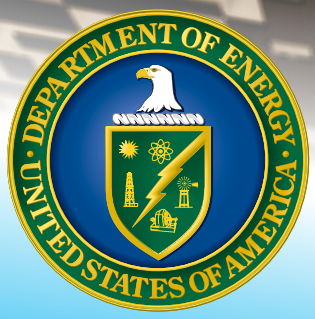

REPORT OF THE 2014
Programming Models & Environments Summit

EDITOR: Michael Heroux

CO-EDITOR: Richard Lethin

CONTRIBUTORS: Michael Garland, Mary Hall, Michael Heroux, Larry Kaplan, Richard Lethin, Kathryn O'Brien, Vivek Sarkar, John Shalf, Marc Snir, Armando Solar-Lezama, Thomas Sterling, and Katherine Yelick

DOE ASCR REVIEWER: Sonia R. Sachs

September 19, 2016

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)

17-CS-3270

Contents

1 Executive Summary

Programming models and environments play the essential roles in high performance computing of enabling the conception, design, implementation and execution of science and engineering application codes. Programmer productivity is strongly influenced by the effectiveness of our programming models and environments, as is software sustainability since our codes have lifespans measured in decades, so the advent of new computing architectures, increased concurrency, concerns for resilience, and the increasing demands for high-fidelity, multi-physics, multi-scale and data-intensive computations mean that we have new challenges to address as part of our fundamental R&D requirements. Fortunately, we also have new tools and environments that make design, prototyping and delivery of new programming models easier than ever. The combination of new and challenging requirements and new, powerful toolsets enables significant synergies for the next generation of programming models and environments R&D. This report presents the topics discussed and results from the 2014 DOE Office of Science Advanced Scientific Computing Research (ASCR) Programming Models & Environments Summit, and subsequent discussions among the summit participants and contributors to topics in this report.

This report presents and discusses the following topics:

1. **Fundamental Drivers for Programming Models and Environments:** Programming model and environment changes are driven by two fundamental forces: (i) architecture changes stemming from reaching scaling limits in traditional architectures and (ii) demands for more complex modeling, simulation and decision-making support that spur increased demand for coupling of capabilities, in addition to reducing approximation error with higher-fidelity representations. These drivers require disruptive changes at all levels of the computing software stack, including programming models and environment.

2. **Emerging Application Design Requirements:** The goal to produce ever more effective computational results leads to demands for coupling previously separate software capabilities to enable simultaneous resolution of multiple physics and scales, and to integrate large-scale data with large-scale modeling and simulation. Tight coupling of these phenomena exposes new computational and data access patterns as well as drives software complexity. Programming models and environments must adapt to meet these emerging requirements.

3. **Disruptive changes in application software architectures:** The design and implementation of

application software is changing. Application software developers are learning how to map their application architectures to system architectures for both performance and portability. Programming models and environments need to adapt in order to support application developers with proper abstractions, and effective transformation to the underlying specific runtime and hardware systems for performance and portability.

4. **Primary programming challenges:** New programming models and environments must assist the application developer in handling massive, dynamic and hierarchical concurrency, performance variability arising from complex hardware control system that limits predictably, much greater attention to data structures, placement, movement and locality and, potentially, explicit handling of resilience at the programming level.

5. **New approaches for design, prototyping and delivery of new programming models:** The emergence of several new programming environments is of particular interest for programming models research. LLVM, clang, and, in the future, Flang, provide modular, extensible community programming environments that enable syntax, analysis and transformation extensions to be inserted into a production software stack. New features in C++ are also providing approaches for exploring programming models, permitting effective and efficient embedded domain-specific languages (DSLs) that enable design and development of new programming models within a supported, portable and standard compiler environment.

6. **Adapting applications to future systems requires a tiered and multi-phased approach:** Modeling and simulation for science and engineering cannot stop while we wait for new programming models and environments to emerge. Furthermore architectures will continue to change over the coming decades. Therefore we need near and long term strategies for adapting applications for effective performance while at the same time keeping application codes viable for producing results. We discuss a three-tier vision path for adapting applications to new systems, now and in the distant future.

7. **Continued funding for fundamental parallel programming models research:** While short and medium term research efforts are necessarily limited by pragmatic constraints, some exploration should focus on the fundamental semantics of programming representation, exploring pathways that may enable disruptive changes for long term parallel programming approaches.

2 Introduction

The creation of new programming models and environments is driven by the needs of user productivity and operational effectiveness of the computing system. Programming models differ by the domain of applications to be captured and the nature of systems upon which the applications are to run. Projected exascale computer systems are expected to stress the means of programming and may demand advances in programming models and the languages they imply.

The intent of this report is to provide a vision for exascale programming and a migration strategy that will provide a meaningful path forward for domain scientists and application programmers from 2016 systems to the 2023 exascale timeframe and beyond. The report highlights key areas of research as well as future Research and Development (R&D) that must be put in place to accomplish this objective. As part of the coordinated research programs being formulated to achieve exascale computing of the next decade, future programming models are identified as crucial to its viability and success, requiring new R&D projects to deliver the necessary programming environments.

Multiple reports have outlined the benefits of continuing the growth in computing performance with the specific goal of building exascale computing systems to be used for modeling, simulation, and analysis of large, complex scientific and engineering phenomena, (e.g., [5, 26]). The primary challenges in building an exascale system are related to energy consumption, which causes problems with both heat density and total cost of ownership, and these concerns at the hardware level will lead to a number of architectural trends that will significantly change the way the systems must be programmed. An exascale system will have 100× more hardware threads than current systems, new memory mechanisms, new communication mechanisms, new synchronization mechanisms, explicit voltage and clock controls, new forms of heterogeneity, and higher rates of errors and performance variability.

Some of these new features will require support in the programming models used for exascale; all of them will require support in the environment: compilers, runtime, and tool infrastructure used to implement these programming models. It is not reasonable to expect most programmers to address these new exascale architecture features "by hand." Thus, while energy is the primary design constraint for hardware designers, the ability to program these systems may be the single biggest factor in their usability, effectiveness, and widespread adoption.

Many of the anticipated hardware changes are not related to overall system scale (i.e., number of nodes),

but instead are about the exascale building blocks: the processors, memory, interconnect, and storage technology, as well as the interfaces between these pieces. The exascale programming techniques are not only needed for the applications that will run on full exascale systems, but also for computations that run on subsets of these systems or on smaller "petascale versions" of the exascale technology.

In addition, applications are evolving and are also imposing new requirements on programming models and environments. The desire to solve increasingly complex problems that span length and time scales, as well as multiple physical domains (e.g., mechanics, fluids and chemistry), are scientific drivers for exascale computing, but they come with new challenges for the programming environment. Multiphysics simulations will require the composition of separately developed codes; the desire for increased model fidelity will lead to sparse and adaptive algorithms that create varying workloads; and the emergence of data intensive problems may introduce new patterns of irregular data access. The usage models are also changing, with increased emphasis on some approaches: ensemble runs will be used for uncertainty quantification and high throughput screening; observational data may be incorporated into or compared with simulations; and I/O limitations will increasingly require in-situ analysis of simulation results.

2.1 Hardware Challenges for Programming Environments

There have been numerous workshops, panels, task forces, and reports over the last seven years highlighting the challenges that will be faced by domain experts and application developers when trying to program projected exascale systems in a manner that fully exploits their key architectural features. A selection of this material can be found on the Department of Energy (DOE) Advanced Scientific Computing Research (ASCR) Web site [1]. One of these most recent reports is the "Top Ten Exascale Research Challenges" report from a subcommittee of the DOE's Advanced Scientific Computing Advisory Committee (ASCAC) [32]. The list of challenges includes the need for programming environments that support massive parallelism, data locality, and resilience. Several of the other challenges, such as energy efficiency, interconnect and memory technology, and correctness, and scientific productivity also directly impact the programming environment. In particular, the impact of energy efficiency on node architectures is discussed in the Abstract Machine Models report [3]. Here we highlight some of the major hardware drivers for changes to implementation and use of programming environments:

Energy Efficiency: Reducing power requirements and increasing energy efficiency is a critical issue. Some power projections for exascale systems based on standard multicore technology are over 100 megawatts,

making such systems impractical. To address the stringent requirements for energy efficiency, hardware technology will continue to evolve, and in some cases, affect the lowest level programming interface provided by the architecture.

Node Architecture: Architectural changes in the node have already disrupted the previously universal programming model of flat message passing parallelism and even the more recent hybrid model that combines message passing with intranode thread parallelism. Fine-grained data parallelism of various forms, limited memory per core, non-uniform memory access effects, and software-managed levels of the memory hierarchy all need to be addressed. In addition, the nodes are likely to become more hierarchical and heterogeneous, and power (voltage and clock) controls are being exposed that may require significant management by software.

Scalability: The scalability of systems, systems software, and applications is a significant issue for exascale computing. Exascale systems will pose unprecedented challenges in system-wide parallelism as well as intra-node complexity. Systems will consist of one hundred thousand to one million nodes, and perhaps as many as a billion cores. Managing and servicing a system of this size will be a challenge. Highly reliable and scalable operating systems and systems software will be needed. Applications must maintain at least a hundred-fold increase in the available parallelism, which is certainly a challenge, but also an opportunity for new models, environments and algorithms.

Reliability and Correctness: Reliability is a significant concern as the number of processors grows and software becomes more complex. Other factors driving up the rate of faults include smaller circuits running at lower voltages, higher likelihood of low probability events, and the increased complexity of hardware (e.g., heterogeneity, voltage controls, etc.), which may introduce more programming errors. Silent hardware errors may become more frequent. The exascale system reliability target is a system with one day between application-level interrupts. More frequent interrupts will require development of a fault model, which will in turn enable co-designed advances in hardware and software reliability as well as new methods for application resilience. Reliability will require some contribution at the programming interface level, but not burden the programmer with devising and specifying detailed fault handlers. The programming model will need to support characterization of success criteria associated with the program operation for automatic testing and validation.

Another challenge, sufficiently important to be documented in the Secretary of Energy Advisory Board (SEAB) Task Force on Next Generation High Performance Computing report [72], is the importance of *program portability*. The report acknowledges that future hardware constraints will be significant enough to

effect a broad and disruptive paradigm shift in algorithms and software architecture. While addressing these challenges opens up renewed opportunity to introduce a higher level of software engineering, at the same time, prior investments must be protected and a migration path from current to future environments must be devised. Performance portability will be a significant concern. Experience has shown that application groups will not develop software for next-generation supercomputers unless there is some assurance that the new software will run on multiple generations of future systems, and run at different scales and on different types of architectures at any point in time. To improve productivity, a programming model that abstracts some of the architectural details from software developers, without sacrificing performance, is highly desirable.

2.2 Application Challenges for Programming Models and Environments

The 2010 ASCAC Subcommittee Report on Exascale Computing [5] describes several application-driven challenges that arise from modern computational science simulations and the numerical techniques within them.

Multiscale and Adaptive Methods: Many physical simulations involve modeling across a wide range of space and time scales. As the physical system being simulated evolves over time, the data structures and computational workload adapts to the need for higher resolution and higher accuracy. The result is variable timestepping, adaptive discretizations based on techniques such as adaptive hierarchical blocks or unstructured meshes, subscale models, and other dynamic techniques. Such methods are already used in some petascale applications today, but there is a trend towards increased sophistication and broader use. In addition, multiscale methods can bridge among multiple different physics models at different scales [42], on a global or local/selective basis. To support high performance implementations of these methods, programming models will need to move beyond static parallelism to facilitating the epxression and efficient execution of irregular and dynamic forms of parallel computation, involving simultaneously multiple types of models and algorithms.

Multiphysics Simulations: Increased computing power also allows multiple physical models to be coupled in a complex simulation. For example, mechanics, fluids and chemistry are all necessary to understanding a combustion engine. Multiphysics simulations may be built by adding a new set of features to an existing code or by combining two or more codes into one, which often requires some complex (in both mathematics and software) communication between the different models. We would like to optimize such communication so that the parts that are communicating can be localized; this locality optimization is a

form of advanced loop fusion. Multiphysics simulations are not a new phenomenon for exascale, but they are increasingly popular and have traditionally lagged in scalability and adaptation to new hardware features relative to simpler applications focused on a single physical model. The implication of multiphysics is similar to that for multiscale methods, where programming models need to move well beyond SPMD.

New Usage Models: The national Materials Genome Initiative exemplifies high throughput computing used to screen many thousands of possible materials for a given application, such as the design of new energy-efficient batteries. Similarly, ensembles of related simulations with different parameter settings may be used to quantify uncertainties, to compare against observational data, and generally to gain confidence in the results. The simulations in an ensemble may execute independently, but running times may vary widely and the choice of which simulations to run can depend on previous results. Exascale systems will need to support sophisticated workflow tools for these applications.

Application Size and Software Complexity: Some scientific applications are written by a single person or small research group and contain only a few thousand lines of codes, but large codes that are used in areas like climate modeling, weapons simulations, and engineering contain often millions of lines of code and involve multiple programming languages, and libraries. Some of these have grown in size by an order of magnitude since massive parallelism was introduced, making disruptions to the programming environment much harder than it was in the early 90s. Programming tools must support maintenance and evolution of such codes for exascale.

Data-Driven Applications: The SEAB report also highlighted an additional challenge of supporting *data-driven applications*. As computer models of scientific phenomena have increased both in scale and in detail, the requirements for increased computational power, typically in the form of FLOPS, have increased exponentially, driving commensurate growth in system capability. The requirement for increasing Floating Point Operations Per Second (FLOPS) is not likely to slacken in the foreseeable future. However, the nature of the workloads to which these systems are applied is rapidly evolving. Even today, the performance of many complex simulations is less dominated by the performance of floating-point operations than by memory and integer operations. Moreover, the nature of the problems of greatest interest to security, industrial, and scientific communities is becoming increasingly data driven. The implication for the programming model and environment is that it should move toward supporting algorithms for the analysis of data. One aspect of this is that the programming should facilitate efficient in-situ analysis.

2.3 Application Software Design Challenges

At the start of the previous disruptive parallel computing transition from sequential and vector computers to distributed memory systems, programming models research attempted to preserve a sequential programming view for application developers. For example, the FORALL statement of High Performance Fortran (HPF) attempted to support distributed memory execution while hiding the details of distributed data and execution from the programmer. Unfortunately, while this approach preserved programmer productivity, it did not enable execution productivity. Once the SPMD design pattern emerged as the superior approach to writing distributed memory parallel applications, first supported via PVM and later by MPI, parallel programming models research acquired a sharper focus guided by the specific needs of the new and emerging SPMD application base.

Today, the parallel application design patterns for emerging platforms is still in an exploration phase. While there is strong evidence that some kind of task management layer is needed as part of new application designs, the details are still emerging. As a result, our programming models and environments research is somewhat hindered and speculative by a lack of clear understanding of what our new application software base really needs.

2.4 Principles of Exascale Programming Models and Environments

Future exascale computing systems have yet to be precisely defined, but many of their attributes can either be predicted or can at least be narrowed to a few alternatives. Indeed, 100 petascale systems are already in development and will soon be available, leaving little doubt about viable exascale designs. In this context, the derivation of classes of appropriate programming models and environments can be considered to sufficient degree of accuracy to guide the creation of research programs through which necessary programming models and environments may be developed. This comes from general requirements as well as more specific architecture characteristics. These will be discussed in more detail throughout this report, but some overall principles and practices can be established. These include:

Asynchrony: Future systems will exhibit long and variable latencies to read data stores in remote nodes or another level of the memory hierarchy. Contention for network, I/O, or memory bandwidth may also create uncertainty of response times to remote accesses and service requests. Programming models and environments will need to address these latencies, probably by embracing asynchrony in performing data

movement and work scheduling to minimize waiting time for such events.

Load Balancing: To best make use of system resources for computations that are adaptive or for machines that are highly variable, the programming environment will need semantic methods for guiding load balancing of data and workflows, and at the same time minimizing data movement by co-locating computation and data, and/or it will have to facilitate automatic methods (e.g., static/dynamic compilers) for such purpose.

Dynamic Adaptive Control: Future systems may be too complex and too dynamic to be managed efficiently by static mechanisms. Programming environments that employ dynamic resource management and task scheduling through the support of runtime system software for introspective operation may achieve higher efficiency and scalability than those that employ static methods, as long memory affinity concerns are adequately respected.

Performance Portability: As a diversity of architecture types, scales, and generations will be served by applications written in the new programming environments, the programmer effort needed to achieve good performance on different systems has to be minimized by hiding some of the differences or allowing them to be managed in an abstract way. Concerns that are specific to one architecture should affect application codes in very limited and isolated parts.

Data Structures, Placement, and Layout: Data structures (both simple arrays and complex irregular structures) need to be mapped onto deep memory hierarchies and across distributed main memory, and those mappings may need to adapt dynamically. The more complex data structures and mapping problems arise in multiscale, multiphysics applications, both today and in future applications; they require sophisticated data representation, organization, and transformation that is easy to use and that conforms to the modalities of the applications and systems. Productivity gains are likely if the data structures' implementation and mapping details can be hidden in the programming model implementation.

