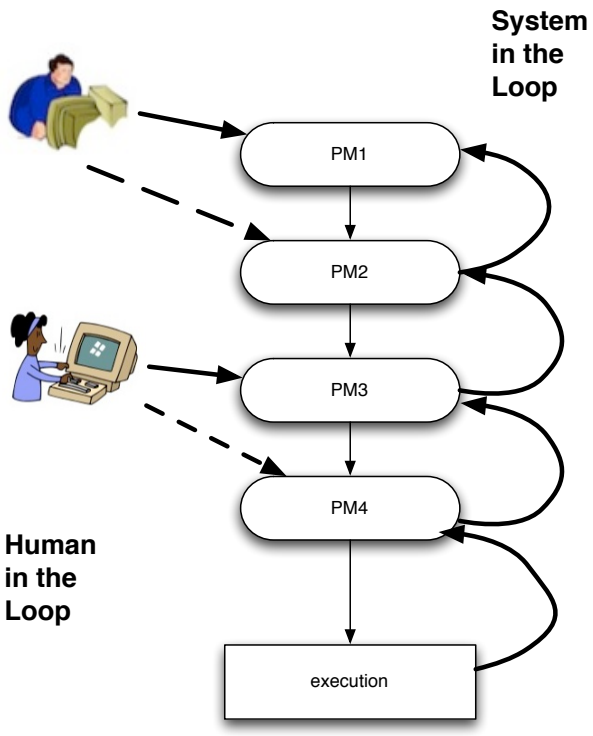


Figure 9: Interactions

4.2 Major Implementation Challenges

In this subsection, we discuss the major challenges in implementing a programming models stack like the one shown in Figure 8.

4.2.1 Exploiting properties of the problem domain

At the very highest level, the programming model is likely to be a domain-specific language. This is because in many problem domains, there are domain properties that can be used to optimize programs: compilers and runtime systems must be able to exploit these properties for performance, and the programming notation used by the application programmer must not obscure opportunities for exploiting them. For example, matrices have a rich algebra, and exploiting these algebraic properties can improve the performance of matrix computations dramatically. If the programming notation allows programmers to use matrices as data types and to write expressions involving matrix operations like addition and multiplication as APL, MATLAB and Mathematica do, it is possible for a compiler to optimize these programs by exploiting the algebraic properties of matrices. However, if the programming notation is like FORTRAN-77 and requires matrices to be expressed as two-dimensional arrays, and matrix computations to be expressed using nested loops and indexed access to arrays, it is much more difficult for a compiler to exploit matrix algebra to optimize programs since the high level abstractions are missing in the application program.

While the example above illustrates in the familiar terms of automating the exploration of different linear algebra algorithms, there are benefits to pushing toward higher level expressions of the problem domain. This can provide additional semantic information to be mined by automated tools for program mapping and also for verification. For example, programmers coding scientific

applications should be provided domain languages for expressing the underlying partial differential equations for the system under study. This can be annotated with information that is known to the domain scientist, such as value ranges, symmetries, and boundary conditions. Furthermore, with the trend toward multi-physics simulations, the different physical models could be expressed at high-level. Particularly with the demands of finding more parallelism/concurrency for exascale, and the criticality of eliminating communication, such *very high level programming* (VHLP) will be essential. VHLP will expose opportunities for new dimensions of concurrency, such as places where the nature of the underlying physics removes the requirement for particular orderings of operations. Or for example, expressing physics at high levels of abstraction exposes opportunities for high-level scheduling transformations that fuse model evaluations at grid points in order to eliminate communications and memory traffic. Recent research in FLAME [17] provide a guideline for high-level specification of a range of linear algebra operations. One particular exemplar of moving to the level of the physics are programming approaches working from the variational form to facilitate exploration with discretization and solver strategies in finite element methods [30].

Exploiting domain properties to optimize programs does not necessarily require a different domain-specific language for every problem domain. A language like C++ that provides extensibility through the use of features like meta-programming and templates might be adequate as a sandbox for prototyping systems that exploit domain-specific properties. New languages can be designed after the utility of such systems has been demonstrated and there is enough experience in using them.

Research Questions: How do we build systems that support programming notations in which programs can be written directly in terms of domain abstractions? How do we build systems that permit domain experts to specify properties important for program optimization? How do we build compilers and runtime systems that can optimize programs using the domain knowledge specified by domain experts?

4.2.2 Unified Abstract Machine Model

At the lowest level, the programming model must expose a unified execution model that provides abstractions for the communication system, memory hierarchy, and core heterogeneity of future high-end machines. This execution model can be parameterized suitably to adapt it to a particular machine. The abstractions should capture all aspects of machine architecture that may substantially impact the performance of programs, without being so detailed as to be overwhelming and intractable for programmers.

Research Challenge: Develop a common machine model that abstracts performance-critical features of future high-end machines such as core heterogeneity, memory hierarchy, and communication topology.

4.2.3 Challenges Particular to Exascale

The problem of building suitable multi-resolution programming models is difficult in general, but it is exacerbated for exascale machines by a number of cross-cutting concerns, including the following.

Parallelism: roughly 1 billion threads

Memory: the amount of available memory, which is likely to be relatively smaller per ALU than in today's systems

Bandwidth and latency: Communication, at all levels of the system (caches, memory, remote nodes) will be the main performance bottleneck and the main source of energy consumption.

Hierarchy: The system storage (caches, memory, disks) is organized hierarchically; accesses to lower levels of the hierarchy consume more energy, suffer from increased latency and are more bandwidth limited. The storage hierarchy becomes deeper, with the inclusion of various forms of persistent memory.

Communication and synchronization mechanisms: The architecture provides a different set of communication and synchronization mechanisms: coherent shared memory, non-coherent shared memory; atomic read-modify-write operations; rDMA; channels; etc. The mechanisms differ at different levels of the same system and across systems.

Energy: Energy consumption per solution becomes a critical resource; the architecture will provide multiple mechanisms to control energy consumption of various subsystems.

Resilience: A lower MTTI may require exposing failures at higher levels of the stack.

Heterogeneity: The system is likely to include different types of computation engines (CPU-like and GPU-like).

All these aspects do not necessarily have to be handled at the top level - indeed, the way we defined the programming models stack, no aspect of the architecture is visible at the topmost level, and all levels of the architecture are visible at the bottom level. *The key issue in the design of the stack of programming models is to decide which aspects of the system are made apparent to the programmer at what level.*

Let us consider each of the previously listed aspects in turn.

Parallelism: There is no consensus about the programming level at which parallelism must be exposed explicitly. The precise form in which parallelism is exposed currently varies between systems:

1. A fixed number of statically mapped threads (e.g., MPI)
2. A rarely changing number of semi-statically mapped threads. The threads are occasionally remapped, for load-balancing (e.g., Charm++)
3. A frequently changing number of dynamically mapped tasks (e.g., Rice's Habanero).

Hybrid approaches such as MPI+OpenMP or even MPI+OpenMP+CUDA are being used to express multiple levels of parallelism in application programs.

As codes become more irregular and dynamic, it becomes more important to support a programming layer where parallelism is dynamic, with mapping possibly controlled at a lower layer.

Research questions: What is the right mix of information provided by programmer, derived by the compiler and derived by the run-time that should be used to expose parallelism and control mapping? How are mapping decisions stacked across layers? At what level is the number of nodes, the number of cores per node, etc., exposed to the programming model, and how? How does one manage the trade-off between load balancing and locality?

Heterogeneity: Exascale computers will likely combine conventional cores with GPU-like SIMD cores and, possibly, cores that support high levels of concurrent multithreading. The different cores present different programming models at the bottom of the stack. Different exascale systems are likely to have a different mix of such engines, with differing architectures, and differing connectivity.

Research Question: At what level of the programming models stack can we abstract the differences between such cores? How?

Memory: The current programming model is that of virtual memory at each node. More explicit memory management may be needed in the future for large NUMA nodes and for systems that replace caches with scratchpads. If a framework is used for the management of the data structures (e.g., meshes), then memory management could be handled in the framework implementation.

Research Question: What additional hooks are needed for efficient memory management and at what level of the programming stack should those be provided?

Bandwidth and latency: The interconnect bandwidth and latency constraints of a system are addressed by managing communication. For many applications, communication already is the main performance bottleneck in current systems, and it will become an even greater challenge in the exascale era. Communication minimization often requires algorithmic solutions, hence needs to be handled as an algorithm design problem. Furthermore, as the number of cores per chip increase, on-chip communication becomes a major issue. For data-parallel computations with a fixed data layout, communication can be largely handled via data partitioning. Codes with more dynamic data structures (AMR, particle algorithms, multi-scale algorithms) may require more dynamic repartitioning and/or software caching.

Research Question: What level of programming support is needed to minimize communication? Is it possible to support good locality, while using a global name space? Is it feasible to handle communication as done in sequential systems – as a temporal locality issue – and having good temporal locality mapped into low communication by hardware and run-time?

Hierarchy: Many codes are currently written with a one-level hierarchy (all MPI codes), with multiple MPI processes per node; an increasing number of codes are written with a two-level hierarchy (MPI+OpenMP). Exascale systems are likely to exhibit deeper hierarchies. Current approaches to handle the HW hierarchy expose the size of each level of the system: e.g., number of cores per node and number of nodes. This impairs code portability. One alternative approach, which has been explored by the Cilk project, is to express parallelism using recursive divide-and-conquer constructs; these can be naturally mapped onto any hierarchical hardware (similar to cache-oblivious algorithms that map to any storage hierarchy) – but, as currently implemented, may not preserve locality.

Research Questions: Can the hierarchical nature of exascale systems be managed, at the top layers of the programming models stack, using "oblivious" divide-and-conquer control and/or data partitioning constructs?

Communication and synchronization mechanisms: The communication and synchronization mechanisms used intra-node are very different than those used inter-node – not only in performance, but also in semantics. This difference is currently exposed in the programming layers used by application programmer. Exascale systems may add one more layer of complexity; e.g., in a chip where cores are partitioned into clusters, with coherent shared memory inside cluster and non-coherent shared memory across clusters.

Research Question: To what extent is it possible to hide at the higher levels of programming the differences between the communication and synchronization mechanism provided by the hardware at different levels of the hardware hierarchy?

Energy: Energy is becoming an essential resource in the exascale regime. Energy consumption is largely controlled by reducing the movement of data, be that access to main memory or interprocessor communication. In addition, future systems will provide increasingly fine-grain mechanisms

for controlling the energy consumption of various subsystems. One needs to decide at what level those mechanisms become visible to the software and to the programmer.

Research Question: How can the programmer, the compiler and the run-time assist in the management of of power consumption.

Resilience: Exascale systems are expected to have a much lower MTTI and to suffer from undetected soft hardware errors. In current systems, we assume that all errors are detected and are either corrected transparently, or cause a fatal exception that stops an entire parallel job. In the latter case, error correction is handled via global checkpoint and restart. Checkpointing and restart can be automatic, but more often than not, involves the programmer who initiates the checkpoints and provides a restart routine. There is no other common API for managing resilience today, and at exascale, the MTTI could be so short, that global checkpointing and restart is not longer a viable option. More aggressive techniques will become necessary, and pairing for fault detection or triple modular redundancy for fault correction are not attractive alternatives as they divert circuitry from useful work and increase power consumption.

Research Question: Can we support enhanced resilience solely via new APIs or automatic compiler and run-time solutions? If not, what additional support for resilience do we need to provide at higher levels of the programming models stack? How can a programming model support more efficient, localized transactions or checkpointing? How can it be extended to support error detection, and where possible, error correction?

4.3 Objective Criteria to Assess Programming Models

Important criteria for assessing proposed solutions are the following.

- Does the programming model permit the specification of domain properties that can be used to optimize programs at the highest abstraction levels?
- Does the lowest abstraction level represent all the features of exascale machines that must be exploited for performance?
- Does the programming model permit users with different skill sets to intervene and guide the lowering of abstraction levels from the algorithm level all the way down to the executable level?
- Does the proposed programming model interoperate with current programming models?

