Workshop on

High-Productivity Programming Languages and Models

Santa Monica, California
May 17-20th, 2004

Edited by Hans P. Zima

Workshop on

High Productivity Programming Languages and Models

May 17-20, 2004, Santa Monica, California

William Carlson **William Gropp** **Ewing (Rusty) Lusk**
John Mellor-Crummey **Daniel Reed** **Vivek Sarkar**
Hans Zima (Editor)

Contents

Preface

High performance computing is a strategic tool for leadership in science and technology, providing the superior computational capability required for dramatic advances in fields such as DNA analysis, drug design, data mining, and structural engineering, as well as in many national security applications. Over the past decade, the dominance of the U.S. in this important area has been threatened by the emergence of technology problems that pose serious challenges for continued advances in this field. Recent government efforts such as the *High Productivity Computing Systems (HPCS)* program [17] and the *High-End Computing Revitalization Task Force (HECRTF)* [18] are specifically addressing these critical issues.

One of the continuing key problems in current high performance computing is the lack of adequate language and tool support. Ideally, high-level programming language design aims at enhancing human productivity by raising the level of abstraction, thus reducing the gap between the problem domain and the level at which algorithms are formulated. But high-level abstractions will be only accepted if they do not impair target code performance in a significant way. In today's dominating programming paradigm users are forced to adopt a low level programming style similar to assembly language if they want to fully exploit the capabilities of parallel machines. This leads to high cost for software production and error-prone programs that are difficult to write, reuse, and maintain.

These were the areas that were addressed by the **Workshop on High Productivity Languages and Models**. The workshop was organized by the Jet Propulsion Laboratory, California Institute of Technology, as an activity of the HPCS project *Cascade* (`http://www.cray.com/cascade`) sponsored by DARPA. Its goal was to ensure broad input from the language community in academia, government labs, and industry to initiate a strong and focused effort for the design of new high-productivity language systems and to find a consensus for a set of common research goals and strategies. The workshop was held May 17-20, 2004, in Santa Monica, California, and was structured around the following four working groups:

1. **Core Language, Compilation, and Runtime Technologies**
   Co-Chairs: Vivek Sarkar, IBM, and **John Mellor-Crummey**, Rice University
2. **Problem-Solving Environments and Domain-Specific Languages**
   Co-Chairs: William Gropp and **Ewing (Rusty) Lusk**, Argonne National Laboratory
3. **Program Development Support**
   Chair: William Carlson, IDA/CCS
4. **Programming Environments and Tools**
   Chair: Daniel A. Reed, Institute for Renaissance Computing, University of North Carolina

Each group was given a charter (summarized in the Appendix) that addressed a specific issue related to the topic of the workshop. This report is a summary of the workshop findings. Its purpose is to provide guidance for the planning of future research programs in the area of language design and related compiler, runtime, and tool technology for high productivity computing systems.

Acknowledgments

The workshop and this report would not have been possible without the dedicated and enthusiastic support of many individuals. I am particularly grateful for the advice and guidance provided by the members of the Steering Committee and for the hard work of the working group chairs in organizing sessions at the workshop and contributing to this report.

Hans P. Zima, *Workshop Chair*

Executive Summary

The four working groups involved in this workshop evaluated the state-of-the-art in programming languages, models, and tools for high productivity computing and developed a set of key findings and recommendations for advancing this important field.

There was general agreement that today's programming language, compiler, and tool technology cannnot adequately deal with the challenges of emerging high productivity computing architectures and the requirements of advanced application development. There exists a critical need for radical improvements in these areas in order to effectively utilize future parallel systems and at the same time increase the productivity of scientists and engineers. Among the major challenges to be addressed are the massive parallelism provided by peta-scale architectures with hundreds of thousands of processors, the severe non-uniformity of data accesses across memory hierarchies, and adaptive, dynamically evolving multi-disciplinary applications often developed and maintained over decades. The requirements implied by these challenges include the need for efficient support for massive multithreading, fault tolerance and robustness, adaptive resource management, intelligent automatic tuning, locality-aware computation, programming-in-the-large, and legacy code migration.

The working groups addressed all these and related issues. A summary of their most important findings follows.

Core Language, Compilation, and Runtime Technologies. This group discussed the challenges, requirements, and solutions for general purpose high productivity programming languages. The group recommends that all future languages, independent of their individual approach, should adopt a number of key properties including object orientation, support for generic programming, safety features (in particular type, pointer, and value safety), highly optimized aggregate operations on general collections, and support for a global name space. The recommendations for short term research focus on easy-to-use powerful concurrency primitives, high-level locality management, and related compiler and runtime technology. Long-term research goals include language extensibility, and support for fault tolerance, correctness debugging, and feedback-oriented automatic performance tuning in synergy with intelligent tools of the programming environment.

Problem-Solving Environments and Domain-Specific Languages. Problem solving environments (PSEs) and domain-specific languages (DSLs) will play a significant role in the future of high-productivity technical computing by providing specialized abstractions of a particular scientific domain, thus reducing the semantic gap between the way researchers think about their computational problems and the way they formulate problem solutions. Key research challenges include support for the seamless and efficient interaction of application components written in different programming languages and paradigms, and legacy code migration, which is a high-priority goal in view of the huge scientific and economic investment in these codes. Additional research efforts include telescoping languages – a technology for automatically specializing libraries for several possible uses – and application-specific code generation, generalizing the approach taken by pioneering systems such as ATLAS and FFTW. Long-term research tasks include the automatic generation of DSLs and advanced analysis methods exploiting domain-specific knowledge.

Program Development Support. This working group addressed the issue of providing support for the production of high productivity software systems. This includes tools to build such systems, configuration management systems to chronicle application history, library infrastructures to facilitate reuse, test harnesses, and program distribution mechanisms. The working group emphasized the need to consider both research and engineering in this field in order to ensure the adoption of research results by a broad community of application and system developers. The major research and engineering efforts recommended include providing application programming interfaces with a well-defined semantics, support for multi-lingual ap-

plication development, and the development of reference implementations for a complete set of tools in this area. The working group also discussed the social issues involved in the acceptance of new technologies.