Data Movement: Exascale systems will likely involve new levels of memory, such as scratchpads and Non-Volatile Random Access Memory (NVRAM), that are under software control. The programming model and environment will need to either expose this to the programmer or manage it within the implementation.

Interoperability and Workflow Management: The programming models and environments of the future have to enable big and complex applications to be constructed from smaller, more simple functions and libraries, and they have to work with execution models and environments developed from other programming

languages and interfaces.

2.5 Ecosystem Considerations

The development of programming models and environments for exascale does not happen in a vacuum, but within an ecosystem that imposes major socio-economical constraints, such as:

Backward Compatibility: Many of the codes that now exist or are currently being developed will continue to be used for decades; the cost of wholesale recoding would be prohibitive. The Department of Energy, alone, has billions of dollars invested in its current application software, and individual applications may have millions of lines of software. Some of the most widely used applications have grown substantially since the last major disruption in high performance software, moving from sequential and vector machines to distributed memory massively parallel ones. Therefore, exascale systems will need to run existing software and, to the extent possible, minimize the need for changes to code.

Application Redesign: Changes in application design may eliminate many present programming model challenges (and expose new challenges) by explicit design. Programming models research is derived in part from an understanding of how today's applications are designed and developed. Broad adoption of message passing in the 1990s eliminated the need for global data and execution concepts in programming models, and introduced other demands. We should expect the same shift in demands in the future.

Technology Reuse: The supercomputing market is small; it is viable economically because it reuses hardware and software technologies that have much broader markets. This is true for programming environments as well; languages and compilers used in High-Performance Computing (HPC) have broad use (C, C++), or are small modifications of languages that have broad use (OpenMP). The tool chain used to support HPC programming environments (languages, compilers, debuggers, linkers, etc.) reuses technology from broader markets. Exascale programming environments will need to achieve a similar level of reuse.

Open source software plays an important role in facilitating this reuse: it promotes commonality across platforms, reduces the level of investment and risk undertaken by system integrators, and enables an ecosystem where small companies and research organizations can contribute components to the exascale stack. LLVM is a good example of such software. While it contains many components, programming environment developers are free to use only those pieces they would like, such as just the code generation portion. Thus, they can choose how to trade off product differentiation against reduced development cost.

Financial Viability: In order to encourage private investment, and thus extend the government dollar, it must be recognized that there needs to be a return on that investment. For industry, this often means that contributing everything to open source is not viable. In some cases, significant private investment has already been made to develop capabilities such as industrial strength C++ and Fortran compilers, advanced optimization systems, and math libraries. There is no justification for spending government funds to try to reproduce such capabilities as open source. In fact, developing open source can undermine the private investment. While the support revenue business model for open source exists, it provides low return and thus it does not always allow developers of advanced technologies to recoup their costs. So it must be recognized that, while some exascale developments could be contributed to open source, some industry technology should be employed, which ought not be open source.

Human Skills Reuse: The community of people that develop HPC codes is small. This community, due to past experience, is reluctant to adopt new programming models, especially if those models have a high learning curve. New programming technologies will generally not be adopted unless they (i) provide great benefit; or (ii) have a low learning curve and low risk.

Diverse and Competing Requirements: It is important to remember that any discussion of programming models addresses the needs of a diverse community that includes domain experts who use codes but do not develop them; application development teams who integrate domain expertise and computer expertise; and library development teams who combine algorithmic expertise with computer expertise. These communities, and the players within the teams, have different needs and are likely to use different programming environments; one size does not fit all.

Furthermore, with the diversity of the community that programming models address, there is the question of who benefits from the adoption of new programming tools. For example, the adoption of a high-level, semantics-rich exascale programming approach may provide portability and increased sustainability of software through multiple generations of hardware platforms and users. This great benefit to the overall program life cycle may not be recognized by an application group solely working toward short-term research goals.

3 Role of Programming Models and Environments Research

There are a number of ways in which a programming environment research program can impact—and has impacted—the practice of parallel programming. Ongoing research is needed to resolve open questions related

to the uncertainty around hardware features, find the best approach for dealing with them, and uncover the potential for radically improved productivity from higher-level programming models. The various research projects (and the discussions at the 2014 workshop) naturally revealed different emphases on which of the hardware or application challenges are most important, and different approaches to hiding or exposing those features.

3.1 Challenges

Approaches to develop new programming models have a "chicken and egg" problem: developers are reluctant to use a new programming environment that is not in broad use and may be discontinued; on the other hand, it is hard to justify large investments in a new programming environment that is not in broad use, and it is hard to properly evolve and evaluate such an environment without a large user community. It is seldom the case that the first version of a new system is "right." Multiple generations are needed before the system is appropriate for broad use. This vicious circle causes most attempts to develop new languages and systems to fail to achieve widespread adoption.

This threshold for adoption is higher with approaches that require more wholesale changes to applications. This is most severe with "all or nothing" solutions that require a major commitment to a new approach, rather than a gradual introduction. This means that innovations in the programming system implementation may be preferred over things that affect the programming model itself, and more incremental solutions may be preferred to preserve prior application software investments. At the same time, the desire to minimize change can limit creative solutions, such as hardware innovations or new high level languages, so it is important that a research program balance investments across potential risk and reward spectrums.

Finally, there should be a process by which research ideas are transitioned to production, perhaps by focusing, initially, on small stand-alone codes that are more experimental and later aim for the gradual replacement of components in large application codes. Such a process involves shared investments of time and funding across disciplines to ensure that new ideas are given due consideration, while acknowledging that application experts, especially those that understand advanced programming notions, are a critical resource. Such activities should therefore be a two-way communication, starting with some education of the programming environments research community to understand the requirements and desires of a given application area.

3.2 Metrics for Success

Here we consider some of the metrics for a successful research program and how results could feed into a production software stack development activity:

- A research project may define a new language, library, or tool that becomes widely adopted by computational scientists. This requires a successful prototype implementation from the research project, in addition to a substantial long-term commitment to fund development and maintenance, possibly combined with ongoing research. The commitment can either come through the DOE computing centers and vendors, so that the programming environment becomes part of procurements, or directly from the DOE research program.

- Specific concepts within a language, library, or compiler implementation or tool can be transferred into existing open source or proprietary software. In this case, the support costs are part of the ongoing cost of developing, say, GNU tools, the MPI vendor implementations, or various OpenMP compilers. However, the bar for adoption is higher than in the first scenario, and possibly requires that other communities outside DOE or scientific computing see them as valuable. UPC was an example of this kind of transfer, where the language constructs were added to GNU tools and the EDG front-end used by many vendor compilers, which encourages new implementations. However, the existence of a complete open source solution (with continued support from DOD) was also part of this picture.

- An improved understanding of programming model or implementation techniques can influence other research projects. This can be important in developing a consensus around certain ideas, just as the many message passing libraries in the early 90's gave the community enough experience to define a standard message passing interface. Community consensus is needed on the semantics of these abstractions, and an understanding of how they can be embedded in the languages programmers are likely to employ (e.g., C, C++, Fortran, etc.).

- While the impact may be less obvious, research programs that show certain techniques to be impractical, inefficient, provably suboptimal, or cumbersome are also important to the success of those ideas that succeed. They help the community move forward and develop the ideas that achieve other forms of impact. Formalizing concepts in the form of a programming language and implementation can help crystallize the ideas and reveal ambiguities or inefficiencies, even if the language itself is not in widespread use.

4 Background, Challenges and Opportunities

The focus on programming models in this report (rather than the explicit concentration on a particular language syntax) permits a more generalized strategy of investigation. Multiple languages or libraries may be devised to realize a particular family of programming models. The model provides a unifying abstraction for which choices of form may be later selected. We present in this section a taxonomy of the design choices for exascale programming models and a few basic assumptions that guide us through this report.

4.1 Characterizing Programming Models and Environments

We use the following definitions:

- A *parallel programming model* provides a set of abstractions that simplify and structure the way the programmer thinks about and expresses a parallel algorithm [67].
- A *parallel programming environment* implements one or more parallel programming models. Thus, Message-Passing is a parallel programming model; MPI is a parallel programming environment. A parallel programming environment may implement multiple parallel programming models: MPI can be used for two-sided message-passing (send-receive), for bulk-synchronous communication (collectives), or for one-sided communication, i.e., Remote Direct Memory Access (RDMA).[1]

New programming models and environments aim to achieve two goals:

1. Improve the *programmer productivity* of code development activities.
2. Improve the *execution productivity* of code as it runs on a given system.

The first item covers the human effort needed to design, code, debug, test, validate, tune, port, support, modify, and sustain applications and libraries. The second item covers the machine resources (time and energy) needed to execute a program. These two goals are partly conflicting forces: code that executes faster may take longer to write, debug, and test, and may be harder to port and maintain.

[1] RDMA is a system level mechanism that can be used to move data between nodes without involving the processors on the remote node. Even on the initiating process, much of the communication work can be offloaded to the network interface. RDMA features in libraries and languages may implement more general mechanisms that perform reformatting, which may either be supported directly by hardware or implemented in software.

The apparent competition between programmer and execution productivity can be reconciled to a large degree by proper application software design. As mentioned in 2.3, parallel application design has not yet evolved to support effective mapping onto emerging scalable manycore, accelerator and hybrid node architectures. As new designs emerge the conflict between the above items can be substantially reduced.

This document focuses on the needs of application programmers and, to a lesser extent, the developers of scientific libraries. This ignores the development of system services and middleware. Application programming for exascale will focus on the concerns that are specific to large-scale parallelism. We describe these concerns, assuming an imperative programming model:

Execution Distribution: The potentially dynamic decomposition of a computation into "execution chunks" (for example *iteration tiles*, or more generally tasks) that can be executed concurrently, and the expression of dependences among these chunks.

Scheduling: The allocation of iteration tiles to execution units (e.g., cores), so as to enhance the utilization of the execution units (load balancing) and reduce communication.

Data Partitioning: The decomposition of global data structures into "data chunks" (for example *data tiles*) that can be stored in distinct locations.

Data Placement: The potentially dynamic mapping of data tiles to specific physical memory locations.

Data Layout: The potentially dynamic organization of the elements of a data tile at each physical memory location.

Communication and Synchronization: The movement of data tiles or iteration tiles that is needed to handle dynamic distributions, to ensure each iteration tile has access to the data tile it works on, and to enforce dependences. Communication and synchronization can happen together (as in a message-driven computation) or separately (an event occurrence that satisfies a dependence).

Error Handling: Detection and handling of faults, such as rollback of the computation and restoration of state from checkpoints, and potentially disabling faulty processors.

Power Management: Dynamic control of voltage and clocks; also maintaining power limits when hardware is over-provisioned, meaning that there are more transistors in the processors, network, and memory than can be simultaneously powered and operated.

A parallel programming model may ignore some of the concerns listed above, assuming that compilers or the runtime system will make the appropriate choices. In fact, the process of creating parallel code can be considered a process of successive refinement, where one starts with a high-level abstract specification of an algorithm and progressively provides more information on the actual binding of the computation to machine resources. The discussion in our community focuses on the exact layering of parallel programming models and parallel programming environments that supports this process. In particular:

- Do we have one parallel programming environment that is used to express algorithms, or do we have a stack of different parallel programming environments, where higher-level systems are implemented on top of lower-level systems? (e.g., using a Domain Specific Language (DSL) that is compiled into MPI + OpenMP.)
- Do we stop programmers from meddling with programming to low-level details of the architecture, in the interest of portability or to avoid constraining the compiler and runtime? In particular, do we expose error handling and power management in the parallel programming model? Do we expose physical resources (e.g., node count), or higher-level, virtual resources?

We make two fundamental assumptions:

1. Programming will be done at multiple levels of abstraction, and the application programming stack will support programmer intervention at all levels of abstraction. A three-level model for these abstractions (see Appendix A for details) is:
    - **High-level:** Domain-specific programming models, perhaps supported by DSLs, that are used by application domain experts, in order to express the application logic in terms that are meaningful for the particular domain. This layer may also be implemented via an application-specific data and work decomposition similar to the approach traditionally used with SPMD applications, but would be extended to include task-level decomposition.
    - **Middle-level:** General-purpose programming models, that expose algorithmic abstractions that are not specific to one application domain and use general data structures. Most programming languages (Fortran, C++, OpenMP, etc.) support such a programming model. The compiler and runtime that maps these languages to the hardware may hide some features of the system—such as the number of cores running an OpenMP program.

- **Low-level:** Architecture-specific programming models, that provide explicit control of the computing resources and, therefore, have well-defined performance models.

The separation between these levels is fuzzy, and one may more properly speak of a spectrum, rather than three well-separated clusters.

An area of discussion in our community concerns the degree to which new execution models needed for efficient exascale computing permeate through the levels of abstraction. In one view, the execution model is only exposed at the lowest level, and reflects the nature of new exascale architectures. In this view, the higher-level programming models are more familiar to the applications programmer, and tools such as compilers map into the new execution model. In another view, the new execution model for exascale is reflected at every level of abstraction, and programmers must take the exascale architecture into account, even at the highest levels of programming.

A programmer can program an application at multiple levels (e.g., some parts using high-level DSLs and other parts at a low level using machine level constructs). The same code can be successively developed at a high level and tuned at a low level. An implication of this choice is that low-level code generated from a DSL program should be, to the extent possible, human readable. It may turn out that application programmers will never have to deal with low-level resilience or energy, if compilers, runtime, and middleware can successfully handle these, with no significant performance loss. But it is too soon to decide one way or another.

2. The application programming stack will support multiple starting points (at high level or middle level) to get to low-level, executable code. Large applications already combine code developed using different parallel programming environments, and there is no reason to assume this will change. Also, any new parallel programming environment will need to coexist with "legacy" code. This makes the question of interoperability paramount.

The different parallel programming environments may eventually converge to one execution model that is supported by the system. The execution model is supported by the hardware operating system and common runtime available on the machine. It defines the set of operations supported by the system and their effect, and the mechanisms for composing them. In addition to this semantic aspect, a Performance Model will provide an estimate of the resources (processors, time, memory, energy, etc.) consumed by an execution. Thus, for a current supercomputer, the execution model may describe a system as consisting of interconnected

nodes, each with a fixed number of hardware threads, each executing one instruction stream and sharing the node's memory; nodes support various intra-node and inter-node synchronization and communication operations. The above simplified model would not describe in an accurate manner heterogeneous nodes with accelerators and different memory types. The model can be refined to account for those. Our community is still debating whether it is feasible to provide one common execution model that would work for all relevant platforms. While, at some distance, all computers look the same, a common execution model may not be able to support an accurate Performance Model.

4.2 Design Choices

This is a survey of the design choice considerations. It provides a general overview of the design space. We will expand this in more detail regarding specific implementation strategies in later sections.

4.2.1 Low Level

We assume that the hardware, OS, and common runtime will be supported by a low-level parallel programming model that is very close to the execution model.

For example, today's execution model is supported by a parallel programming model that is a low-level communication library and a simple thread library: a set of cores is dedicated to the application code; a logical thread is associated with each physical thread of these cores. The threads within a node communicate via shared memory and simple (non-blocking) synchronization operations. Communication across nodes uses message passing and simple (non-blocking) synchronization operations. This lowermost parallel programming model is focused on exposing the performance of the underlying hardware in the most direct way possible. The parallel programming model can be supported by using MPI and OpenMP in a restricted fashion.

They could also be supported by lower-level common runtimes, which more directly embody the anticipated exascale execution model. In this runtime, computation is expressed as lightweight, fine-grained, event-driven tasks with explicit and structured use of memory, dependencies, and synchronization. This runtime will facilitate dynamic fine-grained scheduling, load balancing, and resilience. Such a lower-level abstraction could support multiple types of parallel programming models.

As we move to higher-level programming models, we face a plethora of choices, some of which are listed below.

4.2.2 Uniform vs. Hybrid

Low-level models will use different communication mechanisms inside nodes (intra-node) and across nodes (inter-node). While it is possible to design a parallel programming model that hides this distinction at higher level of abstractions (e.g., by focusing on the transfer of ownership of data tiles while ignoring the mechanisms used for this transfer; the difference will be in the relative performance, not in the semantics), it is highly unlikely that it will be satisfactory. Inter-node latency, bandwidth and control transfer costs are substantially higher than intra-node. Algorithm developers will need to take advantage of lower on-node costs in order to achieve optimal performance. For example, wavefront algorithms that assume light-weight control transfer, synchronization and shared data access are feasible for intra-node execution, but typically not for inter-node. If forced to program at a high-level only, we are destined to have suboptimal performance.

Furthermore, the broader performance-oriented computing community, in particular the vendors, will invest in intra-node parallel programming models and environments. These models and environments will support multiple levels of parallel (not just one), including vectorization/SIMT and one or more levels of tasking. Our efforts must acknowledge and build on these investments, not try to replace or hide them; otherwise we face the same destiny as HPF.

Respecting the independent availability of intra-node models and environments does not mean we have to explicitly write code for each type of node. A low-level model may use different programming models and environments for different types of computation engines, such as CPUs and GPUs. Or, it can use the same model and same language (e.g., OpenMP), and let the compiler and runtime decide how do map iteration tiles among different types of computation engines. In fact, programmer productivity demands that we should strive for performance portability across node architectures as much as our algorithms and programming environments permit, while keeping performance levels acceptably high.

