Workshop on

Software Development Tools for Petascale Computing

http://www.csm.ornl.gov/workshops/Petascale07/

1-2 August 2007
Washington, DC

Organizing Committee: Fred Johnson, DOE Office of Science
Thuc Hoang, National Nuclear Security Administration
Bronis de Supinski, Lawrence Livermore National Laboratory
Bart Miller, University of Wisconsin, Madison
Jeffrey Vetter, Oak Ridge National Laboratory and Georgia Tech
Mary Zosel, Lawrence Livermore National Laboratory

Working Group Chairs: Susan Coghlan, Argonne National Laboratory
Al Geist, Oak Ridge National Laboratory
Jeffrey Hollingsworth, University of Maryland
Curtis Janssen, Sandia National Laboratories
Bernd Mohr, Forschungszentrum Jülich
Rod Oldehoeft, Krell Institute
Craig Rasmussen, Los Alamos National Laboratory
Daniel Reed, Renaissance Computing Institute

Editor: Jeffrey Vetter, Oak Ridge National Laboratory and Georgia Tech

EXECUTIVE SUMMARY

Petascale computing systems will soon be available to the DOE science community. Recent studies in the productivity of HPC platforms point to better software environments as a key enabler to science on these systems. To prepare for the deployment and productive use of these petascale platforms, the DOE science and general HPC community must have the software development tools, such as performance analyzers and debuggers that meet application requirements for scalability, functionality, reliability, and ease of use. In this report, we identify and prioritize the research opportunities in the area of software development tools for high performance computing. To facilitate this effort, DOE hosted a group of 55 leading international experts in this area at the Software Development Tools for PetaScale Computing (SDTPC) Workshop, which was held in Washington, D.C. on August 1 and 2, 2007.

Software development tools serve as an important interface between the application teams and the target HPC architectures. Broadly speaking, these roles can be decomposed into three categories: performance tools, correctness tools, and development environments. Accordingly, this SDTPC report has four technical thrusts: performance tools, correctness tools, development environment infrastructures, and scalable tool infrastructures. The last thrust primarily targets tool developers per se, rather than end users. Finally, this report identifies non-technical strategic challenges that impact most tool development. The organizing committee emphasizes that many critical areas are outside the scope of this charter; these important areas include system software, compilers, and I/O.

Overall, the DOE platform roadmap shows that platforms are growing more complex and scaling to hundreds of thousands of processors. The increase in architectural complexity is rooted in multimode parallelism and heterogeneity. Taken together, these trends create a critical need for tools that can help application teams bridge these complexity and scalability challenges.

Meanwhile, applications are becoming much more multifaceted as teams include a variety of languages, libraries, programming models, data structures, and algorithms in a single application. In fact, application teams are listing scalable tools for debugging, memory correctness, thread correctness, and multimode performance analysis as key factors in their productivity.

In performance tools, emerging heterogeneous, hierarchical architectures will render static, manual approaches to diagnosing performance problems insufficient. Rather, online measurement and adaptivity in performance monitoring are becoming important techniques to dynamically optimize choices of application performance instrumentation and analysis at large system scale and complexity. Furthermore, the architectures and system software must make the necessary performance and reliability information available to these tools so that they can perform root-cause analysis with greater accuracy.

In correctness tools, as in performance tools, the availability of scalable tools is particularly critical. Application teams specifically requested lightweight tools to diagnose memory, threading, and message passing errors that are easy to use and scale from their desktop system to their petaflop platform.

Both performance and correctness tools rely on scalable infrastructures to provide tool communication, data management, binary manipulation of application executables, execution management for batch schedulers and operating systems, and a variety of other capabilities. Tool infrastructures must be efficient, modular, fault tolerant, and flexible. In addition, these infrastructures can speed the development of and reduce the cost of performance and correctness tools by providing standard, portable mechanisms for common capabilities.

In a similar manner, application teams need infrastructures for development environments, where this area includes tools for managing application builds and configurations, mixed language support, dynamic linking, program configurations, remote access, compiler infrastructures for application-specific analysis and transformations, and integrated development environments.

Finally, an array of crosscutting, non-technical issues that can accelerate or inhibit the success of software development tools must be addressed. These challenges extend far beyond solving technical issues into areas that require strategic coordination among industry, government, and academia. Such issues include a well-defined mechanism for sustaining and hardening successful research tools; engagement with application teams, particularly for tool training; access to system testbeds and details about the system architecture and software; modular design and implementation of tool components that could be leveraged across many tools; and, support for international collaboration.

TABLE OF CONTENTS

TABLE OF TABLES

1 INTRODUCTION

Petascale computing systems will soon be available to the DOE science community. To prepare for the deployment and productive use of these platforms, the DOE science community must have the software development tools, such as performance analyzers and debuggers, surpass application requirements for scalability, functionality, reliability, and ease of use. In this whitepaper, we identify and prioritize the research opportunities in the area of software development tools for high performance computing.

Recent studies in the productivity of HPC platforms point to better software environments as a key enabler to science on these platforms. Not surprisingly, application scientists consistently indicate that they need software development tools that can function not only at small scales for initial development but also at the size of the largest runs (e.g., software that scales from the desktop to the petaflop). Clearly, understanding performance and correctness problems of applications requires running, analyzing, and drawing insight into these issues at the largest scale.

Although the architecture of petascale systems is yet to be determined, the largest existing systems can help guide our expectations for those machines. The largest of these systems, the IBM BlueGene/L machine at Lawrence Livermore National Laboratory (LLNL), has 131,072 processors while the second largest system, the Cray XT4/XT3 system at Oak Ridge National Laboratory (ORNL) has 23,016. In fact, the five largest existing systems have over twenty thousand processors each. Further, current technology trends indicate that processor counts will continue to increase as we move towards petascale systems and beyond.

Consequently, research for software development tools for petascale systems must address a number of dimensions. First, it must include elements that directly address extremely large task and thread counts. Such a strategy is likely to use mechanisms that reduce the number of tasks or threads that must be monitored. Second, less clear but equally daunting, is the fact that several planned systems will be composed of heterogeneous computing devices. Performance and correctness tools for these systems are very immature. Third, requires a scalable and modular infrastructure that allows rapid creation of new tools that respond to the unique needs that may arise as petascale systems evolve. Further, successful tools research must enable productive use of systems that are by definition unique. Thus, it must provide the full range of traditional software development tools, from debuggers and other code correctness tools such as memory analyzers, performance analysis tools and tools that support the requirements of building applications that rely on a diverse and rapidly changing set of support libraries.

1.1 Workshop

The remainder of this paper details the findings generated at the Software Development Tools for PetaScale Computing (SDTPC) Workshop held in Washington, D.C. on August 1 and 2, 2007. During the workshop, attendees participated in two separate sessions of two concurrent working groups. On the first day, the Performance Tools and Correctness Tools working groups met. On the second day, the Scalable Infrastructures and Development Environment Infrastructures assembled. The organizing committee emphasized that many critical areas were outside the scope of this charter; these important areas include system software, compilers, and I/O.

The chairs of each working group were chartered with delivering a prioritized list of challenges for their specific topics as rated by the attendees. Although the scheme for rating priorities varied across working groups, in general, they ranked each specific challenge on two dimensions: likelihood and impact. Likelihood is the probability that the technology will not be available in the given timeframe. That is, given current trends, do you expect the target technology to be ready for petascale computing? Impact is

the severity of damage this item will inflict on the goals of petascale computing if the challenge is not addressed. Said differently, how important will the lack of a solution for this challenge be for applications targeting petascale systems. Taken together, these two dimensions were used to rate each technical item, and generate a specific rank.

1.2 Report Outline

We begin in the following section with a short summary of expected petascale hardware directions. We then review the requirements of software development tools for petascale systems as presented by application scientists in Section 3. The next sections examine four technical thrusts: performance tools; debugging and correctness tools; scalable infrastructures; and code development environment infrastructures. For each technical thrust, we review the current state of the art and the requirements for petascale systems, after which, we detail the challenges and our recommendations for addressing these requirements. Finally, we summarize the overall recommendations that emerged from the workshop in Section 8.

2 DOE PLATFORM ROADMAP

DOE is pursuing an aggressive path to capability computing with its leadership class systems. Table 1 lists current and some planned leadership class systems. These systems are notable in their size, from tens of thousands to hundreds of thousand of processors, running a variety of operating systems, runtimes, schedulers, and processor architectures.

Table 1: Present and Future DOE Platforms[1].