Programming Environments and Tools Programming environments and tools encompass a wide range of software, from software development environments to debugging systems and performance analysis tools. In contrast to the state-of-the-art for desktop applications, the tools used to develop, debug and optimize high-performance computing applications have changed little in the past decade, resulting in inadequate support for users developing and maintaining such applications. The working group discussed the current state of the art, the challenges and opportunities associated with high-productivity computing, and possible solutions to current problems. Major challenges were seen in the areas of fault tolerance, scalability, and the requirement for intelligent tools and fully integrated tool environments. Recommendations for short term research include the specification of tool architectures and interface standards supporting integration, rapid standards-based tool prototyping, the exploration of techniques for smarter and more automated tools, and fault monitoring and self-awareness mechanisms. Long-term research goals focus on integrated software design coupling programming models and tools, multiple tool classes supporting different behavioral models, intelligent introspection and adaptation.

1 Introduction

Programming languages determine the level of abstraction at which a user interacts with a machine, playing a dual role both as a notation that guides thought and by providing directions for an abstract machine, which can be automatically translated to the execution of instructions of a physical computation device.

A major goal of high-level programming language design has traditionally been enhancing human productivity by raising the level of abstraction, thus reducing the gap between the problem domain and the level at which algorithms are formulated. But this comes at a cost: even the designers of the very first high-level language in 1957 were aware that their success depended on acceptable performance of the generated target programs. [1] The necessity to find a viable compromise between the dual goals of high level language abstraction and target code performance is a key issue for the design of any programming language. For the emerging class of peta-scale **High Productivity Computing Systems (HPCS)** architectures [2] this is a particularly critical issue due to the necessity of achieving the best possible target code performance for the strategically important applications for which these architectures are being designed. So the overarching goal is to **make scientists and engineers more productive by increasing programming language usability and time-to-solution, without sacrificing performance**.

In the remainder of this introduction we provide an overview of the current state-of-the-art in languages and their supporting programming environments, and outline new challenges resulting from dominating trends in large-scale architecture and application systems development.

1.1 The State-of-the-Art in Language and Tool Support for High End Parallel Computing

Within the *sequential programming paradigm*, a steady and gradual evolution from assembly language to higher level languages took place, triggered by the initial success of FORTRAN, COBOL and Algol in the 1960s. Together with the development of techniques for their automatic translation into machine language, this brought about a fundamental change: although the user lost direct control over the generation of machine code, the progress in building optimizing compilers led to wide-spread acceptance of the new, high-level programming model whose advantages of increased reliability and programmer productivity outweighed any performance penalties.

Unfortunately, no such development happened for *parallel architectures*. Since the 1990s, the integration of commercial off-the-shelf processing and memory components allowed the synthesis of large high end computing systems at affordable cost, either through custom MPPs or commodity clusters. Exploiting these systems, some of which have a theoretical peak performance in the teraflops range, has proven to be difficult: many important applications reach an efficiency of barely 10%.Furthermore, there are many examples where a small modification of a parallel program (such as a change in the decomposition of data to processors) can affect its performance on a large machine by several orders of magnitude. This has led to a situation where users have been forced to adopt a low-level programming paradigm based upon a standard sequential programming language (typically Fortran or C/C++), augmented with message passing constructs. In this model, the user deals with all aspects of the distribution of data and work to the processors, and controls the program's execution by explicitly inserting message passing operations. Such a programming style is error-prone and results in high costs for software production. Moreover, even if a message passing standard such as MPI is used, the portability of the resulting programs is limited since the characteristics of the target architectures may require extensive restructuring of the code. Although there were numerous attempts in the 1990s to develop higher-level parallel programming languages, such as High Performance Fortran (HPF)

[1] John Backus: "...It was our belief that if FORTRAN ... were to translate any reasonable "scientific" source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger..."

[2] The term "HPCS" will be used in this Report to address the systems developed in the DARPA-funded program with the same name, but also including any other efforts leading to high productivity peta-scale systems in the next decade.

and its variants [28, 32], none of them succeeded in gaining wide-spread support in the community, for reasons that include language shortcomings, structural weaknesses of traditional parallel architectures and related performance problems as well as lack of persistent funding for language, compiler and runtime system development.

The state-of-the-art in programming environments for the support of the development, debugging, performance analysis, and maintenance of high end computing applications is similar to that for languages. While integrated software development and testing environments as well as high-level scripting languages are generally used for creating commercial desktop applications, tools for high end computing applications have only slowly evolved over the past decade and are little used by practitioners. Most of these software tools are slightly modified versions of those used on sequential systems and remain focused on programming in the small, not keeping pace with the requirements of large-scale advanced applications.

1.2 New Challenges and Requirements

The rising scale and complexity of the emerging petascale architectures and large-scale applications pose a set of new **challenges** for programming languages and environments, including:

- **Large-Scale Parallelism**
  Future architectures in the petaflops range (10^{15} operations per second) will have tens of thousands processing units, providing massive parallelism of a hierarchical, multi-level structure.

- **Non-Uniform Data Access**
  Architectures will be characterized by severe non-uniformities in data accesses. It will take 1000+ cycles for a processor to access data from memory, compared to $10 - 20$ cycles for an L1 cache access. Access to remote memories will suffer even larger latency. Additionally, the bandwidth available at different levels of the memory/interconnect hierarchies will be substantially different.

- **Large-Scale Applications**
  HPCS applications will increase significantly in size and complexity over the next decade as researchers develop full-system simulations encompassing multiple scientific disciplines. Such applications, which will often be developed by geographically distributed teams, will be built from components written in different languages and using different programming paradigms.

- **The Time-Scale Mismatch**
  Applications are often developed and used over decades, while the hardware and software systems on which they run may change every few years. This mismatch in the timescale leads to a range of difficult problems involving the porting of legacy codes under stringent performance requirements. This problem also inhibits introduction of new architectures since for economic reasons they must perform well on existing applications.

These and related challenges imply a set of key **requirements** that new language systems must address, beyond the general goals discussed above. Requirements include efficient language system support in the following areas:

- **Multithreading**
  Languages must provide explicit as well as implicit mechanisms for creating and managing systems of parallel threads that exploit the massive parallelism provided by future architectures.