  A large number of programming models have been proposed, designed and developed over the past several years to revolutionize parallel programming environments for high-end systems. However, many of them have had limited adoption because of the effort involved in porting applications to these models. Specifically, many DOE scientific applications have had a life-time of several decades, over which millions of lines of code has been written for many of these applications. Further, each application relies on a complete chain of libraries, tools and execution environments for solving various aspects of the computational and communication requirements (e.g., math libraries). For an application to migrate to a new programming model, not only does the application need to ported to this model, but all of the execution environment that it relies on has to be ported as well because many of the new programming models are not interoperable with legacy models. This issue has cornered applications into an all-or-nothing position where incremental migration is not possible.

For revolutionary new programming models and notations to gain traction, an evolutionary path that would allow applications to use them incrementally is desirable. Interoperability with existing programming models would be key in such an environment. Although the conventional wisdom is that legacy applications have to work on exascale as-is, it was observed by workshop participants that most important applications are continuously rewritten. Thus, if an effective programming model and notation for exascale is available, and they provide an incremental path for code migration, we believe that important legacy applications will get ported.

4.4 Objective Criteria to Assess Programming Notations

Important criteria for assessing programming notations are the following.

- Expressivity: The programming notation has to have the expressiveness to describe all classes of important problems in a clear and concise manner. Domain experts should have the opportunity to describe the intent of the program, but should not be required to provide the methods of getting high performance from the code.
- Portability: It is critically important to have an algorithm description that abstract away details that are tied architectural variations. Programs need to be made future-proof by not prematurely binding it into the architecture of the day.
- Effectiveness: While the algorithm description abstracts away architectural details, it is equally important to make sure that either an effective compiler and runtime system can find high performance instantiations for these abstractions or all the tuning parameters are exposed to a performance expert.
- Scalability: It is important to have the ability to write programs that scale from small systems to exascale systems.

5 Compiler: challenges and strategies

Compilers provide analysis, optimization, code generation and essential transformation capabilities that will be required to support software on future architectures. While compiler research is well established in computer science, today's complex and monolithic compilers are in some respects a bottleneck to productivity in the development of DOE application software. The complexity of exascale architectures and the DOE applications that will use them will place significant demands on future compilers. The compiler research community is shackled by the current languages used and largely powerless to change the languages used by application groups. The time scale for application groups to use alternative programming language changes on the order of decades.The introduction of new programming models, as thin layers of abstraction above existing languages, facilitates the development of new compiler technology (and associated run-time support) and can be more tractable on a shorter time scale than new programming languages. It is not a given though that existing programming languages will scale to the new applications, architectures, and execution models for exascale. Significant experimentation with new languages and programming models will be a component of exascale research and development. These new programming models will place new demands on compilers and drive new research and new envisionings of the roles of compilers. This section describes the requirements for new generations of compiler tools to support future petascale and exascale systems, driven by the multi-resolution programming model description in Figure 8.

- **Managing fine-grain parallelism, locality and heterogeneity:** The performance optimizations performed by compilers must adapt to new properties of exascale architectures: support for billions of concurrently-executing threads, more constrained memory systems, heterogeneous processing units and configurable memories.

- **Increasing resiliency:** Concerns about increased component counts, transient hardware failures, increased software complexity, and numerical accuracy at exascale will require new approaches to resilience beyond today's most commonly used strategy of checkpoint/restart. Code that performs non-intrusive sensitivity analysis, applications that characterize numerical properties of their result, self-checking code, transactions and rollback, and other resilience approaches will likely require specific compiler support.

- **Managing energy and power:** Active energy and power management may require remapping software dynamically to adjust to changing resource availability, also requiring compiler support.

- **Modifying the compiler's organization:** An exascale compiler must be fundamentally restructured to support the multi-resolution programming model of Figure 8. Specifically, dynamic software mapping, programmer interaction, heterogeneous hardware, and various other features of exascale system design will demand new bi-directional interfaces between compiler and programmer, run-time and various abstraction layers to support modular systems and an ecosystem for language and compiler development.

- **Support for domain-specific languages and constructs:** Domain-specific languages and tools have emerged as one strategy for managing parallel code generation while retaining a high-level of abstraction for the programmer. A domain-specific tool can be developed by a performance expert for use by domain experts, and can facilitate optimization by exploiting how to optimize codes for the specific domain.

- **Programmatic challenges:** Developing this compiler technology in time for exascale system deployment demands a new strategy that accelerates technology introduction, leverages tools developed by different members of the community and by commercial vendors. To achieve the goals of exascale in a cost effective and timely manner, it is important to draw on all members of the vendor, laboratory, and academic community. Hardware vendors invest significant resources in to their advanced compilers in order to multiply the effectiveness of their hardware. Highly innovative independent software companies with experienced and established compiler teams, can provide technologies that significant private investment and in some cases, government funding have supported. Laboratory efforts has a rich history in knowledge of their applications, the history of machines that they have targeted and an application focus. Academic groups have brought an understanding of context and new ideas. A challenge is to have these groups work together and, more importantly, have the results be sufficiently focused to emerge as robust solutions for DOE. An additional challenge is that without a common compiler framework it can be difficult for separate vendors to work with each other and both labs and academic groups. The structure of the collaborations will play an important role in shaping how broadly different groups will be able to participate.

Key challenges in compiler technology are discussed in the subsequent sections.

5.1 Compiler Organization

An impediment to adopting new architectures or new architectural features is the long lag between architecture introduction and compiler availability, usually spanning multiple years. Usually this is due to several factors, including underinvestment in compilers; a rule of thumb is that the investment needed for a good compiler is the same as the investment needed to develop the hardware, and many system development efforts have failed to invest the needed resources in compilers. A second factor is that typically compilers embody the manual optimization strategies used by uber programmers; there is a lag for the practices and techniques used by programmers to be identified, formalized, and embodied in automated trasformations in the context of other automated transformations.

If we had an exascale architectural design and programming model today, could we have a compiler built in time for the emergence of exascale architectures? The natural lag in compiler development means the answer is without question no, but there may be ways to reduce this gap and accelerate the pace of compiler research and development.

Today's organization of compilers is somewhat monolithic. This monolithic organization has a strong engineering rationale in terms of amortizing the cost of developing intermediate representations over many transformations, facilitating the development of fused and joint transformations to overcome phase ordering limtations, and in eliminating performance-reducing impedance mismatches in transforming code and analyses from one representation to another.

However, it does make it somewhat difficult to integrate new ideas into existing systems. As a consequence, compiler researchers are often compelled to replicate the work of others in their own infrastructure, further slowing the rate of progress in compiler research[2]. Figure 10 shows an abstract conceptual model of a compiler and associated ecosystem, showing an evolutonary path from today's systems.

Current compiler organization: The blue rectangles, circles and connectors reflect the organization of current compilers. Full source-to-binary compilers traditionally consist of two internal representations, one close to the source language, and another close to the architecture. If someone

[2]There are other programmatic reasons for this as well, such as current restrictive DOE policies regarding commercial sources.

wants to add new optimizations to a compiler, they must adopt the compiler's intermediate representation. To modify the input programming language requires extensions to the frontend, which are usually very difficult, and also support in the compiler's intermediate representation. Targeting new ISAs requires extensive backend support for which it may be difficult to replicate the success of vendor backend compilers given vendor investments. A compiler represented by the blue portion of the figure is typically comprised of hundreds of thousands to millions of lines of code, and an investment of a decade or more. Such an extensive investment should be leveraged for exascale, but extended with lightweight additional technology.

Source-to-source compilers. An intermediate strategy for reducing compiler complexity is represented by the left-hand-side of the blue portion, where the output of the compiler is source code. Source-to-source is a productive approach that permits leveraging the vendor specific compiler on the backend generated source code (common to source-to-source technologies). A source-to-source compiler does not need to provide translation to architecture-specific binary, and instead can rely on hardware vendor investments in optimizations for instruction-level parallelism, register allocation, instruction selection and low-level standard optimizations (e.g., common subexpression elimination, constant propagation, dead code elimination, etc.). In addition to reducing compiler complexity, source-to-source compilers are commonly used in the DOE research community because they produce code that is portable across different hardware vendors' platforms, and offer a path for the research community to fill in gaps in hardware vendor compiler technology.

One key challenge in source-to-source compilation is efficiently crossing the source barrier to the hardware vendor back end compiler. Back end compilers typically are not tuned for accepting machine generated code. Syntax-oriented source-to-source compilers can work to preserve the original text of the program to avoid inefficiencies in the process of regenerating optimized programs to source (aka "unparsing."). They use relatively old representations such as parse-trees for performing optimization. Being tied to the syntax so tightly, can make transformations more complex. It is important to move beyond parse-tree based systems in source to source compilation systems. Alternatively, source-to-source systems may use an abstract-syntax tree to remove the complexities of syntax. Still other means internal to source-to-source compiler may operate directly on more abstract representations, such the generalized dependence graph, which eliminates syntactic artifacts, before translations back to a form from which to generate source code as output. Significant work can be done to clean up the generated code to make it suitable for use with vendor compilers that frequently don't handle automatically generated code well.

New interfaces to the compiler. The green clouds and trapezoids in the figure represent additional compiler libraries that could be added to an existing framework. The two trapezoids represent new interfaces to the compiler that are not commonly available in today's compilers. The multi-resolution programming model described in Figure 9 demands a compiler that can leverage optimization constraints and strategies and other information suggested by a performance programmer. Thus, a new interface is needed between compiler and programmer that allows specification of this useful information.

It is vital that these specification of constraints and strategies be limited strictly to those that preserve semantics, to avoid the introduction of bugs through this orthogonal interface. In some systems, this extra information is presented as an external "tuning language" designed to carefully preserve semantics by only affecting the schedule and placement of the computation.

Possibly, these directives could be integrated into Integrated Development Environments (IDE) for performance programmer guided exploration of semantics preserving transformations and visualization of the transformations.

A separate interface between compiler and run-time provides the rich interface that permits the compiler both to tailor its optimization strategy to dynamic behavior and empirical search-

based optimization techniques that use autotuning and machine learning. The latter search-based techniques involve a compiler exploring a range of possible implementations of a computation and through heuristic search identifying the best solution. These autotuning and search capabilities live more naturally in the compiler than in auto-tuning libraries. In exascale architectures, such search-based techniques will be critical to navigating the multiple optimization criteria of performance, energy and resilience.

Composable compiler library technology. The cloud structures include separable analyses, combined analyses and transformation, and search algorithms as examples of composable compiler library technology that can be reused and integrated into different systems. Much of these types of components have internal abstractions that do not need to be tied to the specific intermediate representation of a compiler, and could potentially be made available for wider use. A domain-specific language could be implemented in this way and thus have abstractions carry additional semantics that the compiler may not infer from the base language directly, making it possible to communicate greater explicit and implicit degrees of parallelism available for compiler mapping. Separate compiler passes over a common internal representation typically can share information, but must be organized to understand the information being shared. Through explicit support from within a compiler framework (e.g., common attributes), a compiler developer can share data between independent analysis and transformations, a key step towards composable libraries.

Different compiler technology for different parts of the computation. Compiler optimization strategies are often driven by transformation requirements that may be local to narrow regions of the application (e.g. critical kernels, etc.). Tailoring the compiler support is both an efficiency optimization and a pragmatic approach to supporting large-scale HPC applications. Architectural features may make only local demands for transformations that in turn have local program analysis requirements. Optimizations involving data organization and layout may require global knowledge of the application. This non-uniformity in the program analysis and optimization requirements suggests that compiler support may need to be tailored to the application and its components. Future research may be productive in supporting the time and space complexity of transformation requirements by adaptively using multiple levels of compiler support (analogous to adaptive mesh refinement for some numerical methods for solving partial differential equations).

5.2 Reconceptualizing and broadening the role of the compiler

Because of the radical changes associated with exascale hardware and applications, it is also important to support research that explores opportunities to reconceptualize the role of the compiler. For example, as the level of programming abstraction rises toward the physics of scientific computing application, the compiler will be a natural place to implement optimizations that explore options for discretization, data representation, and even the choice of solvers. In a sense, the techniques that compiler developers use to search spaces of alternatives - analytically and empirically - should admit the informal "know-how" of computational scientists, to remove aspects of drudgery within their responsibilities; and allow them to focus on science. Automation can provide for consistency in results, performance, range of possibilities explored, that are tedious for human programmers.