System	Date	Site	Peak TFLOPS	Processor	Cores per Chip	Cores
Cray Red Storm	2004	SNL	124	AMD Opteron	2	~25,000
IBM BlueGene/L	2005	LLNL	360	PowerPC 440	2	~131,000
IBM Purple	2005	LLNL	93	Power5	2	~10,000
IBM BlueGene/L	2005	ANL	6	PowerPC 440	2	~2,000
Cray XT3/4	2007	ORNL	119	AMD Opteron	2	~23,000
Cray XT4	2007	NERSC	~100	AMD Opteron	2	~20,000
IBM BlueGene/P	2007	ANL	~100	PowerPC 450	4	~32,000
Cray XT4+	2007	ORNL	~250	AMD Opteron	4	~24,000
Cray XT4+	2008	NERSC	~300	AMD Opteron	4	~40,000
IBM BlueGene/P	2008	ANL	250-500	PowerPC 450	4	>100,000
Cray Baker	2008	ORNL	~1,000	AMD Opteron	4	~96,000
IBM RoadRunner	2008	LANL	~1,700	AMD Opteron IBM Cell	4 9	~200,000
Sequoia Dawn RFP	2008	LLNL	~500	TBD	TBD	TBD

From this existing roadmap, looking just two years hence, we can make several observations. While the complexity and scale of these systems are major challenges, the trends in these new systems make the challenges even more complex. Some of these trends include the following.

[1] This table provides illustrative information about the existing and future DOE platforms. Future platforms are subject to budget approval, system and component availability, and other changes.

1. **Increased scale**: As the number of cores per chip increase, the number of total cores per system will increase. This increase in scale of hardware will challenge every aspect of software design, including the application, the programming environment, the libraries, the operating systems, the job schedulers, and the storage systems. At the large scale planned for future systems, the failure rate for system components used by the tools as well as the applications they are used on is going to increase significantly. Thus, innovative fault tolerance mechanisms, both in the applications and the software development tools will be particularly needed.
2. **Multi-mode parallelism**: With single and dual core processors, users could almost ignore the combination of shared-memory and message passing in the same system. As processors grow to four cores and more, the programming models, tools, system software, and applications are going to have to incorporate support and specific optimizations for these cores explicitly.
3. **Reduced memory per core**: A side effect of the multi-core trend is that there is a proportional reduction in the amount of RAM per core (holding platform memory size constant) that will be available to the application. This reduced memory size could add significant complexity to software design and performance optimization.
4. **Heterogeneity**: Several new designs are starting to incorporate multiple processor devices in the same system. For instance, the IBM Roadrunner, currently under development for LANL, will combine a traditional AMD node design with several Cell processors per node. The complexity of the Cell processor, with its radical departure from the conventional design of general-purpose processors, presents significant programming challenges to the application community. In other examples, the Cray Cascade system for DARPA will introduce a variety of processor types, from scalar to multi-core to vector, in the same system.

Taken together, the combination of the existing challenge of scale with these emerging challenges, such as heterogeneity, will require increased activity in the area of development tools if the DOE science community plans to benefit from the petascale platforms.

3 APPLICATION REQUIREMENTS FOR SOFTWARE DEVELOPMENT TOOLS

A rational process for planning future directions for software development tools for petascale computing must reflect both the anticipated directions for petascale architectures and the requirements of applications that will run on them. For this reason, the SDTPC Workshop included presentations from three application team leads: Brian Pudliner from Lawrence Livermore National Laboratory; Robert J. Harrison from Oak Ridge National Laboratory; and John T. Daly from Los Alamos National Laboratory. These user talks provided direct requirements in three of the four technical thrusts addressed by the workshop's working groups -- requirements were implied for scalable infrastructures but the users were concerned with functionality and not the underlying mechanism to provide it at scale. In this section we summarize and synthesize the user requirements.

The user requirements are heavily shaped by the length of the life cycle of the applications. All three talks discussed applications that have both long development cycles and long periods during which the application is in "production." An important aspect of this life cycle is that "code is always in development -- even 'production' code." Thus, the users require assurances of stable support for a programming model, including the development tools that enable its use. Further, "new" applications are almost never entirely new -- they almost always take some existing code base to provide key underlying physics or mathematics functionality from an existing application. As a result, users are not open to tools that only target "new" applications or require significant changes to the established workflow of the application team.

The tool support required can vary with the life cycle stage. Initial code developers need full featured debuggers and performance analysis tools and are willing to work with tools with relatively high overheads, such as some memory correctness tools. Similar functionality is also needed for code being maintained. In addition, support for version tracking, code coverage and regression testing (both correctness and performance) are useful at this stage. Supporting code ready to run at large scales requires yet different tools. Lightweight debugging functionality is essential at these scales, as are low overhead mechanisms for performance profiling and analysis. Codes in production use stress aspects of working with scripts or other mechanisms to interact with applications and large scale systems. In particular, many scientific applications increasingly rely on Python to provide a framework for steering application runs. Another key aspect of this life cycle stage that will become increasingly important with petascale systems is that GUI-based tools must provide a mechanism for fast **and** secure remote operation. Finally, tools to support fault tolerance, with a focus on data integrity, are expected to become even more important during this life cycle stage as the number of cores will increase dramatically in petascale architectures.

Other aspects of the code development process also can shape tool requirements. Developers use tools to make their code correct and performant. However, different developers use tools differently, whether due to personal preferences, time constraints, or level of expertise. Requirements for tool architecture, interoperation, and training should take into account usage aspects if tools are to be accepted and applied effectively in general development practice. For instance, tool refactoring to decompose specific functionality into individual components could allow for targeted use of a tool component with a smaller learning curve. On the other hand, a tool framework that integrates functionality might better support automation of multi-step operations. In either case, it is important to emphasize training to raise the level of tool competency and expected return on tool investment.

One common aspect of the code development paradigm for scientific applications merits specific mention. Applications often rely on many third party libraries. For a variety of reasons, the source of the libraries is generally integrated into the application build infrastructure. However, this integration cannot be complete in order to allow updates of that source and support for using third party libraries in this way is inadequate. As a result, it can represent a significant cost to developers. Further, they anticipate this cost will increase in systems that require cross-compilation, as are commonly proposed for petascale architectures. Simply put, this cost must be reduced, particularly in light of programs like SciDAC that are creating significant support software in the form of libraries for a variety of common needs including solvers and meshing packages.

The increasing prevalence of coupled multi-disciplinary codes has combined with the long life cycle of scientific applications and the use of third party libraries to make codes larger and more complex. As a result, tools must handle larger executables. Tool developers are already seeing demand for tools to handle codes of several hundred mega-bytes to giga-bytes of executables. In addition, the rise of component based programming is resulting in applications that have hundreds if not thousands of shared libraries.

The dominant programming model of DOE Science applications is currently MPI although Harrison noted that computational chemistry codes make heavy use of one-sided communication mechanisms other than that available in MPI-2. This leads to a clear requirement for tools that facilitate the use of MPI, both in performance and correctness, as well as the ability to accommodate alternative communication mechanisms. In addition, application programmers anticipate needing to use multi-level parallelism for expected petascale architectures. These models would include the existing MPI or one-sided parallelization as one level. Harrison mentioned coarse grain task level parallelism (similar to the component model in the Community Climate System Model), as another possible level. More importantly, most application teams expect that efficient exploitation of fine grain parallelism through shared memory threading will be essential on future architectures, petascale and beyond.

The expected addition of thread level parallelism drives many user requirements. Users strongly desire portable tools that provide automatic analysis of the correctness of threaded applications (e.g., freedom from race conditions). The robustness of the support for threads in traditional debuggers and performance analysis tools also concerns them.

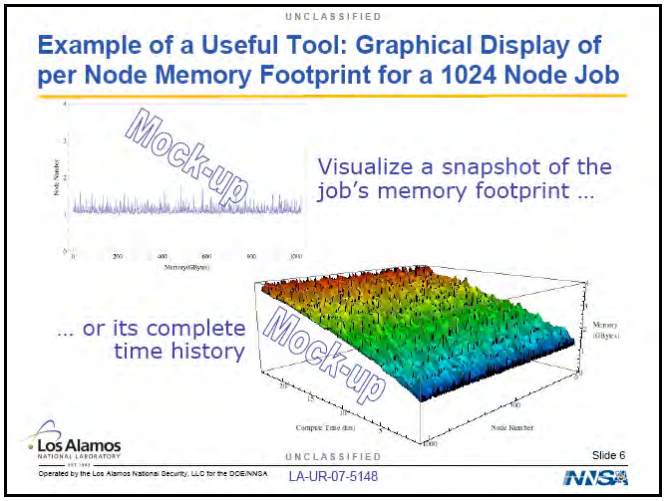

Figure 1: Visualizing Memory Usage over Time.

In general, users do not perceive existing debuggers as scaling beyond about 1024 MPI tasks when applied to real applications. Further work to improve their scaling is needed. However, application developers particularly want lightweight debugging tools that scale to the full size of the platform and provide information that assists in narrowing the problem. Particularly of interest are tools that identify commonalities between MPI tasks or that limit issues to specific aspects of the application or even identify when the issue is due to some underlying hardware problem. These tools can allow effective use of traditional debuggers on subsets of the large job. In addition, users often find print statements are an

effective debugging mechanism although it is easy to see how it can break down at very large task counts. Thus, some mechanism to support this paradigm at large scale is desirable. Tools that monitor application progress and determine if a job is hung or otherwise failing could save significant wasted resources. Pudliner noted that a scalable Python debugger would be nice but was not a priority since the existing tools are adequate in this respect. Likewise, tools must work with applications that use common OS features, such as dynamic libraries.