- **Fault Tolerance and Robustness**
  Although the mean time between failures (MTBF) for individual components in an HPCS architecture is high, the large overall number of components means that failures will occur and need to be dealt with in order to avoid malfunction of the whole system or a large part thereof. System size will also make

centralized handling of fault tolerance impractical, implying the need for a hierarchical decentralized approach.

- **Adaptivity and Automatic Tuning**
  Another consequence of system size as well as the dynamically changing nature of applications is the need for adaptive resource management and automatic or semi-automatic approaches to performance tuning of individual applications as well as the overall system, under a given objective function.

- **Locality-Aware Computation**
  Future architectures may provide hardware-supported global addressing; however, latencies and bandwidths across different memory levels will vary by many orders of magnitude. This makes locality-aware computation a necessary requirement for achieving optimum target code performance.

- **Programming-in-the-Large**
  Emerging advanced application systems will be characterized by a dramatic increase in size and complexity. It is imperative that HPCS language systems support component-based development allowing the seamless interaction of different languages, programming paradigms, and scientific disciplines in a single application.

  The development and use of applications over an extended period of time moreover implies the necessity of life cycle support for all phases and across different versions.

- **Portability and Legacy Code Integration**
  Taking into account the huge investment in applications, their long lifetime, and the frequent change in the supporting hardware platforms makes the automatic or semi-automatic porting and integration of legacy codes – from Fortran 77, C, and C++ all the way to Fortran 95, MPI, CoArray Fortran and UPC – a high-priority requirement. Under the necessary constraint of retaining high performance this is a demanding task requiring a sophisticated approach and intelligent tools.

1.3 Motivation and Goals of the Workshop

The major motivation for organizing this workshop was the belief of a large part of the community that today's programming languages and models as well as current compiler technology are not adequate to deal with the challenges of 2010 architectures and application requirements.

The goal of the workshop was to bring the language community together to initiate a strong and focused effort for the design of new high-productivity language systems and to find a consensus for a set of common research goals and strategies. We use the term *language systems* here to emphasize the fact that this effort is not restricted to the discussion of new (general-purpose) HPC languages but also includes other approaches such as domain-specific languages and problem-solving environments as well as a discussion of the role of programming environments in this context.

More specifically, what is needed is the development of new languages together with the compiler, runtime, and tool technology required to achieve the dual goals of productivity in the sense of short time-to-solution and the highest possible target code performance. Key technical topics in this context include

- Major design features of high-productivity/high-performance programming languages for peta-scale systems, such as multithreading, locality- awareness, generic programming, seamless integration of legacy codes, and performance metrics

- New compilation and runtime technology for these languages

- Problem-solving environments and very high-level language approaches

- Support for multi-paradigm and multi-language program development

- Programming environments supporting fault tolerance, high-level debugging, performance analysis and feedback-oriented compilation in the framework of largely autonomous system operation.

There have been a number of promising research developments that may point the way to successfully address many of the problems outlined above. Examples include new compiler technology allowing HPF-based codes to reach a significant fraction of peak performance on the Earth Simulator [9, 47], techniques for the automatic specialization of libraries [29, 38], software component architectures providing efficient support for programming-in-the-large [3], automatic tuning systems such as ATLAS [44] and FFTW [23] that transform code templates into optimized target code for a range of architectures, knowledge-centric software engineering exploiting the properties of a specific application domain to improve analysis, program comprehension, and performance [41], and initial designs of intelligent, introspection-based software tools for the support of fault tolerance and performance tuning [15].

1.4 Report Outline

This Report is structured as follows. The following four sections represent the insights and results of the discussion in the Working Groups. Section 2 focuses on language, compilation, and runtime issues related to general-purpose HPCS languages. An alternative approach – problem-solving environments and domain-specific languages – is discussed in Section 3. Section 4 deals with issues related to program development while Section 5 addresses the general topic of programming environments and tools supporting HPCS languages.

The Appendix provides details regarding the members of the Workshop Committee, the Workshop Agenda, the Working Group Charters and participants, and the list of attendees.

2 Core Language, Compilation and Runtime Technologies

Co-Chairs: Vivek Sarkar, IBM, and **John Mellor-Crummey**, Rice University

The Working Group on Core Language, Compilation and Runtime Technologies discussed challenges, requirements, and possible solutions for **general purpose HPCS languages**.

In this section, we begin by summarizing a set of assumptions and related properties that we believe will be common to all general-purpose programming languages in the 2010 timeframe. This is followed by a discussion of roadmaps for the solution of key problems in this area.

2.1 Assumptions

Though it is expected that different vendors will explore different system design tradeoffs in dealing with large scale parallelism and non-uniform data access, there are certain assumptions that this Working Group believes will be common to all approaches for building HPCS systems and applications in 2010. For convenience, we list these assumptions below:

1. We foresee that current parallel programming language and compiler technologies will be unable to address the productivity demands for exploiting future HPCS systems, so *new programming models and languages* will be necessary in the 2010 timeframe.

2. The focus on system value will be on *productivity*, thereby leading to *new productivity and usability metrics* for HPCS application development.

3. *Performance metrics* will continue to be important for measuring the value of a system, and will feed into productivity metrics through measurements such as *time to solution.*

4. While *single thread performance* will continue to be important in the 2010 timeframe, our focus is on dealing with large scale parallelism and large non-uniformities in data access. There are market forces in the computer industry that ensure that sufficient attention is paid to single thread performance through incremental technology improvements. Therefore, we will not focus on single thread performance issues other than those that are relevant to parallelism and locality. However, *single thread semantics* will be a key research area in enabling scalable parallel performance in HPCS applications. Some of the important research topics include models for exception handling and floating point operations. In both cases, relaxations of strict sequential semantics are necessary to ensure that they do not pose obstacles to scalable parallelism.

5. On the application side, we expect that future HPCS languages and compilers will need to support *multiple parallel programming paradigms* including data parallelism, control parallelism, divide-and-conquer, streaming, and producer-consumer. A major challenge is to enable these techniques to co-exist in the same application. In addition, a *variety of numerical methods* will be used in the HPCS applications including dense matrix, sparse iterative, sparse direct, spectral, adaptive block structured, particle methods, and unstructured grids.