With the complexity of exascale hardware and applications, the importance of verification techniques will grow to assure the correctness of an implementation through nondeterministic and asynchronous execution. This suggests that approaches based on advanced systems verification technology, such as Foundational Proof Carrying code, be explored for problems of scientific computing. This may require projects to formalize the know-how of computational scientists into knowledge ontologies that can be used for such verification. Such knowledge to be formalized would include numerical methods and even the scientific theories and models that are under study.

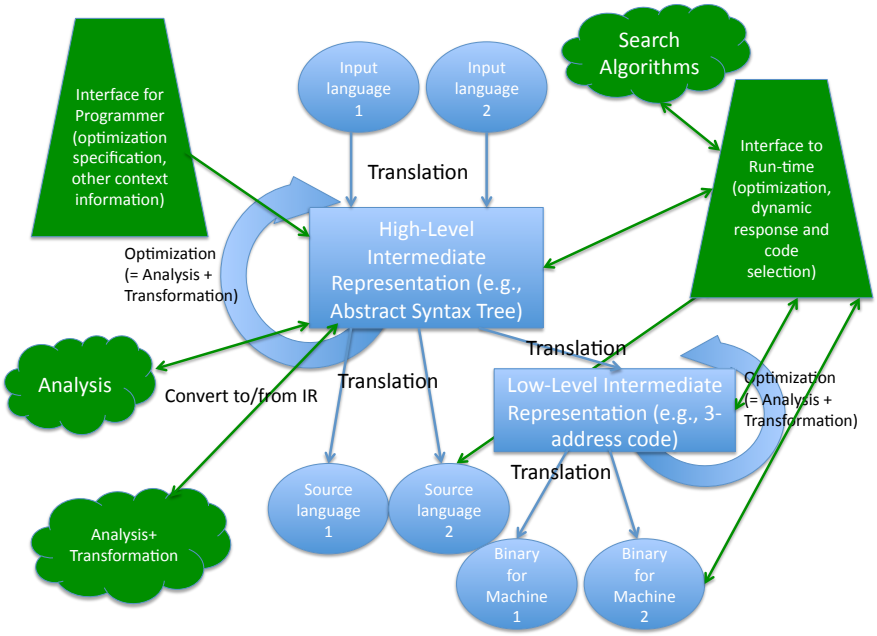


Figure 10: Organizing more modular compilers.

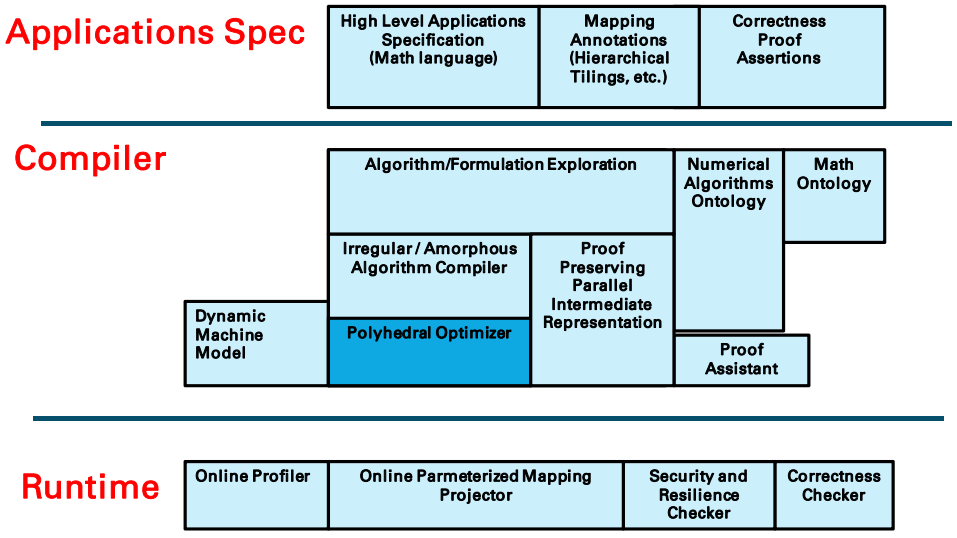


Figure 11: Advanced capability compiler

This would multiply the semantic resources available for automating techniques of uncertainty quantification and would also provide additonal semantic information that could suggest implicit opportunities for optimization, such as new degrees of concurrency or alternative, lower-energy representations and operators.

Framing such knowledge of computational science in a compiler centric manner moves the know how from the opaque bodies of code into reusable modules that can be mix-and-matched with other optimizations. Not only can this result in higher quality of results via verification and higher performance through new optimization opportunities, it can also reduce cost by removing duplication of such efforts across multiple applications and libraries.

A research challenge is in the joint optimization of these new algorithm exploration techniques with established areas of compiler technologies such as automatic parallelization. While programmatically one might want to organize the compiler as distinct modules, the familiar phase ordering problem occurs between algorithm exploration, representation exploration, and traditional compiler techniques such as automatic parallelization. New research into declarative framings of these optimizations into joint solvers is needed.

Another area in which the domain of compiler optimization might be broadened is in support of the increasing degree of multi-physics simulations anticipated for exascale. The opportunity is in automating the scheduling and placement of multi-physics code. This can allow computational scientists to express the computation in terms of distinct physics ensembles, and the underlying physics in terms of the variational forms of the partial differential equations. The compiler could explore different representations, discretations, solver algorithms and different and novel schedulings to fuse different physics modules into regions of high spatiotemporal locality on the exascale architecture.

Finally, the evolution of programming models in more exploratory directions will pose new challenges for compilers. Explicitly concurrent programming abstractions will demand facilities for reasoning about state and operation ordering that are not available on today's compilers. In particular, some of the proposed revolutionary execution model concepts for exascale involve new mechanisms for communication, synchronization, task management, and their regulation/flow control/balancing. These execution mechanisms may depend on complex combinatorial analyses of the problem in order to find separators in the representation that can support good hierarchical placements and schedules of computations over irregular data structures. The research challenge will be in embodying such optimizations for parallelism and locality in the context of other mapping challenges into a newly envisioned supercompilation system.

Figure 11 illustrates a notional modularization of compiler components aligned with this reconceptualization. Algorithms are presented to the compiler in a high-level math form supported by tuning annotations expressed separately and orthogonally. The programmer also provides annotations with respect to correctness, assertions and proof elements for those assertions. The assertion and proof representations may follow from the Curry-Howard correspondence, between assertions/proofs and types/programs, as exploited in contemporaty proof-carrying code compilers used for certifying complex systems code (operation systems, etc.). The algorithm is subject to reformulations performed automatically, guided by a numerical algorithms knowledge base formalized in an ontology. This reformlation would be for the purposes of finding more parallelism, reducing communication, or other other optimization objectives. The knowledge ontology would provide for alternative ways of solving the posed problems amidst declarations of their bounds of applicability, as well as domain knowledge in an expert system form to guide the reformulation to productive outcome. The numerical algorithms knowledge is itself supported by a math ontology. The knowledge is also used to form and check proofs, using a proof assistant tool. The compiler's intermediate representation is designed not only to represent and support explicitly parallel programs, but also

to support the preservation of proof information through optimizations. The runtime performs dynamic optimization, e.g., projecting partial mappings based on runtime information. It also can check the correctness of the implementation and mapping using lightweight checkers.

Achieving this reconceptualization is an ambitious project, but suggests a course for some research. Questions that would be addressed would be such as: how can numerical algorithms knowledge be formalized, in a way that can be injected into parallelizing optimizers? How can power, resilience, and performance objectives be modeled and checked? To embark on this project, technologies developed in the embedded systems community and the certified code and model checking communities, would be juxtaposed with the practice of programming scientific computing systems. While the end-objective is ambitious, such a juxtaposition is likely to result in significant progress toward greater productivity, portability, and reliability.

5.3 Managing fine-grain parallelism, locality and heterogeneity

Three features of exascale architectures will increase the complexity of compiler mapping and optimization techology, and drive changes at the interface between compiler and programming model and between run-time system and compiler. First, exascale systems are projected to have up to billions of simultaneously-executing threads. To identify this much parallel work in important applications, these threads will necessarily be fine-grained, consisting of perhaps just tens to a few hundred instructions. In light of active energy and reliability management, along with varying latencies across the machine, exascale architectures will achieve performance that is different for different runs of the same code. Therefore, these fine-grained threads will execute asynchronously. The compiler will have several roles in managing billion-way parallelism. First, it must translate threads as specified by the programming model into the appropriate run-time calls that will launch an asynchronous thread and notify its dependent threads when it has completed. These fine-grained synchronization operations may involve special hardware support so that overheads are minimized, and the compiler will then be required to generate synchronization instructions. The compiler may also identify fine-grained threads from sequential streams in the application. For example, such as with vectorization technology, it can apply fine-grained automatic parallelization. In addition, the compiler may identify computations that can be migrated into the memory system.

The second challenge in compilers for exascale is the order of magnitude or more anticipated reduction in memory capacity vs. computation rate (i.e., number of bytes per flop). Coupled with the hierarchical processing and memory structures, compiler management of data locality and data movement across the system will become even more important to performance and also essential to managing power and energy. Current petascale and commodity systems increasingly rely on specialized memory structures such as scratchpad memories, configurable caches, non-uniform cache structures, cache sharing across cores and large register files. Along with translating the richer data decomposition semantics of the programming model, the compiler must employ a combination of traditional locality optimizations for cache, emerging techniques for managing software-controlled storage such as scratchpads and registers, and novel techniques for exascale systems such as migrating computation to the memory system.

Locality and parallelism are objectives required for exascale that are in tension. The demand for concurrency will drive toward parallelism, spreading things out, but this can destroy locality. This presents a classical phase ordering problem. Advancing techniques that jointly trade parallelism with locality with a high degree of scope to find a good tradeoff will be essential for exascale.

As the third issue, for efficiency reasons, exascale systems are expected to incorporate heterogeneous processing units with different capabilities, much like today's CPU+GPU supercomputing systems. In addition to supporting code generation across the different processing units, the com-

piler must also generate code that decides how to partition a computation across processing units or selects which processing unit to use given an application's execution context.

5.4 Increasing resiliency

At exascale, the MTTI could be so short, that checkpoint/restart is no longer a viable option for maintaining resilience. New techniques will have to be employed for resilient computational science. In general, only computational scientists will know if an error in their program can be tolerated, corrected, or require a rollback. The compiler will be critical to the efficient implementation of these desired actions. The computational scientist will need to formalize this knowledge, and the compiler role be broadened (aligned with the verification thrust above) to incorporate it into the formation of spaces of legal transformations and to guide new optimizations for resilience.

Compiler support for resilience can include selective use of pairing to detect soft errors, software transactions to allow rollback, and algorithmic tests to ensure the integrity of data. Future work to support resiliency may have to be architecture specific and will almost certainly be application specific. Ideally, compilers would address the above needs automatically, but more realistically they may require direction from computational scientists, who will need new interfaces to describe their resilience requirements. This interface could reasonably be supported by the programming model, or in the near term could be a library or API interface.

A number of approaches to support automated transformations to locally support Triple Modular Redundancy (TMR) are reasonable goals of near future research work. Compiler analysis could reasonably support the transformations and the optimizations required to minimize the associated memory traffic based on different levels of redundancy; per operational unit, per core, per socket, per node, etc. Additionally, memory correctness of data could be verified using more than just the hardware based ECC and errors outside of the range of ECC detected and corrected. Compiler transformations could be used to support this level of additional resiliency. It is reasonable to expect that the analysis to support such transformations, and make them power and performance efficient will be part of future research. Source-to-source transformation is just one way of approaching this problem to lessen the dependence of such technologies on the vendor compiler teams.

5.5 Managing energy and power

Software for future architectures will require either restructuring and/or different algorithms to support the demands of power reduction and overall energy management. The application developer may be able to support this directly, but at some significant burden and likely without much portability. Compiler based transformations may be the only alternative to direct application programmer support for reduced power consumption. This is because the complexity of anticipated hardware controls - voltage controls, clock controls, and schedulers for achieving spatiotemporal energy proportionality - will be high. It will be necessary for the mapped program to adjust these controls on a fine-grained basis in order to maximally exploit dynamic energy saving opportunities. Hardware will likely be overprovisioned relative to power dissipation limits, placing a burden on the software system that extends to the safety and integrity of the hardware itself (preventing melt-down). Consequently, these power scheduling algorithms, which will be tightly integrated with other computation scheduling mechanisms, will demand automation and autotuning not only to reduce the burden on the programmer, but to maximally assure the correctness and integrity of the system.