All of the users noted needs for support for many aspects of memory usage. Since we anticipate reduced main memory per core in petascale architectures, developers need tools that help them understand the scaling behavior of memory allocations and usage. Tools to detect correct memory semantics employed remain important. However, new tools that monitor how often memory regions are touched would support optimizations that simply recompute quantities rather than using significant memory to store them. Even tools that monitor how much memory is being used in a parallel job over time would be useful, as shown in Figure 1. To be applied on petascale application runs, all of these tools must have little overhead, a criteria that many existing memory correctness tools fail to meet.

Several performance issues are anticipated to become of increasing importance. Perhaps at the top of the list is load balancing. Tools are needed to detect load balance problems and to assist application dynamic load balancing algorithms. Other significant performance concerns include mapping application communication topologies to scalable network architectures such as tori.

Pudliner concluded his talk with a list of priorities that he anticipates for software development tools for petascale computing, roughly in priority order:

1. A means of debugging at scale;
2. Memory debugging;
3. Performance analysis tools: serial; parallel (at scale); and thread;
4. Memory characterization tool;
5. Thread correctness tool if necessary;
6. A means of characterizing/optimizing for topology if necessary;

He stated that the top four were close in priority. The sixth was further off in that it depends on petascale architecture directions. However, the major platform directions discussed in the previous section all employ networks that reward communication locality so it is likely to be an important issue. Both Harrison and Daly indicated general concurrence with the points that Pudliner raised.

4 TECHNICAL THRUST – PERFORMANCE TOOLS

Chairs: **Bernd Mohr (Forschungszentrum Jülich)**
Daniel Reed (Renaissance Computing Institute)

The performance tools working group was charged to explore the following topics related to software support and infrastructure for performance analysis, related to future petascale systems:

1. Analysis, modeling, and optimization;
2. Interactive and automatic approaches;
3. Data management and instrumentation;

4. Hardware and operating system support;
5. Visualization and presentation.

This group, like the other working groups, was asked to assess the current state of the art in each of these areas, identify the needs and requirements for performance tools at petascale and present a set of focused findings and recommendations. The latter were to be ordered based on priority and challenge type (technical, finding, policy or training) and the impact (high, medium or low) and risk associated with the challenge (high, medium or low).

The group began the discussion with the sobering realization that the software tool developer community is small, though tightly knit, with members of academia, industry, national laboratories and government. Relatively few academic groups conduct research on scalable performance analysis and, in consequence, the pool of new researchers, particularly those who can obtain a national security clearance and work directly with classified codes, is very small. However, this has led to deep collaborations not only across the academic, laboratory and industry communities but also between groups in the U.S. and in Europe. Collectively, the workshop attendees have long and deep working relationships and shared experiences that quickly focused the discussion on old and new challenges.

The group also acknowledged that it has been debating these challenges and making community policy recommendations for over twenty years. This conundrum is a consequence of the rapidly changing nature of high-performance computing and performance tool dependence on a long chain of hardware and software attributes. Powerful, effective performance tools depend on hardware, operating system, runtime library and compiler infrastructure. Because all of these elements are themselves in flux, particularly early in a product lifecycle, robust performance tools are rarely available when they may most be needed – for early use on new systems. Finally, performance tools are often a loss leader for vendors, dictated by contract specifications but rarely the proximate cause for system purchase. All of these technical, economic and political attributes exacerbate performance tool research, development and deployment.

4.1 Performance Tool Status

There is general agreement that the basic elements of performance tuning (i.e., instrumentation, measurement and analysis) are well understood and effective when applied judiciously. During the past twenty years, the performance tool substrate has been well defined and basic infrastructure support has improved. High-resolution, global clocks (for event ordering), hardware performance counters, and basic performance measurement libraries are now standard on (almost) all platforms.

There are standard software instrumentation techniques for FORTRAN and C, although C++ instrumentation with templates is more problematic. Analysis of message passing applications based on MPI is well supported, as is instrumentation of user code regions and functions, but there is less effective support for OpenMP and thread-level parallelism. This missing support is rising in importance as multicore processors, particularly heterogeneous multicore, become more common.

The standard measurement techniques, based on statistical sampling, profiling and event tracing are well known and are supported by a diverse tool base. However, few of these tools are truly scalable to systems with tens or hundreds of thousands of processors. As this suggests, analysis techniques are the weakest of the three elements of the performance tool chain, as most measurement techniques produce too much data, and the analysis tools are rarely able to identify root performance bottlenecks and suggest effective remediation. Simply put, deep analysis is largely a user function, rather than a tool capability.

We need a new generation of performance tools that automate a larger fraction of data analysis, emphasizing key code segments and execution threads. The visualization techniques of the 1980s, which

showed each thread of control or message interaction, will not scale to 500K parallelism. We lack both the screen real-estate and pattern recognition skills to recognize rare events in a sea of data.

This highlights the fact that performance tool research and development, like that for debugging tools, spans both technical (measurement and analysis) and usability issues. Effective performance tools must not only capture and present relevant performance data with low overhead, they must do so in ways that application developers find intuitive and useful within the standard software development and support life cycle. Given the rapid flux in machine architectures and the long lifetime of scientific and weapons codes, this is a daunting human-computer interaction (HCI) challenge.

Similarly, current analysis tools support only homogeneous systems (i.e., those with little differentiation among processors). The rise of multicore processors, particularly heterogeneous multicore processors, and systems with specialized co-processors (GPUs, FPGAs, Cell and others), poses new challenges, and no extant tools are effective in analyzing any but the most trivial codes.

The working group also believes current tools focus excessively on time as a metric, ignoring the rising importance of memory analysis, particularly as new and emerging systems have ever deeper memory hierarchies. While time-to-solution is the most important metric in HPC, it can be difficult to diagnose and correct performance problems without other metrics, such as locality and reuse of data. In this sense, memory can also be construed to include I/O systems, a limitation for checkpointing and recovery on failure-prone petascale systems.

Finally, although the group devoted little time to discussing performance modeling as an enabling tool for performance optimization, the group did believe modeling is important, both for system characterization and for performance prediction. With this backdrop, Table 2 summarizes the working group's assessment of current performance tool capabilities and techniques.

Table 2: Performance Tool Assessment

Capability	Assessment
Measurement/analysis	WIP
Modeling	WIP
Optimization	LI
Interactive/manual	WIP
Automatic	WIP/Challenge
Data management	WIP
Instrumentation	WIP
Hardware and OS support	WIP
Visualization/presentation	WIP

Legend: WIP (work in progress), LI (little insight), IH (in hand)

As the table suggests, we have effective tools and techniques for smaller, extant systems, but new approaches (both research and development) will be needed for emerging heterogeneous petascale systems. Substantially greater research will be needed for automated and semi-automated code optimization if we are to reduce the already large cognitive burdens based by scientific application developers. We discuss this topic in greater detail below.

4.2 Petascale Requirements

Petascale systems bring new challenges and not simply from the larger number of processor cores. Increasingly complex applications with code coupling, multiple programming models and distributed data sources will exacerbate already complex performance analysis and optimization problems. When coupled with dynamic adaptivity and hardware heterogeneity, new tool approaches will be required.

Specifically, the working group believed performance analysis tools must include greater automation for detecting anomalies, correlating and clustering performance data and behavior and for reducing data to avoid excessive execution perturbation and user confusion. This will also require new support for programming model abstractions and the ability to elide detail and complexity and to reveal them only

when required. The rise of programming model heterogeneity (i.e., both implicit and explicit parallelism), together with hardware diversity and deep parallelism hierarchies, will necessitate new tool approaches.

Petascale systems (hardware, system software, libraries and applications) must be resilient to the inevitable faults expected with million way parallelism and large numbers of commodity components. This suggests that performability (i.e., hybrid assessment of performance and reliability) will be one of the major new frontiers for petascale performance analysis tool research and development. In turn, this will demand new multi-level instrumentation and metrics and scalable presentation metaphors and models that can highlight both performance and reliability problems for computing elements, memory systems and I/O systems.

One immediate consequence of performability analysis and optimization is the fusion of offline and online techniques. In particular, runtime optimization must maximize performance in the face of hardware and software failures, allowing applications to continue execution, albeit perhaps by shifting elements of the computation to other resources, ensuring accuracy given transient failures (e.g., memory or processor bit errors) and balancing on-chip and off-chip parallelism for multicore processors. The working group also believed that additional training and education will be required to create a new generation of application scientists and performance tool developers. As noted earlier, performance optimization complexity is rising rapidly, yet the pool of performance tool researchers and developers has not grown substantially.

