2015 Programming Models and Environments Workshop: Summary Report

March 9-11, 2015
Rockville Hilton, Rockville, MD

Editors:

Sonia R. Sachs, DOE ASCR
William Harrod, DOE ASCR
Thuc Hoang, DOE NNSA

Report Contributors

Pete Beckman, Argonne
Mary Hall, University of Utah
Rob Hoekstra, Sandia
Rob Neely, Lawrence Livermore
Dave Richards, Lawrence Livermore
Martin Schulz, Lawrence Livermore
John Shalf, Berkeley Lab
Armando Solar-Lezama, MIT
Thomas Sterling, Indiana University
Kathy Yelick, Berkeley Lab

Background

Proposed exascale computing architectures present the programmer with a host of challenges. In particular:

- Heterogeneous processing elements and application-managed NUMA hierarchies will require more complex abstract machine models. Portability of these abstract models between different physical architectures may be limited.
- Very large increases in the number of execution streams required to utilize hardware efficiently will force programmers to identify and expose more concurrency in their applications. This, in turn, requires that the overheads associated with creating, managing and synchronizing this concurrency must be minimized.

- If highly asynchronous (and perhaps non-reproducible) execution environments become mainstream, they will present daunting challenges for debugging and performance analysis tools and techniques

To overcome these challenges, programmers need improved programming models and techniques that will allow them to accomplish the goal of writing applications that will perform well across a variety of machines from different vendors or generations.

To address the research challenges outlined above, this workshop convened approximately 45 domain experts in High Performance Computing Programming Models and Environments (PM/E) together for 2.5 days with the following high level objectives:

1. Propose, discuss, and determine the required characteristics or properties of future programming models
2. Review the application requirements for ECI PM/E(s)
3. Identify the key capabilities and elements of future programming environments
4. Determine the research questions that need to be addressed
5. Propose the methods for evaluating PM/E(s) research results

The workshop was a combination of invited speakers and open breakout sessions for directed discussion. The sections in the reports that follow capture primarily the discussions of those breakout sessions, which in turn covered the following topics:

- Application Requirements for Exascale Programming Models and Environments (PM/Es)
- Methods for Evaluating PM/Es
- Characteristics of PMs and PEs
- Outstanding Research Questions

The workshop laid out the progress that has been made in the advancement of new and alternative programming models and environments through the ASCR X-stack Program, as well as application-centric efforts through the NNSA ASC Program and the ASCR co-design centers to explore both evolutionary and revolutionary approaches to developing exascale-ready applications.

Spurring renewed interest in this area is the recent announcement of the ASC Program's Advanced Technology Development and Mitigation (ATDM) program element, which charters the ASC labs with developing the first "from scratch" set of applications in support of the weapons program since the start of ASCI in the mid 1990's. Several ASC teams are actively exploring using next-generation programming models (beyond MPI+X), and work is starting immediately to get those off the ground. This revitalized application effort, along with the pre-exascale procurements in the CORAL (2017) and APEX (2020) timeframes led to a sense of urgency at the workshop for hardened research solutions.

Key Takeaways

Some of the high level issues raised and suggested next-step actions include:

Identified Issue	Suggested Action / Next Steps
Basic terms and their scope such as *runtime*, task-based *models*, *tasks*, *dynamic* vs *static,* and *interoperability* are not well defined	• A process needs to be put in place to adopt common definitions and their scope of these terms that are central to the research community efforts. • Need agreement on what is considered "OS" versus "runtime", while acknowledging there is overlap. Suggested definitions include privileged access (OS) vs user access (runtime)
Programmers interface to programming models at multiple levels, and we need to start distinguishing which of those abstraction levels a particular PM is focusing on	• The proposed taxonomy (see also p. 12) suggested and subsequently adopted for discussion in this workshop was High Level (Domain scientist), Mid-Level (Algorithm specialist), and Low Level (Tuning Specialist). This separation of concerns should be finalized and fleshed out to help focus discussions, and avoid PM's misrepresenting themselves when they are largely focused on just one of those abstraction levels**. UPDATE: The discussion at the April meeting seemed to conclude that only two levels were necessary.**
There is a natural tension when considering programming models and environments between monolithic vertically-integrated solutions, and well-defined interoperable components	• The layers in programming model implementations need to be articulated (to the extent possible), and separation of concerns defined with standard APIs and bi-directional communication so that research can be focused, and vendors can focus on value-added pieces of the software stack. • Interoperability between runtimes is a desired (some say imperative) goal
Introspection, or the ability for the various parts of the runtime and applications to query state is a critical challenge	• Tools efforts (e.g. MPIT and OMPT) are pathfinding efforts in this space, and could define a set of key introspection parameters that runtime systems are expected to provide through standard APIs
Adoption of new PM/Es must be done in concert with application teams who are expected to adopt them.	• DOE must work harder to partner application teams with PM/E developers in a co-design process, and do so early in the design cycle.
Application teams are getting anxious about the timescales for adoption	• Pace of research and development must remain consistent with the schedule for machine deliveries, and what research can realistically be delivered in those timescales in a state suitable for production code adoption.

	• We need to begin moving select research products into a hardened state suitable for early adopters, and fully understand the limitations of MPI+X before asking application teams to change
Pattern-based approaches provided another valuable model for separating concerns.	• Further research is required to understand how pattern-based approaches and/or pattern compilers can help focus PM/E efforts going forward.
Support for improving existing codes and models cannot be ignored in the rush to advance new PM/E research	• A two-track approach (evolution vs revolution) to research PM/Es as part of exascale must be taken, with commensurate funding available to both.
Runtime systems needn't only support resilience, but be resilient themselves *(this probably belongs in the other report)*	• Resilience services, but also design and implementation a exascale runtime services is a critical challenge
The impact of PM/E research needs to be broadly disseminated, particularly to the application community who are largely unaware of the potential	• We need to catalog the open research questions, track those that we're learning the answers to, and work to share peer-reviewed success stories with the broader community
A recurring comment during the workshop was the need for a requirements-based design. Since a primary goal of the exascale project is to deliver science, one measure of success will necessarily be satisfying application requirements.	• This document contains a high level discussion of application requirements on the programming models and execution environment. These requirements need to be further fleshed out and expanded and then broadly disseminated for review and comment.
Application developers need performance models and the associated cost models for the most likely programming models to be supported on exascale systems.	• Since there is no clear, discrete set of programming models for future exascale systems, domain scientists need practical advice on the tradeoffs between the programming model alternatives. Research is required in order to develop these models.
The selection of programming models is the linchpin for exascale research. Much of the work in hardware features, run time systems, programming environments, resilience, algorithms, and even application modifications	• The performance and cost models mentioned above must be elucidated in a way that resonates with application developers. The applications need to illuminate the path to viable programming models.

cannot be done without knowing what programming models will be supported on exascale systems.

In addition to the table above, several of the session authors below elucidate their own research priorities in more detail.

Session I - Application Requirements for Exascale PM/E(s)

Chair: Rob Hoekstra, Sandia

Summary

This section attempts to capture the requirements from the HPC applications community with expectations placed on new programming models and environments (PM/Es), which facilitate the effective use of Exascale computing platforms.

Requirements

The driving focus of application requirements on the programming model is to allow acceptable levels of both productivity and performance. With the ever-increasing complexity of HPC systems there is the inevitable risk that future PM/Es will likewise reflect that complexity. The likely outcome is degradation in both productivity and performance as application teams with finite resources are able to address a decreasing scope of the programming environment. In addition, the current disparity in proposed designs threatens to compound the problem and make effective code portability nearly unobtainable.

Therefore, first and foremost, the requirement that applications would make on future PM/Es is to *manage complexity presented to the developer and strive towards reduction in that complexity whenever possible*. There are two possible leverage points identified that can used to address this concern.

1. **Abstraction of performance aspects of the hardware**. The current dominant model of bulk synchronous parallel (BSP) mapped well to traditional distributed memory platforms but it has become an increasingly ineffective model for more modern architectures. A new model must expose key aspects of the architecture while simplifying them to the essential concepts in an attempt to span a range of architectural designs. An example would be multi-level memory hierarchies. The majority of future hardware architectures incorporate some aspect of this complexity and a programming model that exposes a common abstraction for memory spaces would allow a programmer to address this concern, hopefully in a performance portable manner.

2. **Layered abstractions ranging from hardware aware low level abstractions to domain specific high level abstractions.** A PM/E 'stack' which exposes less and less details of the hardware as you go up the stack allows for separation of concerns. This creates significant opportunities for productivity increases at the application programmer level while still allowing performance experts to delve down into the lower levels and address performance aspects. Admittedly domain specific languages (DSLs) and other 'high-level' abstractions have often been met with cynicism in the past. This cynicism persists but there is a recognition within the application developer community that the time may have come for higher level constructs to manage the ever increasing complexity.