5.6 Support for Domain Specific Languages and Constructs

Domain-Specific Languages (DSLs) define a means of packaging compiler support for domain specific abstractions. Such abstractions may be useful or critical to either a broad application domain or as narrow as a single application. The economics of addressing compiler optimizations using this approach must be weighed against programmer productivity, performance, and possible single source dependence on the compiler support for such abstractions. Embedded Domain-Specific Languages define a DSL using an existing base-language (perhaps a general purpose languages; e.g. C, Fortran, C++, Scala, etc.) and represent a compromise in the development of DLS. While the embedded DSL limits the design of new syntax (having to avoid too drastic of changes from the base-language) they can reuse significant or even the whole base-language compiler infrastructure. Thus Embedded DSLs are pragmatic and can be efficient to build as a way of packaging custom compiler analysis and transformations for users. While many compiler infrastructures can be extended to support concepts worthy of Embedded DSLs, alternatively a compiler infrastructure for less complex base-languages can be extended in ways that permit at least some new syntax and still provide an efficient means to develop DLSs. Much DSL work is related to and leverages existing compiler and/or language infrastructures.

Domain-Specific Languages's can leverage source-to-source compiler technologies to further provide an economical means to exploit specific hardware and existing compiler technologies (e.g. for GPUs; using CUDA and/or OpenCL as just one example). If designed well, and within a common framework, or using a common base-language, DSLs could even be staged to define hierarchies of more narrowly defined compiler solutions. The design of such systems is a worthy research direction since defining composable subsystems (compiler building blocks) may be key to opening the exploration of new DSLs and/or programming models to make future architectures tractable to HPC application teams.

6 Runtime Systems: challenges and strategies

When considering the challenge of supporting billion-way parallelism for an Exascale system, it is widely agreed that the bigger disruption will occur at the intra-node level rather than the inter-node level. This is because the degree of intra-node parallelism needs to increase by about three orders of magnitude relative to today's high-end systems on the path to Exascale, while the degree of inter-node parallelism only needs to increase by at most one order of magnitude. Other challenges at the intra-node level include vast degrees of performance and functional heterogeneity across cores within a node, as well as severe memory and power constraints per core. Taken together, these challenges point to the need for a *clean sheet* approach to intra-node runtime systems. They also have a significant impact on inter-node runtime systems, because any successful inter-node runtime system for Exascale systems must be able to integrate well with new intra-node runtime systems (as discussed in the multiple references to "hybridization" during the workshop).

Power management offers an interesting opportunity in the runtime system for understanding and leveraging trade-offs between energy efficiency and performance. One of the examples cited in the workshop was an application on a production system, for which it was possible to reduce the total energy consumed by 50% while only reducing performance by 10%. A figure of merit such as the energy-delay product can help take both energy and performance into account.

6.1 Runtime Support for Exascale Execution Models

Though there is a variety of execution models under consideration for Exascale computing, there is also a consensus in the community on key abstractions that will need to be present in an Exascale execution model and that the runtime system (assisted by the compiler and architecture) must provide first-class support for these abstractions. Exascale execution model abstractions that will need runtime support include the following:

- *Lightweight tasks:* It is widely accepted that the parallelism exposed by an Exascale application must be over-provisioned by 1-2 orders of magnitude relative to the available parallelism in hardware on an Exascale system. Even with lightweight OS kernels, it is unlikely to be practical to use OS threads for this over-provisioning, especially due to the memory overheads imposed by standard OS threads. Thus, an Exascale runtime system must support large numbers of lightweight tasks that are scheduled on a fixed number of "worker" OS threads (typically, one per core or hardware context). A number of recent multicore programming models (e.g., OpenMP 3.0, Cilk, TBB, and a multitude of dag-parallelism schedulers) already assume the availability of lightweight task parallelism. In an Exascale runtime, there will be additional challenges in supporting lightweight tasks that include support for heterogeneous processors and integration with asynchronous inter-node communications. Support for end-to-end asynchrony across tasks and communications remains a major challenge.

- *Memory hierarchies and data movement:* While memory hierarchy details will vary from platform to platform, there is agreement that Exascale software will have to deal with deep memory hierarchies combined with the need to minimize data movement due to energy limitations. Thus, the task scheduling runtime capabilities discussed above will need to be locality-aware and be capable of supporting function shipping and data shipping as interchangeable alternatives.

- *Coordination of dynamic task teams:* Past approaches to coordinated parallelism such as thread groups and communicators involved a fixed number of threads e.g., the set of threads

or processes synchronizing on a barrier does not change after the first barrier operation is performed. However, we should expect dynamic task creation and termination to be a frequent and common operation in future Exascale systems. Thus, there's an urgent need for synchronization operations such as barriers to be performed on dynamically varying sets of tasks. The coordination can be in the form of a barrier, in which the set of tasks that need to synchronize could potentially change in each phase, or in the form of a special join construct for which the set of tasks can also vary dynamically.

- *Atomicity and transactional semantics:* Recent work on hardware and software approaches to transactional memory has highlighted the benefits of a transactional model for tasks that mutate shared locations. However, it is also widely recognized that global atomicity cannot scale to the billion-way concurrency required for Exascale systems; instead, it is more appropriate to limit atomicity to sets of tasks that execute in the same "coherence domain" (also referred to as a locale or place in Chapel and X10 respectively). Function shipping can then be used to create tasks for execution in remote locales.

6.2 Intra-node Runtime Support for Multicore Processors: Current Trends

While considering future directions for runtime systems for exascale systems, it is worthwhile paying attention to current trends in runtime systems for multicore processors. These runtime systems focus primarily on intra-node parallelism and, in most cases, leave it to the user to deal with integration with inter-node runtime systems.

Runtime System	Prog. Notation	Cont. Support	Greedy Scheduling	Stack Bound	Queue Bound	Scheduling Policy	Affinity Support
OpenMP 2.5	Pragma	No	Yes	Yes	No	Work-sharing	Limited
OpenMP 3.0	Pragma	Yes	Yes	Yes	Yes	Work-first work-stealing	Limited
Cilk/Cilk++	Language	Yes	Yes	Yes	Yes	Work-first work-stealing	No
StackThreads	Library	Yes	Yes	No	Yes	Work-first work-stealing	No
Intel TBB	Library	No	No	Yes	Yes	Help-first work-stealing	Yes
Habanero	Language	Yes	Yes	Yes	Yes	Adaptive work-stealing	Yes

Table 2: Comparison of Selected Multicore Runtime Systems

Table 2 summarizes the comparison of some well-known multicore runtime systems from industry and academia[3]. The Programming Notation column lists how the task parallelism is specified by the user. The Continuation Support column indicates whether or not the runtime system provides support for a worker thread to suspend a task before completion (e.g., when a blocking operation is performed) and move to another task, while allowing the task to be resumed on the same or different worker when it becomes unblocked. The Greedy Scheduling column indicates whether the task scheduling algorithm is greedy i.e., whether it ensures that a worker will never be idle if there's work available to be done by the worker. The Stack Bound column indicates whether the scheduling algorithm is able to limit the maximum stack size needed for a worker relative to the stack size needed by a sequential execution of the program. The Queue Bound column indicates whether the scheduling algorithm is able to limit the total size of all queues in the runtime system. The Scheduling Policy column indicates if a work-sharing or a work-stealing policy is employed by the runtime system, and (in the case of work-stealing) if a help-first or work-first policy is used. The Affinity Support column indicates if the runtime system supports any form of affinity among tasks or between tasks and data.

[3]This table was adapted from Table 7.1 in [22].

Most OpenMP 2.5 implementations followed the SPMD origins of OpenMP that focused on work-sharing in parallel regions. As a result, OpenMP 2.5 runtime systems usually lack robust support for nested parallelism, and can exceed all reasonable bounds on queue size by creating an exponential number of tasks in a queue in the worst case of nested parallelism. There was a significant change in runtime scheduling technologies when moving from OpenMP 2.5 to OpenMP 3.0, due to the introduction of explicit task parallelism in OpenMP 3.0. Most OpenMP 3.0 implementations now use work-stealing schedulers and are able to support nested parallelism to arbitrary depths. Both OpenMP 2.5 and OpenMP 3.0 support limited forms of affinity through the *static* scheduling clause for parallel loops, and (in the case of OpenMP 3.0) the notion of *tied* tasks.

Cilk is a C-based dynamic task parallel language developed at MIT, accompanied by the Cilk runtime [21] which is based on the work-stealing algorithm in [11]. Intel's Cilk++ is a recent extension to Cilk that is based on the C++ programming language [47]. The differences between Cilk++ and Cilk mostly lie in language features rather than the runtime system. Both Cilk and Cilk++ use the work-first work-stealing scheduling policy for all spawned tasks, and do not support any form of affinity scheduling.

StackThreads/MP is a library that supports fine-grain multi-threading in GCC/G++. StackThreads/MP uses a scheduling scheme that is a combination of work-sharing and work-stealing. In StackThreads/MP, the idle workers send steal requests to busy workers. A steal request can be picked up by the busy worker through a polling routine inserted in the program. Once a steal request is picked up, the victim serves the request by preparing the execution context for the thief. After the context is ready, the thief is notified and begins execution. As a library, StackThreads/MP has a very unique stack management model to allow programs to compile with standard compilers (notably, gcc) a standard calling convention, while allow task suspension, resume and migration. In StackThreads/MP's stack model, each worker thread has a logical stack as well as a physical stack. One worker's logical stack frames may be spread across the physical stacks of all workers. Task suspension, task resume and task migration and are all implemented by linking the frames of the worker's local stack. For example, when a task is suspended, the logical stack frames are unwound and saved as continuations. When logical frames are unwound, they still remain on the physical stack. It is known that the StackThreads/MP can exhibit fragmentation in the physical stack and can overflow the stack for large applications (due to the lack of a stack bound).

Intel's Threading Building Blocks (TBB) model is a C++ template library for exploiting task parallelism on multicore processors. The TBB library does not include any continuation support. When a task is suspended and the worker goes stealing, the old stack frames remain on the runtime stack. As a result, the suspended task can only be resumed by the same worker that suspends the task (like tied tasks in OpenMP 2.5); TBB also restricts the stolen task to those deeper in the spawn tree than the suspended task in order to avoid stack over now in the worst case. These two restrictions reduce the effectiveness of the load balancing performed by TBB. It has been shown that the depth-restriction can asymptotically serialize execution in cases where standard work-stealing achieves linear speedup [63]. To mitigate these problems, TBB allows the programmer to manually create continuation tasks. However, this approach is essentially trading productivity for performance. In contrast to the Cilk's work-first execution of all spawned tasks, TBB uses the help-first policy upon task creation. TBB also has an `affinity_partitioner` structure to utilize temporal cache-reuse by binding the same iteration to the same worker thread that previously executed the iteration. TBB allows stealing regardless of the affinity and has a mechanism to reduce counter-productive stealing.

The Habanero runtime system developed at Rice University has been designed to provide first-class support at the intra-node level for the execution model primitives identified in Section 6.1. For lightweight task parallelism, the Habanero runtime includes an adaptive work-stealing runtime

system for terminally strict async-finish computation graphs (which are more general than fully strict spawn-sync computation graphs) [23], and are further extended with data-driven tasks to support general forms of dag parallelism [65]. For memory hierarchies, it supports the hierarchical place tree abstraction [68] along with locality-aware scheduling [23]. For coordination of dynamic task teams, it unifies barrier and point-to-point synchronization in phasers with support for dynamic parallelism and asynchronous collectives [55–57]. For atomicity and transactional semantics, it includes a delegated isolation algorithm [35] with performance and scalability that is shown to be superior to software transactional memory systems and comparable to more complicated user and library approaches that enforce mutual exclusion with fine-grain locking.

6.3 Integration of Intra-node and Inter-node Run-time Systems

Current systems provide a programming model run-time as part of the library that supports that run-time: Thus, in a system running MPI+OpenMP, the MPI library includes a run-time for managing communications and the OpenMP library includes a run-time for task scheduling, synchronization and memory allocation. The two run-time systems share the common services of the OS, but are not integrated, otherwise. This results in performance problems that will worsen as the number of threads per node increase, and intra-node memory accesses become less uniform: The use of threads to support communication is not coordinated with the allocation of computation threads; message-driven scheduling of tasks (e.g., in an active message model) is inefficient; the physical location of communication buffers is not coordinated with the location of physical threads that produce or consume the buffered data; and one cannot intelligently manage the trade-off between critical path scheduling and buffer space reduction.