Finally, there was strong agreement among the working group members that petascale performance analysis is not simply a scaling of terascale problems. New approaches will be required, dictated by code heterogeneity, hardware and software reliability and system scale. This is in striking contrast to terascale experiences, where approaches used with 10-100 way parallelism were scaled to 1000 way parallelism, albeit not without difficulty.

4.3 Findings

As noted above, petascale systems will be of higher complexity and greater heterogeneity than terascale systems. Consequently, petascale performance analysis is qualitatively more complex than that on terascale systems. Today, we rely primarily on manual, labor intensive methods using static and offline approaches. We instrument applications, capture performance data during application execution and then analyze the data after the execution completes. In the petascale regime, the current performance tool practice will be increasingly challenged as the amount of performance information increases. Petascale tools must be complemented with online, adaptive methods that measure and optimize application behavior automatically during execution, rather than solely relying on post-mortem user assessment and adjustment.

Online, adaptive optimization will require greater integration of tool components. It will also place greater stress on system software and runtime libraries for performance tool support, the subject of the infrastructure working group. Intuitively, we need a new set of tool building blocks than can serve as the basis of a diverse suite of experimental performance tools via component reuse. This is the software analog of the flattened hardware hierarchy, where old abstractions are being replaced by end-to-end hardware optimization – so-called holistic design. Finally, we need better mechanisms for hardening, documentation, support and user training. The “valley of death” between research prototypes and robust, easily usable tools must be bridged if we are to identify those ideas that are both intellectually interesting and practically useful. This may require new and more flexible funding approaches, together with longer-term support and transition mechanisms.

Table 3: Performance Tool Challenges.

Challenge	Type	Risk	Impact
User engagement and training	Training	High	High
Additional information sources (e.g. I/O, memory)	Technical	High	Medium
Long-term maintenance and support	Funding and policy	High	High
Funds for technology transfer and deployment	Funding and policy	Medium	Medium
Application-driven development of tools		Medium	Medium
Substantial advances in automation of diagnosis, optimization and anomaly detection	Technical	High	High
Developing live techniques to extend post-mortem	Technical	Medium	Medium
Integrated, persistent monitoring components	Technical	Medium	Medium
Support for multi-component and multi-disciplinary applications	Technical	High	Medium
Detection of load imbalance	Technical	High	High
Support for heterogeneous and hierarchical hardware	Technical	High	High
Support for new and hybrid programming models	Technical, funding, policy and training	Medium	Medium
Add performance analysis to CS curriculum	Training and policy	Low	Low

4.4 Recommendations

Based on the working group's status assessment and findings, the group classified a set of performance tool challenges based on the risk of failure and the potential impact (benefit) of success. Table 2 summarizes these challenges and recommendations. Each of these is broadly connected to the findings discussed above.

Of these, the most important, based on perceived risk and benefit, are as follows.

- ***User engagement and training.*** Long experience in developing performance tools has demonstrated the need for tighter integration between tool and application development teams, from the standpoints both of understanding requirements and of providing adequate training. We discuss details of this cross cutting challenge in Section 8.
- ***Long-term maintenance and support.*** Productization was also identified as particular concern by the performance tool working group. It again actually applies across the areas considered at the workshop and is discussed in more detail in Section 8.
- ***Automated diagnosis and remediation.*** As noted earlier, the scale and complexity of petascale systems will necessitate new approaches to performance optimization, shifting from offline, manual analysis to greater automation. This is a major technical challenge that will require new research funding and technical insights. Failure to address this challenge will increase the probability that emerging petascale systems are used inefficiently.
- ***Load imbalance detection.*** The petascale manifestation of Amdahl's law, load imbalance becomes increasingly critical when a single thread can delay hundreds of thousands of others. Current tools focus largely on metrics, rather than causes. New approaches are needed for analysis and presentation.
- ***Heterogeneous, hierarchical architecture support.*** Finally, the multicore revolution will bring 100-way parallelism on each chip, albeit with diverse cores and capabilities. For almost all systems, this architectural direction will necessitate the use of multilevel parallelism, in general, and threading in particular. Thus, performance tools will need to target multilevel paradigms, including hybrid OpenMP/MPI programs. The Los Alamos Roadrunner system, with Opteron SMPs and Cell accelerators is a forerunner of heterogeneous architectures. Performance tools must capture and relate performance and reliability problems to source code in ways that make multilevel performance optimization possible and practical.

5 TECHNICAL THRUST – CORRECTNESS TOOLS

Chairs: **Susan Coughlan (Argonne National Laboratory)**
Curtis Janssen (Sandia National Laboratories)

5.1 Topics

The primary goal of correctness tools is to provide confidence in application code correctness and to potentially suggest recommendations on repairs to users. The correctness tools working group was charged with exploring the following topics:

1. Instrumentation;
2. Data collection;
3. Data management;
4. Attribution;
5. Automatic correctness recommendations;
6. Visualization;
7. Data mining;
8. Hardware and operating systems support for correctness.

Ensuring that programs run to completion and produce a correct result remains a challenging and expensive problem in all areas of software development. This is no less true for parallel applications that are intended to scale to petascale-class architectures and beyond, which present unique challenges for these goals. Developing applications that run as multiple processes on multiple machines, typically with data communication patterns that tightly couple the processes together, is exceedingly difficult. Petascale machines will layer on top of this the challenge of dealing with additional parallelism models such as threading, or heterogeneous computing resources. It is even likely that many applications will need to reduce interprocess synchronization to run efficiently at such scales by relying on global address space methods and active messages, as is already done in several quantum chemistry applications, for example. Furthermore, by making so much computing power available, petascale machines will change the nature of applications, allowing more detailed coupled multi-scale, multi-physics computations. While developers look for bugs by attempting to reproduce errors with small inputs at small scales, this is not always possible, and it will be necessary to give developers the capability to debug and to analyze programs running at full scale. These multiple dimensions of complexity require that providers of petascale-class machine carefully review the available correctness tools, find their deficiencies in a petascale environment, and chart a direction to resolve these deficiencies. This report will review classes of tools that are required by developers to ensure code correctness and give an initial estimate of the priority of each gap and difficulty in closing the gaps.

5.2 Scope

We will restrict our attention to tools that developers will use to ensure that programs run to completion and give the expected result. We will not discuss validation tools, tools for ensuring that a program's results agree with experiment. Validation is an important area and could not be given adequate attention within the scope of the Correctness Tools Working Group. Also, performance regression in a program can indicate a bug, even if the correct answer is obtained. Since another working group was dedicated to performance tools, we do not discuss performance regression testing here.

Tools that are included are the traditional debuggers that allow programmers to set breakpoints, to run, to step though, and to examine and to modify data in a running program. At large scales, lightweight versions of these tools that provide critical debugging capabilities, while providing more scalability than traditional debuggers, are needed. Other debugging tools include tools that check for incorrect memory references and for efficient memory usage. Also, because of the additional complexity caused by parallelism, tools are needed to enable users to detect incorrect use of parallel programming techniques. Such tools include thread correctness checkers and Message Passing Interface (MPI) usage checkers. A complicating factor is that as machines became larger, and the average time between hardware errors decreases, it will be more and more important for developers to be able to distinguish between application software errors, library errors, middleware errors, and hardware errors. Finally, detecting potential at-scale errors before running at scale could save a great deal of effort. Tools examining trends for a series of small-scale runs, static analysis tools, and formal verification tools could play a role in this regard.

5.3 Debugging Tools

Traditional debuggers allow programmers to manipulate an application by actions that include setting breakpoints, stepping though source lines, and examining and changing data in a running program. Currently, a job running on four thousand processors is the largest supported use of a traditional debugger, and it is expected that the practical upper bound for the number of processors is in the range of one to eight thousand in the best case scenario. On most current systems the practical upper limit is on the order of several hundred processes. Beyond that developers will need a combination of lightweight debugger techniques, becoming progressively lighter and more autonomous in analyzing the application to isolate the problem.

Because of the complexity of petascale applications and the large number of code and hardware groups that are either directly or indirectly involved in an application, it is highly desirable to isolate the cause of bugs as precisely and definitively as possible. Such root cause analysis must also consider possible hardware sources for the error, incorporating information from the machine's Reliability Availability and Serviceability (RAS) system about the health of the hardware.

5.4 Memory Usage Tools

Memory usage tools fall into two broad categories: tools for monitoring memory utilization, including memory leaks and overall memory consumption. and tools to find programming errors in the way memory is accessed. The second category of memory tool includes lightweight tools that identify simple array overruns, as well as heavyweight tools that instrument the running application to monitor all loads and stores while tracking which memory locations contain valid data. All of these tools can locate programming errors before the error manifests itself as an incorrect answer or job interrupt. The heavyweight memory tools are the most informative, as they are the only tools that can detect incorrect memory accesses into incorrectly or correctly allocated memory at the very instruction that caused the problem. Portable tools for monitoring memory utilization are missing and this is a specific issue with developers. Other memory tools are currently believed not to have particular issues, except for the open-source business model and scalable infrastructure issues that were discussed above.