Much of the discussion above implies the value of *pattern* based approaches. Coupled with the programming model abstractions, common patterns can be extremely powerful in representing performance critical aspects of the community's code base. Concepts such as *patterns*, *motifs* or *dwarves* have been proposed in the past with some community acceptance. However, they have not played a significant role in the

development of programming models. The new paradigm we face with Exascale platforms creates a renewed opportunity to explore these concepts with the hope of deriving greater benefit. In fact, the community's embrace of proxy applications seems to be a clear indication of a need in this arena.

Semantic Constructs

There is clearly an increased acceptance by the community that revolutionary changes are in store for our PM/Es. This strongly implies a significant change is semantic constructs. To a great extent, the expectation is that new semantics will be required to better express constructs for fine-grained parallelism, resilience, power and data movement to name a few. The current 'bulk synchronous' model can be construed as dominated by serial constructs with message passing constructs 'wedged in' to support the parallel aspect of the software. A programming model that embraces parallelism and treats data movement on node as well as off node as a first level concept seems necessary in the future. A good example of the former is the emerging directed acyclic graph based task parallel models (e.g. ParalleX).

As mentioned in the previous section a layered approach seems warranted allowing the application programmer to focus on domain specific concepts while lower levels of the programming environment stack address the details of performance. The complexity this implies must be carefully managed however. In many cases this will place a much greater burden on the development and execution environments (including tools) than in the past, and implies a significantly greater cost in both the development and deployment of such. An example on one such issue already impacting the community is interoperability of programming systems such as OpenMP and pThreads. Without significant support within the runtime layers to manage resource contention, this interoperability is nearly impossible.

An additional caveat to this approach is that it must not hinder the development of revolutionary algorithmic approaches that are better suited to the new hardware and a crucial aspect of Exascale computing. Historically, algorithmic advances have outstripped hardware performance improvements by orders of magnitude. These types of advances will likely require R&D that spans a significant fraction of these layers and the PM/Es should allow for this to be viable.

No matter the path we follow, there is a crucial need to scope out both a 'vocabulary' and begin a dialog on open standards for these new PM/Es. The application development community is ramping up engagement with the expectation that they will need to dramatically overhaul their code bases. We must supply them with a target and support those who are willing take risk in investing in these emerging technologies. These 'early adopters' are essential to success as we go forward. As much as possible, the PM/E development must be cognizant of the tremendous resources that may be required to support multiple re-writes of large scale codes and the pain and suffering incurred for those early adopters in particular.

Performance/Portability/Productivity Tradeoffs

The panacea would be that new PM/Es would allow improvement in all three dimensions of concern: *performance*, *portability* and *productivity*. While much of the Exascale discussion focuses on performance, the end goal is delivery of high quality and impactful computational science and engineering. Numerous examples of this tradeoff exist in our community with most application teams settling for 'good enough' performance and

portability due to limited resources in addressing machine specific concerns. Without improvements in the PM/E space the expectation is that this will only worsen.

At the same time, this is an opportunity to explore revolutionary new approaches that have the potential to positively impact all three dimensions. In the end, it will be difficult to find consensus from the application community on this tradeoff. Different workloads drive different value systems that may prioritize a variety of different metrics: new science, throughput, etc. PM/Es must be able to respond to the varying demands. A layered set of abstractions as described previously seems likely to be a successful approach with this in mind allowing those who value performance to invest greater resources at the lower levels. Whenever possible however we want to increase the likelihood that performance improvements developed in one community become readily available in the HPC environment for adoption by others. We recognize that pattern-based approaches hold significant promise in this regard.

Evolutionary vs. Revolutionary

The application community has had the luxury of a relatively stable programming model represented by bulk synchronous parallel (BSP) and expressed most often with MPI semantics. There is significant evidence that message-passing technologies could carry us well into the Exascale era but there is equally damning evidence that BSP is rapidly becoming a barrier both for scalability at the node and system scales. This implies that the programming model almost certainly must change if we expect to adapt to the new technologies represented in future HPC platforms.

Current open standards such as MPI and OpenMP are showing an ability to respond to these new demands and the community expects to continue to rely on them possibly for decades into the future. That being said, revolutionary approaches have by their nature a potential to dramatically improve productivity, performance and portability as noted previously. However, if we are to be successful in deploying these new PM/Es, we must aggressively engage the community to develop common abstractions that can lead to open standards. If history holds true, there will be many failures before success but only through the understanding gained by those failures can success often be reached. To support the application community, a PM/E stack that allows the existing ‘systems’ to be initially used and then replaced by new approaches seems the most effective approach.

A final note of caution; much of the content above drives the application community to dependences on a much deeper, more complex software stack to implement these PM/Es. This is a community that has often written their codes to depend only on the ‘lowest common denominator’ of the software stacks to minimize their risk and dependencies. Early socialization, which relies on the growing ‘Co-design’ community, will be essential. Co-design approaches have already born significant fruit in this regard and we should strive to make them an intrinsic part of our culture. It is clear as well the vendor community must be an integral part of his effort since in the end they deliver the production system software and tools. Their involvement should increase the adoption of common open standards based software stacks and tools which are crucial to the outlined strategy.

Session II – Methods for Evaluating PM/E(s)

Chair: Armando Solar-Lezama, MIT

Charge

The charge for this breakout session was to answer the following questions:

1. What are current best practices for evaluating programming model/environment (PM/E) prototypes using DOE applications?
2. What are the strengths and weaknesses of those approaches?
3. How is parallelism and scalability to be measured and projected?
4. What metrics should be used for future evaluations of PM/E?
5. How can the development of proxy applications be improved to help with this evaluation?

Discussion summary

One of the first questions that was raised in the breakout session was: what is the goal of evaluation was. The following alternatives were raised:

- Measure the research output of the program.
- Measure the value of a particular project.
- Determine what should be transitioned to production.
- Determine when we can recommend that the applications team try something.
- Determine when we can recommend to the applications team that they adopt something.

With regards to the second potential goal, there was some concern that a checklist approach could be used in evaluating projects, leading to the cancelation of promising efforts.

In an ideal world, projects should be evaluated on the basis of the criteria that actually matter:

- Does a PM/E allow us to do science that we can't do today?
- Does a PM/E allow us to work with machines that would be impossible to program with today's approaches.

Unfortunately, evaluating on both of these measures cannot be done until you have full production quality programming environments, and by then it may be too late to discover that an approach didn't work, so the goal is to find good proxies that will allow us to make predictions regarding these two outcomes.

There was some discussion as to the extent that ease of adoption should be a criteria. It was agreed that there could be more tolerance for a painful adoption process if stronger benefits could be demonstrated, as illustrated by Figure 1.

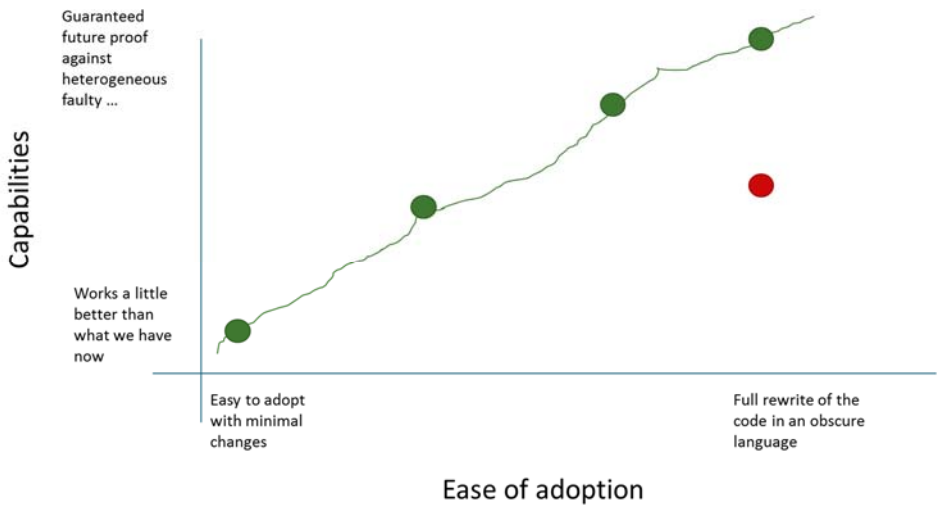

Figure 1 Space of tradeoffs between capabilities and ease of adoption

Among the capabilities that would be desirable and that could be evaluated were: Raw performance, memory consumption, support for exascale specific issues such as fault tolerance, heterogeneity, explicit cache management, future proofing and performance portability, as well as analyzability and the extent to which certain classes of bugs can be ruled out by the model.

One point that Thomas Sterling emphasized was that we need to separate the programming model from its implementation, and in particular, to make sure we don't rule out potentially good programming models because of not-so-good implementations.