Future systems will need a better integration. There are two possible approaches to this problem:

1. A design for run-time interoperability, that would enable multiple run-time systems within a node to to coordinate their activities

2. A common run-time that provides services to all programming layers running on a node.

The first approach has the advantage of enabling the coexistence of multiple run-time systems, each possibly associated with a different programming language or library; it facilitates separate development and support of these languages and libraries, and facilitates reuse of existing investments. During the workshop, this approach was referred to as the MPI+X runtime or PGAS+X runtime for the cases when the inter-node runtime system is based on MPI or a PGAS implementation and the X refers to an intra-node runtime system as in Section 6.2.

The second approach can provide better integration; it represents a clean sheet approach that is better suited to a future unified programming model that can handle both intra-node and inter-node parallelism and communication.

Whichever approach is followed, a modular organization will facilitate development and maintenance. The two alternatives are, essentially, whether to group run-time functions according the library or language they support, or group them according the resources they manage and services they provide: memory management, CPU management, power management, communication resource management, etc. As is of ten the case, the answer is likely not be be black and white – with a lower layer of services that are resource specific, and a higher level of services that are programming model specific.

6.4 Runtime Interoperability

The discussion in the previous section also applies in the case where multiple programming models are supported on a node. Computing research has produced a rich variety of parallel programming models and runtime systems, each with its unique set of capabilities and advantages. Different programming models provide differing idioms for expressing parallelism, managing the work of a computation, and operating on distributed and shared data structures. Their associated runtime systems provide the low level capabilities needed to support a variety of high level models as well as the layer of abstraction needed to efficiently harness increasingly complex hardware systems.

Many of the challenges that arise when combining multiple programming models stem from the incompatibilities between runtime systems of these models, which are not able to communicate with each other and share resources, often resulting in poor or erroneous application behavior and resource stalemates. For new programming models to be successful, it might be desirable for their runtime systems to be interoperable with existing runtime systems that applications use, including all the other libraries they are composed of.

In short, for a completely new model, that provides its own separate runtime system, to be practically usable many existing libraries including math libraries, simulation tools and other frameworks, would all need to be ported to work with this new model. This might be an expensive proposition as it would not take advantage of the previous investment from DOE and other funding agencies. With a runtime system that is interoperable with legacy runtime systems, it might be possible to alleviate some of these issues.

6.5 Decentralized Monitoring, Data Aggregation, Data Analysis, Data Mining, and Introspection

There is a vast gulf to be bridged between hardware performance monitors (HPMs) that provide detailed low-level information on the behavior of individual cores, and the holistic view that an application programmer will need to understand the behavior of their program on millions of cores in an Exascale system. It is essential that the monitoring of the whole program be decentralized. Further, scalable approaches to data aggregation will be needed to help application programmers and algorithm designers gain high-level insights into their behavior of their programs, since it won't be practical to just stream all the HPM information from all cores in an Exascale machine to a single development node where the programmer runs their performance tools. Instead, it will be important to use the Exascale machine itself to perform aggregation, analysis, and mining of performance data. These aggregation techniques will also need to perform extrapolations to predict counts for events that are not directly tracked by HPMs (e.g., communication events from the interconnect) but can be reliably correlated with other HPM events. Mapping back from the HPM level to the source-code level also poses significant challenges.

Having aggregated the data, the Exascale system should also be leveraged to support Data Analysis and Data Mining, by performing scalable on-the fly analysis of the aggregated data as opposed to detailed analysis of individual traces. Implicit in this analysis is a set of important decisions about what information to collect, analyze and discard or retain. The retained data must be organized in a performance database that provides meaningful information to subsequent executions or phases in the current execution, but also provides efficient access for on-the-fly analysis. In general, on-the-fly analysis can be viewed as a form of introspection that enables users and tools to monitor potential performance bottlenecks.

6.6 Fault Tolerance: Detection and Recovery

Given the resilience challenges for Exascale systems, it will be necessary for the runtime system to support multiple techniques for error detection and recovery (e.g., checkpointing, redundant computation, suspend-retry) while leveraging available monitoring techniques (Section 6.5) as well as machine learning techniques.

Efficient, localized error handling will require the involvement of the run-time in error recovery and, possibly, in error detection. This will require new run-time functions and new interfaces between the OS (that manages changes in the configuration of the "virtual user machine" and alerts from the hardware) and the run-time.

The run-time can provide enhanced protection from programming errors, e.g., a combination of compiler, runtime and hardware can provide efficient race-detection in shared memory, or deadlock-detection. It may be desirable to support different levels of safety, even within one execution, and the run-time will have to support such choices.

6.7 Compiler/Runtime/User Interface

As previously stated, both codes and systems become more complex, dynamic and irregular: The behavior of codes change as they adapt to the evolution of the simulated system; the behavior of computers change as they adapt to failures or manage power. Therefore, "run-time compilation" where the executable is changed dynamically during execution becomes increasingly important. "Run-time compilation" methods range from the simple, such as the use of code that is parameterized by machine parameters, or the selection of different methods, according to input or machine characteristics, to the complex, such as auto-tuning and run-time code morphing.

Run-time compilation requires a tighter interaction between compiler and run-time, as the run-time provides to the compiler performance information that is needed to tune the code, and the compiler can provide to the run-time information's about the code that can be used to improve its execution environment.

Such a tighter interaction will also help a looser loop of off-line code tuning, where a compiler improves code based on the properties of previous runs. This is essential when the expert programmer is in the loop, and program tuning results from a close interaction between compiler, ran-time and programmer" The information produced by compiler and run-time have to be exposed to the user in a framework that relates to the source code, so as to guide refactorings of this code.

6.8 Runtime/Architecture Interface

New hardware features can reduce the overheads of thread management, memory management, communication and synchronization. Most of these features will be used by the run-time. Therefore the interaction between run-time designers and architecture designers is an essential component of the co-design process. New run-time prototypes should be guided not only toward the efficient use of current machines, but also toward the testing of new hardware features for exascale machines,

6.9 Interaction between OS and Runtime

The design of an exascale machine will require new thinking about the division of labor between OS and run-time. The OS isolates one application from another, and provides resources to the "virtual user machine" that runs each application; the run-time manages resources within one "virtual user machine" Current OS structure reflects the need for time sharing of one physical node by multiple processes (each being a "virtual user machine"); One system where nodes are dedicated to one

application and communication outside the "virtual user machine", e.g., for I/O, is filtered by a smart NIC, it is not clear why allocation of node resources (CPU, memory and power) should involve the OS. Experience with systems with no OS, such as Anton, show that they can achieve ultra-low communication latencies (200ns).

7 Path Forward

7.1 Introduction

The timeline for Exascale is driven by the hardware, which will arrive in 2020 whether or not the software is ready. Therefore it is critical to ensure that technologies necessary for the effective exploitation of exascale platforms will be available in a timely manner, both to influence the co-design process for the systems and to ensure delivery of computational discoveries soon after deployment. This is a very short time span in a normal lifecycle from a research project to a production system. This will require careful consideration to find the least resistant path from current technology to what is required to be deployed on exascale platforms. Thus, it is important to look at evolutionary solutions to exascale platforms. On the other hand the current state-of-the-art software stack is far from ideal. It is already showing many weaknesses that will only get amplified as we move to exascale systems. Thus, it is imperative for the community to explore revolutionary alternatives that can fundamentally alter the software landscape and eliminate many of these vexing problems. These could be deployed by 2020 if research progresses fast, or later, otherwise.Therefore we propose a multi-prong approach to going forward. An evolutionary path will take the existing MPI based approach and develop an MPI+X solution that heavily rely on the current technology when possible and develop novel approaches when necessary. A revolutionary program will look at innovative alternatives that can fundamentally eliminate vexing problems and provide a platform to build portable, high-performance software in a cost effective manner while achieving greater flexibility and productivity. The program as a whole should include a balanced mix of lower risk projects and higher risk, higher return projects. The evolutionay path should provide a complete, low-risk solution to the problems of exascale programming. The revolutionary program is not a standalone path to exascale, but a set of opportunities to fundamentally alter the evolutionary path.

The program will have to do with the well-known problem of "crossing the chasm", or "crossing the valley of death" in discontinuous innovations [43], illustrated in Figure 12: Crossing the gap between the first "visionary" adopters and adoption by a large group of pragmatic users. While the first group will be willing to put up with much pain, the second group will require a technology that is robust and well supported on all relevant platforms. This is especially true for new programming models: As a rule, programmers are reluctant to change programming model, unless they are forced to do so; i.e., unless some key goals cannot be achieved using the old programming models. The evolutionary path will provide the users with a familiar programming model, but will require a continuous porting effort to get from one system to another. The revolutionary path will require a complete rewrite of the software stack, with the hope of performance portability that will eliminate or significantly minimize requirements for porting to future platforms.

7.2 Evolutionary Path

We assume that the collaboration between DOE and the vendors will define an approach to the achievement of exascale performance that minimizes risks and mazimizes returns. One approach that minimizes risk assumes as few changes as possible in current programming models. The approach is currently defined as MPI+X, and assumes that MPI is used for communication across nodes, while "X", which is most likely an evolution from current OpenMP, is used for intra-node parallelism. An evolved OpenMP will have to address all of its current limitations, such as lack of scalability, lack of composability, no dynamic parallelism, and no load balancing. We shall call this the *base programming model plan.*

This plan is not risk-free and requires DOE research and advanced development for its implementation:

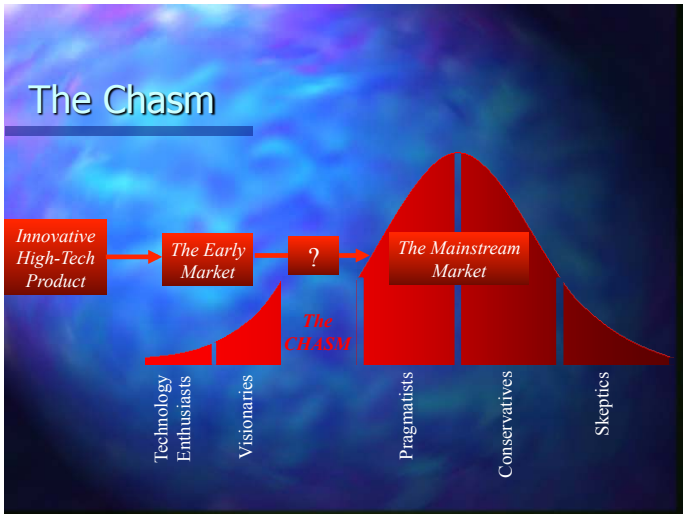


Figure 12: Crossing the Chasm

MPI has to evolve to support higher-levels of parallelism, to better use the communication hardware of current and future supercomputers and to provide better interfaces to nodes that run a large number of threads. Many of these issues are being considered by the MPI3 forum and continued investments will be needed to support MPI evolution.

OpenMP has to support a better management of locality in a heavily NUMA platform, provide better interfaces to GPU-like accelerators, and better interact with MPI. Some of these issues are being considered by the OpenMP Architecture Review Board, and investments will be needed to ensure the availability of good OpenMP implementations to these specifications. OpenMP implementations are typically supported by hardware and software vendors investment in their product lines; separate DOE funding in OpenMP implementations is likely to be redundant.

Work is also needed on a better interoperability between MPI and OpenMP to avoid the current model of serializing MPI calls.

Note that an "enhanced OpenMP" could embody a programming model that is quite different from the current OpenMP programming model, where parallelism is expressed as a flow graph of tasks and inter-task dependencies [16]. This would provide increased asynchrony; if the tasks are executed transactionally, then the model can provide error isolation. Similarly an "enhanced MPI" could provide a programming model that is quite different from the current MPI programming model: If MPI processes are virtualized, as done in Adaptive MPI [26], then the model of a fixed number of processes can be replaced with a model of a dynamically changing number of processes, with run-time load balancing. A better design and implementation for one-sided communication as demonstrated by ARMCI [44] and GASNet [12] and will move MPI to support a programming model similar to that provided by libraries such as Shmem [8] or GA [45], or PGAS languages such as UPC [18] and Co-Array Fortran [49].