4.2.3 Physical vs. Virtual

Resources in the system, such as memory, cores, or nodes can be virtualized by adding a layer that maps logical resources onto physical resources: paging is used to virtualize memory; thread schedulers virtualize cores by dynamically mapping logical threads onto physical cores. Nodes are virtualized in a package such as Charm++ [50] in a system that supports the migration of "virtual nodes." Virtualization frees the programmer from the need to manage the mapping of computation or data chunks to physical locations; the

price is a possibly suboptimal mapping, the need for over-decomposition (e.g., having more logical threads than physical threads) to ensure high utilization, and the overhead for dynamically scheduling threads or moving data that restricts the frequency at which mappings can change. Thus, systems such as TBB [65] or Cilk [17] can manage the dynamic scheduling of an execution every few microseconds (thousands of instructions); Charm++ migrates virtual nodes only a few times in an hour. Concurrent Collections (CnC) [21] provides a fully virtualized space of control tags and data items that can be mapped strategically to reliable and volatile storage to provide resilience [81].

4.2.4 Scheduling and Mapping

The mapping of virtual resources to physical locations can be done with a variety of constraints: the mapping of threads can happen once, when the thread is created, or multiple times, with migration. Thread migration can be restricted either to a set of cores or to one node; alternatively, threads could be migrated (together with their private data) across nodes.

In order to reduce communication, it is necessary to properly align the mapping of computation and the mapping of data. Most memory references in a computation should be made to local data. Determining the optimal alignment is a hard (NP hard) problem, and various heuristics have to be used.

One approach is *control parallelism*, which focuses on the distribution of control and moves data where it is needed. Most shared-memory programming models and environments, including OpenMP, implement control parallelism: they provide the programmer some control of how computation is laid out among cores, but no control on data location. These models were designed for systems where efficient use of cores was paramount, whereas communication overheads to shared memory were negligible. They are less appropriate for environments where communication costs are a major issue (as for modern CPUs).

The dual approach is *data parallelism*, which provides programmer control over the distribution of data and moves execution where the data resides—by using an "owner compute" rule so that each operation is executed near the location of its stored result. High-Performance Fortran is an example of such an approach.

Most systems exist in between these two extremes: *object parallel* models, such as Charm++, use objects that combine data and control as the unit that can be migrated. Message-passing systems and PGAS languages have a fixed association of data and control. However, since data allocation is controlled locally in a message passing application, data migration can be efficiently executed by using portable libraries such as

Zoltan [46, 30] and Metis [52]. Use of these libraries within a message-passing application enables high-fidelity mapping of control and data parallelism, informed by computation and communication costs known only to the application developer, and the ability to decide if and when to remap control and data if load imbalance occurs during execution, e.g., if the given application tracks front as part of an evolutionary simulation.

Some highly optimized kernels—such as neutral territory methods, systolic array algorithms, and communication-avoiding algorithms—combine control scheduling tightly with data scheduling in order to optimize communication.

While scheduling and mapping of computation and data appear to be extremely challenging problems in the presence of emerging node-parallel architectures, we anticipate that new application architectures–which explicitly manage work-data partitioning at a task level–will provide the primary means of assuring local data access, in the same way that distributed memory application architectures have addressed this issue in the past. Intra-node tasking application architectures will have more flexibility than classic MPI-based approaches, for example enabling light-weight control transfer via futures mechanisms, but will, by design, enable co-location of computation and data.

4.2.5 Communication

Communication moves data from one physical location to another. It can be *one-sided*, issued by one thread only; *two-sided* involving two threads; or *collective*, involving multiple threads. It can be *implicit*, as a side-effect of a reference that moves data from memory to cache or can cause a cache line eviction; it can be *explicit*, as with a cache prefetch or a put or get operation. It can be *synchronous*, or *blocking*, as with a read that must appear to complete before the next instruction executes; it can be *asynchronous*, or *non-blocking* or *split-phase*, if the operation that initiates the communication is distinct from the operation that completes it. Data movement can be an explicit *copy*, where the source and destination have distinct names; or *caching*, where the physical location changes, but not the name of the datum. Caching is *coherent* if hardware or firmware can locate the latest copy of a piece of data, or *non-coherent* otherwise. Data can be reformatted, distributed, or computed on as it moves, for individual communications and collectives. Communication can also effect synchronization, as in a message-driven computation that starts a task upon message arrival.

4.2.6 Synchronization

Synchronization affects the relative (perceived) ordering of concurrent operations. Thus, a send-receive communication is synchronizing since the send must start before the receive ends. Synchronization can be two-sided or collective. Message-passing typically uses *ordering synchronization* that determines the relative order of operations at two threads; a signal-wait pair is an ordering synchronization; shared-memory often uses *non-ordering synchronizations*, such as locks. Non-ordering synchronizations introduce *nondeterminism*, where different execution schedules may lead to different interleavings and different outcomes. Some limited forms of *nondeterminism*, such as changes in the relative order of operations in a reduction, are more benign than others, as they only cause rounding errors. Nondeterminism makes debugging and testing more difficult, and should be avoided whenever possible. It would seem that avoiding nondeterminism is difficult and costly for some classes of computations, such as some graph algorithms, but some numerical algorithms can manage to get the right result with benign nondeterminism.

Excessive use of synchronization can result in arbitrary dependence graphs, which can make code hard to understand. Many systems restrict synchronization so that the dependence graph is a series-parallel graph. Such a model is often called fork-join; fork-join is also used for programming models where control dependences form series-parallel graphs, but where synchronization operations can add additional dependences. The same effect can be obtained by a restricted use of synchronization, such as in a bulk-synchronous programming model. The execution of such a code has serial semantics.

Some in the exascale programming community are developing systems for arbitrary synchronization—specifically event-driven programming models. This comes from the recognition that a very promising source of the additional concurrency needed for exascale is to avoid restrictive synchronization models that can only approximate the true dependences in the application. While programming at this level is verbose and hard to understand, compiler technology exists that can generate efficient schedules expressed in an event-driven model from high-level programming abstractions [22]. Furthermore, task-centric application designs will naturally enable expression of relaxed synchronization, since, by default, tasks are asynchronous execution units that are annotated with synchronization restrictions.

4.2.7 Global View vs. Local View

Global view, or more accurately aggregate view, approaches provide pre-defined support for managing data across more than one memory image and dispatching execution across more than one thread. Local view approaches support logically sequential execution accessing a unique data source. Modern programming models and environments can support aggregate and local views at various levels of the system architecture. For example, an application using message passing is a local view approach at the inter-node level, but may use OpenMP as an aggregate approach within a node.

Global or aggregate views typically improve programmer productivity, but can hinder performance if the aggregate execution does not effectively map computation and data. Programming languages such as UPC and Co-Array Fortran support dual views of data with clear process-data mapping with some success. However, the most common and reliable performance, which blends global view productivity benefits and local view performance is obtained from application frameworks that provide global view programming support but local view execution underneath.

4.3 Examples

The following are brief examples of the types of systems currently used to execute parallel programs.

4.3.1 Shared Memory Systems

Shared memory languages, such as OpenMP, TBB, or Cilk provide a global (flat) address space—hence a global view of data. Communication in these languages is implicit (via shared memory accesses), while synchronization is explicit (via parallel control constructs, atomic operations, or synchronization operations).

TBB and Cilk++ encourage, but do not enforce, a fork-join programming model. OpenMP supports nested loop parallelism that is a form of a fork-join programming model. All allow more general synchronization.

4.3.2 Distributed Memory

MPI+C (or C++ or Fortran), which is often called *flat MPI*, provides a uniform programming model; it can support a distributed address space, with explicit communication, but can also be implemented on top of

shared memory hardware. Communication can be one-sided, two-sided, or collective, as well as synchronous or asynchronous. It can be used in a bulk-synchronous programming style with all communication happening in a separate phase, possibly as collective operations.

Synchronous PGAS languages, such as UPC, also support a homogeneous programming model. In these languages, communication is implicit through pointer dereferences and array indexing, and can also be done with more explicit bulk copy operations as one-sided communication. PGAS languages can be implemented for shared or distributed memory hardware, and can take advantage of the shared memory system by using simple load and store instructions for accesses to "remote" data. In UPC and Co-Array Fortran, the scheduling of threads and the mapping of data is static. UPC provides a global view of data, and a local view of control. Chapel supports a global view both of data and of control.

4.3.3 Hybrid Models

MPI+OpenMP is a hybrid model, with different notation for shared memory parallelism and distributed memory parallelism. It reflects the popularity of multicore processor building blocks for parallel systems, and directly uses the load/store operations in shared memory—but explicit communication between nodes. While MPI is used in an estimated 90% of HPC applications, a smaller fraction use OpenMP and still fewer use a node programming model specifically for GPUs, e.g., OpenACC or CUDA.

4.4 New Approaches for Programming Model Design, Prototyping and Delivery

The emergence of several new programming environments, tools and language capabilities is of particular interest for programming models research. For many years the computing community has wanted a common backend compiling environment. In the past few years, the emergence of LLVM has provided the right layers of abstraction and adaptability for almost universal standardization of intermediate representation (IR) of source code, and efficient adaptation of the IR to a specific target processor. Similarly clang has emerged as a new community effort to provide a C/C++ compiler environment built on LLVM, and the very recent announcement of renewed Flang effort provides hope for a similar impact on Fortran. All of these efforts also promote modularity and the opportunity to insert custom syntax and IR analysis and transformation layers within an otherwise production software stack, enabling localized innovation in programming models

that can be used by application codes with minimal risk exposure.

Another emerging capability is effective and efficient support for embedded domain-specific languages (DSLs) in C++. The series of 2011, 2014 and 2017 C++ standards enable design and development of new programming models within a supported compiler environment. C++ 2017 will also include a parallel algorithms library. New programming models, expressed as embedded DSLs, can be written in standard portable C++ and delivered to application teams with far less concern for sustainability issues. Furthermore, DSL concepts, if general enough, can be considered as candidates for future C++ standard features. Array views are a concrete example of this kind of migration.

4.5 Current State

Most existing DOE applications are written in flat MPI, where each MPI rank is a single process and thread. There has been some progress in making ranks multi-threaded, usually through the use of OpenMP for CPUs and CUDA or OpenMP for GPUs. These applications are typically written in Fortran, C or C++, with C and C++ being increasingly used.

Porting a flat MPI application is relatively straightforward, but is often not the best strategy for good performance, especially with manycore chips. Accelerating such an application with OpenMP or OpenACC can be challenging. Offloading portions of the application onto GPUs adds further complexity and requires additional work.

Various compiler analysis tools are available (of varying quality) that can help identify parallel regions amenable to using OpenMP (or OpenACC). The best of these tools provide sufficient dependence analysis to help the programmer identify what can and cannot be parallelized to the compiler.

New applications today tend to be written in some combination of Fortran, C, C++, and Python. While Fortran is often dismissed as an out-of-date language, and its viability as a commercial product is often weak, it is still very important to the HPC community. Any comprehensive plan for preserving our application base must include credible Fortran plans. C++ (with C as an important subset) has emerged as the preferred portable parallel programming environment in portions of the HPC community, and is widely used in the performance-oriented technical market, by companies such as Google. Python is a preferred workflow and scripting language, but is structurally not suitable for most large-scale application codes.

There has also been some experimentation with DSLs. Embedded DSLs, built directly within or upon

existing languages, have shown some early benefits in their ability to leverage existing tool chains (e.g., debuggers and performance analysis tools). However, these experiences have been fairly limited to date.

In addition, some newer languages and programming models are starting to see very limited use for applications. While many of these are early in their research phases, a few have seen use for production applications, or have at least made significant progress on benchmarks or mini-apps. Charm++ has been used successfully in production for the molecular dynamics code NAMD and several other applications. Chapel has shown some promise in programmability and portability, including an implementation of the shock hydro mini-app LULESH, but it still needs to demonstrate sufficient performance.

Much of the ongoing efforts in programming work by hardware and software vendors is focused on continued improvements of existing programming environments. MPI advances include better support of one-sided communication, support for fault-tolerance (in experimental libraries), support for new MPI3 features, and better performance with large thread counts. Work on OpenMP includes support for new OpenMP 4 features, better scalability with high thread counts, and compilation for GPUs. Vendors are also working to bring support for parallel execution to the underlying programming languages, such as in the most recent C++11 and C++14 standards.

Along with this evolutionary work, some vendors are also investigating potentially revolutionary approaches to HPC. This includes work on Chapel by Cray [24], X10 by IBM [70], Concurrent Collections (CnC) [21] by Intel, and R-Stream [56] by Reservoir Labs. While these are interesting research projects, history and current market size limits suggest that the broad HPC community will not adopt them because of portability and sustainability concerns.

A major trend in several vendor products (OpenMP, Cilk, TBB) and research projects is support for a lightweight task model running in shared memory—i.e., the dynamic scheduling of entities that are lightweight threads, not visible to the Operating System (OS). In these projects, work stealing is used for load balancing and for regulating the amount of currency to balance with the amount free execution hardware resources [16]. Work stealing can be made very low overhead through the use of fast data structures, deques, for task queues. In research ,it has been demonstrated that work stealing can scale to very large numbers of tasks and nodes on distributed memory hardware [31].

A global name space, together with RDMA communication, provides the internode communication mechanism for many of the emerging languages and runtimes; the objects moved can be logical tiles, rather than address ranges. A key issue for such a model is the efficient scheduling of tasks upon the arrival of data they

require. The efficient support for such models is a key goal of projects such as Qthreads with Portals [75] and Argobots [9].

Such models can also be expressed using message driven objects as in Charm++ [51] or with a dataflow model that expresses dependences across objects, such Legion [15], Dague [20], CnC [21], or SMPSs and OmpSs [62, 34].

This event-drive, lightweight task, model has many attractive features:

- Latency hiding: Having many tasks "in-flight" enables latency hiding, as stalled progress on one task can be covered by execution of another task.
- Dynamic load balancing: Similarly, many tasks mean that work can be dynamically scheduled, with sensitivity to temporal locality issues.
- Natural resilience strategy: If tasks communicate only with data that is read when the task is dispatched and data written when the task has completed, then the task parent can re-dispatch a task that fails or times out—assuming that errors are detected before task completion.[2]
- Domain scientist productivity: Task code does not need to be highly scalable itself, since parallelism is obtained by concurrent execution of many tasks. This permits a domain scientist to write simpler code—paying attention to vectorization or SIMT concerns, perhaps modest thread parallelism to exploit low-level shared caches, and hyper-threading—but otherwise write correct sequential code in common languages.
- Increased concurrency: The event-driven task model can express dependence synchronization with fewer approximations, becoming a source of additional parallelism.
- Multi-language development: Task code can be written in Fortran, C, or C++, or be generated automatically from higher-level abstractions.
- Multi-level memory systems: Tasks can be spawned recursively, and data scoped dynamically to enable efficient use of multi-level memory architectures.

On the downside, the envisioned event driven lightweight task model requires an overdecomposition, where the number of tasks is much larger than the number of executing threads; this makes the task granularity

[2]This is a good assumption. Exascale hardware that is being designed will with extremely high probability detect faults: RAM, latch, and register state will have parity or ECC. Combinational logic will be designed with circuits for which faults will be unlikely, even running at Near Threshold Voltage (NTV).

more fine, and reduces the computation intensity (i.e., computation to memory access ratio) of the tasks, unless the tasks can be scheduled in ways that find locality among tasks. Also, it has proven difficult so far to find a compromise between load balancing and locality preservation. The problems becomes even more difficult if task dispatching and load balancing targets the entire system, rather than one node only. System-wide task management strategies are best integrated into applications where particular knowledge about the cost of task migration across nodes can be used to inform migration choices.

It is important to note that new application architectures are emerging that are expressly designed to exploit lightweight tasking runtime environments, as they mature. When the application architecture itself provides a task data and work decomposition, programming model requirements become much simpler. In the same way that distributed memory application designs bypassed the need for HPF in the 1990s, task-based application architectures will significantly simplify programming model and environments requirements for many applications.

5 Vision

Future applications will need to expose massive concurrency, tolerate dynamic execution variability, exploit locality in many levels of memory, and at the same time tolerate latencies that vary by many orders of magnitude. Applications will also need resilient implementations so that the impacts of faults and halts are localized and repaired.

Furthermore, to preserve domain scientist productivity, new applications must present a programming framework that supports insertion of new functionality as simple code that is close to the mathematical representation of the science. Although sophisticated parallel algorithms and implementations are certainly required for application scalability, in many cases the concerns of introducing new modeling features can be separated from the details of parallel implementations. For example, well-designed MPI applications are implemented in this way. Domain scientists introduce new functionality as sequential code, understanding when they need to perform a halo exchange, or a collective operation such as dot-product. But these operations are performed via abstraction layers—whose implementations are sophisticated, scalable, and adaptable, but also outside the scope of a domain scientist's concerns. For future applications, domain programmers can be expected to write efficient code that can be vectorized and compiled for modest thread-scalable execution, but should not need to write complicated parallel code that is best left to application

framework developers. Today's domain programmers seldom make direct calls to MPI functions when executing a halo exchange operation. Instead the application framework supports this programmer by providing an "exchangeHalo" function.