Also, measuring against some of the exascale issues is difficult because we do not yet have exascale machines. With regards to performance portability, it was suggested that comparing performance portability across the existing variety of machines. It was also suggested that it is important to focus on how the performance portability happens: is it fully automatic, is it manual but with a strong separation between correctness and performance as in Halide where the mapping from high-level to low-level implementation can be defined separate from the high-level DSL description of the algorithm. With regards to aspects like fault tolerance or energy efficiency, the challenge in evaluation is that we do not yet know how the ExaScale machines will look like.

Measuring ease of adoption was universally agreed to be harder. The main criteria that were brought up were

- Is it close to what people already use?
- Is it clean? Easy to learn?
- Is it general and expressive?
- Is there an ecosystem for it beyond us?
- Does it support incremental migration, particularly for software that is already modular and well designed?
- How transparent is the system in terms of performance and for debugging purpose?

Composability was also suggested by Ron Brightwell as a good metric.

Kathy Yelick suggested thinking in terms of the process to go from research to production, and thinking about metrics that are relevant at every point in that transition. The transition is illustrated in figure 2 and involves the following steps:

1) Start talking to applications people
2) Start to engage with vendors (in parallel with #1)
3) Reference implementation of PE
4) Write application component jointly: PE and Apps group (goto #1)
    a. Evaluation done jointly
5) High scoring implementation (at least for some case on some metric)
6) Experiment with production code
7) Develop multiple implementations for risk avoidance
8) Toolchain support
9) Full documentation, support, hardening
10) Applications using programming models to get science results on exascale

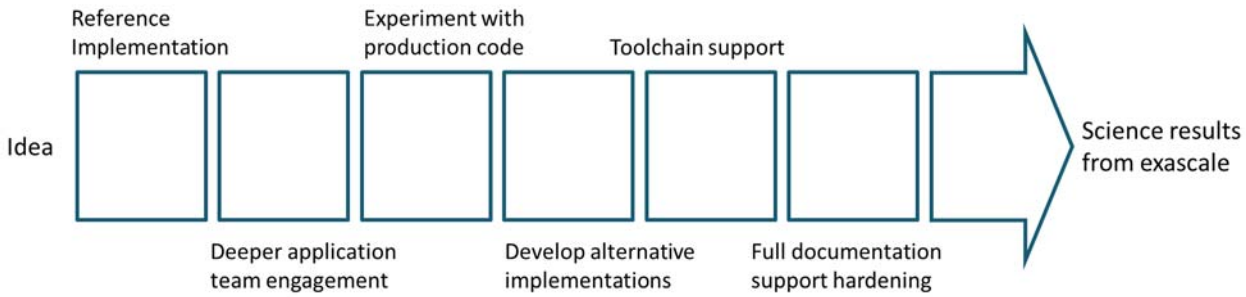

Figure 2 Transition from idea to production tools

So for example, while mini-apps will not be enough to convince people to use a tool, they may be useful in demonstrating that the idea is promising and that science groups should start having some deeper engagement with the application teams. But once it gets past that stage, then it is important to have something more complex than miniapps to really demonstrate that a PM/PE can work for real codes.

Finally, there was a lot of discussion about the difficulty in measuring human-factors type of issues, as these things are difficult to measure, especially before you have production-ready prototypes that can be tried out.

Sessions III – Characteristics of Programming Models

Chair: David Richard, LLNL
Chair: John Shalf, Berkeley Lab

Charge

The overarching questions for the programming models breakout discussion were

1) How to map application requirements (taken from breakout session 1) onto future programming model constructs.
2) Identify dependencies
3) Clarify the role for runtime systems in programming environments

The programming model abstractions and their semantics were enumerated as follows

- Parallelism
- Global Name Space
- Locality/Distribution/memory hierarchy
- Synchronization
- Introspection
- Resilience
- Communication
- Time/Space/Energy management

The task of the breakout was to attempt to map the set of requirements (high-level domain abstractions in the PM/E report) that were enumerated in the applications discussion onto various programming abstractions (mid-level abstractions according to the PM/E report).

Multiple Levels of Abstraction

Although it may seem tempting to drive programming model research to converge on a single exascale programming model with a unified set of abstractions, such an effort is likely to fail. No such convergence exists with current hardware or applications, and no single programming model or set of abstractions is likely to meet the diverse and even contradictory requirements of all exascale programmers and workloads. To give just one example, how can a single programming model expose all possible concurrency in a given workload while simultaneously aggregating or grouping concurrent work to maximize utilization of specific hardware resources such as caches, scratchpads, SIMD units, or warps?

In actual practice programmers interact with code at multiple levels. Each level has its own set of goals, concerns, and abstractions. The table below shows key abstractions needed in programming models needed for three different levels. At the high level, models for domain scientists express the geometry and equations of a science problem without concern for algorithmic or machine details. Concurrency is captured through expression of dependencies rather than attempting to specify parallelism directly. Ideally, significant code at mid-level models could be machine generated from the high level descriptions, but mid-level code hand written by algorithm specialists will prevail in regimes where domain specific abstractions are difficult to codify. Mid-level abstractions are closer to actual hardware concerns, but still attempt hide specific details behind appropriate abstractions. Low-level models are the home of tuning specialists concerned with performance critical details such as explicit memory placement and

movement. At this level programming models must match the architecture closely. For performance portability to be a reality we must identify methods to generate the vast majority of an application's low level code directly from higher level models, leaving hand tuning for only the most critical kernels.

Abstractions Needed/Expressed by Three Levels of Programming Models		
High Level (Domain Scientist)	Mid-Level (Algorithm Specialist)	Low Level (Tuning Specialist)
• Science geometry o Locality (implicitly?) • Equations • Dependencies instead of parallelism • Data abstraction to express dependencies (references, aliases) • Algorithmic name space(s) • Data flow • Problem state Introspection support for tools & diagnostics	• Locality/hierarchy • Dependencies • Parallel patterns o Scan, foreach, task DAG • Error sensitivity & recovery hints • Containment • Communication • Global name space(s) Time/space/energy management	• Explicit data placement & movement • Explicit control of functional units • Synchronization & Locking • Pattern compiler & runtime • Introspection • Resilience notification & actions • Idempotence • Global name space(s) • Communication • Time/space/energy management Roles of runtime and compiler support

Table 1 Separations of Concerns between different categories of programmers demands different levels of programming model abstractions

The observations from Table 1 correspond strongly to the discussion of "separation of concerns" that was identified in the first chapter (Figure #1) of the first ASCR Programming Models Report (July 2011 in Marina Del Rey, CA.). It also connects very well to the discussion of different levels (high, mid, low) of programming abstractions in the 2014 ASCR PM/E report.

Developer Roles	Domain Expertise	CS/Coding Expertise	Hardware Expertise
***Application**: Assemble solver modules to solve science problems. (eg. combine hydro +GR+elliptic solver w/MPI driver for Neutron Star simulation)*	Einstein	Elvis	Mort
***Solver**: Write solver modules to implement algorithms. Solvers use driver layer to implement "idiom for parallelism". (e.g. an elliptic solver or hydrodynamics solver)*	Elvis	Einstein	Elvis
***Driver**: Write low-level data allocation/ placement, communication and scheduling to implement "idiom for parallelism" for a given "dwarf". (e.g. PUGH)*	Mort	Elvis	Einstein

Figure 3: This "separation of concerns" diagram shows the stratification of developer roles and expertise on a complex code project. The matrix represents the expertise of these respective programmers in the areas of the science domain, numerical algorithms, and the low level details of computer architecture and tuning. A person is an "Einstein" if they are an expert, an "Elvis" who is conversant in the area, and a "mort" who is a non-expert. The programming environment offered to a domain may require a different set of criteria than the environment offered to the tuning experts at the driver level.

Mapping Between Levels of Abstraction

The vision for the separations of concerns identified in Table 1 falls apart if there is no way to map between the different levels of abstraction. The mapping would implicitly be defined as an API at its most base level. There is general agreement that an automatic system cannot be relied upon to infer these mappings from end-to-end. The mapping process will require hints to make an impossible expert-system reasoning task into a process that is automatable and practical. Also, for any mapping decision that is particularly expensive, it may still need to expose additional information at the highest level to guide the decision-making and mapping process. For example, rather than having the programming system infer the optimal data layout on the underlying machine, it might be better for the upper level to describe the shape and dimensionality of the data so that the lower layers can automate the mapping process. Rather than have the compiler automatically infer parallelism, it may be beneficial to describe data dependencies at a high level so the lower layers of the system have more information available to make decisions about how best to schedule work on the underlying computing elements.

The biggest concern about layered abstractions is brittleness – particularly of the intermediate layers of the software. How can work our way towards the higher-level software abstractions without the intermediate layers becoming ossified? One option discussed was for higher layers of the hierarchy of abstractions to be more declarative in order to give the lower layers more freedom for optimization, and for the lower layers to be more imperative to provide precise control over the hardware resources. Spiral was given as an example of this approach, but other codes use this approach effectively to build more durable abstractions. In general, this kind of brittleness has been the biggest impediment to creating durable high-level abstractions (PM/E has largely remained confined to lower levels of the hierarchy of abstractions as a result).