Ultimately MPI and OpenMP specifications could converge, nominally evolutions of the specifications but adopting the more revolutionary execution model concepts that have been proposed, such as uniform treatment of parallelism between the dimensions of node and system. With exascale hardware promising very deep hierarchies of hardware partition (chips with themselves 4 to 5 levels of hardware partitioning and hierarchy) and system level packaging with another 4 to 5 levels

(carriers, modules, boards, chassis, cabinets, ...) the notion of node will blur making irrelevant the current basis for the distinction between MPI and OpenMP.

7.2.1 Considerations

Given such a base plan, and given the difficulty of moving from research ideas to deployed technology, we believe that the following questions should be asked for any new research in programming models:

To what extent is the new technology necessary? The report identifies several issues in exascale computing that are likely to require handling in a programming model, even if one follows the base plan. One approach is to handle these issues within the context of the evolutionary approach, in order to minimize risks. This would follow the approach above where the evolutionary approach increasingly integrates new research products.

Simultaneous DOE investment in alternative technologies is critical as the evolutionary approach has a high risk of not being able to handle these issues by itself; these alternative approaches can be tested and ready for integration as the evolutionary approach encounters anticipated barriers. A pool of innovative approaches may initially seem to be of limited capability compared to today's established production MPI-based DOE application-supercomputing system complexes. However, the rate of improvement in current established programming approaches is low. New, innovative, revolutionary technologies, unburdened by source legacies, may achieve a more rapid rate of improvement, to not only match but surpass the established technologies [20]. These technologies would address the goals below.

Scalability MPI will need to scale to higher process counts and to provide more scalable collective operations. One-sided communication layers, whether integrated into MPI or separate, need to scale as well. Both may need to handle a hierarchical model of the system topology. OpenMP will need to scale better to large thread counts.

Functional Portability The portability of MPI and OpenMP have been essential to the HPC ecosystem, but given the divergence of node architectures, portability will be harder to maintain in the future. Programming abstractions – whether in OpenMP or in some new model – that perform well across a variety of heterogeneous and homogeneous nodes are needed.

Performance Portability If too many machine specific optimizations are embedded in MPI or OpenMP, it will become difficult, if not impossible, to get good performance in a different machine without extensive rewriting and performance tuning. It is important to study how to bring performance portability to programs and evaluate alternatives to MPI and OpenMP that will provide performance portability with minimal programmer effort.

Finer granularity Strong scaling requires support for finer grain tasks, hence a reduction in the overheads per inter-node communication and intra-node synchronization. It is not clear that the per message overhead can be significantly reduced for MPI. Alternative solutions (e.g., PGAS languages, codelet approaches) might be needed.

Locality The energy limitations of future platforms will require significant reductions in communication (in space) and in storage (i.e., communication in time). While the MPI model provides explicit control of communication, the OpenMP model provides no support for reducing communication and improving locality. It is essential to fix this problem. Polyhedral methods provide ways to analytically manage locality in static and dynamic form. Cilk and Habanero

provide elegant approaches based on cache oblivious algorithms for achieving complex hierarchical locality.

Hierarchy Even though many computer scientists think of "THE programming model" and "THE programming language," domains that have succeeded in exploiting parallelism like dense linear algebra and databases take a hierarchical approach to programming, using a hierarchy of models and languages that exploit abstraction. In linear algebra for example, one has carefully tuned BLAS at the lowest level, something like LAPACK on top of that, and MATLAB at the highest level. Designing the interfaces between layers carefully, performing cross-layer optimization, permitting higher layers to choose implementations in the lower layers (e.g the implementation of join) are all essential to achieving performance. This is central to the conception in the Galois system or the linear algebra DSL of Norris at al. [9]. As an increasing number of DSLs, libraries and frameworks are built on top of MPI+X, more work can shift to address these higher programming models, facilitating the shift to new models.

Heterogeneity Nodes are likely to have accelerators (GPU, MIC, FPGA). The "X" node language needs to provide a suitable support for those, in order to avoid an MPI+X+Y" solution. High-level compilers can avoid the MPI+X+Y problem by accepting high-level descriptions of algorithms and automatically performing the low-level code generation in the current X+Y; retargeting the compiler would address changes in X and Y without rewriting the application base. In any case, one needs to support better interfaces between the various programming models, e.g., to avoid sequential bottlenecks between MPI and OpenMP.

Explicit Memory Space Management Nodes programming may well have to handle software managed scratchpads with affinities to different node components, as well as a level of memory between current DRAM and disk. Both intra-node and inter-node programming are affected to avoid unnecessary copies as one communicates data through multiple memory spaces.

Asynchrony The increased variation on execution speed of various components, due to error recovery and power management, will require codes that are more tolerant to noise, hence, more asynchronous. Increased asynchrony can also be a major contributor to energy reduction. While it might be possible to achieve this goal using the MPI+X *programming notation*, this will require a change in the prevalent, bulk-synchronous *programming model*. Tools and methodologies for the conversion of bulk-synchronous code to asynchronous code are important.

Resilience It is essential to reduce the overheads of the current global checkpoint-restart approach for error correction. It is likely that this will require some programming model support to take advantage of new non-volatile memory technology. To make applications resilient to individual component failures without a full application restart, programmers will need enhanced error detection.

7.3 Revolutionary Path

In the revolutionary path we are looking for radically different approaches that can eliminate many of the current and anticipated problems with the evolutionary solutions. Although MPI is the preferred method of high performance programming today, everyone agrees that it is far from ideal. MPI makes it possible to get high performance by exposing many of the underling architectural details and asking the programmer to manually perform the bulk of the performance tuning. Thus, the evolutionary path will still require extensive program modifications and manual performance

tuning when porting a program to each new architecture. While the evolutionary path will look at the novel problems introduced or existing problems magnified at the exascale level, it may be difficult for any MPI based approach, which was originally designed under very different set of constraints, to provide viable solutions to these problems. When the exascale imposed constraints are approached with a clean slate using modern methods, tools and technologies, we hope solutions that emerge will eliminate some of these vexing problems. For example, recent advances in machine learning based approaches to autotuning programs, coupled with the right language design, may be able to eliminate much of the performance tuning burden from the programmer. If the community does not invest in finding novel and radical solutions, in the long run the life of the high performance programmers will get increasingly difficult and the cost of creating and maintaining high performance programs will become unsustainable.

The revolutionary solutions we are looking for have to demonstrate viability for high performance programming. These solutions have to demonstrate that they can handle a large class of programs that are considered important by the community. The "mini applications" provided by the application community will be a good start. However, it is also necessary to demonstrate that these systems can scale up to realistic applications. They also have to demonstrate that it is possible to achieve high performance comparable to, or better than, the evolutionary approach on exascale system.

As revolutionary solutions will require more investment for adoption than the evolutionary approach, it is important to demonstrate clear benefits of these approaches. We are looking for solutions that will clearly reduce the burden we are placing on high performance programmers. A revolutionary approach need not focus on yet another language – it needs to focus on the process of generating software from the mathematical specification to the executable, and of modifying such software to provide new functionality or to port to a new platform. A better programming language may be one component of the solution but it is highly unlikely to be the entire solution. Researchers should be encouraged to think about the entire software development process and of ways to accelerate it. For example, a novel approach could radically reduce the programmer involvement in porting to a new architecture, tuning a code, or testing it.

When looking at revolutionary approaches, it is unrealistic to ask for a comprehensive solution that is close to deployment. The researchers should be able to address one or two specific problems. However, it is important to demonstrate the viability of such a solution in a broader context by perhaps integrating their solution into an existing framework. While revolutionary solutions may not lead to a product at the inception exascale systems, there should be a clear path to integration and productization in a reasonable timescale.

7.4 Global Considerations

Is the technology likely to be become available from other sources? The DOE funding for exascale computing will be limited; it is important to focus investments on technologies that are unique to exascale, while reusing as much as possible technologies that have a broader application domain. For example, investments in core compiler technologies should be considered with care, since core compiler technology has a broad applicability; it must be more productive to extend successful open source and commercial COTS compiler infrastructures (gcc and LLVM; possibly, Open64 and Rose, PGI, icc, R-Stream, etc.) than develop new ones from scratch. Decisions in this area have to carefully balance the unique needs of exascale with the savings accrued form using technologies with a broader applicability; and the intrinsic advantages of a particular infrastructure with the viability of this infrastructure, due to its position in the market. This is the critical "ubiquity" concept which already recognizes that exascale systems are not alone in facing certain

challenges. For example the Near Threshold Voltage (NTV) innovations likely to provide significant power savings for exascale are emerging from commercial investments targeted primarily at mobile devices.

To what extent does new technology improve productivity? Even if all the problems listed in the previous paragraph can be solved within the context of the evolutionary approach, it is quite possible that the solutions will significantly increase the programming effort needed to achieve good performance at exascale with codes of interest. Alternative technologies that can achieve the same goals with significantly lower programming efforts should be considered – provided that the expected payoff is very significant. Work in this area has to be in the "high-risk, high-payoff" category, because of the many obstacles in the adoption of totally new programming models: The X10 goal of HPCS has not yet led to changes in the way we program; furthermore, code programming is a very small fraction of the overall code development effort that includes verification, validation and code tuning. It is quite possible that the more important impact could be achieved with new V&V or code tuning methodologies, rather than with totally different programming models. For this reason, changes to the programming models to enhance V&V efforts, such as increasing the semantic content and enabling the use of automated theorem proving, could be very valuable.

To what extent is there a clear path to technology deployment? Research on programming models for exascale needs to have a well-defined path for timely deployment, if successful. We shall want every significant technology to be well-supported on all exascale platforms; a non-proprietary implementation is also highly desirable to mitigate the risk of a vendor bailing out. Successful deployment will be needed within eight years or less from research start. Therefore, it is important that research not only provide a proof of context, but also a good quality implementation that can be easily ported by vendors or by a DOE team. Such a port is facilitated if the prototype development used a broadly available infrastructure (e.g., broadly available compilers, run-times and libraries).

To what extend is there a clear path to application migration? Application running on exascale platforms will have been mostly developed before new programming models or technologies created by research starting now will see fruition. Thus, the deployment of a new programming model cannot depend on the assumption that exascale codes will be developed entirely from scratch in this model. A migration path should include solutions for *interoperability* – i.e., mixing the new with the old; and for *portability* – ie.e., transforming the old into the new.

How easy it is to develop the new technology into a complete solution? A new programming model cannot stand on its own: It requires good support from programming environments (such as Eclipse), debugging and performance tools, compilers and run-times; codes need to be able to use existing computational libraries, parallel I/O libraries, visualization libraries, etc. This "complete" support will need to be deployed this decade, with limited funding. Point solutions that require significant additional work to be deployed in broad use may not be appropriate.

Is the work done at the level that most facilitate insertion? Functionality required for exascale may be introduced at different levels of the programming stack, resulting in different trade-offs. Consider, for example, the issue of providing more advanced functionality for data distribution and communication control and for co-locating of data and computation, in environment where nodes come and go, and where both data structures and computations change dynamically.

This issue is critically important for reducing energy consumption and providing resilience. It can be handled (a) transparently, in a run-time; (b) as part of the programming language, as done by Chapel and X10; or, (c), as part of a framework for distributed data management that sits above a programming language such as C++ and a communication library such as MPI (Heroux' model). Each of the choices has advantages and disadvantages: a pure run-time approach would be ideal for the viewpoint of the application programmer, but might not be feasible. A language approach facilitates optimizations, and is likely to be more elegant. However, it requires significant investments in solution that has a high threshold for adoption and is less flexible. A framework approach may impose a bigger burden on the library developers and may limit possible optimizations, but is much easier to ingest in the existing infrastructure; it can evolve, over time, from a simple framework implemented in C++ and MPI into an embedded domain-specific language, supported by its own run-time and optimization framework. Such trade-offs must be carefully considered when assessing research projects.

Several high-level compiler projects have demonstrated that it is possible to automatically generate good OpenMP from high-level specifications of application kernels, including complex hierarchical nestings of doalls and complex tilings. It may be that OpenMP and MPI will naturally evolve to being good target languages for higher level programming tools, rather than the applications programming layer.