5.5 Tools Specifically for Parallelism Correctness Checking

Users need several tools in order to detect incorrect use of parallel programming techniques. Such tools include thread correctness checkers and MPI usage checkers. Thread correctness checkers are very similar to memory tools. However, instead of being concerned with the validity of memory references, they verify properties including that no potential race condition exists. No threading tool that runs on all

processors of interest to DOE currently exists. Further, no existing threading correctness checker supports the hybrid threading/MPI paradigm, even at small scales, that will be relevant for petascale systems due to slowdown incurred when using existing tools.

Tools for checking MPI parallelism exist, however the full potential of this category of tools has yet be realized. For example, trends recorded by tools from a series of runs at small scale could be used to predict potential undesirable behaviors that would arise at large scale.

5.6 Static Analysis and Formal Verification

The best time to find a bug is before the application is executed. Static analysis tools to check program style and find certain programmer errors have been available for some time. In fact, compilers continue to have more and more static analysis capability built into them. However, from a petascale tools perspective, additional checks could be done that are outside the scope of ordinary static analysis tools. An example of such checks include locating integer operations that could overflow at scale and checking parallel programming constructs for correctness.

It is also possible to detect flaws in algorithms before the programming stage by using formal verification techniques. Such techniques would permit parallel programming constructs to be rigorously verified for guaranteed correctness, eliminating the need for a trial and error approach to implementing and debugging a faulty algorithm.

Table 4: Correctness Tool Challenges.

Challenges	Priority	Difficulty
Scalability of traditional debuggers being able to do the same things on more nodes is too limited	8	Med/High
Memory leak/high water mark tool to instrument libraries is needed	7	Medium
Existing thread correctness checkers are neither multi-platform nor designed for multilevel parallelism	6	High
Need lightweight tools to perform root cause analysis	4	Med/High
Interface between traditional debugger to light-weight debugging tools (tool collaboration interface) is needed provide to a smooth transition from super light-weight to super heavy tools	4	Med/High
Not enough extreme scale lightweight debugging tools	4	Med/High
Missing lightweight tools to debug core files at large scale	4	Medium
Missing the ability for RAS systems that applications and system software can use to learn about hardware causes for program faults and to allow the fault to be handled	3	Medium
Open source support model needs to be elucidated, as well as the role of vendors	3	Medium
Compiler infrastructure needed to build the static analysis tools is missing	3	Med/High
New ways to represent the output of debuggers, pre-analyzed for users, are needed	3	Med/High
Operating system support (overriding issue for pretty much everything) remains an issue for many tools	3	Med/High
User education efforts insufficient for many tools	2	Med/Low
Hardware/system software coverage test suites are not exhaustive	2	Medium
Common open infrastructure (with performance expectations that are documented) is missing	2	Med/High
Many static analysis tools are still needed, particularly for Fortran	2	Medium
Ease of use often missing for many tools	2	Med/High
Tools that use formal verification methods to identify deadlocks, livelocks, race conditions, and other errors in parallel software are missing	2	Med/High
Need MPI usage trend tools that run at small scale could be (but are not) used to predict issues that could occur at large scale	1	Med/High
Build related failures are not identified, including software that are not built consistently (different flags, different compilers)	1	Medium
Extensibility of the debuggers for more user-driven analysis capability is needed	1	Med/High
Impossible to develop tools for scale without access	1	Medium
No standardized test harness for regression testing exists	1	Medium
Design by contract, assertions, parallel assertions are at best poorly supported	1	Med/High

5.7 Miscellaneous Issues

Other issues, some of which are not represented in the gaps below, are worth discussing. First, no standardized test harness for regression testing is widely used. In addition, over time, the performance of

a system can degrade and hardware components (such as memory) can start to fail. Pre-emptive performance and correctness regression testing on a system should be done at regular intervals to ensure that jobs accomplish useful science rather than debug system problems. An exhaustive regression test suite as well as research to determine when and what to test to minimize testing time and to maximize the system's protection, is needed. Second, consistency in software builds becomes an issue in complex applications involving many libraries. Using different compiler flags for different parts of the application can result in errors, or limit the functionality of tools. Finally, there is no formal support for design by contract in languages typically used by the parallel computing community, and scalable support for assertion checking (and the subsequent error reporting) is nonexistent. Assertion checking involving explicit parallelism (say, ensuring that a variable has the same value on all nodes) is also lacking.

5.8 Correctness Tool Working Group Findings

Table 4 lists a number of gaps in correctness tools for petascale computing that the correctness tools working group identified. However, a tool or requirement not being listed as a gap does not mean that the tool is not essential. Failure to support existing, successful capabilities, or not addressing the gaps affecting all correctness tools could result in additional gaps.

Working group participants were asked to prioritize the gaps by distributing a maximum of seventy points among all the gaps, with more points being allocated to higher priority gaps (the higher the priority, the greater the impact of addressing the gap). Everyone was also asked to rate the difficulty of closing the gap as low, medium, or high. Table 4 gives the averaged priorities. The difficulties were averaged over just those responses that rated the difficulty of each particular gap.

The ratings exhibited considerable variability. For nearly all gaps, the standard deviation of the priorities was greater than the average priority. The only exception was for the thread-correctness tool gap, which had a smaller standard deviation and, hence, for which there was a greater degree of consensus. The highest priority needs are 1) improvement of the scalability of traditional debuggers, 2) memory usage tools, and 3) thread-correctness tools. The total average number of priority points assigned to these three were twenty-one out of seventy. Just below that was a cluster of four gaps that totaled to sixteen points. These gaps all related to lightweight debugging capabilities. Beyond that, a number of gaps were identified for a total of twenty-four in all.

6 TECHNICAL THRUST – SCALABLE INFRASTRUCTURES

Chairs: **Al Geist (Oak Ridge National Laboratory)**
Jeffrey Hollingsworth (University of Maryland)

The area of scalable infrastructures is defined as software and hardware that are used primarily by tool builders in creating tools. However, many types of infrastructure may also be useful by applications teams in building internal tools and even for directly instrumenting their programs. For example, the PAPI performance counter library would be considered tool infrastructure since it is used to build other tools. However, many applications groups also directly use PAPI.

The goal of tool infrastructure is the development of useful bits of software that make it easier to create new tools. By packaging commonly needed features, tool innovation can be encouraged by allowing a

tool developer to concentrate on the new aspects of their tool rather than re-creating commonly needed infrastructure.

6.1 Topics

Tool infrastructure complexity can range from a relatively simple library with a few hundred lines of code to complex systems with dozens of functions and hundreds of thousands of lines of code. Developing the infrastructure described here can sometimes be done by tool developers working alone. However, frequently tool infrastructure is at the edge of tools and other disciplines, such as architectures, operating systems, compilers, and execution environments. We try to note where there are interactions required from other areas. In this report, we group tool infrastructure into six thematic areas:

1) **Tool Communication** includes the areas of communicating information within a tool when tool components are located on different nodes in a parallel system. It also includes the need of tools to get information from external sources such as the hardware, operating system, compiler, scheduler, and runtime system (libraries and scheduler). Another important aspect of tool communication is the ability of tools to exchange information between tools. An important aspect of communication is reuse. Reuse can include both tool components (code) and information about applications (data).
2) **Data Management** includes all aspects of gathering, reducting, and storing information about applications. It includes not only directly measured information (e.g., trace of operations, or sampling of counters), but also metadata about the execution environment in which the data was gathered (e.g., machine configuration, library versions). Data reduction techniques include any ways in which the collected performance data can be aggregated, distilled, or otherwise condensed to reduce the volume of data resulting from long running programs executing on leadership class machines. Since leadership class machines are national assets with a geographically distributed user base, an important aspect of data management is ensuring tools work well when there are large latencies between users and the machines.
3) **Scheduler Issues** involve the relationship of tools to the batch scheduler on the system. Tools must closely coordinate with the scheduler on issues such as: tool deployment (launching the tool onto nodes of the system) and tool composability (the coordination of multiple tools that may wish to run at the same time on the same application). Tools also need to be able to make requests of the scheduler for additional resources (e.g., extra nodes or memory) for either the tools themselves or for tool functionally such as fault tolerance. In addition, tools need information from the scheduler about the network topology and other attributes of the assigned nodes.
4) **Operating System Issues** for tools include the availability of process control interfaces (e.g., /proc or ptrace), access to thread information including thread creation and dispatch, and low overhead access to hardware counters. In addition, the underlying operating system must include sufficient support for the scale of the machine (e.g., maximum number of open file descriptors).
5) **Binary Manipulation** is the ability to gather data from compiled programs and to insert code into binaries to create new modified binaries. Related issues include compiler hooks to provide information about transformations and optimizations. Other challenges include binary analysis of optimized and stripped programs, and the need to generate new binaries with instrumentation. In addition, other information such as a stack un-winder is required to allow the creation of runtime tools.