As a consequence of these assumptions, there are some common attributes that we expect to see adopted as goals for all new programming models for HPCS systems in the 2010 timeframe. None of these attributes is viewed as controversial by this Working Group, though there are certainly open research problems relating to their efficient implementation on the large-scale HPCS systems that will be built in the 2010 timeframe:

1. Support for *object-orientation.* The productivity benefits of object-oriented programming — including modularity, encapsulation of state, separation of concerns, inheritance, and polymorphism — are well understood.

2. Support for *generic programming.* This will include at least the use of polymorphism as in object oriented programming, and may also include type parameters. For instance, we expect to see one code template for a common operation such as matrix multiply, abstracted with respect to the sparseness of the matrix, its base type, and the underlying data distribution. Generic programming will contribute to productivity by reducing the cost of software maintenance, and increasing opportunities for code reuse.

3. Support for *safe programming.* We expect language design to progress so that large classes of errors will be ruled out by a combination of static and dynamic checks. The classes of errors to be checked will include:

   - Type errors — no operation will be performed on an object for which it is not defined
   - Pointer errors — no dereference will be permitted of an unbound pointer (such as a null or "bad" pointer)
   - Array indexing errors — no array index operation will be permitted with an index value that is out of array bounds
   - Initialization errors — no read access will be permitted on data that has not been previously initialized
   - Branching errors — no branches will be permitted to undefined code locations

   These safety checks will improve productivity by enabling errors to be caught earlier during compilation (for static checking) and unit test (for dynamic checking) phases of the software lifecycle, rather than in debugging of production deployments. We expect the scope of errors that will be caught by static and dynamic checking to progressively increase with improvements in validation and verification technologies, so as to eventually include higher-level semantic errors such as data races, and writes to immutable data.

4. Support for *aggregate operations on large collections.* We expect HPCS languages to support general types of collections, including arrays, sequences, sets, and graph-based data structures, and to provide efficient implementations of aggregate operations such as reduction, parallel prefix, and other collective communications as well as point-wise operations. Aggregate operations are used extensively in current HPC programming models such as MPI and OpenMP, and improve productivity by providing concise notations for operations whose implementation may require complex inter-processor interactions.

5. Support for *programming in the large.* We expect support for modern, type-generic module and component systems to be adopted in future HPCS programming languages. Additionally we expect the availability of a rich set of *standard libraries* (*e.g.,* collections, I/O libraries, concurrency libraries, etc.) to be widely available in the HPC domain. Finally, we expect to see standards for inter-language interoperability to emerge (such as the models being proposed in the CCA forum), allowing modules written in multiple languages to be deployed independently and to interoperate with each other, in a high performance and massively parallel context. A supporting environment for programming in the large will improve productivity by enabling common components to be shared more easily among multiple applications.

6. Support for a *global name space.* We expect to see all future HPCS programming models offer some support for a global name space, even though the latency and bandwidth for accesses to different locations in this name space will be highly non-uniform. Support for a global name space can improve productivity by simplifying the implementation of parallel algorithms and data communication.

7. Support for a *managed runtime system.* We expect the runtime system of future HPCS languages to evolve into a managed runtime system that can support memory management, thread management, sharing of runtime facilities and objects across multiple languages, reflection, and mobility of control

and data (including seralization and checkpointing). A managed runtime system improves productivity by relieving programmers of the burden of implementing customized runtime services for their own applications.

8. Support for *portability.* We expect new applications to be written in ways that will ensure *functional portability* across different vendor systems. *Performance portability* is a harder problem since different HPCS systems may have features that favor different classes of applications. However, the focus on *performance transparency* in the new programming models is expected to reduce the effort required for performance portability compared to current programming models.

9. Support for *separation of concerns.* We expect to see separations of concerns across multiple dimensions:

   - Separation of concerns between application scientists and system experts. We expect end-user programming, performance and debugging models and tools to evolve so as to enable application scientists to specify information about the underlying domain and the intended computation in terms familiar to them.

     Side by side, we expect system experts more familiar with the nuances of compiler design, computer architecture, memory caches, distribution, parallelism and memory consistency to express ways in which this application-specific information can be harnessed to improve performance on the target system.

     Thus we expect to see a separation of concerns across *function* (what needs to be done), *method* (how it should be done, sequencing and parallelism) and *location* (where it should be done).

   - Separation of concerns between policies and mechanisms, especially in the runtime system

   Separation of concerns can improve productivity in general by enabling each stakeholder in HPCS application development to focus their attention on only those aspects/concerns in the software that are pertinent to their role. In particular, the goal of a 10× improvement in development productivity can only be realized if non-system-experts can contribute directly to the construction of production-quality HPC applications with little or no involvement from system experts.

2.2 Research Roadmap 2005–2009

The previous section provided a discussion of assumptions that we believe to be essential for all future general-purpose HPCS languages. Here we propose a list of specific research areas that will need to be addressed in the near term to satisfy key requirements for high productivity.

2.2.1 Concurrency Primitives

HPCS architectures will have ten thousands of processing units providing parallelism at potentially different levels of granularity. Past research has clearly demonstrated that compiler-controlled automatic parallelization of sequential algorithms does not in general yield satisfactory results. As a consequence, providing **easy-to-use massively parallel programming constructs** that are capable of generating the large scale of parallelism needed by these systems is a key research area. Such constructs will also be geared for use in *adaptive parallelism* and *hierarchical parallelism* contexts.

Some of the limitations in current and past programming models that will have to be addressed by the new concurrency primitives are as follows:

- *Granularity*: while current programming models exhibit coarse-grained parallelism (e.g., MPI processes or OpenMP tasks), they are ill suited to generating the massive scale of parallelism that will be required by future HPCS systems.

- *Coordination and synchronization*: current programming models use low-level constructs such as locks and synchronized messaging to enable parallel threads of control to coordinate their execution. The complexity inherent in using these constructs greatly limits the productivity of HPCS application development.
- *Compositionality*: current programming models, especially the Single-Program-Multiple-Data (SPMD) model, make it difficult to create and deploy HPCS applications that represent composition of components with different forms of parallelism, *e.g.,* the combination of data-parallel algorithms with divide-and-conquer algorithms.