Parallel programming environments must be developed pragmatically. A strategy that balances between evolutionary development and revolutionary development may be best.

On the evolutionary side, although the research community can and must experiment with new languages, we must recognize that application portability, cost of compiler development and support, and the need for our applications to be part of a larger ecosystem limit our ability to move new programming languages into a production setting. Our programming environments research must be conducted with an eye toward delivering new capabilities in existing or emerging industry-supported languages. This certainly increases the expectations on compiler developers to produce efficient code for complicated expressions. Compiler-generated code must also interact with new runtime capabilities that are needed to support asynchronous task-centric dataflow execution environments. The cost of the evolutionary approach alone is that it may be very difficult for exascale (to manage exascale hardware features) and it may constrain forward portability (applications become baked to one architecture).

The evolutionary side would emphasize that application teams adopt new programming models as embedded DSLs in languages like C++. The C++ standards community has and continues to introduce new language features that make C++ embedded DSL features feasible. Application developers can even tolerate modest language extensions, such as CUDA, OpenMP, or OpenACC. These incremental extensions can be implemented within an existing compiler environment, reducing costs of both application and compiler developers.

The revolutionary side has investment in, and use of, new programming models, new automatic optimization tools, and new environments. This aims for greater productivity (automatic code generation to manage exascale hardware), greater portability (code mapped to new architectures through retargeted tools), and greater performance (optimizations greater than can be achieved with traditional programming). This requires sustained investment and a good partnership with software tool vendors to develop and maintain tools.

In the remainder of this section, we consider three strategies for programming models and environments of the future, along with a baseline strategy of continuing on the current petascale path.

5.1 Baseline

The cost of porting current HPC software to new programming models is likely to be prohibitive. Even if cost was not an issue, there is not enough time to rewrite much code in the next eight to ten years. It takes multiple years to firm up a new programming model—often more than a decade; and the number of programmers with HPC skills is limited. Therefore, it is essential to support current programming models in the exascale timeframe, with best possible performance.

Current evidence indicates that some, possibly many, codes written using MPI+OpenMP will scale up to billions of threads: MPI has been used with real application with up to 7.8M ranks [12], and, experimentally, with over 100M ranks.[3] OpenMP has been scaled up to 128 cores [55] and to 240 threads [71], with speedups of 70-100, for various applications.

It is not yet clear how broad the set of codes is that can achieve good performance in this model, and whether alternative models can expand this set. Also, with new exascale application requirements (e.g., multi-physics) and with new exascale hardware (e.g., explicit voltage and clock controls), it may be costly and difficult to write code in the traditional low-level way, and the portability of such code will be even lower.

Still, due to the value of existing code, and the infeasibility or rewriting it, it is essential to perform R&D to ensure that MPI and OpenMP codes scale as well as possible. These include evolution of the MPI and OpenMP standards, evolution of the implementations, and application code refactoring. We list below some of the required work for this baseline course.

OpenMP Language Enhancements. OpenMP provides limited support for locality of computation and for locality of data. This will be needed as shared memory systems become non uniform (NUMA) and have a deeper hierarchy. Richer synchronization primitives, such as asynchronous and point-to-point synchronization, are needed to achieve high performance on large numbers of cores. More support is also needed for heterogeneous computing.

OpenMP Runtime Scalability. Current OpenMP runtimes are not designed for the use of hundreds of threads. Work is needed to avoid scalability bottlenecks in the runtime.

OpenMP Scalable Programming Model. OpenMP supports multiple programming models: atomic execution units that can thread (parallel sections); iterates (parallel loops) or tasks (task constructs).

[3] See https://www.westgrid.ca/westgrid_news/2013-01-14/ubc_researchers_use_westgrid_explore_exascale_computing

The interaction between these different constructs is complex, and different OpenMP implementations use different scheduling algorithms for scheduling these constructs. As a result, it is too easy to write innefficient code, and too hard to write code with portable performance. The pragmatic solution is to promote a recommended programming model and to orient runtime implementations toward the efficient support of this model (say, task parallelism, with a work-stealing scheduler). This model should be chosen so as to facilitate the development of asynchonous code.

MPI Scalability. Current implementations often use multiple data structures of size proportional to the number of ranks at each node. This must be avoided when the number of ranks increases faster than the amount of memory per node. One can also improve the scalability of various collective operations.

MPI Resilience. MPI must provide a means of continuing an application after the loss of a node.

MPI Interface with OpenMP. MPI defines an interface to processes. OpenMP supports threads, tasks and parallel iterators, as units of control; threads is the least useful of the three. The interface between MPI and OpenMP has to be standardized and well implemented, so as to support concurrent MPI invocations for parallel OpenMP code. In particular, it is important to define precisely how MPI interacts with tasks, and to provide a tight coupling between communication and task scheduling.

MPI Support for Multithreading. MPI performance suffers when multiple threads are associated with the same MPI rank. This is due in part to the limitations of current MPI implementations (e.g., coarse locking) and in part to inherent problems with MPI semantics (e.g., the complex rules for matching send and receives). Work on implementation, and possible changes in the MPI standard, could alleviate the problem. The problem can also be alleviated by providing a thread model for MPI, where MPI ranks are associated with threads, rather than processes.

MPI Support for Deep Memory Systems. Work is needed to ensure "zero-copy" implementations of MPI communication, irrespective of the location of source and destination.

MPI+OpenMP Scalable Programming Models. As for OpenMP, MPI supports different programming models, and implementations may favor one over another. It is important to promote a style of programming in MPI+OpenMP that can be supported efficiently, and to favor this style in implementations. For example, it may be desirable to increase the use of one-sided communications, and optimize their implementation.

In addition, the MPI+OpenMP model can benefit from many of the technologies described in the previous sections. DSLs, frameworks and libraries can target MPI+OpenMP as their low-level programming model. Tools for autotuning, refactoring tools that pinpoint troublesome parts of the code and facilitate transformations, compiler hints, just-in-time compilation for MPI or OpenMP, autotuning, formal verification, etc., are all applicable to the MPI+OpenMP model—and should be applied so.

5.2 Path I: Migration of Applications to New Programming Environments

While MPI+OpenMP is a viable option for applications at exascale, it may not provide the best performance, especially for strong scaling that requires support for finer grain parallelism. It may not provide a good way of expressing new types of concurrency needed for multi-physics and multi-scale, and may not be a good way to implement power controls. It may also not provide the best sustainability, as code written to MPI+OpenMP will be bound very tightly to particular hardware models, and thus costly to port to new architectures and generations of hardware.

Concurrent with exploration of more disruptive parallel programming model development, application developers can realize significant performance portability by refactoring applications to expose more task and SIMD/SIMT data parallelism. Such approaches have already had value [57, 58], and represent a ubiquitous modification to application architecture that preserves a large portion of the computational source code base.

Migration to new application architectures can improve the performance of existing applications, or may enable use of these codes for a broader range of inputs and scales. Migration will also facilitate continued maintenance and evolution, in a future where other programming models become dominant. In this section we describe two basic models for preserving the value of existing applications as we move forward to new programming models.

5.2.1 Inplace Migration

An inplace migration strategy involves refactoring an application for a new system architecture, while at the same time keeping it functioning and full-featured. This approach has been used in situations where application teams cannot afford the resources or disruption associated with more substantial refactoring efforts. Although this approach can be effective when architecture changes are incremental, it has not historically been effective when architecture changes are disruptive. For most applications preparing for

modern exascale systems, an inplace approach may give quick and incremental performance improvement; the long-term impact effect is either increased cost, poor scalability, or both.

5.2.2 Clean Slate Migration

During the MPP disruption of the 1990's, the most successful approach to developing a truly scalable MPP application followed the approach outlined in Figure 1. In this model, the basic data movement and dependence requirements are distilled and represented by a minimal set of modeling capabilities from the existing application. Then a new, clean slate application framework is developed to support the scalable execution of the minimal modeling capabilities. Once the new framework is robust and scalable, the modeling capabilities from the original code can be "mined," refactored, and integrated into the new application framework.

Recent experiences with the Concurrent Collections (CnC) model [21] has shown how dependence requirements of applications can be captured in executable models with a well-defined semantics. These executable models can also support automatic code generation for event-driven runtimes such as the Open Community Runtime (OCR) [44]. While it is well understood how to infer data movement requirements for a given domain decomposition or data distribution, there is a need for more experience with specifying data movement requirements at a more abstract level (e.g., in the form of communication avoidance [10], hierarchically tiled arrays [40], CnC affinity groups [25], or lower bounds on data movement [37], at the level of the experience gained with expressing dependency requirements in CnC).

5.2.3 Phased Migration

Independent of inplace vs. clean slate migration strategies, application teams must also decide how to best manage the (eventual) replacement of the MPI-supported SPMD bulk synchronous parallelism in an application, which may occur by migrating to future versions of MPI that have critical adaptations. Presently, most application teams are eager to introduce parallelism underneath MPI, replacing their sequential-only approach with a threading model. Many application teams are open to radical refactoring underneath the SPMD (MPI) layer. As a result, although we may need to replace the MPI layer with a more dynamic task-driven inter-node approach, the best practical strategy is a two-phased approach. First, focus on introducing a new node-level parallel programming approach. Then, expand the new approach to include inter-node activities. It is important to explore node-level parallel programming approaches that can be

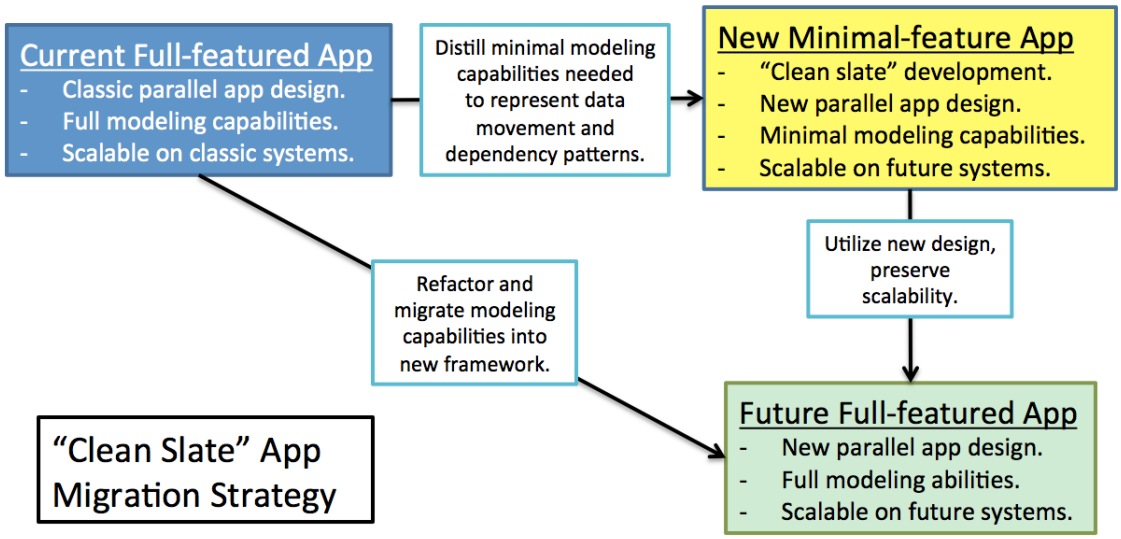

Figure 1: Basic "clean slate" application migration strategy for preserving existing application capabilities on emerging computing platforms. Application developers (domain experts) are often in the best position to identify the minimal modeling capabilities required to represent the full application requirements. The architecture of the minimal-feature app is often informed by design space exploration using miniapps or other application proxies.

integrated seamlessly with (possibly asynchronous) inter-node communications, so as to go beyond many current approaches to hybrid programming that require (for example) that all communications be performed by one thread in an MPI process. This could be achieved either by evolving or replacing MPI.

5.3 Path II: Developing New Applications in New Programming Environments

This section outlines the vision for developing brand new applications in a new programming environment. So-called "green field" development of applications is relatively rare these days—most applications rely on code or libraries developed from previous projects—so any vision for application development must address questions of incremental adoption and compatibility with legacy code. Nevertheless, the development of new code—or the migration of legacy code to new programming environments—offers important opportunities for improving performance and maintainability for scientific applications.

5.3.1 Requirements

The four guiding requirements behind a new programming environment should be: separation of roles, reuse, automation, and debugging/correctness. Due to the growing complexity of writing scalable software,

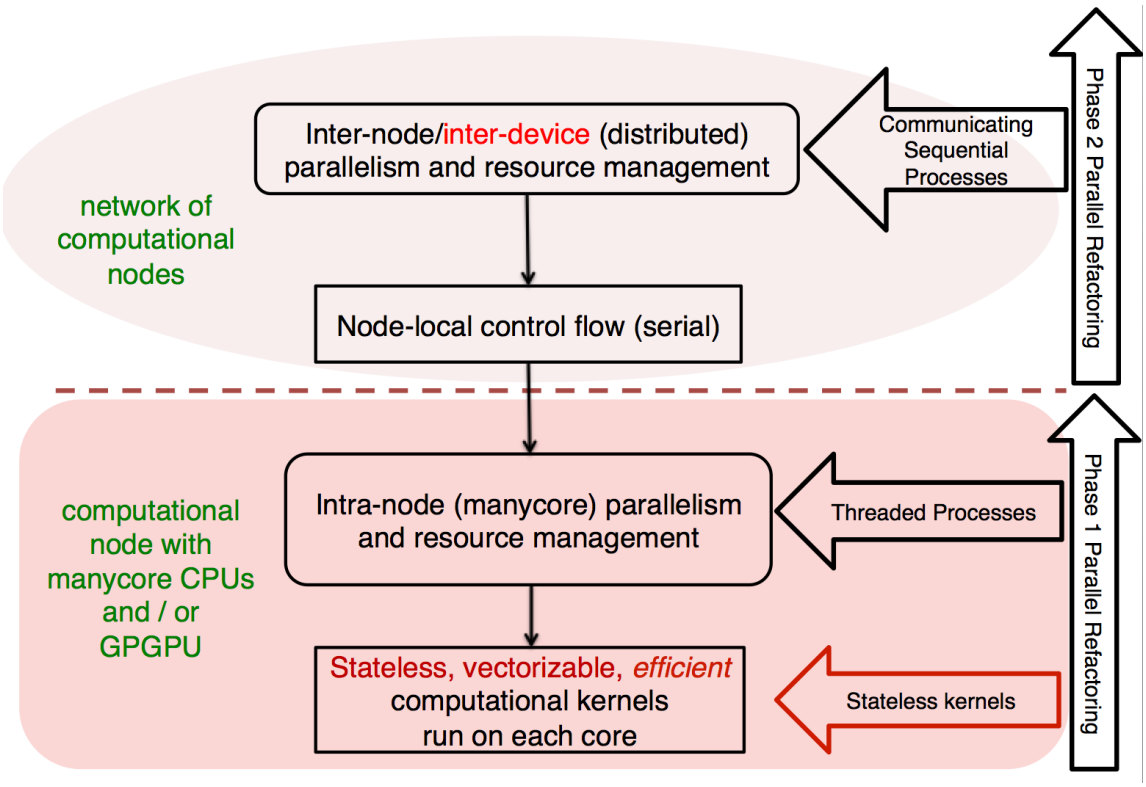

Figure 2: Most application teams are moving toward an MPI+X application architecture as sketched in this figure. The upper layer is executed as bulk synchronous using MPI. The bottom layer is the focus of application refactoring efforts at this time. Preparation for future extreme scale computing environments should leverage the application community's willingness to adopt agressive on-node refactoring as Phase 1 of a two phase effort.

future programming systems must support users with varying levels of expertise, from domain scientists to architecture experts, consistent with the multiple levels of abstraction described in Section **??**. Reuse has been recognized as one of the most important drivers of productivity, but it has proven challenging to achieve while maintaining performance. Automation, in turn, will be crucial in making applications forward scalable and easily ported as architectures change. Finally, if we provide higher abstraction levels to make programmers more productive, the system must continue to provide support for verifying correctness and debugging at the same level of abstraction.

Separation of Roles. One of the biggest challenges in developing high-performance scientific applications is the need for close interaction among experts with widely dissimilar backgrounds. Any non-trivial application in this domain will at least require the involvement of domain scientists, experts in numerical methods—usually applied mathematicians or computer scientists—and performance experts—generally computer scientists with a good knowledge of programming environments and tools, and a strong background

in parallel computer architecture and parallel algorithms.

The programming environment can have a significant impact on the efficiency with which these different groups of experts can interact. For example, the improved support for programmer-defined abstractions offered by C++ (compared to Fortran or C) has facilitated the development of rich frameworks and toolkits that help isolate domain scientists from the details of the low-level algorithms and data structures needed to support the desired functionality. In addition to simplifying initial development, the use of such frameworks and toolkits also helps isolate domain scientists from changes to the architecture that may require reimplementation of the framework but do not affect the client code. Despite the success of frameworks, however, a number of obstacles remain in achieving true separation of roles. It is difficult to achieve high performance with completely general frameworks, and specialized frameworks that can achieve high performance tend to be limited in their applicability and difficult to extend. In practice, these obstacles have meant that, despite the promised separation of concerns, successful use of frameworks, and especially the use of multiple frameworks in the same application, very close interaction is still required between domain, algorithm, and system experts.