6) **Miscellaneous Issues** include the ability of tools to survive (or at least not crash the application) when a machine fault occurs, and scalable support for tracing such as parallel file I/O support. Additional issues include the need for tools to work with applications that are partially written in scripting languages.

6.2 Current Status

Before looking at the future tool needs, we briefly review the status of tool infrastructure in the six areas outlined above.

In the area of tool communication, many tools develop their own styles of communications. The major two common infrastructure components are PAPI for hardware counters, and MRNet for inter-tool communications and data reduction. At the lower level, OS specific counter libraries such as perfmon2 (Linux) and hpm (AIX) are available, but not on all platforms. Data management tools consist of the research projects PerfDB, PerfDMF, and PerfTrack. There are SciDAC efforts in the SDM and PERI projects to develop interoperation between these data management frameworks.

Operating system and scheduler interfaces are available in various forms. While there is diversity in implementation, functionality is generally available on machines using traditional operating systems (e.g. Linux and AIX). However, a major limitation for tool builders currently is the limited OS support for tool functionality on node micro-kernels such as the Catamount and BlueGene kernels. Process control, sampling, and memory information are all inadequate on these micro-kernel systems today. Scheduler support for co-allocation of nodes for tools remains limited. Support for getting network topology information has been limited at best. Some thread information is available for MPI via the threadDB interface, or for openMP via the performance interface detailed in an OpenMP ARB white paper.

A variety of research has been done in the area of binary manipulation. For online binary editing, tools such as Dyninst have provided multi-platform support. In the area of static binary re-writers, a collection of platform specific tools such as Atom, Pin, and Valgrind are (or have been) available. Many tool groups have built their own binary analysis tools to meet their needs, but sharing analyses between tools has been limited to date. Compiler hooks to support analysis and instrumentation have been included in gcc and the ROSE source-to-source translator. However, to date well documented interfaces to this type of information are not available from compiler vendors. To support open tracing of programs, the Open Trace Format (OTF) standard has been developed. To date, several tools can read and write this format, but additional adoption and standardization of information in the traces is still needed.

6.3 Infrastructure Findings

Today, tool infrastructure reuse is uncommon. The emphasis on research prototypes rather than production tools has limited reuse. The lack of reuse has limited tool innovation since the effort required to explore a new tool idea is unduly large when vast amounts of code must be written to explore an idea.

Due to the growing complexity of petascale systems, applications and systems will be more dynamically adaptable. Runtime support for fault tolerance, adaptive load balancing, and varied compute models will all require additional tool support, and the need for tools to measure and respond to changes during program execution.

Tools need communication abstractions beyond TCP/IP sockets. With the increased use of micro-kernels and high-speed networks, having communications between tool components rely on TCP/IP networks is no longer adequate (or even possible in some cases). To promote reuse and integration, these tool communication abstractions must be standardized

The cost of supporting tools for multiple platforms and operating systems is straining tool developers, both commercial and academic. The diversity of platforms and operating systems has made this worse. Additional operating systems such as Cray's Compute Node Linux and Linux on Power systems has only made the situation more complex by adding additional OS choices to existing hardware platforms.

Going to petascale will increase the need for anomaly detection and (dynamic) data reduction. Anomalous performance of nodes (due to either hardware or system software) will become the norm at the large node counts expected. Likewise, the volumes of data that can be generated from these larger numbers of cores will require new techniques in dynamic (during program execution) reduction and processing.

Table 5: Tool infrastructure challenges.

#	Challenge	Risk if nothing done	Impact if challenge solved
Tool Communication			
	Within a single tool	Medium/High	High
	Interfaces to the OS/machine data	Medium	Medium
	Lack of reuse	High	High
	Tool interop/data exchange	High	Medium
Data Management			
	Measurement data and meta data from runs	Medium	Medium
	Data reduction & presentation tools – graph tools	High	High*
	Availability of Additional Hardware Data – memory, etc.	Medium	High
	Distance between user and machine – latency and data volume	Medium	Medium
Scheduler Issues			
	Tool Deployment – launch	High	High
	Co-scheduled jobs or "spare" nodes in a job request for tools or fault tolerance	High	High/Medium
	Topology map info, control of placement of "spare" nodes	Medium	Medium
	Tool composability – all fighting for same hooks	High	High
OS Issues			
	OS scaling issues for tools – enough sockets	High	High
	Process control	High	Medium
	Threading issues	Medium	Medium
	Low overhead access to performance data	Medium	High
Binary Manipulation			
	Binary re-writing	High	High
	Compiler hooks	High	Medium
	Binary analysis	Medium	Medium
	Stack un-winder	Low	High
Miscellaneaous Issues			
	Tools survive machine faults	High	High/Medium
	Tools for scripting languages	Medium/Low	Low
	Perf. data I/O support: parallel file system, avoiding I/O via reduction	Medium	High
	Sampling support for signals/interrupts	Medium	High
	Clock synchronization	High	Medium
	Scripting languages for tools	Medium	Medium/Low

6.4 Recommendations

Funding for petascale tools infrastructure is important to the success of an overall scalable tools effort in order to:

- Supply the scalable, dynamic, capabilities needed by next generation tools;
- Reduce redundancy of having each tools group develop their own infrastructure capabilities;
- Promote integration of developed tools by standardizing infrastructure APIs;
- Reduce cost of life-cycle tools support by having the infrastructure handle much of the diversity of platforms and operating systems.

A second major recommendation is to fund the full tool life cycle from design, hardening, support, and long term maintenance. While, this can't be done for every tool idea, it needs to be done for those tools and infrastructure capabilities that are the most popular and useful.

Table 5 shows the workshop assessments of various challenges to the tools infrastructure at petascale. It is divided into the categories given in the Topics sub-section. For each item the risk to the success of future tools if nothing is done is scored as high, medium, or low. With a similar scale the impact if the challenge would be solved was assessed by the workshop attendees. The highest scoring item with a unanimous vote was data reduction and presentation capabilities supplied by the infrastructure. Several other items scored high in both in risk and impact. These include: lack of tool reuse, coordinated tool launch, tool composability, scalability of OS features needed by tools, and binary rewriting.

7 TECHNICAL THRUST – DEVELOPMENT ENVIRONMENT INFRASTRUCTURES

Chairs: **Rod Oldehoeft (Krell Institute)**
Craig Rasmussen (Los Alamos National Laboratory)

7.1 Topics

This group has been assigned the task of identifying the requirements and gaps in the development environment necessary to meet the petascale challenge. The working group considered the following topics:

1. Integrated development environments;
2. Build environments (e.g., make, libtool);
3. Compiler support for development environments;
4. Mixed language support;
5. Refactoring;
6. Automatic interface generation and validation.

7.2 Scope

Programming models (and the languages and libraries implementing these models) are a very important component of the development environment. Like developer tools, the programming models will need to adapt to the growing levels of on-chip parallelism and the increasing depth of memory hierarchies. It is important that programming models be considered in meeting the challenges of petascale computing, in particular in those areas related to the placement, movement, and access to memory. However, programming models are outside the scope of this working group and we restrict our attention to tools that aid the developer in designing, creating, modifying, building, and running applications.

Tools specifically related to program correctness, debugging, and performance are in the purview of other working groups and are not considered here. However, development architectures supporting the *integration* of the full range of tools affecting application development are within the scope of this working group and are considered. In addition, it should be noted that compiler infrastructures, while they may lie outside the scope of other working groups, are an important component in tools considered by these groups, in particular those tools related to program correctness and performance.

There are many classes of tools used by the scientific community in the development of scientific applications. These include: UML tools used for software design and modeling; tools that support a group of geographically distributed developers and tools that support remote development (isolated from the petascale computer); revision control tools; configuration and build tools and libraries and runtime environments; tools supporting development in mixed languages; tools providing program transformations and refactorings; and compiler infrastructure support for general tool development. These tools and tool classes were all discussed by the working group, although some were excluded from detailed consideration early on, as described below. In addition, we considered integrated development environments that provide an infrastructure to integrate a broad range of development tools.

7.3 Tools and Issues Not Considered, and Considered

After our preliminary discussions developed an extensive list of tool categories and issues, we excluded several that we felt were present in the current computing environment and did not have a direct affect on petascale computing, or that were well on their way to being handled at this scale. These include software design tools, systems for source code control and bug tracking, database services, and project management issues.

Two areas of software development were recognized as important and relevant to petascale computing, but we did not consider them owing to time constraints. These are fault tolerant methods in software design, and software design that is informed by the hardware systems' inter-processor and processor-memory topologies. We recommend that these not be ignored, because both can greatly affect the performance of petascale applications. These topics should be included in future discussions.