2.2.2 Locality Primitives and the Memory Model

The non-uniformity of data accesses across different levels of the memory hierarchy makes locality-aware computation an important consideration with regard to performance. Many algorithms display strong locality properties.

Even though we expect HPCS languages to provide a global name space, a key challenge will be to combine the convenience of the global name space with performance transparency that captures non-uniformities of data access so as to simultaneously enable high performance and high productivity. Past experience has shown that providing the illusion of uniform data access can be counter-productive, because it may obfuscate the performance tuning necessary for HPCS applications. While optimizing compilers can greatly aid in performance tuning, it is unrealistic in general to expect compiler technologies to automatically implement the illusion of a uniform data access model on a large-scale non-uniform HPCS system in a way that delivers scalable performance.

One aspect of this problem has been addressed by languages such as High Performance Fortran (HPF) and its variants [28], which allow the specification of data distribution and alignment as well as data/thread affinity in a declarative manner. The core research problem here is to generalize the HPF approach to be applicable to more complex data structures than dense Fortran-style arrays, and to more general models of parallelism than offered by SPMD. A promising way to deal with this issue is via type modifiers and type systems that capture notions of local/remote storage classes.

Of particular importance is efficient compilation and runtime support for aggregate operations under locality constraints, and the efficient implementation of accesses to individual components and substructures.

Another research area will be in high-level semantic integration of data access operations with concurrency primitives, so as to hide the low-level semantic complexities of "memory consistency models" and "data races" from the programmer while still exposing their performance implications. Memory models and data races are known to be significant inhibitors to productivity in current parallel programming models. We expect future programming models to combine ease-of-use with a robust semantics, so that there is no need to expose programmers directly to the notion of a memory model. However, a memory model will be included in the implementation specification of the programming language so as to provide a precise contract between the language environment and the underlying system hardware and software.

2.2.3 Optimizing Compiler Technologies

There has been significant progress in optimizing compiler technologies for vectorization, parallelization, and locality optimization in the last three decades. We expect HPCS compilers to both leverage and improve these technologies in the 2010 timeframe, not necessarily as the dominant mode of extracting performance but as a key enabler for productivity through optimization frameworks that address low-level system details thereby leaving the programmer to only worry about higher-level algorithmic aspects of performance visible in the Abstract Execution Model.

Some areas in which improvement over existing optimizing compiler technologies will be needed in the 2010 timeframe include:

- Analysis and transformation of explicitly parallel programs
- Analysis and optimization of (implicit or explicit) communication and synchronization
- Management of locality and data movement
- Extension of vectorization, parallelization and locality optimization techniques to irregular codes

2.2.4 Abstract Execution Model, Performance Transparency, and Runtime Issues

Research in runtime systems focuses on the definition of an Abstract Execution Model (*AEM*), as introduced in Section 2.1. The goal of the execution model is to provide high-level abstractions of features for expressing and managing parallel activities as well as data and thread locality, combined with a performance model. It can serve as a target for defining the functional semantics of new programming languages as well as a framework for reasoning about performance and other non-functional characteristics of applications. The abstract execution model will provide a key foundation for supporting performance transparency in HPCS programming models, while still hiding low-level system details and parameters.

The Abstract Execution Model should enable the programmer to be aware of the high-level cost (memory, concurrency, communication) of various language constructs by providing a reasonably accurate algorithmic performance model. More detailed performance exploration can be performed by tools that show the user how the Abstract Execution Model gets mapped to a specific architecture via performance feedback from actual executions. Finally, if there is a priority to obtain increased performance on a specific architecture even at the cost of productivity, "wide-spectrum" languages can be used to enable users to program at levels of abstraction below that of the Abstract Execution Model – this may be warranted for frequently used high-performance components and libraries.

Some of the key research topics that will need to be addresssed in the runtime support for HPCS languages are as follows:

- Adaptive optimization and parallelism — adapt optimization and parallelism to available resources in the system and various runtime configuration parameters in the application
- Dynamic compilation — enable application code to be compiled and recompiled at runtime, especially in combination with adaptive optimization and parallelization
- Optimization trade-offs between resource utilization (load balancing) and locality (data affinity)
- Scalable memory management — allocation and garbage collection
- Synchronization — automatic lock management
- Dealing with memory contention
- Support for parallel I/O
- Support for data persistence and migration
- Support for fault tolerance and migration

2.3 Research Roadmap 2010–2015

2.3.1 Extensible Languages and Specialization

In addition to a single base language for programming in the large, a key research problem is the design of extensible languages that enable the definition and optimization of user-defined primitives in specific domains. There is a productivity benefit in the using extensions of a common core language instead of multiple domain-specific languages whenever possible, so as to promote portability and reuse of code. The extension techniques may range from modules, libraries and components to first-class language constructs. In all these cases, research is needed in how to design languages that support specialized implementations of domain-specific types and frameworks that deliver comparable performance to what would be obtainable from domain-specific languages. Examples of known domains that would benefit from such extensions include streaming computations, matrix data types, and tensor computations.

2.3.2 Automated Performance Tuning

With the dramatic increases in complexity of system architectures, system software and application programs over the last decade, the ability for application programmers to realize the tremendous performance potential of today's high-end computing environments has diminished, and will dramatically worsen as we scale to the HPCS 2010 systems.

Going beyond adaptive optimization and parallelization, a key research area for the next decade will be applications that are self-tuning, where the process of mapping to large numbers of processors and porting from one platform to another is automated by a suite of tools. In the narrow domain of scientific libraries, self-tuning systems such as ATLAS have demonstrated promising results in managing the memory hierarchy and obtaining portable high performance across a range of platforms. What is needed is an extension of the concepts in ATLAS and other self-tuning libraries to apply to more general application code and go beyond the memory hierarchy to perform optimizations across processors.

Such a strategy will demand changes to the languages, compiler optimization, code generation, run-time systems, performance monitoring and perhaps even architectural support in these large-scale parallel systems. At the language level, we can provide linguistic mechanisms for specifying parameterized code or algorithm variants, where underlying code generators and run-time systems empirically select the appropriate variants or parameter values. In general, we need a broad and integrated suite of tools that exploits the sheer computational, memory, and I/O capabilities of these large-scale systems to derive high performance and portable implementations of the applications of 2010.