Reuse. The software engineering community has recognized for many years that the key to productivity is reuse; i.e., the cheapest code is the one you do not have to write because it is already there. In the high-performance community, frameworks have attempted to draw from the best practices in software engineering to achieve a high-degree of reusability; however, the intense performance requirements of the domain have introduced several challenges in achieving high degrees of extensibility and reuse. The main problem is that many standard approaches to achieve reusability—such as relying on indirection and dynamic dispatch—tend to significantly reduce performance.

One consequence of this is that designers of frameworks often have to trade off performance, generality and usability. Generality and performance can be achieved by providing the user with many configuration options to specialize the behavior of the framework to a particular use, but this is often at the expense of usability. Similarly, frameworks often make assumptions about data layout and organization, which limits their applicability. In some cases, the assumptions are very strong and explicit: the data must be in a data-structure provided by the framework. But even frameworks that leverage generics (templates) and polymorphism in order to provide flexibility on the data representation have implicit assumptions about data layout; for example, iteration patterns in the framework often imply that some data layouts will be

more efficient than others. Finally, developing and maintaining high-performance frameworks that work efficiently across a variety of architectures is extremely difficult. Only a handful of frameworks have been able to do this. The programming systems of the future will have to overcome these challenges to provide the necessary separation of roles required for high productivity.

One of the challenges for future programming systems will be to allow for high levels of reusability without strong performance penalties and without an undue burden on the programmer.

Automation. Increased architectural complexity and the rapid rate of change in architectures will make performance optimization increasingly challenging. For example, the growing diversity in architectures will mean that optimization for a particular architecture will not be limited to tuning block sizes for loops and messages, but will require exploring many alternative algorithms, data structure choices, optimization strategies, and their combinations. Additionally, the performance characteristics of future architectures are likely to be much more dynamic because of power management. This means that developers will not be able to evaluate the performance implications of different design choices by timing a small number of executions, but will instead need to rely on statistical analysis and architecture models to evaluate the impact of different optimizations.

The programming systems of the future will have to provide very high levels of automation, while avoiding many of the pitfalls that have mired previous attempts at automation. The most important of these is avoiding all-or-nothing automation, where the programmer is entirely reliant on a heroic compiler to achieve high performance. When the compiler falls short, the programmer is left with a few unappetizing choices which include writing compiler plugins, mastering cryptic annotation systems, or most often simply rewriting the problematic functionality in low-level, architecture-specific code. By instead providing access at various levels to the implementation of mapping, automation can work in collaboration with developers to achieve the desired result. For example, even if code needs to be rewritten at a low level, the changes in the code should be well-contained and easily reversed, so as not to invalidate the automation. A "code" is more than one source file; it becomes a set of representations at different levels, with well-understood relations between the different representations. Also associated with this code object is the information on various experiments done with the code for performance tuning, decisions made by autotuners, refactorings made by programmers, etc. The programming environment should enable all this information to be used by higher-level automation tools, and to be exposed to the programmer in a convenient manner.

Checking and Debugging Support. Testing and debugging are among the most onerous aspects of development. Chasing bugs that only manifest themselves at scale will be particularly challenging, since traditional debugging techniques do not scale beyond a few dozen cores. The programming system will have to provide enough support for automated testing and computer-aided debugging. This may include increasing use of formal methods, combining static analysis and dynamic assertions, as well as automated test generation. The information generated while debugging and testing should also be part of the extended "program object."

5.3.2 DSL-Centric Development

All of the previously mentioned requirements can be met using the domain-specific approach outlined in Section A.1. The developer of a DSL is an expert programmer who understands both the domain and the right set of optimizations to map applications in that domain to a variety of architectures. The users of a DSL can be less expert, thus providing a multiresolution programming system (i.e., separation of concerns). Reuse is achieved by encapsulating domain-specific abstractions and their implementations. Automation, partial automation, and debugging support are achieved by the implementers of the DSLs.

5.3.3 Separate Mapping Meta-Information

The problem of generating optimized code that ports across architectures places too much burden on either programmer or compiler in the general case. An optimization that is very effective for one application or for one architecture may actually hurt performance in another context. Therefore, optimization strategies must be tailored to both application and architecture.

To achieve separation of roles, a desired goal is for the parts of an application to be expressed with a separation of concerns. The expression of the intended computation, which would be the responsibility of the domain scientist, should articulate a high-level (or perhaps mid-level) architecture-independent specification. Additional programmer specifications to describe details of data and computation partitioning on a particular architecture, as well as guidance for managing energy or resilience, should be separate from the specification of the algorithm. A separate specification means that it is possible to describe multiple mappings of the same high-level code to different architectures, power/energy constraints, or resilience requirements.

In Section A.4, we described a process of mapping from high-level, domain-specific implementations to

mid-level and subsequently low-level abstractions. This mapping was described as automatic, user-directed, or mixed. Any user-directed mapping requires such specifications. We would like the underlying mapping framework to be based on a rich set of optimizations and code generation capabilities that are general, which can be composed in a variety of ways to develop specialized mappings of different domains to different hardware.

5.3.4 Role of New Programming Models

Starting with a clean slate makes it possible to rewrite some or all of an application in new, more architecture-appropriate programming model. When DSLs are embedded into another programming language, then using new programming models is a somewhat orthogonal issue.

The introduction of new programming models and systems faces many obstacles, as listed at the introduction to Section 4. The great majority of newly designed programming languages die in their childhood. The few survivors often owe their survival to serendipity, or to major investments by vendors. The benefits of new programming models have been made clear in this report; a R&D environment and procurement policies should be developed that encourage sustained private vendor investment.

New programming models should enter the community incrementally, generating code for portions of applications that could not be practically implemented by hand, allowing focus of human investment on critical factors. These new programming models should be frequently evaluated for their effectiveness. As promise is realized, support for increasing breadth of application should be provided.

5.4 Path III: Performance portability beyond Exascale

Some of the research outlined in Path II will continue into the exascale era and beyond. A longer timeframe will create new problems, as well as opportunities for more radical changes in the way supercomputers are programmed.

As devices approach atomic scale, silicon scaling becomes increasingly difficult and expensive. Moore's Law, which governed the evolution of MOS technologies, is coming to an end in the next decade. New materials and new structures might help to continue the increase in the performance of integrated circuits; silicon manufacturers seem to be confident about scaling to the 5nm node. However, it is not clear that they can offer improved cost/performance beyond that and it is clear that they are subject to the same limitations

of size, as silicon devices cannot shrink to less that a few hundreds atoms.

Nevertheless, the evolution of supercomputers will not stop at exascale. More specialized hardware and software architectures can provide performance improvements of one or two orders of magnitude with the same technology and power budget. In the longer term, new information processing technologies will emerge.

5.4.1 Algorithm-Specific Hardware

It is well known that special-purpose architectures can achieve 50-100× better performance than general-purpose systems. An example is the Anton architecture for accelerating molecular dynamics computations [73]. The use of special-purpose computers has been limited in the past because of the longer development time needed for such systems, and the difficulty of refreshing the technology used in such systems. However, if semiconductor technology evolves more slowly, then longer development time is less of an impediment. Also, progress has been made in technologies for the synthesis of special-purpose systems.

A possible obstacle to this approach is that special-purpose computers are algorithm-specific, not application-specific: they accelerate one particular kernel, such as the computation of inter-atomic forces in molecular dynamics. DOE applications often mix multiple algorithms and models, so that in order to exploit special purpose hardware for them, multiple special-purpose systems may need to be tightly connected. This may not be an insurmountable obstacle: future microprocessors are expected to have more logic than can be powered at once. It has been suggested that one possible approach to leverage this situation is to populate the chip with specialized compute engines that are used as needed, and not used all the time [27].

The software stack for algorithm-specific hardware should share components with that for general-purpose exascale hardware. Specialized architectures will still have general purpose components that should be programmed with the general-purpose tools (compilers, etc.). Specialized accelerators will require specialized code generators. Verification-oriented compilation tools should have continuity with the system-level verification of the specialized hardware.

5.4.2 Reconfigurable Computing

The selective use of specialized accelerator engines on a chip is a simple example of reconfigurable computing. Another approach is to combine standard microprocessor logic with Field Programmable Gate ArraysField Programmable Gate Arrays, thus providing some hardware reconfigurability, where it is most effective, with-

out sacrifing the performance of custom designed components. This approach has a long history [39], but has become more practical in recent years. Some companies are now marketing processors with Field Programmable Gate Array accelerators, and Field Programmable Gate Array chips can now contain embedded microprocessors, memory controllers, I/O controllers, and other higher-level components. Intel is offering computing chips with in-package FPGAs, and this trend is expected to be continued, with speculation that the FPGAs will be brought on-die.

FPGA accelerators are increasingly being brought into the fold of other accelerator programming; for example, FPGAs can be programmed with OpenCL. However, it is critical to note that the OpenCL written for GPUs, which emphasize data parallelism, will not necessarily run well on FPGA. To get good OpenCL performance on FPGA, new structural idioms within the OpenCL are needed that emphasize pipeline parallelism. This illustrates the value of the application programmer writing code at a higher-level than OpenCL. Such higher-level code could be retargeted from GPU to FPGA, generating the OpenCL variant with the right target-specific idioms using a compiler.

5.4.3 Specialized Architecture

Hardware specialization is not a black-and-white situation. Current supercomputers already use cores and accelerators that are specialized for numerical applications; future supercomputers will use more specialized hardware to reduce their energy consumption. Some of these specializations may be unique to supercomputing; many others would have broader applicability, but might be deployed first on supercomputers. Examples of such specializations are listed below. Many of these have a strong chance of appearing in exascale architectures, because they can contribute so strongly to power efficiency.

Cache-less Computing: Caches consume a large fraction of the power budget of a chip: caches use a large fraction of the chip surface; each cache access requires multiple SRAM accesses (because of associativity) and cache misses can generate tens or even hundreds of such accesses. It is widely believed that the use of scratchpads could significantly reduce the power consumption of chips [11]. However, the manual generation of code for non-coherent address spaces is tedious, and current codes will not port easily. High-level compiler technology exists that can generate the code needed to manage scratchpads, including the generation of code that software pipelines DMAs or block reads and writes to the scratchpads for streaming computations.

Near-memory Computing: A significant fraction of the power consumption of chips is due to communication. This fraction is reduced if computations are done close to memory, e.g., in the memory controller of a Hybrid Memory Cube [49]. The programming model must be extended to support message-driven computation, also known as active messages or parcels, which invoke a computation starting at a particular data address upon arrival of an event or data token. To be profitable, such data-driven computations must be sufficiently coarse grained to amortize the overhead of sending the control-oriented messages that initiate computation. This implies that careful data and computation partitioning is still needed in using near-memory computing. This granularity selection is directly addressed with modern high-level compiler iteration and data tiling techniques.

Variable Precision and Approximate Computing: The use of 64-bit addresses and numbers increases the amount of physical memory consumed by applications and increases the power consumption, as more data has to be communicated and processed. It is well known that full precision is not needed everywhere in scientific computations: lower precision can be used in many places without reducing the accuracy of the results [8, 53]. While such techniques can be applied on current systems, using 32-bit and 64-bit arithmetic, better hardware support for variable precision could make them more prevalent. Early results indicate that precision choice could be largely automated [69]. Technology has emerged in the compiler community for careful management of error in the evaluation of mathematical expressions implemented in floating point [61]; although such work is framed in terms of reducing the error for normal precision floating point, it could potentially be used to choose optimal precision, or minimize error when reduced precision floating point hardware is available.

More dramatically, it seems feasible to use arithmetic circuits that consume much less energy, but commit systematic errors in low order bits, for some inputs [33]. The practical use of such circuits will require compilers that have significant awareness of the numerical properties of the compiled code. This might be achieved using some of the technology for management of floating point error described earlier.

Probabilistic computing: There is a fundamental trade-off between the power consumed by a device and the probability the device will have incorrect transitions: a larger energy gap between a zero and one reduces the probability of spurious transitions, by increasing the energy needed for that transition. By the same token, this increases the energy needed to switch state. Traditional computers have been built with logic where spurious transitions are rare and always detected. Ongoing research is considering

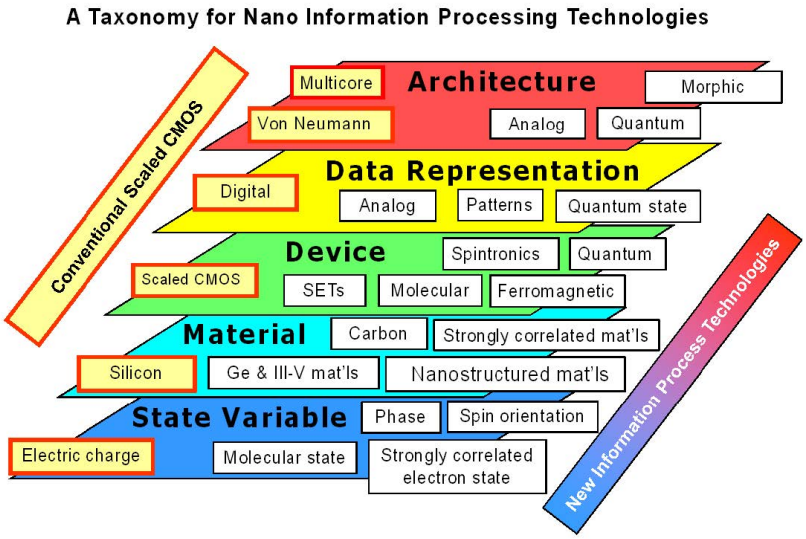

Figure 3: Information processing technologies of the future may use new state variables, materials, devices, data representations, and architectures (reproduced from the ITRS 2013 report).

the impact of forsaking this assumption. Algorithms are designed to tolerate frequent silent errors in memory or in the CPU, while still converging to a useful result, thereby reducing the amount of energy needed for computations [60].

Current work is mostly ad-hoc and algorithm-specific. The widespread use of probabilistic computing will require the automation of transformations that "bulletproof" computations.

5.4.4 Beyond Moore

Figure 3 indicates possible choices for future information processing technologies. State is often currently represented by a charge on a capacitor. In the future, it could be represented by a spin, or a phase in a material, or a molecular state. New materials can be used. New devices are being tested; e.g., spintronics for manipulating state that is represented by a spin, and so on. While not all the combinations are possible, the choice of future technologies is dauntingly large. However, some of these technologies have been studied in more detail and some observations can be made about their implication on programming models. Here

we mention one, Cryogenic Computing. A future study continuing from this report should comment in more detail on quantum computing, including adiabatic quantum computing.

Cryogenic Computing: IARPA has a large program aimed at developing technologies for cryogenic computing.[4] It has been estimated that the use of such technologies could produce a petaflop computer with a consumption of 25 kW. However, many obstacles still remain.

The use of cryogenic technologies has interesting consequences for software:

Cryogenic technology consumes very little energy for transmitting signals (as it uses superconducting wires), and very little energy for storing bits (as these are stored as a signal rotating on a superconducting loop). Almost all energy is consumed on circuits (including fan-out). This is a major reversal as compared to our current emphasis on reducing communication.

Cryogenic devices are now manufactured at relatively low densities ($\approx$ 200nm feature size). Devices can switch very quickly ($\approx$ 1ps), and long distance communication—even off-chip communication—will use very little energy. But signals will not propagate any faster; thus extensive use of latency-hiding technologies is paramount.

Cryogenic memory devices require a large number ($\approx$ 30) of Josephson junctions. As a result, cryogenic memory has very low density. Cryogenic computers will have very small amounts of fast, low-energy memory; this will need to be complemented by slower, denser memory. The design of software and algorithms for such a memory structure will be very demanding.

A possible software architecture for a cryogenic computer was studied more than 15 years ago by the HTMT project [79]. The proposed software organization was very different from that used in a conventional von Neumann machine.

6 Summary and Conclusions

Parallel programming models and environments R&D faces its most challenging period in history, perhaps ever, but at least since the start of distributed memory computing in the early 1990s. At the same time, extensible community programming environments and language standards such as LLVM, clang, Flang (in

[4] http://www.iarpa.gov/index.php/research-programs/c3

the future) and C++ features provide ways to develop new models and extend environments that result in rapid protyping, co-design with application developers and production-quality, portable tools that can be more easily adopted by applications. Whenever possible (and it is not always possible), new R&D efforts should be done within these environments to reduce cost, improve quality and accelerate use in production computations.

Opportunities for R&D impact include not only new models and environments, but also collaboration with application, runtime systems and hardware teams to design and produce application architectures that are more readily mapped to emerging systems. Distributed memory application architectures (SPMD with message passing) emerged to replace high level programming models like HPF, but drove the need for message passing models, environments and standards. New application architectures will help clarify what is needed for future models and environments that must support asynchrony, load balancing and the other attributes discussed in Section 2.4. R&D efforts must be done in collaboration with forward looking application teams for most impact, and there is a sense of urgency since our existing application base is not well prepared for scalable performance on emerging systems.