On the other hand, we included one class of problems that some felt were not directly related to petascale computing, but that nevertheless currently cause significant difficulties for application programmers. In particular, program building and configuration, including library ordering during linking, continue to take too much effort during application development.

7.4 Application Build Tools

The current state of tools for program configuration and construction is deplorable. Applications must be built for multiple systems, including perhaps one or more petascale machines. We found that too much complexity results from multiple compilers, operating systems, libraries (and their versions). Common option sets and command-line interfaces are missing. We are concerned that the lack of shared libraries and dynamic linking capabilities on petascale systems currently in development will contribute more difficulties. We recommend consideration of new tools (make is still broken), improved tools (e.g., for managing linking order), and more attention to interoperability of program build tools.

7.5 Mixed Language Environments

Because of the complexity of petascale applications and the large number of code teams that are either directly or indirectly involved in developing an application, mixed-language programming becomes increasingly important in petascale application development. For example, as the number of physics packages and libraries rises, the likelihood increases that different components of an application will be written in different languages. In addition, programming paradigm changes will be required to handle the increased levels of parallelism associated with leadership-class machines. User feedback clearly indicates that this will include the hybrid OpenMP/MPI paradigm as well as a likely increase in usage of PGAS languages such as UPC and Co-array Fortran.

We found that developers are using Python to script application runs and to prototype algorithmic ideas. We recommend that language interoperability efforts continue at both the tool and language-standards level and that increased automation be used to decrease the level of programmer effort. It is a concern that tools that require the ability to load libraries dynamically, like Python, will not be able to be used with some operating systems on petascale computers.

7.6 Compiler Infrastructure

Several tools are needed to enable petascale computing. Many of the tools discussed here, and in the context of other working groups, require, or could use the benefit of, static analysis capabilities provided by a compiler infrastructure. For example, static analysis is used to instrument codes for performance measurements automatically, is used in source-to-source transformations to enhance performance (e.g., loop transformations), and is used in tools to aid programmers in ensuring that applications perform correctly (e.g., lint tools).

We found that both developers and tools would benefit from a compiler infrastructure. In particular, knowledge of estimated performance at the source-code level (cost estimates are routinely a part of an optimizing compiler) should be provided. This is particularly important with regards to IO costs and memory usage, as memory latency dominates the performance of most scientific applications.

We recommend that a flexible (easily adaptable), portable, and open-source compiler infrastructure be supported to provide for the tooling needs of petascale computing. Vendors should also be encouraged to open up portions of their compilers to provide users and tools with as much information as possible.

7.7 Program Transformations

Currently, scientific applications must use a variety of hardware architectures in distinct runs. Even today it is a challenge to achieve performance across multiple architectures and this challenge will only be exacerbated as petascale platforms are delivered. We found that source-to-source transformations have successfully adapted existing codes automatically to new computer architectures. We recommend that projects be supported that explore the usage of program transformation tools in achieving architecture independence for scientific applications.

7.8 Software Development for Remote Systems

Because of the substantial costs associated with leadership class facilities, most users will likely be located at a site that is remote from the petascale computing resource. Even if a user is at the same institution as the computer, the user will likely be isolated from the machine itself. The file system, compilers, libraries, tools, and other resources will always be remote from a user's desktop and will probably not be the environment under which a user's application was developed.

A remote environment provides challenges for a user as files must be transferred, differences in environments taken into consideration, and often several different levels of authentication negotiated. Perhaps most importantly, it complicates the use of software development tools. Overall, these issues imply that we must provide mechanisms so that application programmers can effectively work remotely. In particular, we need high performance support for secure remote GUI operation. It would also be beneficial if the development environment hid many of the details of remote operation so that the user need not substantially modify their usual workflow. We recommend investment in a client-server-based shared infrastructure for remote development, including improved communication efficiency.

7.9 Integrated Development Environments (IDEs)

Petascale applications will have much larger and more complex requirements for program control, data management, and visualization, in addition to the normal developer activities of designing, coding, building, debugging, and performance optimization. IDEs are designed to increase developer productivity in all these areas by providing an integrated workbench of tools, sharing data between tools, and automating many common tool operations. However, they also tend to have steep learning curves and must not interfere with desire for a set of lightweight tools, as was clearly expressed in the user talks at the start of the workshop. Further, most scientific application developers have existing workflows that would require any petascale IDEs to support the use of their components in stand-alone mode. Thus, we found that although IDEs are widely used outside of scientific computing, they have had little impact in HPC. Projects sometimes put together specialized environments for a single application, which do not generalize to future use. Single stand-alone tools are much more common. While IDEs are unlikely to gain adoption in HPC easily, they might provide productivity gains. For this reason, we recommend limited investment in IDEs that focuses on pilot projects to explore their possible advantages and to establish their ability to support existing workflows as well as the revamped integrated workflow for which they are known.

7.10 Findings of the Development Environment Infrastructures Working Group

The working group identified a number of gaps in application development tools and environments for petascale computing and these gaps are listed in the Table 6. However, we should note that a tool or requirement not being listed as a gap does not mean that the tool is not essential. For example, vendor-supplied optimizing compilers are essential in developing high-performing applications. Failure to support existing, successful capabilities, could result in additional gaps.

Working group participants were asked to prioritize the gaps by providing a rating of high, medium, or low to indicate the priority of addressing the challenge. The ratings were normalized to a scale of one to ten and entered in the table. Each participant was also asked to indicate the impact on the ability to reach petascale computing goals if a gap were *not* closed, again on a scale of high, medium, and low. Both priorities and impact in the table are averages of participant responses.

It should be stressed that *all* of the gaps listed below received a medium to high priority and impact rating. Areas found to have a lower priority were not considered fully by the group and are not listed here.

Table 6: Development environment infrastructure challenges.

Challenges	Priority	Impact
Application build and configuration issues are a continuing problem.	10	High
Compiler infrastructure is needed to support a wide variety of development of tools.	9	High/Med
Tools are required to facilitate the development of applications from remote sites.	9	High/Med
Mixed language support needed for migration to new programming models.	8	High/Med
Scalable dynamic linking support is needed from vendors and static linking a continuing problem.	8	High/Med
Program transformations required for portability across hardware architectures	6	Medium
Lack of an integrated development environment for the integration of tools.	5	Medium

8 STRATEGIC NON-TECHNICAL CHALLENGES

The working groups all agreed on a set of important non-technical challenges: policy, funding, business, intellectual property, or training/education. We collate and highlight these crosscutting issues in this section. In most cases, a solution to each of these issues would have significant and immediate impact

on the success of software development tools for HPC. Steps beyond technical research should be taken to eliminate these challenges.

1. **Funding and model for sustaining/hardening tools.** Software development tools are a financial burden for HPC vendors, as their features rarely if ever determine acquisition outcomes. Similarly, academic and laboratory performance tools researchers and developers rarely possess either the skills or the desire to transition research ideas to production code, with concomitant support. Nonetheless, many of today's successful research tools could benefit from sustained funding to transition the tools to production software. However, the government rarely funds long-term maintenance and tools support. The pathway from research prototype to a software tool that is widely available, production quality and actively supported is not clear. In most cases, the funding researchers receive is targeted toward specific research goals, and not necessarily to provide tool porting, testing, documentation, standardization, or user support. A new model of software tool support is needed if we are to address current and future needs.
2. **System diversity.** Architectures and software systems for HPC are quite diverse when compared to just a decade ago. In many cases, existing tools must be ported and validated against these new systems. This task is made much more difficult if the new systems use novel architectural features, or non-compliant or proprietary software.
3. **System testbeds and development access to target platforms.** Access to system testbeds for software development and testing continues to be a challenge for development of software tools. In particular, software development tools must be able to run at production scale and in the same environments as production users. In some cases, these developers must be able to modify the system software, such as the operating system, to perform their tests.
4. **Access to and engagement with applications and domain experts.** All too often, performance tools are developed in the absence of detailed understanding of user and application challenges. Conversely, users are often unaware of the technical difficulties underlying tool design and support. Bridging this gap with a collaborative tool development and extension process, where promising ideas are identified and tested early, then enhanced and supported across the application development and support cycle, would ameliorate the expectations gap. Recent experiences in both the Office of Science and NNSA affirm the distinct advantages of having computer science experts engaged with applications and domain experts. Working together, these two groups can best map the applications to the architectures.
5. **User training.** Software development tools can be very flexible and powerful in their own right. The developers of these tools should make it a priority to train the user community on tool capabilities and usage. Furthermore, usability should be a major requirement included in any funding focusing on transition to production software.
6. **Interactions with vendors.** Successful software tools require intimate knowledge of the target architecture and software system. Vendors must provide this knowledge to external developers in some form, either by adhering to standards or by providing specifications, documentation, software, and early access to systems.
7. **Standardization.** Aside from standardization of target system architecture and system software components, developers of software tools could benefit from standardization within their own community. For example, APIs, tracefile formats, and user interfaces could all benefit from standardization. This standardization would promote tool interoperability among other benefits.