With respect to compiler optimization, we must separate the mechanical and well-defined portion of the compiler that performs analysis and code transformations from the more machine-specific and fragile portion that searches the set of transformations to be applied and determines values for the parameters of the optimization, such as the tile size for a loop nest. The optimization search space is becoming increasingly complex, and an area of recent interest is replacing the somewhat ad hoc compiler heuristics commonly used to explore the optimization search space with either the empirical techniques of ATLAS or machine learning approaches. While both approaches are promising in terms of improving the effectiveness of optimization, we must use them carefully so that they can scale to optimizations for large parallel systems. For this purpose, compiler models, even if imprecise, can be used to prune large portions of the optimization search space and provide domain knowledge to the machine learning techniques.

Underlying performance monitoring tools and performance databases can guide the optimization search, and lead to programs whose performance and portability improves over time. This will also involve a collaboration with hardware performance monitoring support in the architecture. For example, the granularity of tracking the caching behavior of individual address ranges, as in the Itanium-2, is far more useful to the optimization process than simply providing an overall measure of cache misses.

2.3.3 Fault Tolerance

Fault tolerance is a key requirement for HPCS architectures, addressing the important issues related to reliability in future large-scale systems. This advanced research topic deals with program constructs for redundant data and control, as well as constructs for error recovery based on redundant data and control information combined with the ability to recover execution from a previous checkpoint. Techniques such as deterministic replay and reverse execution can provide further exploratory options in this space. While there is a large body of work in the area of fault tolerance in loosely-coupled distributed systems and transactional systems, little or no attention is currently paid to the support of fault-tolerance in tightly-coupled HPC systems.

2.4 Radical New Ideas

The previous subsections discussed topics suitable for an aggressive research roadmap for the Core Language, Compilation and Runtime areas in future systems. Here we briefly outline a few topics that offer the promise of being "radical" new ideas, compared to the topics outlined in the previous subsection:

1. *Use of Functional and Declarative Languages for High Performance.* This topic is intended to address the need of creating large amounts of parallelism for future HPCS systems. Functional and declarative languages have been very effective at generating large amounts of parallelism, but they have also been very challenging to implement with high performance on large-scale systems. It would also be desirable to combine imperative and declarative concepts in a single language that is able to generate large amounts of parallelism, while still retaining the conveniences of an imperative language.

2. *Use of Transactional Programming Models.* An extremely difficult and error-prone area of parallel programming is management of complex updates to shared data. Transactional techniques hold great promise in providing a semantically clean approach to this problem, while also providing a foundation for fault tolerance. Of the four ACID properties (Atomicity, Consistency, Integrity, and Durability) the durability property need not be enforced as strongly in an HPCS programming model as in traditional transactional systems. In contrast, the performance expectations from HPCS programming models are much stronger than in transactional systems. For these reasons, it is not possible to simply use a commercial relational database system to support transactional programming in HPCS applications.

3. *Programming by Example.* The underlying hypothesis in this topic is that it is easier for the user to describe paralell execution of specific instances of a parallel program than to define a parallel program in its full generality. In this case, a programming environment can be developed with appropriate tools to capture specific instances, and then synthesize the complete parallel program by induction over the examples coupled with queries to the user about corner cases. If this approach is successful, it could result in a huge improvement in development productivity for HPCS applications.

2.5 Infrastructure Roadmap

Many good research ideas in the area of languages, compilers, and tools over the last decade have never made it into practice. There are several reasons for this. The primary reason is that progress in this area is labor intensive, requiring many years of infrastructure development and significant funding to realistically deploy a new language or develop compilers for radically new architectures. In addition, there is almost no way for one research group to leverage the work of another since the effort required to replicate the work of another group would be prohibitive.Thus, since deploying software technology is dependent on a combination of the efforts of distinct groups, progress in this area has been severely limited.

In the late 1990s, an effort to address this problem led to the National Compiler Infrastructure program, which funded the Stanford SUIF compiler group to build an open-source platform for compiler and languages research. This program led to some successes, in that many research groups that did not previously have compiler infrastructure began using SUIF, and several satellite research programs centered around SUIF developed from former members of the SUIF group. Nevertheless, the impact of the National Compiler Infrastructure effort was limited, and at this time, there is no long-term effort to maintain and extend this infrastructure. SUIF was largely a single-investigator activity, and key members of the research community who already had their own research infrastructures did not participate in its development and deployment. It is clear that future productivity tools will become increasingly complex, and will exceed the capabilities of a single organization. What is needed is a suite of productivity tools for which there is community participation, so that the tools can evolve to incorporate new research ideas and provide a public forum for testing and using these new ideas. A community-led infrastructure, with a critical mass of participation from the high-end computing languages and compiler research community, would increase the breadth of capabilities and ensure long-term impact. Finally, the ultimate customers of this infrastructure (such as government labs and industry users of HPC technologies) need to collaborate with HPC vendors to define a mutually beneficial business model that can provide HPC vendors with sufficient incentive to invest in productizing the new infrastructure.

There are some promising examples of community-led infrastructure in open-source software projects such as the gcc compilers for C and C++, the Jikes Research Virtual Machine for Java, and the Eclipse platform for integrated tools. However, none of these open source projects have devoted attention to addressing the special requirements of HPCS systems. New funding will be necessary to build a community-led infrastructure that is needed for HPCS systems (whether built from scratch or by extending existing open-source projects). The eventual goal would be an HPCS development environment that integrates programming, debugging, visualization, and performance analysis of large-scale parallel programs in a common framework.

In summary, it is clear that progress in high-level languages and compilers will require cooperation and organization within the research community and government support of these activities to face the challenges of 2010 systems.

3 Problem-Solving Environments and Domain-Specific Languages

Co-Chairs: William Gropp and **Ewing (Rusty) Lusk**, Argonne National Laboratory

Domain-specific languages (DSLs) and problem solving environments (PSEs) will play a significant role in the future of high-productivity technical computing. Though there are significant opportunities for general-purpose high-productivity languages to raise the level of abstraction at which code is written, their *general purpose* nature limits the extent to which they can be expected to accommodate the specialized abstractions of any particular scientific domain. Thus, the development of new general purpose-languages can be expected to *reduce* but not eliminate the semantic gap between the way researchers think about their computational problems and what it takes to implement them in a general-purpose language. DSLs and PSEs provide the primary means by which the semantic gap can be closed.