An example: For illustration, consider a potential "revolutionary" project that would develop a new programming language that can be used to manage parallelism and communication both intra-node and inter-node, thus replacing both MPI and "X".

Replacing both MPI and "X" has a good potential for improving productivity, so that such a research direction is certainly worth considering; such technology is unlikely to come from another market.

But this is not sufficient: Such a new language will have to provide performance comparable to MPI+X at scale, provide good control for locality, handle heterogeneity, enhance asynchrony and provide a solution to resilience. The new language should interoperate with an MPI+X model and tools should be provided for migrating MPI+X code to the new language. One will need to provide robust optimizing compilers and debugging and performance tools for the new language on multiple platforms. It should be clear how vendors could support such a new language in a time frame that is consistent with the exascale deployment goals. Finally, the new language should offer clear advantages over alternative ways to support a similar programming model, e.g., through extensions to C++.

It is unlikely that all these issues can be addressed in an initial research projects. On the other hand, such an initial research project should be able to outline a rationale why the proposed approach would be able to handle these issues, and a path toward the deployment of the proposed technology.

A research plan in exascale cannot be judged only by the quality of each of the components; one also needs to assess the overall research portfolio for its consistency and complementarity.

In particular, one needs to pay attention to the following issues

Is the portfolio sufficient? Do we cover all the bottlenecks to exascale computing?

Are efforts complementary? The DOE exascale program will be under budget pressure; therefore, it is essential to ensure that efforts in different areas are complementary – optimizing the whole

more than the individual projects. For example, research in run-time, compilers, debuggers and performance tools should be complementary to research on new programming models.

One important way to ensure complementarity is to invest in infrastructure that can serve multiple projects, e.g., a run-time that can support multiple programming models, or an auto-tuning infrastructure that can be used with many programming models.

Does the portfolio properly balance risks and returns? A good research portfolio has to mix low-risk low-return work with high-risk, high-return research. Investments should not only be guided by the intrinsic merit of each project, but by the need to achieve a proper balance. Also, the portfolio has to provide more coverage to those areas that are seen as more problematic, and less to those where the base plan is clearly adequate.

7.5 Cross-Cutting Issues

Research in new programming models will face a set of organizational issues that are cross-cutting across of components of such research:

7.5.1 Internal Collaborations

The research will involve DOE Science labs, NNSA and academic partners. These different partners have different missions, different environments, different skills and different funding models. It is important to organize the research so as to leverage the advantages of each of these partners.

7.5.2 External Collaborations

Research in programming models will need to collaborate with research in other areas, such as applications or architecture. The interactions have to be though both in terms of needed flow of information, needed coordination, and scheduling constraints. For example, research that impacts architecture should become available in 2015 or so. New programming models will not be used to code for exascale, unless they are ready and well supported on existing platforms by 2015, or if code porting is made painless by the use of advanced tools.

7.5.3 Vendor Interaction

Vendors should be involved at an early stage as partners to ensure research technologies match their constraints and can be easily moved to industry. It is necessary that the same programming models be supported by all vendors, so as to provide code portability. This will require research teams to work with multiple competing vendors – leading to complex IP problems. Note, however, that there are many successful precedents of multi-vendor research partnerships.

DOE has a rich history in interactions with computer vendors. But little recent language, compiler or run-time research has made its way back from DOE to the system and independent software vendors. Similarly, there might be cases in which DOE, due to implicit and explicit policy constraints at headquarters or in the labs, has duplicated technology that could have been obtained COTS.

7.5.4 IP

DOE has a strong preference for open source software; open source software also is the environment of choice for research. On the other hand, many of the components of the software stack provided

by vendors – in particular, compilers – are proprietary. If research is to include areas such as compilers, one will need to carefully manage the IP issues this situation raises.

One possible approach would be to focus on open-source technology for exascale systems, e.g., request that the compiler infrastructure be based on open source compiler software such as LLVM or Open64. The use of LLVM by vendors such as Apple and Cray increases the plausibility of such an approach.

An alternative is to develop research technology using open source in the expectation that it will be adopted and ported to vendor products. For example, compiler transformations could be first developed within DOE closer to the application groups and then migrated back to the vendor compiler groups. Open source and source-to-source compilers are an open and plausible vehicle to communicate both analysis and transformations that are important to support both applications and whole application domains on emerging computer hardware. This approach could automate and raise the levels of abstraction for application groups while still addressing the detailed requirements of vendor solutions. This bridging could be an important basis for interactions with vendors in the co-design process. Such an approach is viable only if research uses non-viral open source licenses. Furthermore, this approach is cost effective only if a means can be found for DOE compiler development to be performed in a context where optimization baselines comparable to the significant capabilities of commercial tools (the products of significant commercial investment) are used; typically open source tools are not on par with commercial tools. Alternatively, another mode for DOE to work on tools development is in the context of commercial source bases.

This issue is is of major importance to vendors. Restrictive and short-sighted DOE policies that mandate open source, and DOE laboratory implicit policies of only taking in open source, may well be one of the causes of neglect of these technologies for solving DOE HPC software challenges, and duplication within DOE of capabilities available elsewhere. While commercial software may involve some licensing costs for DOE, those costs are without question lower than recapitulating the technology as open source. Using government resource to duplicate software as open source also undermines vendors. It is very difficult for open source to attract and sustain capital investment because ROI on open source software is significantly lower than closed source software. Finally, developing critical technologies - and the programmign system for exascale is clearly a strategic technology - as open source hands technologies to our national competitors in high performance computing, in some cases further undermining US vendors by seeding offshore low-wage competitors[4].

It will be important to develop models for collaboration with US vendors that can avoid all of these pitfalls.

The role of open source vs. proprietary technology in the HPC stack has been an item of intense discussion in DOE for many years, and this report will not resolve the issue. Different vendors have different strategies with respect to open source software and a different willingness to use open source software in their products. Vendors that differentiate their products in hardware may be more willing to adapt open source, as compared to vendors that differentiate themselves in software.

The DOE exascale program will need to interact with such different vendors, further complicating the decision process.

[4]Is is hard to imagine DOE mandating that commercial hardware vendors publish their schematics and fabrication process specifications; strategic software technology should be treated equally carefully.

7.5.5 Common vs. Vendor Specific Technology

Hardware vendors and and independent software companies have large established and experienced teams in technology specific to their platform, in particular, compiler development, and deep portfolios of advanced technology; not just in node-specific software, but also in parallel and HPC software such as parallelizers, libraries, programming models, debuggers, profilers, threading systems, messaging systems, and thread checkers. In some cases, these technologies have been partly or even fully funded by the government. But they are typically not open source, so that the vendors can preserve some competitive advantage and achieve a fair return on investment.

Part of the exascale software stack will be developed by the hardware vendors that will focus on their specific hardware, and will have internal road-maps for the details about features that can be leveraged. This is especially true for "back-end" compilers, node scientific libraries and other node-architecture specific software.

At the other end, part of the software stack has to be developed in close interaction with application developers. This is especially true for high level programming models, frameworks and libraries. Hardware and software vendors could be encouraged to have greater interaction with application developers.

Finally, programming models should be common across systems, in order to support portability. Such a commonality should go beyond the semantic of the programming model used, and also include performance portability (a code that is efficient on one platform does not need to be rewritten to run with reasonable efficiency on another platform). It is highly desirable to have a common set of tools and a common integrated development environment that can be used for code development on either platform, so that programmers need no learn how to use different tools and environments.

These forces are pulling in different directions and will need to be carefully managed. They possibly indicate that DOE research should focus on the higher levels of the stacks where commonality across platforms and strong interaction with applications is most important. However, such a research cannot ignore the lower levels of the stack, and collaboration with vendors will be essential. Such a collaboration should be aimed at influencing vendors so that they provide at the lower layers those interfaces and services that are required for building efficient higher layers; and at defining proper interfaces between layers. For example, one can imaging that higher level DSLs and programming models are supported by a parallel run-time and by a source-to-source compiler, such as ROSE, PGI, or R-Stream. The parallel run-time has to be supported by a node run-time that could be partly provided by the vendor, while the target source code is further compiled by a vendor "back-end" compiler. Such an approach suffers from a lack of integration (it is harder to design debuggers or performance tools) but provide a logical division of labor. For it to work, one needs to define rich interfaces that enable to move across layers as much information as possible.

7.5.6 The Vendor Ecosystem

Vendor contributions to exascale programming models can come from a variety of vendors: Microprocessor vendors, such as Intel or NVIDIA are developing compilers and integrated development environments for their hardware: The Intel Parallel Studio and the NVIDIA CUDA technology. Intel is developing several systems also supporting advanced parallel programming concepts such as Array Building Blocks, Cilk++, and Concurrent Collections. Independent software companies can provide significant technologies for exascale. The Portland Group Compiler offers highly tuned and robust Fortran and C++ compilers, and has demonstrated the ability to rapidly innovate in programming models with their CUDA Fortran products. The R-Stream compiler, developed with

government funding and available to the government under common FAR and DFARS source rights, offers unique capabilities for automatic parallelization and scheduling using polyhedral methods. Platform vendors such as Cray or IBM are developing their own libraries, compilers and programming environments, both for node programming (e.g., OpenMP and thread tools), and for global programming (e.g., Chapel, X10, UPC, CAF and parallel performance tools). This landscape is continuously evolving, and could significantly change by 2020.

Since collaboration with vendors is important, it is important for the research teams to collaborate with those companies that are most likely to use their research. Similarly DOE should take advantage of and sponsor commercial researchers, who can have significant unique proprietary assets to increase research effectiveness. A better definition of the expected role of various vendors in exascale will facilitate such interaction.

7.5.7 Common Research Infrastructure

Projects that are likely to coexist in the long run will benefit from using a common infrastructure. Such an infrastructure should also facilitate research and the exchange of code components.

7.5.8 Integration vs. Specialization

Research in each area (architecture, run-time, compilers, programming tools, libraries) has strong dependencies on research in the other areas. Such interactions are better handled by collaborative research of larger teams of researchers that represent these various areas and work together towards a common goal. At the same time, small and focused groups are sometimes better positioned to provide best-of-breed individual pieces of technology, so the underlying approach must facilitate composing separately-developed technology through common interfaces.

7.5.9 Work replication

The limited research budget for exascale dictates that replications be avoided, while not stifling the natural competition that leads to better outcomes. It is also important to minimize the waste of resources through duplication effort recreating technologies already available. One may want to push for more commonality and less replication in infrastructure that is not a proper part of the research

7.5.10 Pedagogical Role

Research in academia and in DOE research labs creates not only new technology, but also new educated researchers that can move this technology to vendors or other labs. It is important that the research funding models encourage the involvement of students and contribute to the education of the generation of exascale developers and users.

7.6 Programmatic Challenges

7.6.1 Programming models

The uncertainty about future architectures, and the limited budgets devoted to exascale computing means that it is essential to have a lean, agile infrastructure for constructing layered systems of programming models – e.g., developing DSL's or frameworks. The infrastructure must support their efficient implementation and must support the creation of "reverse engineering tools", i.e., debuggers and performance tools that present information to the programmer at the level of the

programming model she uses. It must guarantee semantic preservation across layers: a lower layer either correctly implements a higher layer or throws an exception. It must also preserve performance semantics: a lower layer either provides the performance expected at the higher level, or indicates why it cannot do so.

7.6.2 Rapid Prototyping of New Language Concepts

Novel production quality languages have to be built using tried and proven concepts and features as a single bad feature can break the language. One of the biggest barriers for research on new language features is the current inability to get quality feedback from programmers. The dominant reason for a success of a language is that large number of programmers find it to be productive for their requirements and start using it. As many of the early adopters will be already using a language they are familiar with, the advantage need to be significant enough to justify the learning curve. Thus, programmer experience is perhaps the primary reason for a success of a language. Unfortunately there is a chicken-and-an-egg issue when it comes to novel languages. It is hard to get programmers to use something new and without programmer feedback it is harder to evolve a language. It is not easy to evaluate a language for multiple reasons:

1. It is hard to get a programmer to invest their time onto a research language. With a short term perspective, there is nothing in it for them. The language/compiler will be buggy; has very little supporting infrastructure (libraries, debugging); it keeps changing rapidly making it necessary to update/port the program; and it may not have the longevity thus requiring the programmer to redo the program using a different language.