8. **Modular infrastructure development.** Modular tool infrastructures that allow the development and composition of a set of tool components emerged in multiple sessions. This infrastructure could provide components using a variety of mechanisms that include libraries, runtime systems, software source code, user interfaces, and standard APIs.
9. **International collaboration.** A large number of software developers reside outside the USA. In order to facilitate collaborations across these communities, the employers and funding agencies must embrace and facilitate these collaborations by providing joint funding opportunities and bridging gaps in policy.
10. **Education and workforce.** As is the case in other areas of HPC and computer science, there is a specific need to educate new students and workers in order to ensure a sufficiently large and capable workforce.

In many cases, these cross cutting issues are symptoms are consequences of larger topics. First of all, the HPC market is relatively small when compared to the consumer and enterprise computing markets. As a result, finding a business model for the development, porting, and support of the tools is a challenge. Proprietary products are only feasible through a combination of licensing agreements and engineering contracts and expose the labs to the risk of the product becoming unavailable due changes in the business case for the company's support of petascale machines, including changes due to the transfer of ownership of the intellectual property involved. Open-source software gives the labs the opportunity to fix bugs and add features through internal efforts and external contracts. Communities can develop around open-source software resulting in a spread of the cost; however, we must recognize that the community of leadership class facilities is, by definition, small. Also, we must find ways to ensure open-source research projects evolve into robust, easy to use, well-documented software. Whether we chose open-source software or proprietary software, the community must be engaged in software development tools and their underlying infrastructure.

A second issue is that we find similar underlying functionality is needed for a variety of tools, and this functionality is often repeated. Open tool infrastructure that can, say, start tools on nodes and collect and filter data are needed, so that more time can be spent on developing the required tool capabilities. Some tools require an interface to hardware features, and some may need a detailed knowledge of the operating system/application interface.

APPENDIX 1 – SDTPC WORKSHOP AGENDA

8.1 Wednesday, August 1

Start Time	Activity	Speaker/Chair
7:30	Continental Breakfast	
8:30	Welcome, Introductions, Goals	Jeffrey Vetter, ORNL Fred Johnson, DOE
9:00	Applications Experiences #1	Brian Pudliner, LLNL
9:30	Applications Experiences #2	Robert Harrison, ORNL
10:00	Break	
10:30	Applications Experiences #3	John Daly, LANL
11:00	DOE Platform Futures	Fred Johnson, DOE ASCR Bob Meisner, DOE NNSA
11:30	Tools Futures	Bart Miller, Wisconsin
12:00	Working Group charter	Jeffrey Vetter, ORNL
12:15	Lunch on your own	
1:15	Poster Session	
2:00	Working Groups convene (1) Performance Tools (2) Correctness Tools	
3:20	Break	
3:30	Working Groups continue	
5:00	Working Groups formulate findings	
5:30	Adjourn	

8.2 Thursday, August 2

Start Time	Activity	Speaker/Organizer
7:30	Continental Breakfast	
8:30	Working Groups convene (1) Scalable Infrastructures (2) Development Environment Infrastructures	
10:00	Break: 15 min	
10:10	Working Groups continue	
11:15	Working Groups formulate findings	
12:00	Lunch on your own	
1:00	WG report #1	WG Chair
1:30	WG report #2	WG Chair
2:00	WG report #3	WG Chair
2:30	WG report #4	WG Chair
3:00	Closing comments and action items	Steering committee
3:30	Adjourn	

APPENDIX 2 – SDTPC WORKSHOP ATTENDEES

The following people participated in the SDTPC workshop in Washington, DC on 1-2 August 2007.

Table 7: SDTPC Workshop attendees.

First/Middle Name	Last Name	Organization	Email
Dong H.	Ahn	Lawrence Livermore National Laboratory	ahn1@llnl.gov
Sadaf R.	Alam	Oak Ridge National Laboratory	alamsr@ornl.gov
Ron	Brightwell	Sandia National Laboratories	rbbrigh@sandia.gov
Cary D.	Butler	US Army Corps of Engineers, ITL	hopkint@wes.army.mil
Susan	Coghlan	Argonne National Lab.	smc@alcf.anl.gov
David	Cronk	University of Tennessee	cronk@cs.utk.edu
John Thomas	Daly	Los Alamos National Laboratory	jtd@lanl.gov
Larry Paul	Davis	DoD High Performance Computing Modernization Program	larryd@hpcmo.hpc.mil
Bronis R.	de Supinski	Lawrence Livermore National Laboratory	bronis@llnl.gov
John V.	DelSignore	TotalView Technologies	jdelsign@totalviewtech.com
Luiz	DeRose	Cray Inc.	ldr@cray.com
Douglas W.	Doerfler	Sandia National Laboratories	dwdoerf@sandia.gov
Thomas	Epperly	Lawrence Livermore National Laboratory	epperly2@llnl.gov
David A.	Fisher	HPCMO	dfisher@ieee.org
Robert J.	Fowler	RENCI/University of North Carolina	rjf@renci.org
Jim	Galarowicz	Krell Institute - Open|SpeedShop	jeg@krellinst.org
George A.	Geist	Oak Ridge National Laboratory	sonewaldc@ornl.gov
Howard	Gordon	NSA	flash@super.org
Richard Leigh	Graham	ORNL	rlgraham@ornl.gov
Robert	Harrison	ORNL	harrisonrj@ornl.gov
Thuc	Hoang	DOE NNSA	thuc.hoang@nnsa.doe.gov
Adolfy	Hoisie	Los Alamos National Laboratory	hoisie@lanl.gov
Jeff	Hollingsworth	University of Maryland	hollings@cs.umd.edu
Marty	Itzkowitz	Sun Microsystems	marty.itzkowitz@sun.com
Curtis	Janssen	Sandia National Laboratories	cljanss@sandia.gov
Fred	Johnson	DOE	fjohnson@sc.doe.gov
Karen L.	Karavanic	Portland State University	karavan@cs.pdx.edu
Darren J.	Kerbyson	Los Alamos National Laboratory	djk@lanl.gov
Allen Davis	Malony	University of Oregon	malony@cs.uoregon.edu
Bob	Meisner	NNSA	bob.meisner@nnsa.doe.gov
Barton	Miller	University of Wisconsin	bart@cs.wisc.edu
Bernd	Mohr	Forschungszentrum Juelich	b.mohr@fz-juelich.de
David R.	Montoya	Los Alamos National Laboratory	dmont@lanl.gov
Shirley Victoria	Moore	University of Tennessee	shirley@cs.utk.edu
Jose L.	Munoz	NSF	jmunoz@nsf.gov
Wolfgang E.	Nagel	TU Dresden / Center for Information Services and High Performance Computing	wolfgang.nagel@tu-dresden.de
Rod	Oldehoeft	Krell Institute	rro@krellinst.org
Avneesh	Pant	NCSA	apant@ncsa.uiuc.edu
Abani Kumar	Patra	NSF	apatra@nsf.gov
Douglass Edmund	Post	DoD High Performance Computing Modernization Program	post@hpcmo.hpc.mil
Brian Scott	Pudliner	Lawrence Livermore National Laboratory	pudliner1@llnl.gov
Craig E.	Rasmussen	Los Alamos National Laboratory	crasmussen@lanl.gov
Daniel A.	Reed	University of North Carolina at Chapel Hill	dan_reed@unc.edu
Rolf	Riesen	Sandia National Laboratories	rolf@sandia.gov
Philip Charles	Roth	Oak Ridge National Laboratory	rothpc@ornl.gov
Martin	Schulz	Lawrence Livermore National Laboratory	schulzm@llnl.gov
Dolores	Shaffer	DARPA/Science and Technology Associates	dshaffer@stassociates.com
John	Shalf	Lawrence Berkeley National Laboratory	jshalf@lbl.gov
David Eugene	Skinner	LBL	deskinner@lbl.gov
Lauren L.	Smith	High Performance Computing, NSA	llsmit1@nsa.gov

Valerie	Taylor	Texas A&M University	taylor@cs.tamu.edu
Rajeev S.	Thakur	Argonne National Laboratory	thakur@mcs.anl.gov
Jeffrey S.	Vetter	ORNL	vetter@computer.org
Greg	Watson	IBM Research	grw@us.ibm.com
Mary	Zosel	LLNL	zosel1@llnl.gov

APPENDIX 3 – OTHER SDTPC CONTRIBUTORS

The following people contributed to this final report but were unable to participate in the workshop.

First/Middle Name	Last Name	Organization	Email
Bob	Lucas	USC/ISI	rflucas@isi.edu
John	Mellor-Crummey	Rice University	johnmc@cs.rice.edu