3.1 Key Challenges of Long-Term Application Development

Perhaps the most difficult challenge facing application development is the mismatch in the timescale of application and computer hardware. Many applications are developed and used over a period of decades; hardware rarely lasts more than three years. Applications development thus must strive for both portability and efficiency across a wide range of architectures.

Significant productivity gains in the development and maintenance of large scientific application codes will be realized when developers are not only able to use high-level abstraction but also trust that the implementation of the language will generate high-performance code. Unfortunately, developers have learned to program defensively and to the common denominator of languages and language features in order to preserve the value of their software across different platforms and over time. The use of high-level language abstractions often introduces unacceptable performance penalties. The abstractions are often unevenly supported across language implementations and computing platforms. Fortran array notation and copy-in/copy-out semantics are examples [12, 4, 45, 16, 40, 2].[3] High-optimization compiler options sometimes produce incorrect or even unrunnable codes, and compile times can balloon far out of proportion to the size of the input source code. Standards for high-performance programming languages are promulgated at a level of complexity and revised at a rate that outstrips the capacity of the market for the language to support such change. Complete, correct, and up-to-date implementations of the latest version of Fortran, for example, are very hard to find, and quality third party development tools are in short supply. Direct input and feedback from users is lacking. Much of the current progress in high performance scientific computing is occurring in spite of rather than being facilitated by programming languages. A key challenge for High Productivity Programming Languages will be to correct issues that currently discourage application groups from taking advantage of high-level language abstractions.

An additional problem faced by applications is the need to exploit multiple programming languages and models. At the simplest, an application may use Fortran for numerically-intensive kernels and C for portable file processing. The use of scripting languages such as python is also becoming more common. Another reason to use multiple programming models is to enable incremental porting to new programming models. For example a parallel code that is currently using message-passing may want to exploit a new programming model (such as a global name space model) but without requiring that the entire application be rewritten. The lack of omnipresent, scalable and accurate performance and correctness debuggers also impedes application development.

In considering the key challenges, there are two major branches: existing applications (also called legacy applications) and new long-term projects. (A third branch, new short-term projects, was not considered by this Working Group.) Application groups can take advantage of domain-specific languages and problem

[3]Fortran array notation, typically well-implemented on vector systems, has been so poorly implemented for microprocessor-based systems that a number of large application groups have deprecated its use from their formal coding standards. Note that [2] provides a counter example but its publication in 1995 precedes the publication in 1999 of [16] by one of the same authors in which Fortran 90 array notation is now discouraged.

solving environments to address both of these challenges. One example of the use of these methods is the Weather Research and Forecast Model (WRF) application.

The Penn State/NCAR Mesoscale Model (MM5) is a large community weather model originating in the 1970s. The model was adapted to scalable distributed-memory systems in the mid-90's using a strategy that involved developing a custom source translator that is distributed with MM5 and invoked automatically when the code is built for parallel systems [35, 33]. MM5 continues to be maintained and distributed in its original non-MPP form. The Weather Research and Forecast Model (WRF) [46] was officially released as the successor to MM5 in May, 2004. Designed and implemented completely from scratch, WRF addresses performance and portability through its design: a software hierarchy with well-defined interfaces between scientist-developed code and the other layers of the framework. This allows scientists to develop "Model Layer" components using familiar, predominantly Fortran 77, idioms without concern for parallelism and other infrastructure issues.

Notably, this software framework-based approach satisfies portable performance and developer productivity requirements but relies very little on base language abstractions. However, neither a framework-based approach nor one based on currently available language-abstractions is completely satisfactory. For example, neither provides a way to ensure optimal per-processor performance on both vector- and microprocessors from the same source code. A number of studies have shown the best choice of array index and loop nesting order on vector processors is not the best choice for microprocessors [5]. Yet such decisions must be made early in a code's development and, once made, are very difficult to change. At present, the best one can do is to chose a compromise between storage and loop-nesting order. Performance portability and overall productivity would benefit enormously from a source-to-source translator or efficient language abstraction that automatically converted array index and loop nesting order. This is only one example of the potential benefits from powerful source-to-source translation tools, even for "future-legacy" codes being developed today, such as WRF.

Many efforts are underway to develop flexible and programmer-definable language translators based on compiler-level knowledge of the user's program [19].

A key issue is the need to achieve excellent performance. Experience has shown that this requires the attention of expert programmers to low level details, even using assembly language when necessary. While much of the concern in the community is focused on parallelism, the reality is that it is still too hard to get good single processor performance. This is best shown by the difference in performance on dense matrix-matrix multiply written in simple and clean Fortran and achieved by routines drawn from careful implementations of the level 3 BLAS [44]. A related issue is whether a chosen programming model or language provides performance that is robust or fragile against small changes in the code.

3.2 Problem-Solving Environments (PSEs)

3.2.1 Characterization

Problem-solving environments (PSEs) are problem-oriented software frameworks that provide the functionality needed for solving a target class of problems without requiring specialized knowledge of the underlying hardware/software system. PSEs serve as an easy-to-use interface to HPCS resources, assisting users with knowledge about the target domain and the formulation of problem solutions, and allow rapid prototyping of new ideas.

Component Architectures and PSEs. For the purposes of this Workshop, software component architectures can be thought of as tools for addressing certain productivity-related issues in large-scale software development efforts, particularly the composition of applications from many modules ("components") and the reuse and interoperability of those components. Component architectures can be classed with Problem Solving Environments because they provide an environment which facilitates the construction of PSEs, either intentionally, or by accumulation — a component model being so widely used that much of what is needed to assemble broad classes of scientific application is available "off the shelf".

3.2.2 Overview of Key Systems

There are many problem-solving environments in wide use. In the domain of scientific computing, the following is a sampling of some of the most widely used or best known systems:

SCIRun (`software.sci.utah.edu/scirun.html`) is a package for building sophisticated, fully-integrated problem-solving environments, including computational steering and visualization. BioPSE is an example of an application-specific PSE built using SciRun.