Relating to Application Requirements

The organization of the programming model abstractions into this hierarchical model resolves some long-standing debates such as the role of MPI in our programming environment. The primary criticism of MPI is that it is almost an assembly language level of abstraction with regard to implementing parallel constructs. However, the layered abstractions allow that an application programmer should rarely or never need to code directly in MPI – the communication layer would exist at the lowest level of this hierarchy and utilize encapsulation to hide the details from users at the middle and upper layers.

The debate over programming models should therefore shift from which low level abstraction to use and determine how to map between the layered abstractions and whether the higher-level abstractions are ultimately serving the needs of the domain scientists (e.g. are we able to use encapsulation to properly embed the lower levels such as MPI communication). All too often, the upper layers of abstraction are implemented in a manner that is specific to a specific application. There are opportunities for common programming systems to supply higher-level abstractions to bridge the gap from low level to high level. However, candidates to fill this role of any given programming abstraction must identify its target audience (domain scientist, algorithm specialist, or tuning specialist) and must be vetted in the context of utility to the end user at that level of abstraction.

Given the separation of concerns defined in Table 1, *there is no single level of programming model abstractions that satisfy the requirements of the applications scientists*.

To evaluate different programming approaches in the context of application requirements, the question should be recast as:

1. What abstractions automate tedious things that programmers do in practice to optimize codes or to make them portable across systems?
2. Which programming models/products offer abstractions or features that automate porting and tuning tasks that are tedious, complex and expensive?
3. What does it look like to use them? Is it productive? Is it understandable? Is it maintainable?

Key Issues and Open Questions

Building an ecosystem that supports interoperable programming models across all three levels will require significant efforts in both research and development. Key issues that must be addressed include:

1. Domain specific languages (DSLs) have proven to be very effective high-level models in certain domains such as stencil calculations. How transferrable is this success to other domains? How can we produce and maintain code generation systems that produce efficient low-level code in a variety of machine specific low level models across a wide variety of DSLs for a large number of domains? How to bridge the gap between application abstractions and DSL abstractions?
2.

3. When DSLs are not available what are the alternatives that best support portable performance? What abstractions will likely prove to be durable across vendor platforms and over time?
4. What are the proper abstractions and techniques to express, compute, layout, and schedule dependencies? Can we express them independently? Are they dynamic or static, implicit or explicit? What are the alternatives for control?
5. Libraries have proven their value as resources for portable performance. How will libraries interoperate in codes that are composed in a variety of programming models?
6. Exascale Machines will have massive on-node parallelism. What is the appropriate way of expressing fine-grain models? What is the role of task-based models at various levels of granularity? How do we express/ensure locality?
7. As flop-to-bandwidth ratios continue to fall, an increasing number of workloads will find that performance is limited by memory and network bandwidth and/or latency. What models and abstractions will be most effective in helping programmers maximize their use of bottlenecked resources?
8. What a resilience model would look like? What is the impact into each level of programming models?
9. How will models at every level interact with debugging and analysis tools?
10. Issue of performance vs performance portability (need to expand on this).
11. Issue of data and task parallelism (need to expand on this).
12. What are the implications of these issues for applications improvements/re-writing?
13. Exascale machines will be even harder to understand than current machines. What are the requirements of programming models to support debugging and performance tools? How do we tie things back to the original code from where a bug or performance bottleneck was found? Attribution problem. APIS should be defined that makes this easy to do.
14. Memory and I/O is much more complex on exascale machines. What are the mechanisms for managing memory and I/O? What are the mechanisms for managing data locality and data movement?
15. Mappings between the abstractions and mapping to the machine will be needed. What are these mappings? What is the role of code generation?
16. Optimization questions. Abstractions have interfaces that need to be optimized. What performance optimizations can be automated through translation or RTS support or auto-tuning? What AMM needs to be exposed? Abstractions need to have well defined semantics and interfaces.
17. Performance and cost models questions.

The process of addressing these questions, and other that we encounter along the way will need to be a collaborative effort between application scientists and programming model developers. The right timing of such collaborations will be essential and expectations have to be managed carefully. If you do not have a complete tool chain then it can be difficult to interact with an app - but if you wait until the tool chain is complete then you will miss important opportunities to interact.

Finally, the pace of research and development must remain consistent with the schedule of machine deliveries; at some point we have to figure out what can be delivered in time for a particular generation of machines, keeping in mind that there will also be machines delivered in the future.

Findings & Recommendations

1) No single programming model or unified set of abstractions is likely to meet the diverse and even contradictory requirements of all exascale programmers and workloads.
2) There is no single level of abstraction for programming models.
    a. 3 levels were identified: domain scientist, algorithm specialist, low-level tuning.
    b. The correct abstraction depends on the target audience.
3) The quality and value of a given abstraction can only be understood in the context of its target audience
    a. Evaluating a low-level abstraction such as MPI or tasking
    b. Encapsulation of low-level constructs in higher level constructs is
4) Encapsulation of low-level abstractions to form higher-level abstractions can either be provided by the programming environment or by application codes or by libraries.
    a. Programming languages should not be the sole determining factor for the success of the PM/E program.
    b. Library encapsulation and software engineering to create layering of abstractions should be considered as a co-equal path to success as a new programming language or tool.
5) Application-level (high level) abstractions are relatively weak in current PM/E relative to available low-level abstractions.
    a. Such abstractions have high value for productivity because they can automate expensive and tedious tasks currently handled by programmers
    b. Such abstractions have higher risk because they potentially have a narrower scope of applicability

Session IV – Characteristics of Programming Environments

Chair: Mary Hall, University of Utah
Chair: Kathy Yelick, Berkeley Lab

Overview

Beyond the programming model, which is used to encode an application, the *programming environment* contains tools that are used to help programmers understand and develop their code. This section of the report combines the output of two independence parallel breakout sessions.

As an initial organizing activity, the list of programming environment examples provided in the preparation materials, and then used this list to narrow down on some key examples which are discussed in detail below:

- Compilers
- Code transformation
- Code synthesis
- Code generation
- Debuggers
- Auto-tuning
- Workflow management system
- Data analytics
- Visualization
- Storage Systems
- Optimized libraries
- Performance analysis and visualization
- Performance modeling
- Correctness tools
- Resilience tools
- Build-and-test frameworks
- Software quality tools

Performance Analysis and Correctness (and Data Analytics and Vis) at Scale

The dominant theme of this discussion was how to detect performance bottlenecks and correctness issues for applications running at exascale. We noted that data analytics and scientific visualization pose similar challenges, but we did not discuss these further. We also pointed out that energy management was considered a subset of performance and could be treated in the same way. The current state of the art for petascale computing is already proving to be a challenge for application programmers, and the issues are expected to significantly worsen given anticipated hardware and software changes for exascale. A current challenge is that performance analysis and correctness debugging during production runs is made difficult by a reduced software stack on compute nodes, intolerable overhead, and batch scheduling. That means that much of this analysis and debugging work needs to be done on scaled down application configurations typically running on different hardware, possibly with a different software stack. Further, techniques for verification of implementation, both static and dynamic, may not be practical at scale. Consequently, intermittent problems or ones that only

arise at scale are difficult to pinpoint. For this reason, the applications often build in their own checking that can be run at scale, and rely on checkpoint/restart to recover from hardware failures. As a result, bugs and performance bottlenecks remain challenging to detect and correct.

The challenges associated with current approaches will grow in an exascale regime, assuming 100x more threads, more frequent soft errors, dark silicon, more dynamic or asynchronous parallelism, more complex multi-physics applications, and more complex software stacks. Our group talked about how integrating information across the software stack, and interacting with programmers in new ways would be required. If we anticipate non-deterministic execution behavior due to soft errors, dynamic scheduling, or other factors, then the coverage testing phase of an application for performance and correctness must consider the design space of anticipated execution environments. We considered the possibility that fault injection might be required during testing if soft errors become prevalent.

Dynamic adaptation of the execution environment or the application's execution will demand on-the-fly attribution of bottlenecks and errors to a cause, providing information directing the programmer to the problematic portions of the source code. Attribution to a cause would demand sophistication in the hardware and software stack to not only collect measurements from a variety of sensors but also detect anomalies, patterns of execution behavior, and erroneous results. This sort of sophistication may require significant interaction between layers of the software stack, and due to scaling issues, must collect, analyze, draw conclusions and then discard data. Further, measurements may need to be tracked over longer time scales, rather than using instantaneous data.