2. There is a difference between evaluating a language feature(s) and building a full featured language. A full featured language has to get everything right (one ill-conceived construct can kill a language). Thus, you need to use proven features. The issue is how to get a novel feature to a stage where it can be adopted into a full featured language.

3. Even with a full featured language, market forces will not drive for evaluation and evolution. There is a herd mentality in programmers. In order to get a programming language adopted, it already have to be successful and stable. This works against evolution (example: programmer resistance in moving to Python 3.0)

4. Novel language features need time to evolve. It is hard to get it right the first time. Unfortunately, evolution makes programs obsolete, making evolution extremely difficult. The more success a language has, more difficult this becomes (design by committee, programmer resistance etc.)

5. Need to understand the productivity of an expert programmer. A language that is similar to existing languages are easy to learn, but may not provide too much benefits in the long run. A totally novel programming construct may take time to learn, but once perfected can be very beneficial (or not). Thus getting good feedback requires long term commitment from the programmers. A one-week experiment will not do it (for example, multiple inheritance looks like a great idea at first, until you try to write a complex program with multiple inheritance.)

6. All programmers are not the same, thus need many programmers to use a language before getting a representative view.

In many cases, a successful new language have to provide a functionality not available in existing languages in order to gain adoption – it has to be essentially the only viable way to achieve some

goal, e.g., while Java has many good features, it's adoption was largely due to its support for applets. It is possible that the same will happen for exascale – with a new language being the only effective way of handling resilience or locality. This necessity can be used to achieve other desirable goals that, by themselves, would not cause a paradigm shift.

7.7 Application Engagement

The work on programming models has to be closely coordinated with the work in applications. The application teams are working to provide compact applications, mini-applications and application skeletons that can be used to validate programming model work "in the small". It is less clear how the work on programming models and supporting technologies can impact the development of exascale applications, because of the short run-way; and how one can test issues related to "programming in the large": e.g., how easy it is to express large multi-physics and multi-scale codes and manage the complexities of such codes.

7.8 Strategies for Validation and Metrics

The ultimate measure of success for a new programming model is its adoption by the user community. The design of a language or framework is an important contributor to its success, but still is one of many success factors: does the new model fill a known gap, is it well supported, does it provide the required performance, is it likely to persist – those are factors that may have more importance that the elegance of the design. Furthermore, one important goal for a new programming model, namely programmer productivity, is extremely hard to measure: Productivity on toy codes is different than productivity on large codes; productivity of a novice is different than the productivity of an expert programmer; productivity does not correlate much with simple measures, such as lines of code; and productivity is highly dependent on the programming environment, not only the programming model. Therefore, it will be hard to provide simple, intrinsic measures of success for work on programming models, and it will be hard to evaluate work in this area in its early stages.

This does not mean that evaluation has to be avoided – it means that an objective evaluation using quantitative metrics will necessarily be incomplete, and has to be complemented by a qualitative evaluation.

Quantitative evaluation should include measures such as performance on existing compact applications: measured performance on existing systems, and extrapolated performance on exascale systems, taking into account failure rates, and measuring performance both by time to solution and energy consumption.

Qualitative evaluation should include a measure of the expressiveness of the proposed model. One possible approach is to consider the main parallel programming patterns used in HPC codes (such as captured by Berkeley motifs and other similar work [2]) and explain how these patterns can be expressed in the proposed programming model. Equivalently, one could ensure that the mini-apps used to evaluate the proposed programming models be representative of all these motifs.

It is important that the insistence on metrics does not undermine the healthy DOE investment in novel or revolutionary research approaches. Typically any innovative technology cannot match the performance of established technologies. However, what is important, in the medium and long term, is *the rate of improvement*. Established technologies typically flatten their rate of improvement, as ever greater spending is required to make progress against their intrinsic limits. Innovative technologies without these limitations can escape the bounds of the established technologies, and can be on a much steeper rate of improvement. This is the McKinsey S-curve argument [20]. While

this business literature makes these cases in colorful stories such as the triumph of steam over sail, an example of this in our own field of high peformance computing is the infamous "Attack of the Killer Micros" [14] that overturned the vector supercomputing establishment. *It is certain* that several of the so called "toy" programming systems that our academics are working on today will provide the answers to some of our most vexing programming challenges, provide they are kept alive. They must be nurtured, even if they seem weak and barely threaten our venerated programming approaches that are serving us so well now. Consequently, any regimen of metrics for comparing programming approaches must not be so restrictive as to overlook consideration of the scalability, rate of improvements, intrinsic limits, and the changing hardware and applications landscape.

8 Bibliography

References

[1] S. Amarasinghe. Why have compilers failed to help parallel programmers. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, E. Lee, N. Morgan, G. Necula, D. Patterson, et al. The parallel computing laboratory at uc berkeley: A research agenda based on the berkeley view. *EECS Department, University of California, Berkeley, Tech. Rep*, 2008.

[3] A. Avritzer, F. P. Duarte, R. M. M. Leao, E. de Souza e Silva, M. Cohen, and D. Costello. Reliability estimation for large distributed software systems. In *Conference of the Center for Advanced Studies on Collaborative Research*, 2008.

[4] L. Bairavasundaram, R. Goodson, B. Schroeder, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. An Analysis of Data Corruptions in the Storage Stack. In *USENIX Conference on File and Storage Technologies, (FAST)*, 2008.

[5] P. Balaji. Evolutionary support for revolutionary programming models and runtime systems. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[6] J. Balfour and W. Dally. Design tradeoffs for tiled CMP on-chip networks. In *International Conference on Supercomputing*, 2006.

[7] R. Barrett. Preparing multi-physics, multi-scale codes for exascale hpc. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[8] R. Barriuso and A. Knies. Shmem user's guide for c. Technical report, Technical report, Cray Research Inc, 1994.

[9] G. Belter, E. Jessup, I. Karlin, T. Nelson, B. Norris, and J. Siek. Exploring the optimization space for build to order matrix algebra. Technical Report ANL/MCS-P1890-0511, Argonne National Laboratory, May 2011.

[10] G. Blelloch. Nested parallelism and hierarchical locality. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[11] R. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[12] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

[13] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[14] E. D. Brooks. Attack of the killer micros, 1990.

[15] S.-E. Choi. Five things about hpc programming models that i can live without. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[16] A. Duran, J. Perez, E. Ayguadé, R. Badia, and J. Labarta. Extending the openmp tasking model to allow dependent tasks. *OpenMP in a New Era of Parallelism*, pages 111–122, 2008.

[17] V. Eijkhout, P. Bientinesi, and R. van de Geijn. Towards mechanical derivation of krylov solver libraries. *Procedia Computer Science*, 1(1):1799 – 1807, 2010. proceedings of ICCS 2010, http://www.sciencedirect.com/science/publication?issn=18770509&volume=1&issue=1.

[18] T. El-Ghazawi, W. Carlson, and J. Draper. Upc language specifications v1. 1.1, 2003.

[19] P. K. et al. Exascale computing study: Technology challenges in achieving exascale systems. http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf, 2008.

[20] R. N. Foster. *Innovation: The Attacker's Advantage*. Summit Books, 1986.

[21] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[22] Y. Guo. *A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism*. PhD thesis, Rice University, Aug 2010.

[23] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar. SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, Apr 2010. IEEE Computer Society.

[24] W. Harrod and S. R. Sachs. Introduction to the ascr exascale programming challenges workshop. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[25] M. Heroux. Next generation programming environments: What we need and do not need. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[26] C. Huang, O. Lawlor, and L. Kale. Adaptive mpi. *Languages and Compilers for Parallel Computing*, pages 306–322, 2004.

[27] The international technology roadmap for semiconductors (ITRS). http://www.itrs.net.

[28] C. Jansen. Pushing back the point of diminishing returns for parallel performance. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[29] S. Kale. Composable and modular exascale programming models with intelligent runtime systems. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[30] R. C. Kirby and A. Logg. A compiler for variational forms. *ACM Trans. Math. Software*, 32:417–444, 2006.

[31] P. Kogge. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. *CSE Dept. Tech. Report TR-2008-13*, 2008.

[32] A. Koniges. Challenges and application gems on the path to exascale. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[33] S. Krishnamoorthy. (de)composable abstractions for a changing architectural landscape. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[34] R. Lethin. Reconceptualizing to unshackle programmers from the burden of exascale hardware issues. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[35] R. Lublinerman, J. Zhao, Z. Budimlic, S. Chaudhuri, and V. Sarkar. Delegated Isolation. In *Proceedings of OOPSLA 2011*, 2011.

[36] B. Lucas. Exascale: Can my code get from here to there? Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[37] T. Mattson. Unintelligent design for asynchronous exascale systems. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[38] J. Mellor-Crummey. Lessons from the past, challenges ahead, and a path forward. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[39] S. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer. *IEEE Tran. on Device and Materials Reliability*, 5(3), September 2005.

[40] D. A. B. Miller. Rationale and challenges for optical interconnects to electronic chips. In *Proc. IEEE*, pages 728–749, 2000.

[41] D. A. B. Miller and H. M. Ozaktas. Limit to the bit-rate capacity of electrical interconnects from the aspect ratio of the system architecture. *J. Parallel Distrib. Comput.*, 41(1):42–52, 1997.

[42] J. Misra. The challenge of exascale. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[43] G. A. Moore. *Crossing the Chasm*. Harper, 2nd edition edition, 1999.

[44] J. Nieplocha and B. Carpenter. Armci: A portable remote memory copy libray for ditributed array libraries and compiler run-time systems. In *IPDPS'99*, pages 533–546, 1999.

[45] J. Nieplocha, R. Harrison, and R. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[46] K. Pingali. Why compilers have failed and what we can do about it. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[47] I. C. Plus. http://software.intel.com/en-us/articles/intel-cilk-plus/.

[48] D. Quinlan. Challenges for compiler support for exascale computing. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[49] J. Reid and R. Numrich. Co-arrays in the next fortran standard. *Scientific Programming*, 15(1):9–26, 2007.

[50] V. L. Rideout, F. H. Gaensslen, and A. LeBlanc. Device design considerations for ion implanted n-channel mosfets. *IBM J. Res. Dev.*, 19(1):50–59, 1975.

[51] V. Saraswat. The return of logic. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[52] V. Sarkar. Programming constructs for exascale systems and their implementation challenges. Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[53] J. Shalf. Functional vs. imperative languages (dataflow 2.0). Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[54] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. *VECPAR*, pages 1–25, 2010.

[55] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. 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.

[56] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[57] J. Shirako and V. Sarkar. Hierarchical Phasers for Scalable Synchronization and Reduction. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[58] H. Simon, R. Stevens, and T. Zacharia. Modeling and simulation at the exascale for energy and the environment town hall meetings. http://www.er.doe.gov/ascr/ProgramDocuments/Docs/TownHall.pdf, 2008.

[59] H. Simon, R. Stevens, and T. Zacharia. A platform strategy for the advanced simulation and computing program, 2008.

[60] T. Sterling. Future Directions in High-End Computing, April 1999. Invited Presentation at the University of Vienna.

[61] R. Stevens and A. W. et al. Scientific grand challenges: Architectures and technologies for extreme scale computing. http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Arch_tech_grand_challenges_report.pdf, 2010.

[62] B. V. Straalen, J. Shalf, T. Ligocki, N. Keen, and W.-S. Yang. Scalability challenges for massively parallel amr application. In *In IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[63] J. Sukha. Brief announcement: a lower bound for depth-restricted work stealing. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 124–126, New York, NY, USA, 2009. ACM.

[64] S. Swaminarayan. Exaflops, petabytes, and gigathreads... oh my! Presentation given at DOE 2011 Workshop on Exascale Programming Challenges, July 2011.

[65] S. Tasirlar and V. Sarkar. Data-driven tasks and their implementation. In *Proceedings of International Conference on Parallel Processing (ICPP)*, 2011.

[66] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.

[67] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific computing kernels on the Cell processor. *International Journal of Parallel Programming*, 35(3):263–298, 2007.

[68] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Languages and Compilers for Parallel Computing, 22nd International Workshop, LCPC 2009*, volume 5898 of *Lecture Notes in Computer Science*. Springer, 2009.