Basic research in programming models requires continued efforts in programming semantics including mixed task and data, replacement of local vs. remote explicit control to forms of shared memory computing that simplifies programming and yields performance portability, asynchrony management through event driven control, reliability clauses that permit user understanding of error responsiveness and more. Even if these kinds of concepts do not have sufficient mass appeal, pattern explorations that lead to reusable parallel programming patterns can have broad impact.

R&D efforts will have multiple timelines. It is clear that fundamental work is needed–separate from today's practical environments–that is highly speculative and has no obvious path for application migration. At the same time, some new R&D efforts must be done in tight collaboration with applications, runtime systems and hardware teams in order to produce practical models and environments with a migration path, an ability for domain scientists to write simple code and a utilization of industry and community programming environments.

Programming models and environments R&D is essential for the future success of our parallel applications. While we can persist with classic message passing and modest OpenMP approaches for a few years and for some applications, our existing code base is not ready for emerging platforms. The only way to sustain and increase the number of applications that are truly scalable on leadership systems, and to address the

ever-growing complexity of our emerging application base, is to develop new parallel programming models and environments that can enable these advances.

References

[1] Department of Energy Advanced Scientific Computing Research (ASCR). http://science.energy.gov/ascr/.

[2] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 19–31. ACM, 1976.

[3] J. Ang, R. Barrett, R. Benner, D. Burke, C. Chan, D. Donofrio, S. Hammond, K. Hemmer, S. Kelly, H. L. A. Leung, D. Resnick, A. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. Wright. Abstract machine models and proxy architectures for exascale. In *LBNL/Sandia Technical Report*, Feb 2014.

[4] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[5] S. Asbhy, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, T. Mezzacappa, P. Moin, M. Norman, R. Rosner, V. Sarkar, A. Siegel, F. Streitz, A. White, and M. Wright. The opportunities and challenges of exascale computing. Fall 2010.

[6] A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, et al. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[7] R. Babich, M. A. Clark, B. Joó, G. Shi, R. C. Brower, and S. Gottlieb. Scaling lattice QCD beyond 100 GPUs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 70. ACM, 2011.

[8] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.

[9] P. Balaji, S. Seo, H. Lu, L. Kale, Y. Sun, E. Meneses-Rojas, C. B. adn George Bosilca, T. Herault, S. Krishnamoorthy, and J. Lifflander. Argobots specification. https://collab.mcs.anl.gov/display/ARGOBOTS/Argobots+Specification, 2014.

[10] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[11] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78. ACM, 2002.

[12] P. D. Barnes Jr., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. Warp speed: executing time warp on 1,966,080 cores. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, pages 327–336. ACM, 2013.

[13] G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From relational verification to SIMD loop synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 123–134, New York, NY, USA, 2013. ACM.

[14] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *ACM Sigplan Notices*, volume 44, pages 219–228. ACM, 2009.

[15] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[16] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual IEEE Conference on Foundations of Computer Science (FOCS '94), Santa Fe, New Mexico*, pages 356–368, November 1994.

[17] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[18] S. Boldo. *Deductive Formal Verification: How To Make Your Floating-Point Programs Behave*. PhD thesis, Université Paris-Sud, 2014.

[19] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis. Wave equation numerical resolution: a comprehensive mechanized proof of a C program. *Journal of Automated Reasoning*, 50(4):423–456, 2013.

[20] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.

[21] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşırlar. Concurrent collections. *Scientific Programming*, 18:203–217, August 2010.

[22] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu. Runnemede: An architecture for ubiquitous high-performance computing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 198–209. IEEE, 2013.

[23] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. *Programming Models for Emerging Architectures*, 1(1):1–9, 2009.

[24] B. Chamberlain and J. S. Vetter. An introduction to chapel: Cray cascades high productivity language. In *AHPCRC DARPA Parallel Global Address Space (PGAS) Programming Models Conference, Minneapolis, MN*, 2005.

[25] S. Chatterjee. *Runtime Systems for Extreme Scale Platforms*. PhD thesis, Rice University, Dec. 2013.

[26] J. Chen, A. Choudhary, S. Feldman, B. Hendrickson, C. Johnson, R. Mount, V. Sarkar, V. White, and D. Williams. DOE ASCAC report on Synergistic Challenges in Data-Intensive Science and Exascale Computing. March 2013.

[27] A. A. Chien, A. Snavely, and M. Gahagan. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *Procedia Computer Science*, 4:1987–1996, 2011.

[28] P. Colella, D. T. Graves, J. N. Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. V. Straalen. Chombo software package for AMR applications design document, 2012.

[29] T. Coquand and G. Huet. The calculus of constructions. *Information and computation*, 76(2):95–120, 1988.

[30] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[31] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 53. ACM, 2009.

[32] DOE ASCAC Subcomittee. Top ten exascale research challenges. Technical report, Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Feb. 2014.

[33] P. D. Düben, J. Joven, A. Lingamneni, H. McNamara, G. De Micheli, K. V. Palem, and T. Palmer. On the use of inexact, pruned hardware in atmospheric modelling. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2018):20130276, 2014.

[34] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[35] H. C. Edwards, A. B. Williams, G. D. Sjaardema, D. G. Baur, and W. K. Cochran. Sierra toolkit computational mesh conceptual model. Technical Report SAND2010-1192, Sandia National Laboratory, 2010.

[36] R. G. Edwards and B. Joo. The chroma software system for lattice QCD. *arXiv preprint hep-lat/0409003*, 2004.

[37] V. Elango, F. Rastello, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. On characterizing the data movement complexity of computational DAGs for parallel execution. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 296–306, New York, NY, USA, 2014. ACM.

[38] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: Library-based syntactic language extensibility. *SIGPLAN Not.*, 46(10):391–406, Oct. 2011.

[39] G. Estrin. Organization of computer systems – the fixed plus variable structure computer. In *Proceedings of Western Joint Computer Conference*, pages 33–40, 1960.

[40] B. B. Fraguela, J. Guo, G. Bikshandi, M. J. Garzarán, G. Almási, J. Moreira, and D. Padua. The hierarchically tiled arrays programming approach. In *Proceedings of the 7th Workshop on Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, LCR '04, pages 1–12, New York, NY, USA, 2004. ACM.

[41] M. Garland, M. Kudlur, and Y. Zheng. Designing a unified programming model for heterogeneous machines. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 67:1–11, 2012.

[42] T. C. Germann, J. F. Belak, and A. McPherson. Anticipated programming models for scale-bridging materials science at exascale. http://http://xsci.pnnl.gov/ppme/pdf/Germann_pres.pdf.

[43] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proc. 42nd ACM Symp. on Principles of Programming Languages*, 2015.

[44] Habanero Extreme Scale Software Research Project. CnC on the Open Community Runtime (OCR). https://github.com/habanero-rice/cnc-ocr.

[45] S. Haney and J. Crotinger. PETE: The portable expression template engine. *Dr. Dobb's journal*, 24(10), 1999.

[46] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing '95*. ACM, December 1995.

[47] P. Hudak. Domain-specific languages. *Handbook of Programming Languages*, 3:39–60, 1997.

[48] F. Immler. Formally verified computation of enclosures of solutions of ordinary differential equations. In *NASA Formal Methods*, pages 113–127. Springer, 2014.

[49] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88. IEEE, 2012.

[50] L. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object-Oriented System based on C++. *ACM Sigplan Notices: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 28(10):91–108, October 1993.

[51] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[52] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comp.*, 20(1):359–392, 1998.

[53] H. Kaul, M. Anders, S. Mathew, S. Hsu, A. Agarwal, F. Sheikh, R. Krishnamurthy, and S. Borkar. A 1.45 GHz 52-to-162GFLOPS/W variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 182–184. IEEE, 2012.

[54] F. Lorenzen and S. Erdweg. Modular and automated type-soundness verification for language extensions. *SIGPLAN Not.*, 48(9):331–342, Sept. 2013.

[55] G. R. Luecke and W.-H. Lin. Scalability and performance of OpenMP and MPI on a 128-processor SGI Origin 2000. *Concurrency and Computation: Practice and Experience*, 13(10):905–928, 2001.

[56] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin. R-Stream compiler. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1756–1765. Springer, 2011.

[57] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. Investigating applications portability with the uintah dag-based runtime system on petascale supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 96:1–96:12, New York, NY, USA, 2013. ACM.

[58] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. Preliminary experiences with the uintah framework on intel xeon phi and stampede. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, XSEDE '13, pages 48:1–48:8, New York, NY, USA, 2013. ACM.

[59] V. Menon and K. Pingali. High-level semantic optimization of numerical codes. In *Proceedings of the 13th International Conference on Supercomputing*, pages 434–443. ACM, 1999.

[60] K. V. Palem. Energy aware algorithm design via probabilistic computing: from algorithms and models to Moore's law and novel (semiconductor) devices. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 113–116. ACM, 2003.

[61] P. Panchekha, A. Sanchez-Stern, J. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, 2015.

[62] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151. IEEE, 2008.

[63] M. Püschel, J. M. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[64] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[65] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[66] Reservoir Labs. Power API. https://github.com/reservoirlabs/power-api.

[67] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, 1993.

[68] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *SIGPLAN Not.*, 46(2):127–136, Oct. 2010.

[69] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 27:1–12, New York, NY, USA, 2013. ACM.

[70] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM, 2007.

[71] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. Müller. Assessing the performance of OpenMP programs on the Intel Xeon Phi. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 547–558. Springer Berlin Heidelberg, 2013.

[72] Secretary of Energy Advisory Board. Report of the task force on next generation high performance computing. Technical report, U.S. Department of Energy, Aug. 2014.

[73] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, et al. Anton, a special-purpose machine for molecular dynamics simulation. *Communications of the ACM*, 51(7):91–97, 2008.

[74] M. Snir, A. Maccabe, and J. Kubiatowiczs. Exascale Software Architecture: FastOS Common Vision, August 2014.

[75] D. Stark and B. W. Barrett. Opportunities for integrating tasking and communication layers. Technical report, Sandia National Laboratories (SNL-NM), 2014.

[76] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, Apr. 2014.

[77] L. R. Symes. Evaluation of NAPSS expression involving polyalgorithms, functions, recursion, and untyped variables. *ACM Symposium on Mathematical Software*, 1970.

[78] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM, 2011.

[79] K. B. Theobald, G. Gao, and T. Sterling. Superconducting processors for HTMT: issues and challenges. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 260–267, Feb. 1999.

[80] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[81] N. Vrvilo. Asyncronous checkpoint/restart for the concurrent collections model. Master's thesis, Rice University, Aug. 2014.

[82] Z. Xu, S. Kamil, and A. Solar-Lezama. MSL: A synthesis enabled language for distributed implementations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 311–322, Piscataway, NJ, USA, 2014. IEEE Press.

A Abstractions and their Compositions

In this section, we elaborate on a three-level decomposition of program abstractions. The highest-level abstractions are domain specific, and seek to achieve the previously stated goal of programmer productivity. Beneath these are domain-independent abstractions, whose aim is to provide functional and performance portability across machines. At the bottom of this pyramid are the execution-level abstractions, which provide direct access to and control over the execution of code and the placement of data. We also describe how to automatically or semi-automatically map from higher to lower levels, and the tools that might be employed to support this mapping process.

A.1 Domain-Specific Abstractions

It is an established idea that the best high-level programming abstractions are domain-specific [47]. For HPC, DSLs allow experts in a science to describe processes and algorithms in a language that is as close as possible to what they would use to describe an algorithm in a paper or on a whiteboard. Today, there are already many examples of systems that allow developers to code at a very high level of abstraction. Most of these systems fall into two broad categories: (1) stand-alone DSLs; and (2) frameworks. The use of DSLs, embedded DSLs, and frameworks should be expanded within the programming approach for exascale, because high-level expression provides productivity and sustainability, and it enables opportunities for domain-specific optimizations to be performed that can enable very high performance. New approaches for embedding domain-specific languages into general-purpose languages could bring about even more opportunities for optimization. C++ embedded DSLs are particularly attractive since they are portable and can deliver high performance when using template meta-programming techniques.

Stand-alone DSLs generally allow developers to express computation at a very high level of abstraction, and rely on domain-specific program representations to support high-level optimizations with minimal analysis. A very successful example of a DSL is the Tensor Contraction Engine (TCE), a domain-specific language for many-body theories in chemistry and physics [6]. TCE allows domain scientists to express their computation in terms of tensor operations and is able to produce implementations that would be prohibitively expensive to develop by hand. Other examples of successful DSLs include Pochoir—a DSL for stencils that can produce parallel and efficient (particularly with good cache behavior) stencil implementations competitive with hand-optimized code [78]—and Spiral, a system that can generate highly optimized

implementations of many signal processing applications [63]. Chroma [36] is a data parallel DSL for lattice field theory that can generate optimized code for a range of processing targets including GPU [7].

While the benefits of DSLs are positive, there are well-known challenges to developing a good stand-alone DSLs and seeing it used. First, it requires adequate investment, for both the expertise to properly engineer the compiler and the expertise in the domain.[5] By properly engineering optimizations and using "real" compiler technology, this course addresses the second challenge of a DSL, which is standing up an entirely new specialized compiler. Third, the development course for a DSL requires proper resourcing to fill out the ecosystem—with user training, syntax-aware editors, debuggers, profilers, etc. Finally, any custom DSL that requires a special toolset, presents a severe sustainability challenge. Numerous software studies have shown that 70–80% of total product cost is post-delivery maintenance. Unless a DSL has a broad, active developer and user community, the support cost of the DSL will be too large to warrant the investment. Because of this embedded DSLs are typically better.

Embedded DSLs rely on the abstraction and encapsulation mechanisms in a general purpose language to support domain-specific abstractions as a framework. Examples of this approach are Chombo [28], a framework for solving partial differential equations using adaptive mesh refinement, and Sierra toolkit [35], a framework to support solving PDEs over unstructured meshes. Most frameworks include a set of generic data structures that programmers can specialize to fit their problem domain, as well as libraries of operations that can be performed on these data structures. Some of these frameworks rely heavily on polymorphism (generics/templates) and high-order functions (often implemented using object-oriented inheritance mechanisms) to allow programmers to specialize the behavior of the framework to solve a particular problem instance. Frameworks address one challenge of the DSL approach, since they do not require a separate ecosystem of tools; programs written in the framework can be edited and debugged using the tools of the host language.

New embedded DSL technology is emerging that can help bridge the gap between frameworks and stand-alone DSLs. One example is illustrated by the Selective Embedding Just In Time Specialization (SEJITS) [23] approach, which leverages the reflection facilities available in most productivity level languages (python, javascript, ruby, etc.) to manipulate at runtime the high-level code written by the programmer. Traditionally, metaprogram execution would be at compile time; SEJITS extends this to metaprogram execution at run time. (The benefit of mixed static/dynamic optimization of the application itself for parallel computing is

[5] A typical course seen in the development of DSLs is for the team to initially under-invest in the compiler aspects, and then to find itself needing general compiler optimization technology. At this point, the DSL team works to re-engineer the DSL compiler to interface to a proper compiler. For example, TCE is currently making use of Rose; Chroma is moving away from using the Portable Expression Template Engine (PETE) solely, to using LLVM.

already well established [14].) The use of a dynamic language helps in to blur the line between library and DSL, although it is not strictly required. Halide [64], for example, follows a similar approach to embed a high-level DSL for stencil computations in C++. At run time, the programmer-provided DSL code actually builds an internal representation that is then manipulated, compiled, and linked with the program so that the code can execute efficiently.

A.1.1 Language Features to Enhance Support for Embedded DSLs

From the point of view of the programming environment, the most important thing is to make it easy to define domain-specific notations and their mapping to low-level code. Languages like C++ (and especially C++11) already provide important features that allow programmers to define domain-specific abstractions that have been used very successfully by a number of frameworks. Some areas for programming language R&D that will enable greater and more powerful use of DSLs are as follows:

- *Syntactic Extensibility:* C++ supports some syntactic extensibility in the form of operator overloading but in general, DSLs embedded in C++ tend to be much more verbose than stand-alone DSLs because of the syntactic restrictions imposed by the language. There is a significant body of work concerning the addition of syntactic extensibility to languages [38, 54]; some of this technology is being incorporated into the Rose compiler as part of the DTec X-Stack Project.

- *Symbolic Manipulation:* One of the major benefits of DSLs is that the exposed high-level semantics show optimization opportunities that would be hard for lower-level optimization tools to analyze. For example, given a set of matrix multiplications, it is easy to optimize the multiplication order at a high level through algebraic manipulation of the expression, but if instead we are given a series of nested loops implementing the said multiplications, program manipulation becomes significantly harder [59]. Expression Template [80, 45] programming (as in PETE) provides a means to accomplish this, and is used in HPC embedded DSL such as Chroma, but such programming is extremely difficult to master and understand and it makes libraries very complex. Also, expression rewriting is a limited (and somewhat crude) form of optimization; modern compiler optimizations are typically based on some form of dependence graph (e.g., SSA, Array-SSA, Concurrent-SSA, GVN, PDG, GDG, ...) representation of the program. As footnoted earlier, DSL optimization projects that start with expression rewriting typically discover along the way that it just better to plug in a proper compiler that provides the modern representation and optimizations. PETE is particularly complex because it co-opts the type system

into a data structure to represent expression trees, which is a highly stylized and refined programming technique at best. The Scala language has been very successful in supporting compile-time computation through lightweight modular staging [68] and has shown significant potential in applying the technology to develop high-performance DSLs [76]. R&D in this overall area is desperately needed.