While some bottlenecks or errors may be the result of the execution environment, others might arise from intentional optimizations introduced by the application programmer or programming system. For such application-specific optimizations, it may be valuable to have application or programming systems to control and/or verify that they have not harmed the accuracy of the scientific results. For example, a number of approaches to reduce computation or avoid data movement, including communication-avoiding algorithms, variable-precision arithmetic, and relaxing data dependences across multi-physics computations, may affect accuracy of the scientific result. As another example, an application programmer may wish to reduce the power or energy requirements of their application at the cost of some performance. When making such accuracy/energy/performance tradeoffs, what tradeoffs are tolerable is dependent on the application or application domain, and therefore must be expressed as part of the application or its underlying domain-specific libraries and programming systems. We also need to build into applications the verification of scientific results. Further, we may want to provide mechanisms for introspection so that applications can self-diagnose bottlenecks and errors.

Performance Portability

Ideally, application programmers would like to write and maintain a single application code that runs well across many platforms. This type of performance portability may not be practical with current and expected machine diversity – already applications have multiple code bases or "ifdefs" in the code to switch implementation and even algorithmic approaches based on architectures. An important goal of the exascale program is to minimize, if not eliminate, the changes needed to move applications across machines and still obtain excellent performance.

Both breakout groups identified that the approaches to performance portability roughly fall into three broad categories: *encapsulation*, *translation*, and *auto-tuning* – including some combinations of these three. These are discussed in more detail below.

Encapsulation

Encapsulation involves refactoring code to separate (roughly) the mathematics from the computer science, i.e., by writing the code in such a way that the inner loop nests can be easily identified and therefore modified when one is moving code to new node architectures. This provides little real performance portability, but makes it easier to make the changes between platforms. User-friendly abstractions that make use of operator overloading and templates, e.g., Kokkos, RAJA and TIDA, can also help with this type of encapsulation. At a higher level, techniques that separate algorithms (e.g., expressed as a DAG of dependent tasks) and schedules can also aid in performance portability; in this case the scheduler may need to be rewritten to take advantage of features like heterogeneity, and some level of common portable must be possible within the tasks (e.g., it doesn't handle different SIMD intrinsics), but still provides a fairly narrow part of the application to manually port. Applications that can take advantage of highly optimized libraries (LAPACK, PETCs, FFTW, etc.) also gain performance portability as the libraries may be manually ported and tuned without changing the application. Sparse matrices and stencil computations are examples of things that don't fit as well into this class, since the set of matrix structures or stencil operators is too large to encapsulate in a library.

Translation

Solutions based on translation take some type of input, e.g., a standard program text or domain specific language, and convert that into highly optimized and specialized code for each platform. These have shown promised for areas like stencils because the computations are fairly simple loop nests and can be analyzed and translated for sufficiently constrained languages. The approaches include separate domain-specific languages (DSLs), general-purpose languages with possible annotations, DSLs embedded within a general-purpose language, and use of template metaprogramming to perform translation at template instantiation. Both full-fledged compilers and special-purpose translators (specializers) are used.

Domain-specific programming systems – languages, libraries or optimizations -- can provide specialized implementations from high-level mathematical or algorithmic specifications. Not all domain-specific systems aim for high performance, as their main advantage is programmer productivity, but we saw examples in the plenary talks that with underlying architecture-aware mappings, domain-specific systems can produce code that matches the performance of manually-tuned code. Future research in this area will need to address composition of domain-specific tools for applications that draw from multiple domains, and tools that ease the development of domain-specific programming systems.

Auto-tuning

Translation and encapsulation provide ways of generating architecture-specific code, but don't provide techniques, on their own, for selecting the right version of the code. Traditionally, a performance model built into a compiler would perform that selection, but in recent years the use of auto-tuning has proven useful. Auto-tuners select from a set

of possible optimized versions by running them on the target system and choosing the best possible within a class of inputs, e.g., matrix size. This is used within libraries, compilers for both general purpose and domain-specific languages. For some settings, wholesale algorithm replacement may be necessary, and for this auto-tuners sometimes include support for algorithmic choice, in which a set of equivalent algorithms can be specified and used as part of auto-tuning.

Many in the breakout session were familiar with using auto-tuning to adjust the value of optimization parameters such as number of threads or tile sizes – the same parameters that are exposed by the mid-level mapping tools. What was less familiar were tools that support expressing and selecting among code variants, or distinct implementations of the same computation that are functionally equivalent but may be better suited to particular architectures or input data sets. Today, libraries and application code that has been optimized by auto-tuning are not uncommon in DOE applications, but we were not aware of production applications that employ auto-tuning as part of the build process. Future research and deployment of auto-tuning must support code variants and integrate auto-tuning into the build process so that it can be used to achieve performance portability and apply to future architectures.

Dynamic Application Building and Execution

The techniques for performance portability in the previous subsection all benefit from the ability to specialize implementations for their unique execution context (including target architecture, application configuration and input data set). A general strategy for optimization specialized to execution context is to configure the application at run time once execution context is known. Most or all of the techniques our breakout session discussed in this category are not used in production HPC codes, but the question arose as to whether they might be more applicable in an exascale regime. For example, dynamic code generation and just-in-time compilation could be used to generate code tailored to a specific execution context. Auto-tuning could be used to assemble code variants and adjust optimization parameter values at run time. We also discussed how asynchronous many-task run-time abstractions could be supported to dynamically map code to hardware. These would only be profitable if the performance gain from specialization was greater than the overhead of assembling applications dynamically.

A concern with all these approaches is the potential for non-deterministic application behavior if code is never tested for a specific execution context (see subsection on performance analysis and correctness).

Correctness Debuggers

Many of the cycles used on the DOE computing centers are consumed by large community codes that may have hundreds of thousands to millions of lines of code. It is important to separate the problem of *validating* a numerical model (a mathematical formulation approximating a physical system), and *verifying* the implementation of that model. The programming environment tools address only the latter. Moreover, changes to simulation codes can introduce bugs that are difficult to detect, because rather than causing the program to crash or produce obviously unexpected behavior, they may have subtle problems in the numerics that are only visible with detailed analysis. Such errors could be the result of simple indexing errors (e.g., off-by-one array access), bugs in packing or unpacking numerical data to send between processors, insufficient precision in the numerical representation, and any number of other sources. Thus, while

traditional debugging and program analysis tools may help to detect such errors, they do not specifically address the issues that arise in debugging numerical code.

The group discussed both the current state of debugging and also expectations of what would be needed for exascale computing. The DOE centers use parallel debuggers such as Totalview and Alinea, and GDB, although these are often installed well after the system has been up and running applications. The general perception among the group was that many codes are still debugged using print statements. For large applications or frameworks used to support a class of applications, debugging support is often built-into the code. This may be used to ensure that higher level properties, such as conservation of forces, in addition to lower level correctness assertions. In addition, parts of the code used for checkpointing due to system errors are invaluable in debugging: an application that has crashed can be restarted from the last checkpoint and run with debugging support to examine the error. That said, checkpoint can itself be a debugging challenge, since data structure and type information may be lost in the checkpoint file, making it easy to introduce bugs in either the save or restart code, which go undetected for a long time. Tools are need to inject errors into running application to test various resilience approaches. Debugging on current petascale systems has its own set of challenges due to the size and complexity of the traces across all of the cores, and exascale applications will be even worse. Sometimes large parallel applications can be debugged by tracking only a single representative process, or by running at a more modest scale (e.g., one thousand threads) and the debugging states merged.

In addition to those current tools future exascale programs will have higher degrees of thread-level or task-level concurrency, which will require better support for race detections and the ability to visualize complex task graphs. Many of the applications in the broader HPC community still are written as pure message passing parallelism, where race conditions are much less likely to occur, so understanding how to identify and debug data races in multithreaded code will require training for the broader programming community. Even with current OpenMP code, the anecdotal evidence suggests that race conditions are not a major problem at the application level, although they are more frequent in the runtime systems themselves. Reproducibility of numerical results is particularly important in debugging and in some production applications. Reductions that are optimized to a particular hardware configuration or atomic updates on floating point values can both lead to irreproducible behavior. Programming models that can ensure reproducibility in reductions and for (if possible) atomic updates would be valuable; this is viewed a feature that should be available on demand, since it may be expensive and is not required for all usage scenarios. Finally, if the frequency of transient hardware errors rises, support for detecting these will be important; they can easily be confused with data races, for example. Other types of system errors, such as energy reports, will also require debugging support.

Overall, the problem of correctness (and performance) debugging is largely one of reverse engineering problem the mapping of the program to the machine, to understand what happened. As runtime system become more dynamic and more sophisticated, this reverse engineering problem becomes more difficult.

Performance and Energy Optimization Tools

While correctness is essential and performance somewhat option, the general opinion at the meeting was that Performance tools are used more than correctness tools. This is probably due to the subtlety of identifying performance errors. Tools of different types

(tracing vs. sampling, local vs. parallel) were discussed, including Valgrind, Vampire, TAU and HPCToolkit. There is probably a need for two different kinds of tools, or at least usage modes, to address both expert and novice use. For example, course grained statistical analysis may be run to identify egregious performance problems, possibly even by default on a system.

In general, performance tools for Exascale should allow for composability across application components (being able to performance tune separately or in whole) and the ability to specific measure the success criteria. For example, if power or energy utilization will be constrained then tools are needed to measure these in addition to running time or communication load. One can imagine scenarios in which these metrics are part of the allocation limits on a system, e.g., it may not be possible to run the entire machine in a "high power" model due to power caps, so a mix of high and low intensity jobs are required. There are also proposals to give the programmer some control of power and reliably, e.g., running longer or with higher error rates at lower power; tools will be needed to support any such interaction.

Storage Abstractions

One group touched briefly on the topic of storage abstractions, although a separate report has been done on storage systems and I/O, which includes requirements for programming environments. New models of storage will be important on Exascale systems, and they need to be integrated into future programming models and environments. These systems will contain new types of non-volatile memory, with much higher bandwidth than today's disk-based systems; these may be used for fault tolerance and for preserving scientific data across complex workflows. Overall, the storage architecture will be more complex that on today's systems, with levels of memory and storage that will be under user level software rather than hardware or system control. The storage systems may expand value of Exascale systems and accommodate new types of applications from data analytics and other environment.

Performance Models

Performance models are an important way to understand performance on current systems and to predict performance on future ones. While "performance" is most often about running time, there may be other performance metrics, such as energy use or failure tolerance, that may be modeled as well. There were three types of performance models discussed in this meeting: *benchmarks* to model application workload in the design of compilers and tools; *simulation tools* used to test and evaluate future architecture and microarchitecture designs; and *analytical models* to aid in estimating performance. Analytical models were generally viewed as useful for understanding performance bounds, e.g., computing running time limits on performance based on bandwidth requirements, but they can be inaccurate or very complex for detailed analysis of hardware features with discontinuous behavior. Understanding failures is an area where better modeling is needed to help in the design of programming environments; even a coarse model of which types of faults will be most common, whether they are uniformly distributed (which seems unlikely) and what their domain of failure is would all be valuable. There is work ongoing to collect databases of failure information from existing systems, which may help in some of these predictions, although Exascale systems are expected to have higher error rates due to technology effects and scale.

Productivity Models

Productivity of Exascale systems is critical to the success not just for Exascale applications, but to the impact of this technology on deployment smaller systems (e.g., Petascale racks) and for applications outside of the DOE space. The goal of the Exascale Computing Initiative is to develop useable and productive Exascale systems, not systems that merely achieve high performance (10^{18} floating point operations per second) but are too complex for most people to program or too unbalanced for most applications. That said, productivity has been notoriously difficult to measure or model, because it is highly dependent on the particular expertise of the community and is not easily quantifiable.

A number of productivity metrics were discussed:

- Technology adoption, i.e., use of Exascale hardware and software technology
- Utilization and demand for deployed Exascale systems
- Science and DOE mission results that come from the use of Exascale systems
- Number of programs that run on Exascale technology (use the architectures and programming models)
- Trickle down to petascale in a rack, to cloud data centers, and other domains

Further discussions will be required to identify the general goals for productivity and to define a metric by which it can be measured.

Session V – PM/E Research Questions

Chair: Thomas Sterling, Indiana University

Introduction

The Exascale Programming Model (XPM) is an abstraction of the interface between the user application program and the exascale computing system. It represents the conceptual strategy by which a user computation is structured, the classes of objects that comprise it, and the flow control methods that guide its concurrent execution. It is not a language or library but rather a family of languages and libraries that could be implemented consistent with its semantics.

Exascale computing will impose unprecedented challenges to future programming models due to extremes related to:

- Scalability
- Efficiency
- Architecture heterogeneity
- Resilience
- Energy demand
- Performance portability
- Productivity, and
- Generality

In addition to these challenging properties of exascale computing are two additional assertions to be addressed: the exploitation of runtime system software with possible hardware architecture support to exploit dynamic adaptive resource management and task scheduling and interoperability with other concurrent programs either on the same computing platform or potentially on other interconnected high performance computers at the same time.

This report identifies and describes a set of strategic research questions that must be answered prior to the establishment of a working and supportive programming model for exascale computing. Key issues are then expanded in detail to clarify and refine specific questions in preparation for planning future research programs.

Strategic Questions

An exascale programming model may prove to be significantly different from conventional practices in terms of its semantics and abstractions due to the fundamental issues of essential innovation driven by the distinguishing properties of future extreme computing demands. While there are many detailed questions to be resolved some depending on specific approaches and assumptions, overriding strategic questions critical to the success of exascale and relevant to all likely models to be pursued. These strategic questions are delineated and briefly described as follows:

1) ***Parallelism*** – what is the semantics of parallelism as a defining component of future exascale programming models that will yield billion-way concurrency required for trans-exaflops performance regime? Such new semantics must provide a unified programming interface that represents many levels of task

granularity and precedence constraints as well as parallel control synchronization.

2) ***Latency*** – how is the delays resulting from the propagation of access requests and responses to remote resources to be mitigated to avoid imposing a critical bottleneck? What is the role of future programming models in supporting the full system methodologies to minimizing and hiding latency effects? In particular, how will the programming model facilitate the amelioration of the efficiency degradation due to the variability and uncertainty of latency asynchrony?
3) ***Runtime Dynamic Execution*** – in what ways will the programming model exploit the opportunities of dynamic execution, possibly including adaptive methods, to improve efficiency and scalability through the application of runtime system software? What support does the runtime system have to provide? How will the programming model interoperate with and inform the runtime system? What is expected of advanced introspection methods and what can be their capabilities?
4) ***Name Space*** – What are the semantics of naming of referential objects that are first-class and the hierarchy of such name spaces that are represented in the programming model? How does this related to control and continuations?
5) ***Control State*** – what is the parallel control state, the semantics of parallel control, and the distribution and dynamic migration of continuations as represented within future exascale programming models?
6) ***Productivity*** – What attributes of the programming model facilitate the combination of ease of use and sustained performance?
7) ***Invariants*** – What aspects of the programming model will guarantee performance portability including invariants that are common to all target platforms?
8) ***Locality*** – How is the distributed aspect of the data and control flow is reflected within the programming model to benefit from locality to minimize latency and overhead and to exploit the distribution of computation for exploitation of concurrency for performance.
9) ***Interoperability*** – Programs must be able to work with each other without having to be explicitly designed to do so. How is this to be achieved intrinsically within the programming model?
10) ***Resilience*** – in support of active methods how will the programming model inform fault tolerance strategy and mechanisms to strengthen reliability?
11) ***Energy Efficiency*** – what aspects of the methods to achieve energy efficiency and power bounded computing will be supported through the programming model and the interfaces based on them? What information can the programming model provide to influence optimization with respect to energy?
12) ***I/O*** – how does the programming model reflect and represent external information flow of sufficient generality to adapt to diverse interface conditions?
13) ***Performance model*** – what is the objective function that will be used to drive development and optimization of programming models?

Detailed Questions

Data management is among the greatest of challenges in parallel computing and is likely to become more severe and of greater concern as we enter the exascale era. Both the requirements of the applications and the challenges to effective use of the emerging highly parallel architectures are driving improvements and increased sophistication in the creation, distribution, naming and dynamic placement of problem data. Efficiency of

access, exploitation of parallelism through distribution, reduction of latency effects through locality aggregation, and repartitioning for load balancing are all areas to which the programming model and resulting interfaces contribute in combination with input data set description, system specific loaders, compilation, and runtime systems. A number of detailed issues must be resolved including:

- Data abstraction model and representation schema
- Expression of the dimensionality of arrays
- The challenge and opportunity of cache oblivious programming for large problems such as the global array model
- Automatically tiling of arrays and how to express array dimensionality
- Consistent data models and bindings depending on level of memory hierarchy

Naming and Addressing are closely associated with data management but even more strongly define the nature of the programming model semantics. They differentiate shared memory models from distributed memory models, establish requirements and limitations of cache coherence, open opportunities for new kinds of memory hierarchy such as scratch pad memory and NVRAM layers, and determine how abstract data may be mapped to physical memory locations. For programming models there are open issues as to what degree the programmer is responsible for management and control of the way objects are referenced and the relationship between such referenceable objects and their physical distribution and access. Such issues include:

- Global naming
- How to express constraints on references to facilitate analysis?
- Distributed memory with a global name space?
- Remote versus local: write in one model – symmetry of semantics for accesses?
- Oblivious to shared memory versus distributed memory
- Translation of virtual to physical addresses both locally and globally
- PGAS

Parallel control flow in the broadest sense of a programming model determines the semantics of parallelism, task delineation and precedent constraints, synchronization, granularity and ultimately efficiency and scalability of parallel execution. The programming model distinguishes the degree to which the programmer explicitly defines the program parallelism or depends on automatic methods for exposing and exploiting computation concurrency to deliver performance as well as other aspects of operation such as latency hiding. Models are distinguished by their approach to parallel control. Among the issues to be considered are included:

- Unified or hierarchical model of parallelism
- Form and function threads (if they exist) and their interrelationships
- Functional (value oriented) or mutable side-effects
- Pre-emptive or non-preemptive
- Synchronization semantics like dataflow and futures
- Break for subset tasks
- Granularities versus overheads of control
- Support for conventional methods

The **abstract machine model** is the overall conceptual execution framework visualized by the parallel programmer through the programming model semantics and syntax. It establishes the general philosophy and the operational principles of the programming strategy that is to be shared across computer platforms on which programs are to be

run. It influences the relationship between the user interface, its environment, and the system software. It also determines performance tradeoffs of processing, memory access, and communication. Many questions about the abstract machine model have to be answered explicitly or implicitly in the formulation of the programming models. Among such issues are included:

- The essential aspects of abstract machine model that defines target for programming model. Can there be a portable abstract machine model to support a programming model across disparate platforms?
- Functional computing is one possible abstract machine model with a strict pass by value and single-assignment semantics. While probably too purest and restrictive, elements of this paradigm could contribute to the abstract machine model
- What is the hierarchical organization of the abstract machine model and what are the modules of which such hierarchy structure is comprised?
- Multiple layers of programming models from very high level (possibly DSL) to low level (C like but with parallelism). At each level it is possible to consider more than one programming model. What differentiates them and what roles do they serve?
- How does the abstract machine model definition contribute to the separation of concerns for the programming model?
- What aspects of the abstract machine model are implemented by the architecture, the runtime system, and the compiler? How is the information between levels and across domain spaces bridged?
- Can machine independent parallelism be built into compilers, not just sequences?
- Is it possible to avoid explicit communications management?
- How do we define the machine abstraction of a parallel task?

Resilience is a property of a system that establishes means and mechanisms for delivering reliability in the presence of faults. The programming model may treat the system as a failure free platform, crash when encountering a fault, or contribute to response actions upon an interrupt. Key issues include:

- What is the right resilience model?
- How can programming models capture resilience information?
- What is the balance of responsibilities between the programming model, the runtime control system including operating system, and the hardware architecture?

Debugging is a critical part of the programming process and the programming model has to incorporate interface hooks that enable to programmer to determine program state and establish correctness conditions. Debugging of parallel programs can be much more challenging than for sequential programs due to the multiplicity of concurrent execution paths possible to producing a solution. Additional problems like race conditions and the management of synchronization variables complicate the programming process. Issues include:

- Fault model
- Methods of error detection
- Kinds of information provided by the application program for detection and recovery
- Runtime system support of programmer debugging

A **performance model** is required to guide the programmer in determining choice of program structure, flow control, distribution, and scheduling alternatives to achieve optimal performance through efficiency and scalability. This will lead to means of performance optimization and tuning as well as cross-platform performance portability. But what that performance model is and how it is employed through the programming model and environment of tools is far from determined for exascale computing. Research questions include:

- What is the performance model that will drive the development of application programs?
- How does the programming model relate to the performance model?
- How will the runtime system combine the performance model and information from the programming model to manage task scheduling and resource management for best performance? Included is automatic aggregation and locality management.
- Instrumentation for measurement of program behavior is guided by expectation of performance sensitivities and designed within the context of a performance model

Hardware architecture is evolving as it traverses the pan-petaflops performance regime during this decade. Current trends emphasize multi/many-core and GPU accelerator processor architectures within symmetric multiprocessor shared memory nodes integrated in large-scale parallel systems through high bandwidth system area networks. But new processor core designs are being considered to minimize energy, enhance reliability, reduce overheads, and manage latencies for greater efficiency and scalability than feasible with conventional designs. Mobile and embedded computer architectures such as ARM-64 are being explored. While vendors suggest their roadmaps are set, in truth, the final exascale architectures are yet to be defined. Programming models have followed computer architecture to facilitate their application. But without a firm definition of those architectures in the next decade, questions remain that must be resolved through research in order to prepare future programming models. Such questions include but are not limited to:

- Name space and global address space and hierarchies reflected by architecture
- Thread management architecture support for control and switching
- Tradeoff space for load balancing versus data movement costs
- Native semantics of parallel flow control supported by ISA
- Bandwidths, latencies, and overheads of architecture structure

Application codes: programming models are representative schema to capture the knowledge, requirements, data structures, and control patterns inherent to the definition of an application. Exascale applications in many cases will impose unprecedented needs on the target computational platforms and the programming models and interfaces must reflect these. A number of interrelated issues and questions concerning future exascale programming models will have to be resolved to achieve the means of crafting exascale application codes. These include, but are not limited to:

- How do STEM codes differ from conventional commercial codes and what can be learned from general commercial programming practices?
- In what ways will exascale applications differ from past generations of HPC codes?

- How can programming models capture user intent with minimal syntactical burden?
- Apps and algorithms are more restrictive than the language; how can the idioms of particular problems be exploited to optimize performance?
- Can general classes of algorithms be converted to generic codes?
- How do we do interoperability in the large? What are the ways of composing applications? Do DSLs offer a possible approach?

Domain Specific Languages (DSL) are a promising strategy to achieving early adoption of exascale computing and addressing the challenges of user productivity. But the needed specification does not make this a general solution to the programming models problem. If DSLs can serve as part of the solution, some questions have to be addressed:

- How transferrable across various domains is a DSL useful and across a wide set of applications
- How to accelerate the creation of DSLs
- How do we do interoperability in the large? What are the ways of composing applications?
- Will DSLs solver enough problems?
- What are the areas we know we can push it and what does the DSL need to look like? What are missing pieces?

Session VI – Research Questions for Programming Environments

Chair: Martin Schulz, LLNL

Charge

In this session the participants were asked to identify the key research questions in the area of programming environments. In particular, they were asked to answer the following questions:

- What are the key research questions in programming environments that must be answered by the research community before widespread development and adoption?
- Is there an order and preferred method for pursuing these?

Structure

The participants decided to follow the list of elements needed for programming environments identified in the preceding breakout session (see page 18) and group them into a smaller number of topics to be discussed:

- Code Generation/Compilation
- Debugging/Correctness Tools
- Performance Analysis/Optimization Tools
- Resilience Tools
- Storage Management
- Code/Software Management and Workflows
- Crosscuts / Overarching Infrastructure

The topics of "Libraries" and "Analysis and Visualization Tools" were excluded, since they could not be covered adequately and would have warranted a separate workshop. Further, the topic of "Testbeds" was excluded, since testbeds don't pose any research questions by themselves, yet all participants agreed that they would be paramount to have as one of the key drivers for research in this area.

The rest of the session was then spent in identifying the research questions for these seven identified topic areas. In the end, the group did not find a preferred or priority order for these research questions. Answers to all of them are necessary for a successful exascale software stack. However, all of them address both immediate and longer term needs and a prioritization within each topic area should be done to establish this differentiation. Additionally, the group noted that we need to address these questions on two tracks concurrently: how can we improve support for existing codes and models in the near future (evolutionary), while at the same time have the support for new models ready when they are needed (revolutionary)?

Research Questions

Code Generation and Compilation

The main area of discussion for this topic was research around "smarter" compilers. The main example was the need for continuously optimizing compilation and included topics like background (re)compilation based on performance feedback (taking it a step further than existing feedback based compilation) as well as to dynamically change code properties. The group assumed that this was likely only feasible for on-node techniques, although more global approaches could be of interest as well. The group also noted that

this could cause new research questions in debugging, as such adaptive and continuously changing systems make pinpointing bugs and other problems more difficult. Additionally, the group identified the need for faster and/or more parallel compilers as a research questions, especially in the light of new language extensions, heavier use of complex templates, etc.

Debugging/Correctness Tools

The group divided the area of debugging and correctness tools further into three subcategories: static tools, dynamic tools and approaches that require both static and dynamic techniques. The research questions on the static tool side focused on how to detect bugs and errors before codes execute, while the dynamic side centered around how to deal with non-determinism (How difficult and costly is it to enforce determinism and can we establish dynamic "off" switches for non-determinism?), how to deal with scale (how to automate debugging to make it easier in low-availability scenarios, how to scale interactive tools – both from the interaction and the data collection point of view), when do we need interactivity, and how to establish testing frameworks at scale? Further, the group identified the ability to reproduce errors at lower scale as a major research question.

On the combined static/dynamic side, the group identified questions around techniques to identify scaling bugs before executing on large systems (not just for the applications themselves, but for any level of the software stack) as well as a need for developing techniques to help debug applications written in new task models, as developed in the DOE/ASCR X-Stack efforts. An additional topic of discussion was how to enable visual debugging, i.e., the use of visualization tools for debugging large scale problems.