- *Autotuning Support:* The high-level programming model should allow developers to make explicit statements about choices that can affect the performance, accuracy, or energy efficiency of the resulting implementation without affecting correctness. This can be accomplished through the good old fashioned concept of *polyalgorithms* [77], in which choices are expressed simply through idiomatic use of conditionals and branched code in the host language. A modern example of this is PetaBricks [4], which allows programmers to explicitly introduce choices as part of their code.

- *High-level Algebraic Properties:* The programming system should also give programmers the ability to specify high-level algebraic properties of functions or data structures that can be exploited for automatic or manual program manipulation. Some of the most important properties include equivalences among sequences of calls, as well as properties such as associativity and commutativity.

- *High-level Semantic Information:* This can include facilities to express precondition/postcondition in order to support expansion of optimization and mapping alternatives, to assist with verification and validation, to support composability, and to enable sustainment. The community for programming language research has very advanced and established technology available for reasoning about programs (e.g., Coq [29]), and new approaches for reasoning about and composing models correctly relative to deep properties of concurrency, security, and math (e.g., CompCert [43]). This community is beginning to show results in applying such technology to the formal proof of domain specific numerical solvers and floating point programs in general [19, 48, 18] and the ability to generate new floating-point math library routines with guaranteed improvements to accuracy [61]. Such technology could be a route to the general introduction of mathematical ontology to reason about correctness of domain-specific algorithms, particularly with regard to their concrete implementation in floating-point programs and with mixed or non-standard precision.

- *Debugging Support:* As the DSL reflects a higher level of abstraction than the code that will execute on the architecture, it is important not to forget to install the plumbing to reflect back to the programmer the state of the execution in terms that are representative of the high-level program.

In addition to these language features, the future success of DSLs will be assisted by general investment in the tool ecosystem that provides the ability to rapidly construct and extend them, as this can help overcome the long development cycle of general-purpose compiler frameworks which has been a limitation to their adoption in the application community. Further, for multi-physics applications, it is not unreasonable to imagine multiple DSLs composed together into a single code; thus, composability of DSLs must be supported. These two additional goals imply the need for an underlying meta-framework for embedded DSLs.

These high-level abstractions then rely on tools that take this domain-specific expression lower to the mid-level abstraction, which is more general across many domains but still hides details of the architecture or machine instance target.

A.1.2 Pragmatics

Support for DSLs is not a specific HPC requirement. In fact, DSLs are much more prevalent in application areas that have large user communities. For DOE, where the communities are smaller, a strategic approach led to successful use of DSLs. Suitable tools and language features can reduce the cost of developing a DSL. For maximum effectiveness, DSL development should integrate "real" compiler technology and involve "real" compiler expertise to avoid spending time and money rediscovering known optimizations and reimplementing them in silos. This requires proper investment. The investment in a DSL will achieve a good return because it enables significant high-level optimization for performance, documents semantics for sustainability, enables advanced verification approaches, and facilitates porting from petascale to exascale and beyond. Even when the DSL is for modest size user communities, the general ecosystem of the DSL approach can be shared across many DOE programs and a variety of applications. In general, library development with technology silos should not be proceeding without a DSL strategy.

A "co-design" effort of the CS and Applied Math community with domain scientists could help identify opportunities for new libraries, frameworks, and DSLs, and couple the development of language and tool technologies with their use in the implementation of new DSLs and frameworks.

The development of DSLs for scientific computing is facilitated due to the existence of a mathematical definition of the desired output, and a mathematical definition for many of the transformations involved in optimizing the DSL-generated code. Systems such as TCE and Spiral express their transformations as algebraic manipulations. It would likely be profitable to explore the extent to which such a rigorous algebraic approach can be applied to PDE solvers and other scientific kernels.

Ultimately, the impact of any particular DSL will be determined by successfully maintaining it over the required many years, if not decades, that an application written in terms of the DSL requires. If the DSL is part of the application code itself, or if it is portable code, such as template meta-programming in C++, then sustainability is not a major issue. However, if it is a stand-alone DSL, the cost of sustainability is a serious concern. The HPC community is too small and specialized to warrant heavy investment in stand-alone DSLs as a core strategy for programming models and systems research.

A.2 Domain-independent Abstractions

High-level abstractions encourage programmers to write their applications in the most portable and semantically rich way possible, often by using application-specific or domain-specific constructs. Low-level abstractions aim to provide maximal control and transparency to the programmer, and thus typically offer a minimally abstracted view of the underlying platform. Between these two extremes of the abstraction spectrum lie mid-level abstractions that provide a portable means of expression without relying on domain-specific notation or semantics. These abstractions are supported in general-purpose languages, such as C++, and their goal is to provide a domain-independent means of describing a parallel computation, and the data on which it operates, in a manner that still provides both functional and performance portability across a range of possible machines.

Machine architectures in the exascale time frame will exhibit deep, heterogeneous hierarchies of both processors and memories. They will support parallel execution at many levels—from instruction-level parallelism to coarse-grained task parallelism—and storage of data at the many corresponding levels of the memory hierarchy. The precise organization of these hierarchies may differ substantially between target machines, as will their favored mix of parallelism and data layout. Consequently, a program that seeks reasonable performance portability across a range of targets must be able to express the structure of parallelism and data in a manner that abstracts these choices. Given such an abstracted, target-independent program representation, the various components of the programming environment—including tuned libraries, runtime systems, compilers, and autotuners—should be capable of making a set of choices that achieves reasonable performance.

There will, of course, be cases in which automatic mapping by the programming environment produces performance that is lower than desired. The program may not adequately expose the inherent parallelism of its computation to effectively utilize a given target, or it may provide too little information about the

structure of its data. There may also be cases in which, even given the appropriate information, generating optimal code is beyond the ability of the provided tools. In these cases, it is important to both provide analysis and visualization tools which allow the programmer to understand the behavior of their program, and the means by which a motivated programmer could selectively write lower-level, target-specific code for critical portions of the program that can interoperate with the rest of the application. This interoperation is most easily achieved if both the portable abstractions discussed here, and the low-level abstractions discussed in a later section, are embedded in the same language.

A.2.1 Programming Model Components

Mid-level programming model abstractions express the logical structure of parallelism and locality in the program, without committing to a specific mapping onto the resources provided by the platform. The expression of parallelism and locality should also be in a form that can be mapped onto the varieties of parallelism that the platform might require.

Such abstractions often follow a form combining three fundamental components: (1) a pattern of computation/communication; (2) a collection of data being operated upon; and (3) user-provided function bodies that execute within the pattern. These abstractions may be embedded in the program in several ways, including extra-linguistic directives or language-supported constructs. The most common example of such a construct is a parallel loop. This may be written using directives, such as:

```
#pragma omp parallel for
for(int x=0; x<N; ++x)  F(x);
```

or as a language-supported parallel loop:

```
parallel_for(x : interval(0, N)) { F(x); }
```

This loop exhibits the three components described above. It names a computational pattern—a loop. It describes the data this pattern operates over—the half-open interval $[0, N)$. And it specifies an arbitrary function body that is embedded within the pattern—the loop body F(x). The named parallel pattern (parallel_for) permits different iterations to be executed concurrently and separately on different processors, sequentially on a single processor, or some combination thereof. The compiler and runtime environment, in cooperation, must decide precisely how to execute this loop. By supporting arbitrary function bodies, these

loops provide general-purpose functionality that can be easily reused.

Parallel loops are one of the simplest parallel patterns. Other fundamental data-parallel operations include reductions, prefix sums, and data partitioning operations. Common task parallel patterns include pipelines and fork-join recursion. A fully-realized programming model should provide a complete collection of flexible, reusable patterns that covers the space of necessary patterns. Attempts at assembling suitable collections of parallel patterns can be seen in directives models such as OpenMP and OpenACC, as well as C++ libraries such as TBB and Thrust.

Accompanying this set of parallel patterns, the programming model also requires a corresponding model of data. Programs require a collection of data types—such as arrays, lists, or sets—to structure the data on which they operate. Moreover, they require the means to hierarchically decompose these data structures into smaller sub-structures. Just as parallel computation patterns give the system information about tasks that can be executed across different processors, the ability to hierarchically structure collections of data gives the system information about data that can be stored across different memories. Separating the logical structure of the data from the physical layout also provides opportunities for the system to optimize the layout both for architectural characteristics (e.g., cache sizes) and algorithmic access patterns.

Real computations seldom involve a single instance of a parallel pattern or a single data structure. Rather, they consist of an interwoven chain of many patterns and nested data structures. It is therefore necessary that the programming model make apparent to the programming system the dependences between the various pieces of the program. This may be purely implicit, such as information produced by dependence analysis of the source program. It may also be provided in the form of side-band information, such as data directives decorating the program, or directly embedded in the program via mechanisms for composing operations (e.g., an explicit parallel pipeline pattern).

The full set of abstractions outlined here requires a higher level of expression than directives approaches such as OpenMP currently offer. Flexible languages, such as C++, with support for generic programming may be able to encompass the necessary set of abstractions. They may also be conveniently embedded in new languages such as Chapel. Regardless of the language in which they are embedded, the design of the programming model should be driven by the set of parallel patterns and data structures prevalent in parallel programs, and these patterns should be common across languages.

A.2.2 Target-specific mapping

A program written using abstractions, such as those described above, expresses the logical structure of parallelism in a target-independent fashion. The programming environment—in the form of the compiler, runtime, and standard library implementations—is responsible for constructing a target-specific mapping for executing this program on the target machine and for mapping its data structures onto the machine's memory hierarchy. These mappings, or *schedules*, will in general embody static and dynamic decisions. Certain scheduling decisions can be made statically when the program is compiled, but others can only be settled program has started to run and the dynamic configuration of the machine can be observed. Some scheduling decisions may even be data dependent, and thus emerge during the execution of the program.

One part of scheduling is to map the mid-level control structures—the parallel patterns such as loops, tasks, and reductions—to low-level execution constructs. The environment must decide which portions will be executed in parallel, and which will be executed sequentially within threads. A placement of this work on the machine resources must be chosen, and is generally driven by information about the affinity of work with the data it must operate upon.

Similarly, the programming system composed of the compiler, runtime, and supporting libraries will select how to map data structures across nodes and across different memory levels within a node. They may select the order in which elements of a structure are ordered (e.g., row major, column major, or block major, for arrays). They may select a specific low-level representation for a high-level data structure, such as a set (bit vector, list of elements, hash table, etc.). This mapping is now static (mapping across nodes), or managed by hardware (mapping to caches), but is likely to become more dynamic. With multiple memory levels and possibly non-coherent nodes, as well as the changes in program access patterns through different phases of algorithms, it will be important for the programming system to include automatically transposing storage layouts, redistributing data structures, or reconfiguring the use of fast memories during the execution of the program.

The compiler and runtime improve locality by aligning, to the extent possible, control mapping to data mapping so that operations are executed near the data they need. The problem for finding a good alignment is complex, even when the only degree of freedom is choosing an execution order that improves temporal and spatial locality in cache, for a sequential code, with a fixed data layout; e.g., by tiling loop nests and by performing loop fusion. The problem becomes much more complex with multiple execution locales, multiple

storage locales, and the dual abilities to move data to execution and execution to data. The two mapping problems are tightly coupled and will need to be managed jointly. Programming abstractions will need to be designed with the goal in mind of exposing the full space of mapping options to the compiler and runtime. It is not clear that the current abstractions used for expressing programs are sufficient for the needed automatic analysis and optimization. Even when they are expressed, new methods for co-optimizing execution and data placement will also need to be developed.

The compiler and runtime components that identify opportunities for parallelism and locality optimization should not be restricted to "flat" data types such as arrays or tensors. These properties must be expressible over more general data structures—such as sets, graphs, tables, and trees—or even algebraic structures—such as intervals or polynomials. This suggests that the required programming abstractions should have the ability to cleanly express polymorphism over data type, and also compose with the various available parallel patterns such as loops and reductions.

A.3 Execution-level Abstractions

The low-level abstractions provided by the programming environment are responsible for providing access to and an abstracted representation of the platform(s) used to execute the desired application. They must serve the needs of programmers who find it necessary to write low-level code, application libraries that seek to exercise careful control over the execution of their computations and the placement of their data, and the mid-level abstractions of the programming environment that will be mapped onto this lower level.

There are two broad categories of abstractions described in this section. First are the set of programming model abstractions that describe the computations to be performed and the data on which they are performed. Second are the set of platform interfaces that are used to describe and control the platform where the data is stored and the computations are executed.

The set of low-level abstractions provided by the programming environment should provide a complete programming target in themselves; it should be possible (though not necessarily practical or desirable) to write an entire program solely in terms of these low-level abstractions. Moreover, they should collectively support the needs of a broad range of programs, although individual abstractions may potentially be more narrow in scope. The functionality exposed at this level reflects capabilities that roughly align with the actual mechanisms provided by the underlying platform and can be expected to be implemented efficiently

by the platform. Low-level abstractions reflect how the computation will actually be performed, providing explicit control over the creation of parallelism and synchronization amongst parallel activities.

A.3.1 Underlying Platform Abstractions and Interfaces

The programming environment targets an underlying platform comprised of hardware, operating system software, and machine-dependent portions of the runtime system. The outlines of an exascale platform have been described in other reports [3, 74]. This section summarizes the platform terminology and components that most directly impact the form of the low-level programming abstractions.

A *container* is the set of physical resources (e.g., cores and memory) at each node in the system, dedicated to one application for a period of time. An application executes within a container on a collection of *threads*, each of which represents a sequential execution of a portion of the application code. All threads in a container access memory via a single, shared address space, and thus normally communicate via this shared memory. The shared address space within a node could be divided into one or more coherence domains. A *coherence domain*, shown in Figure 4, is potentially a subset of a node (otherwise usually true within the entire node) where the memory consistency model is strong, (i.e. where reasonable expectations of the visibility of loads and stores applies such as sequential consistency). Outside of a coherence domain, the consistency model is much weaker. While loads and stores that cross coherence domains might be provided by the hardware, there are no guarantees that references are kept fully consistent with references within a domain—there may be caching of the locations that will not be detectable from outside the coherence domain. Containers will usually run within one coherence domain, but could contain multiple coherence domains.

An *enclave* consists of two or more containers that do not directly share memory, and thus communicate via message passing or RDMA. The allocation of resources may be handled globally by a single operating system, or hierarchically by nested enclave operating systems that allocate resources to successive sub-enclaves. The low-level abstractions provided by the programming model will be restricted to a single enclave, but extended with abstractions that make it convenient to support high-level and mid-level abstractions.

Within a container are entities to execute instructions, called *execution units*. These roughly correspond to hardware threads or sets of threads executing a specific instruction stream. These units should have some parameters, such as absolute performance (relative to time) and performance/energy. Performance characteristics could also be expanded in various ways, e.g. floating point versus integer, vectorizable or not, etc.

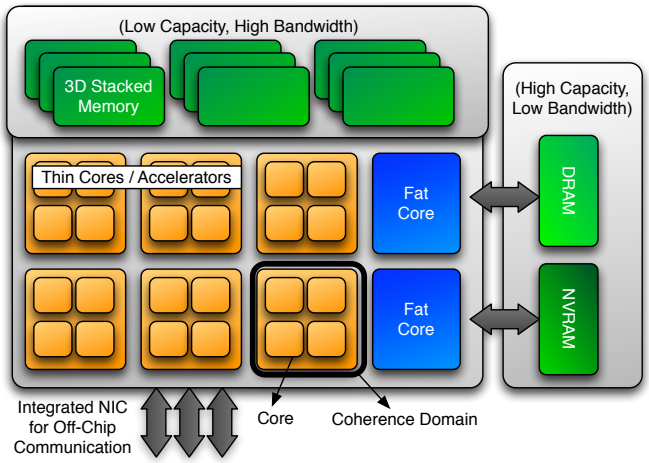

Figure 4: The increasingly hierarchical nature of emerging exascale machines will make conventional manual methods for expressing hierarchy, locality, and data movement very challenging to carry forward into the next decade.

A *storage* abstraction is also present for storing data. To the first order, this consists of explicitly addressable storage such as scratchpad or normal DRAM memory, but could also include various types of cache storage used for optimization. This abstraction has parameters such as bandwidth, capacity, and volatility. Latency could also be included. External storage is not considered here, since it is not within the container. It is expected that some storage technologies will allow for configuration—for example between scratchpad and caching style of operation—and this configurability may also need to be reflected. Storage may also provide some additional semantics, such as full/empty or trap-on-reference. Note that non-traditional processing elements should also be considered as part of storage. These could include concepts such as atomic memory operations performed at the memory, more complicated processors-in-memory, and processors-in-network.

Alongside the low-level programming model abstractions described in the previous section is a set of low-level interfaces for interacting with the hardware/software platform that provides the actual runtime environment for the program. These interfaces may be selectively used by programs that seek to adaptively tune their behavior to different machines, but their use is not required—programs could just as easily choose to ignore these interfaces.