Performance Analysis/Optimization Tools

Similarly to the area of debugging tools, the group split the area of performance tools into static, dynamic and hybrid static/dynamic subtopics. On both the static and dynamic side, the group identified questions on how to get more information from compilers and runtime systems respectively and how to turn this information into actionable insight as key questions. Additionally, on the dynamic side, questions identified included how to deal with performance information in light of non-determinism, variable execution and highly adaptive systems as well as how to include new metrics, such as energy, power, or reliability.

The hybrid static/dynamic area led to a lengthy discussion and the identification of the following research questions:

- How to create new representations for visual performance analysis?
- Which tools are needed for detecting memory access patterns?
    - This was also seen as a good candidate for new visualization techniques
- How to develop tools for remapping memory and re-tuning?
    - Static (before), Dynamic (for next run), Dynamic (on the fly)
    - Separate data layout from computation (late mapping)
- How to exploit hardware support for custom memory remapping?
    - Requires detection and analysis to extract mappings (example: 2D memory regions)
    - What are we willing to "pay" for it? How much change in hardware can we push for by demonstrating its benefits?
- How do we develop tools to evaluate precision requirements?
- How to improve performance modeling, especially in light of adaptivity and new metrics?

- How can we create tools that provide us with “goodness” metrics for matching codes to architectures?
- How to enable and ultimately exploit continuous data collection and analytics to study long term trends?

Resilience Tools

The group discussed a wide range of questions in resilience, particularly referring to programming environments. The key ones were:

- How to avoid global recovery?
  - A single failure shouldn’t break the complete system (live recovery/migration and how to do this efficiently)
    - What abstractions are good for individual applications?
    - What abstractions are good for coupled workflows?
  - What characteristics do we want in a resilient model?
    - Incl. MPI – how to hotswap or shrink?
- What model is useful in which applications?
  - How can we automate the geometry mapping so shrinking resources can be used efficiently?
- We need global checkpoints at some point (limited job time, catastrophic machine failures,) - how do we optimize them?
- How transparent should resilience be?
  - How far do/can/should/must we burden the programmer?
  - What to expose to application level?
  - How do new (tasking) models influence this?

Storage management

The group saw the integration of storage management into complete workflows (arbitrary sources and sinks, coupled simulations, integration/reaction to decisions of the resource managers, etc.) as a key research question that is currently unanswered. Related to that, the group raised the question on how the existence of on-node and/or global NVRAM changes this picture and how to establish the necessary APIs and usage abstractions for these new storage options. Further, the question was raised what the right I/O storage abstractions are and if files are still the right answer.

Code/software management and workflows

The last topic discussed in storage management also continued into this topic area: what are the right abstractions for I/O and storage and their flow? How can we understand the flow through the I/O system, tag/index data based on its properties, and establish provenance information? As a more general question, how can we manage large volumes of data?

The topic area also led to a discussion on tools to help the code development process itself: how can we develop better tools to understand code complexity? While good commercial solutions exist in this area, they are not HPC focused, create too many false positives and do not scale. How can we extend these solutions to match the HPC needs? Further, what tools do we need to help map codes to the architectures (incl. accelerators) that best match their properties?

Finally, the group posed research questions on how to integrate continuous performance gathering and the associated analytics into the daily workflows, including performance regression suites.

Crosscuts / Overarching infrastructure

Additionally, the group identified a wide range crosscutting and overarching research questions:

- How to develop new techniques for visual correctness and performance debugging and how to exploit the intersection with the DataVis and InfoVis research communities? How to address scaling limitations and how can we use such techniques to visualize the evolution of algorithms and/or data structures (at various levels, including the physics level, DAGs, etc.)? How can we integrate this into existing visualization and analytics workflows?
- How do we interface programming environment tools with runtime systems? How can we efficiently get performance and debug data? How can we identify and control knobs to change performance properties? How can we tell the system, especially in the light of adaptive systems based on learning components and models that we are doing well right now?
- How can we efficiently and couple tools and application components while staying modular? How can we match runtimes, enable computational and analysis steering, and interact and coordinate with resource management?
- How can we create production quality, scalable infrastructures for visualization, data analysis, debugging and performance tools? How can we share infrastructure between those components and eventually make it a system service? How do we get away from hacky scripts, avoid data management through shared files?

Agenda:

An abbreviated version of the agenda is below:

	Topic	Speaker
Monday, March 9, 2015		
8:15am – 8:45am	*Welcome and Introduction –DOE Perspective*	*William Harrod* *Director of Research, Advanced Scientific Computing Research (ASCR)* *Thuc Hoang* *Program Manager, Advanced Simulation and Computing (ASC)*
8:45am – 9:10am	*Implications of Hardware Trends on Programming Models*	*John Shalf* *Lawrence Berkeley National Laboratory (LBNL)*
9:10am – 9:40am	*ASC Application requirements for PM/E(S) in exascale era*	*Rob Neely* *Lawrence Livermore National Laboratory (LLNL)*
9:40am – 10:00am	*Fusion applications requirements for PM/E(s) in the exascale era*	*Stephane Ethier* *Princeton Plasma Physics Laboratory (PPPL)*
10:20am – 10:40am	*Genomic applications requirements for PM/E(s) in the exascale era*	*Rob Egan* *LBNL and Joint Genome Institute (JGI)*
10:40am – 11:00am	*Climate application requirements for programming models and environments in the exascale era*	*Rob Jacob* *Argonne National Laboratory (ANL)*
11:00am – 11:20am	*Library requirements for programming models and environments in the exascale era*	*Lori Diachin* *LLNL*
11:40am – 12pm	*Application requirements for PM/E.*	*Jim Belak* *LLNL*
12:40pm – 2:40pm	*Parallel Session I: Application Requirements for Exascale PM/E(s)*	*Chair: Rob Hoekstra* *Sandia National Laboratories (SNL)*
	Parallel Session II: Methods for Evaluating PM/E(s)	*Chair: Armando Solar-Lezama* *Massachusetts Institute of Technology (MIT)*
2:40pm – 3:00pm	*ExMatEx Co-Design PM/E Efforts*	*Jim Belak* *LLNL*
3:20pm – 3:40pm	*Exact Co-Design PM/PE efforts*	*Pat McCormick* *Los Alamos National Laboratory(LANL)*
3:40pm – 4:00pm	*Kokkos*	*Carter Edwards* *SNL*
4:00pm – 4:30pm	*The Future of MPI*	*Rajeev Thakur* *ANL*

4:30pm – 5:00pm	*The Future of OpenMP*	*Barbara Chapman* *University of Houston*
Tuesday, March 10, 2015		
8:20am – 8:30am	*Workshop Goals and Objectives*	*Sonia Sachs*
8:30am – 8:45am	*Parallel Session I report*	*Rob Hoekstra* *SNL*
8:45am – 9:00am	*Parallel Session II report*	*Armando Solar-Lezama* *MIT*
9:00am – 9:40am	*X-Stack D-TEC Programming Environment*	*Dan Quinlan* *LLNL*
9:40am – 10:20am	*X-Stack Traleika Programming Environment*	*Wilf Pinfold* *Intel*
10:40am – 11:20am	*X-Stack DEGAS Programming Environment*	*Kathy Yelick* *LBNL*
11:20am – 11:40am	*Tilearchy*	*Ben Bergen* *LANL*
11:40am – 12:00pm	*RAJA*	*Rich Hornung* *LLNL*
12:40pm – 1:00pm	*ASCR Programming Environments Summit Report Summary*	*Armando Solar-Lezama* *MIT*
1:00pm – 3:15pm	*Parallel Session III: Characteristics of PM/PE(s) (set 1)*	*Chair: Dave Richards* *LLNL*
	Parallel Session IV: Characteristics of PM/PE(s)(set 2)	*Chair: John Shalf* *LBNL*
3:45pm – 4:30pm	*Continue Parallel Session III: Characteristics of PM/PE(s) (set 1)*	*Chair: Mary Hall* *University of Utah*
	Continue Parallel Session IV: Characteristics of PM/PE(s) (set 2)	*Chair: Kathy Yelick* *LBNL*
4:30pm – 4:45pm	*Report Back of Session III*	*Dave Richards* *LLNL*
4:45pm – 5:00pm	*Report Back of Session IV*	*John Shalf* *LBNL*
Wed, March 11, 2015		
8:15am – 8:30am	*Report Back of second Session III*	*Mary Hall* *University of Utah*
8:30am – 8:45am	*Report Back of second Session IV*	*Kathy Yelick* *LBNL*

8:45am – 10:15am	*Parallel Session V Breakout: Research questions for programming models*	*Chair: Thomas Sterling* *Indiana University*
	Parallel Session VI Breakout: Research questions for programming environments	*Chair: Martin Schulz* *LLNL*
10:30am – 10:45am	*Report Back of Session V*	*Thomas Sterling* *Indiana University*
10:45am – 11:00 am	*Report Back of Session VI*	*Martin Schulz* *LLNL*
11:00am – 11:30am	*Summary of Workshop*	*Chairs: Pete Beckman, ANL* *and Rob Neely, LLNL*