Machine Representation and Introspection. The previously mentioned platform abstractions all require a representation, often parameterized, for building the machine model. They provide information on

the configuration of the platform, the topology of the resources and abstractions, and details on the specific hardware capabilities.

The abstract machine representations supported by contemporary programming models such as OpenMP and MPI are very simple, and can usually be specified by one or more integers (e.g., number of processes or number of threads). However, we anticipate the machine abstractions for exascale platforms to be much richer, largely due to increasing heterogeneity of processors and memory, increasing need to manage affinity and data movement, and increasingly deep machine hierarchies.

In addition to providing interfaces to manage processor heterogeneity (i.e., different kinds of cores), these abstractions should also provide facilities for managing performance heterogeneity. Ostensibly identical components may perform quite differently due to issues such as thermal, power management, manufacturing variation, wear and tear, and resiliency effects. Thus, part of the introspection interface is the capability to report the potential variability that may be experienced.

Power Monitoring and Control Interfaces. Given the importance of managing power consumption for exascale, interfaces will be required that provide detailed information regarding the power and/or energy being consumed by different abstractions when performing different operations for an application. In addition, some level of power control is expected (and is starting to appear today [66]), which will also be accessible through platform interfaces. These controls might include concepts such as running execution units more slowly for less power, or providing less bandwidth from storage for less power.

Resilience Interfaces. The low-level resilience interface starts by focusing on error detection and reporting. In response, reconfiguration will often be possible to avoid faulty components. While low-level system software (e.g. the OS) may perform automatic reconfiguration without the application's knowledge, more often some interaction will be needed with the application at run time to understand the effects of the fault, and to decide and direct the subsequent machine changes. The reliance interfaces would enable the application and/or higher levels of abstraction in the programming environment to reason about the needed resilience of a particular computation must be and handling recovery from errors.

Data Collection Interfaces for Profiling and Autotuning. Profiling interfaces provide detailed information regarding the performance of the various low-level abstractions. For example, they can report the number of instructions issued or floating point operations retired in a given time interval. They can also report on

the performance of various aspects of data movement. These interfaces are used for performance tuning and potentially autotuning frameworks, which use the data automatically to generate and select from a set of possible alternate implementations of an algorithm.

Note that if the goal is to tune for power consumption, or for performance at a given power level, then the power monitoring interfaces must also be consulted in concert with the profiling and performance interfaces.

A.3.2 Programming Model Abstractions

This set of execution-level abstractions forms part of the programing model, and every parallel program running on the system will use some portion of them, either directly or indirectly. They should provide an explicit, low-level interface whose form is closely aligned with the underlying mechanisms provided by the platform. The primary goals of these abstractions should be transparency and control, since they are the fundamental requirements of performance-critical code and software layers that implement higher-level abstractions.

Means of Creating Parallel/Concurrent Tasks. The first requirement for parallel programming is the ability to create parallel work. In some systems, programs may be initiated in a pre-existing parallel state. For example, distributed SPMD programming systems are often designed to create one or more processes per node before the application code begins executing. However, applications that must both exploit hierarchical heterogeneous machines and adapt to irregular computation patterns will often require additional mechanisms to dynamically create new parallel tasks under program control.

Low-level constructs for dynamically creating new parallel work build directly upon the underlying thread model provided by the platform. Therefore, they typically accept a *body*—specified as a code block, function-valued expression, or similar form—and create one or more *execution agents*, each of which executes the given body. An execution agent is a logical unit of work enumerated by the programming model construct. The mapping of these logical units onto physical threads is handled by the programming language environment in cooperation with the runtime system. The set of all execution agents created in a single invocation is a *task*.

These constructs should also support the ability to *configure* the task being created. Two configuration options are of particular importance. First, programs require the ability to specify the creation of an arbitrary number of execution agents in a single task (rather than only one at a time) in order to scalably create the

massive amount of parallelism necessary to fully exploit the underlying machine. Second, the program must be able to specify what scheduling assumptions are required for the correct execution of the tasks being created, e.g., whether it must be concurrent with its caller and whether its constituent execution agents must be concurrent with each other. These scheduling assumptions may also address resilience requirements. For example, a program might wish to indicate that a particular task can tolerate errors, thus indicating that the runtime is free to use aggressive voltage/frequency scaling that might improve throughput at the cost of higher error rates, or conversely that such scheduling policies would be undesirable because a given task must be reliable.

Explicit Synchronization Operations. Once parallel tasks have been created, they require synchronization primitives for correctly coordinating their activities. Programs require constructs for at least two fundamental synchronization patterns. The first pattern has a group of two or more concurrent activities that must synchronize with each other. Programming environments must efficiently support both the case where individual pairs must synchronize (which is common in point-to-point communication patterns) and where potentially large groups must synchronize at a barrier (which is common in bulk-synchronous programs). The second pattern has signaling the occurrence of events, either directly via an event abstraction or indirectly via mechanisms such as futures.

Synchronization constructs should also be integrated with the constructs for creating work, in order to more clearly indicate the scheduling dependences between tasks. Run-time schedulers can more efficiently utilize machine resources when scheduling dependences are made explicitly visible to them. For example, abstractions for events can be integrated with the constructs for work creation to indicate which tasks must be completed before the newly specified task is ready to run.

Explicit Data Movement / Communication Primitives. In addition to coordinating their activities with synchronization primitives, parallel tasks must also be able to exchange data. Constructs for communication must describe the movement of (potentially large) blocks of data in a way that the underlying implementation can efficiently leverage hardware resources such as RDMA engines. They must also support message passing in a way that can efficiently leverage high-performance Network Interface Controllers (NICs) and interconnects.

Means of Placing Data and Tasks. In order to efficiently utilize machines that are both deeply hierarchical and heterogeneous, low-level code may require the ability to reason about the location of program elements, especially data and tasks. This need is best addressed by providing an abstraction of the *place* where data resides and where tasks are executed. Abstractions of place are important both for inspecting and for controlling the location of program elements. For example, a program module may wish to learn the location of the memory holding a given data structure, so that it may create a new task executing nearby to create good locality.

Place abstractions provide an abstracted view of the available machine resources, specifically the memories and processors available for holding program data and executing program tasks. To provide programs with both flexibility and fine-grained control, place abstractions should be capable of representing machines at different levels of granularity. For instance, a program may at different times wish to refer to the set of all processors in a node, the set of all processors satisfying some criterion, or a specific processor. Place abstractions must also be able to represent the relationship amongst processors and memories, such as determining memory resources "near" a processor or set of processors. Finally, they must also be integrated with constructs for work creation, storage allocation, and so forth. This allows the program to specify, if it so chooses, the desired location for the storage being allocated or work being created. It also allows reasoning about the capacities of the various memories and execution units.

A.4 Mappings Among Levels

As previously stated in Section 4, the mapping of code and data to physical resources is performed across the high, middle, and low levels. To meet the productivity goal of Section 4, there should be a fully *automatic* path to perform this mapping. In many cases, automatically mapped code will be high-performance, embodying complex high-performance schedules that would be impractical to write "by hand." In cases where the automatic code does not meet performance goals, or the programmer wants more control, the system must permit programmers to intervene at any of these levels to steer the mapping, which we call a *directed* approach. The programmer should also be able to link in hand-optimized code at any location.

In theory, a programmer can always, given enough time, produce a hand optimization that matches or beats automatically generated code; however it will not be practical at exascale to write more than a small amount of such code by hand, since the number of hardware considerations for programmers to manage for performance at exascale will be large. Also, hand-optimized code is not likely to be performance portable

(and probably not even portable), since it will reflect close optimization for performance to specific hardware structures. A *mixed* mapping strategy, in which portions of the mapping are automatic and others are directed, allows human resources to be focused. It can also minimize production of non-portable code. Such a system should be *multi-resolution*, targeting different user capabilities, as it is likely that the programmer performing the directed mapping may be a computer scientist or computational scientist expert programmer whose efforts are encapsulated in the system to benefit the domain scientists.

To meet the need for automated mapping, a programming system must address the unprecedented complexity and diversity of future architectures. This can be addressed using a breadth of automatic mapping technologies, relying on current and new capabilities to be developed.

Compilers are a key part of the mapping strategy. There is broad and advancing knowledge of parallel program optimization to bring into practice, including advanced parallelization, explicit scratchpad memory management generation, explicit communication generation, joint parallelization and locality optimization, joint scheduling and data placement optimization, energy proportional scheduling, and so on. Optimization techniques that were once restricted to narrow program forms are expanding to more general program forms. Compilers can now optimize mathematical expressions in terms of reals so that, when they are implemented in floating point, error is minimized. Compilers can generate optimized event-driven task programs for exascale. Compilers can generate optimized OpenMP, MPI, CUDA, and OpenCL—in some cases generating program schedules with complexity far beyond what a human could generate by hand.

Even with all this compiler technology to choose from, there is much to gain from investment in research into new parallel optimizations, via new intermediate representations, analysis techniques, and optimization algorithms.

Moore's law helps with compilation—we have faster hardware and bigger memories on which to run our compilers—and with optimization algorithm advances, compilation approaches are now practical which were once considered infeasible "and then a miracle occurs" white board technology. The surplus of hardware on which to run compilation opens other opportunities. For example, since it is long settled that no one compilation heuristic applies to all input programs [2], modern compilers will often try many different heuristics and simply choose the one that performs best, or use other advanced search procedures (e.g., genetic algorithms). Modern compilers also rely on advanced mathematical optimization libraries (e.g., Mixed Integer Linear Programming (MILP) solvers like Gurobi or CPLEX); framing optimizations mathematically allows for the employment of powerful algorithms (e.g., Gomory cut generation) to rapidly explore and prune

vast spaces of potential mappings. There will be advantages to expanding compiler techniques to use new modern optimization techniques (e.g., convex optimization).

All modern compilers also expose knobs beyond command line flags to allow them to be plugged into autotuning frameworks. When the amount of time available for optimization is high or the kernel is particularly critical, superoptimization can be applied.

With complex hardware, static machine models used to guide compiler optimization are approximate. Mixed static/dynamic optimization techniques allow the compiler to make partial mapping choices, and generate code that completes the mapping by itself (e.g., with a compiler-generated inspector-executor form), or in concert with an advanced runtime (e.g., using compiler-generated dynamic dependences and affinities).

Compilers often run into trouble when presented with overly optimized low-level code. Analyzing such code that is overly "baked" to one hardware architecture, that uses impenetrable concurrent programming constructs, or that omits critical semantics, can be impossible. Exascale programmers should be encouraged to express their code using as high a level as possible, to avoid this pitfall. Compilers should be developed with control interfaces that allow them to be plugged into DSLs, frameworks, and libraries (e.g., telescoping libraries).

When compiler techniques fall short, or in order to get some more performance from the compiler, empirical optimization (aka autotuning) can play a critical role in mapping. Because empirical optimization can be very expensive and slow, recent research has used various techniques to explore a set of different mapping decisions in a systematic way. These techniques and associated tools will be even more critical as architectural complexity and diversity increase, and in consequence increase the cost of making a suboptimal decision.

The following is a description of tools that may be involved in the mapping process.

- **Compilers:** Use advanced program representations, analyses, programming language technology, and mathematical optimization technology, to rapidly generate correct optimized code variants.
- **Autotuners:** Autotuners perform empirical measurement of points from a search space of possible implementations, arising from programmer specification, compiler optimization, or some combination.
- **Heuristic Search and Machine Learning:** Various techniques must be employed to prune the

search space arising from autotuning so that only a manageable number of points in the search space are considered. Most commonly, the search space is explored using some sort of heuristic search, such as a simplex method or genetic algorithm. In some cases, machine learning is employed based on a priori training; for example, in cases where the best solution relies on features of the architecture or input program.

- **Synthesis and Verification** Synthesis is an emerging technology that can help derive code satisfying both certain structural constraints (e.g., constraints requiring the use of vector operations) and a functional specification. Recent work has shown how this can be applied both to support automatic [13] and human-guided [82] optimization from a high-level implementation. Verification can also help establish the correctness of transformations that cannot be proven correct using more traditional analysis mechanisms.

- **Architecture and Application Models:** Using models of the architecture and of how the application will map onto the architecture, the process of generating parallel code can be seeded with promising starting points, and the search space for autotuning can be significantly pruned.

- **Dynamic Runtime Systems:** Many decisions, including selection of optimized code, can be deferred to runtime when the execution context is fully known and information on the execution can be captured.

At all levels of the mapping process, the programmer should be permitted to intervene and direct the mapping. The following interactions with the mapping process may be performed by the programmer to provide direction or respond to feedback.

- **DSL:** The DSL can indicate to the programmer whether the constructs they are using are likely to perform well on the target architecture, and ask for information that can enable more aggressive optimization of the code. Programmers could also suggest algebraic equalities that can help the compiler better optimize a given program in the DSL.

- **Compiler:** Similarly, the compiler can mark code that is likely to be inefficient, and ask for information that can allow the compiler to rewrite the code to a form more amenable to optimization.

- **Models:** Models can predict the expected performance of the code on a target architecture, providing upper bounds that capture inherent properties and indicate to other tools whether further optimization is likely to be profitable.

- **Performance Monitoring Tools:** Tools that detect or help pinpoint performance anomalies can provide users with insights into performance bottlenecks in the code.
- **Runtime:** The runtime system can flag significant load imbalance, cache misses, power-hungry computations, and other sources of inefficiency.
- **Visualization and Interaction:** Development environment (e.g., Eclipse) plugins provide the programmer insight and control over program analyses and optimizations, and provide command line or scripting control over the application of optimizations.
- **Autotuner:** The autotuner can mark which code variants and parameter ranges led to the best and worst performance, in order to help the programmer understand this relationship.

In summary, the mapping process must rely on a collection of tools and interfaces to make intelligent decisions, or guide the programmer to intelligent decisions.

B Glossary of Terms

bulk-synchronous A programming model where execution consists of a fixed number of threads, and all synchronizations have barrier-like semantics: The computation proceeds in successive phases where any operation by a thread at phase i appears to precede any operation by any thread at phase $i + 1$, and is not ordered with respect to operations on other threads at phase i.. 26, 28

coherence domain The collection of processing elements in a node or system where reasonable expectations of the visibility to one processing element of individual loads and stores from another processing element apply. Often considered strongly or sequentially consistent. 68, 78

container A set of physical resources (for example, cores and memory) which are dedicated to one application for a period of time. 68, 78

data tile a subset of the logical values of an array grouped into a block to improve locality. 19, 23

dependence graph A graph that describes the order in which instructions or tasks have to be executed (or appear to be executed) for a correct execution. 26

DSL Domain specific language, say more. 20, 21, 29, 58–60, 62, 75

enclave A collection of multiple containers, typically not part of the same coherence domain. As such they normally communicate via message passing or RDMA. 68

Execution Model The lowermost parallel programming model that is exposed by the system hardware OS and common run-time.. 21, 22

execution productivity A measure of the functionality that a programmer produces versus the cost or effort of writing the code. 18

execution unit Hardware that executes a sequential stream of instructions. 19, 68

Field Programmable Gate Array An integrated circuit designed to be configured after manufacturing. FPGAs are typically less dense and less fast than custom-designed logic, but can be customized to better the needs of an application than a general-purpose CPU.. 44

fork-join A programming model that results in executions that are series-parallel graphs.. 26, 27

framework There are two common definitions of framework in the programming language community: (1) in contrast to a library, where a user program calls the library for some service, a framework is a set of templates and procedures that exerts more control over the user code and data structures, even to the point of inverting control so that the user code plugs into the framework, and the framework is calling the user code; and (2) as a means for interfacing and coordinating parts of programs written in more than one programming language, as in Google Protocol Buffers. Scientific computing frameworks bring both aspects—significant imposed functionality and structure, as well as multi-language support. There is no clear separation between a framework and an embedded DSL—the term embedded DSL is mostly used when the hosting language is more flexible, such as SmallTalk or Python, so that the programmer is (almost) not exposed to the embedding language. 58

Hybrid Memory Cube A memory package that consists of multiple DRAM chips layered atop a logic chip, connected with vertical vias.. 45

iteration tile a subset of the iterations of an algorithm grouped into a task for improved locality. 19, 23

parallel programming environment The collection of compilers, libraries and tools that implements one or more parallel programming models. 18, 20, 21

parallel programming model A programming model provides a set of abstractions that simplify and structure the way the programmer thinks about and expresses a parallel algorithm. 18, 20, 22, 23, 78

Performance Model A model that provides an estimate fo the resources consumed by a program. 21, 22

programmer productivity A measure of the functionality that a programmer produces versus the cost or effort of writing the code. 18

RDMA Remote Direct Memory Access. 18, 22, 30, 68, 78

serial semantics Said of a parallel language where the outcome of the execution of a parallel program is equivalent to the execution of a sequential program that can be easily derived form the parallel program. Sequential semantics facilitate debugging and testing.. 26

thread A software vehicle for the execution of a sequence of instructions. 68

work stealing A task scheduling algorithm whereby spawned tasks are queued locally and normally executed by the local thread; threads that are idle "steal" tasks from remote queues. 30