PETSc (`www.mcs.anl.gov/petsc`) is a set of tools/solvers for the scalable (parallel) solution of scientific applications modeled by partial differential equations.

Chombo (`seesar.lbl.gov/anag/chombo`) A set of tools/solvers for implementing finite difference methods for the solution of partial differential equations on (parallel) block-structured adaptively refined rectangular grids. Both elliptic and time-dependent modules are included.

Fluent (`www.fluent.com`) Fluid flow and heat transfer modeling software suited to a wide range of applications.

The Common Component Architecture (CCA, `http://www.cca-forum.org`, [3, 10]) is a component architecture specially designed to meet the needs of the high-performance scientific computing community.

Cactus (`http://www.cactuscode.org` [1]) is a PSE with some of the properties of a component architecture, in that it is possible to "plug in" code modules ("thorns") at relatively high levels in the simulation workflow. It was originally developed for numerical relativity simulations, but has been used in a fairly broad range of domains.

CUMULVS (`http://www.csm.ornl.gov/cs/cumulvs.html`, [30]) is an environment for visualization and steering of parallel scientific computations, which also provides fault tolerance (user-directed checkpointing) and task migration capabilities.

ESMF Earth System Modeling Framework [20] is a NASA/HPCC funded collaborative project to develop a "high-performance, flexible software infrastructure to increase ease of use, performance portability, interoperability, and reuse in climate, numerical weather prediction, data assimilation, and other Earth science applications."

NWChem (`http://www.emsl.pnl.gov/docs/nwchem`) is a computational chemistry package that provides methods to compute the properties of molecular and periodic systems using standard quantum mechanical descriptions of the electronic wavefunctions or density.

Netsolve (`http://www.cs.utk.edu/netsolve`) is a client-server system that enables users to solve complex scientific problems remotely by accessing distributed hardware and software computational resources. Netsolve searches for appropriate resources on a network, chooses the best one available, and returns the answers to the user.

KeLP (`www.cs.ucsd.edu/groups/hpcl/scg/kelp`) Kernel Lattice Parallelism [21] is a C++ library that provides abstractions to separately manage data layout and motion for dynamic block-structured applications.

3.2.3 PSEs vs. General Purpose Languages: Pros and Cons

Problem-solving environment technology can allow researchers to work in a manner more consistent with the scientific process and removed from the mechanics of utilizing computational resources, thereby providing for productive utilization of valuable high-performance computing resources [13].

However, PSEs come with several disadvantages. The two most important are the need to fully support the PSE (including debugging) and the fragility to changes in the domain of applicability, limiting the ability to extend an application beyond the domain originally envisioned for the PSE. Other issues include problems interoperating with other tools caused by a requirement that the PSE be in complete charge of the application and the ease with which a poor PSE can be designed.

3.3 Domain-Specific Languages

3.3.1 Characterization

For this report, a domain-specific language is first a language, by which we mean that it has a well-defined syntax and semantics. It is domain-specific because it emphasizes a particular application domain.

3.3.2 Overview of Key Languages

Ellpack (`www.cs.purdue.edu/ellpack/ellpack.html`) is a high-level language, designed as an extension of Fortran 77, for the solution of elliptic boundary value problems. First released in 1985, it is one of the oldest PSEs in numerical computing [39].

Spiral (`http://www.spiral.net/`) The SPIRAL code generation system is focused on the automated, tuned code generation of those signal processing algorithms that can be labeled performance critical, such as the discrete Fourier and discrete cosine transforms (DFT/DCT), that is, algorithms that are ubiquitously used in many scientific applications. The intermediate code is generated in SPL and then implemented in either Fortran or C, and then iteratively optimized using advanced compilation techniques for performance that is highly tuned to the target architecture.

StreamIt (`http://www.cag.lcs.mit.edu/streamit/`) is a programming language and a development/ compiler infrastructure designed for high-performance streaming applications such as streaming media, router software, and distribution base stations. StreamIT supports the programming of large-scale streaming applications in either graphical or textual forms. The system is designed to generate code for a variety of target architectures, including COTS processors, multi-clustered architectures, and grid-enabled processors.

SQL The Structured Query Language was created by IBM and adopted by ANSI in 1986. It is used as the programming language of relational databases such as Oracle, DB/2, MySQL, Sybase, Informix, Microsoft SQL server, Postgres and many others. SQL is not a portable standard (many mutually incompatible dialects exist) and it is not suited for general purpose programming.

S+ , developed by Chambers and others at AT&T, is among the most flexible and powerful licensed statistical software systems. It provides fast vector- and matrix- arithmetic and can deal with large datasets. Most built-in functions were originally written in Fortran or C. See [8] for a description of the S language family.

R (`www.r-project.org`) is built on another implementation of S. R is freeware developed by academic statisticians around the world. It provides functionality very similar to S+.

For a discussion of S, S+ and R, see [43].

SAS (`www.sas.com`) is primarily user-oriented, with procedures and a macro language for programming.

SPSS (`www.spss.com`) is popular among non-statisticians, especially in the social sciences. It is mostly used as point-and-click statistical spreadsheet, but does have a command line form with SAS-like procedure syntax.

Lispstat is an object based (prototype based) language using lisp syntax (built on top of xlisp!) with a reasonable graphics subsystem, developed by Luke Tierney at U of Minnesota. It is used in academic communities for research ([42]).

Graphical Programming Languages (`http://c2.com/cgi/wiki?GraphicalProgrammingLanguages`) attempt to make the compositional aspects of programming more intuitive and visually accessible. Typically programs are formed by drag-and-drop operations that result in visual collections of hierarchical codes. In many cases these languages, or systems, are constrained in the sense that only *correct* compositions or sub-units (templates) can be assembled, and therefore the program design is the source code. It is for this reason that the composition can only be created in verifiably correct forms — that some graphical programming languages are increasingly used in industrial applications. Note, however, that algorithmic issues are suppressed and are handled with varying degrees of detail in different systems. Examples include:

- StreamIT (see above)
- VisualBasic `http://www.engin.umd.umich.edu/CIS/course.des/cis400/vbasic/vbasic.html`
- Simulink `http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/ug/basics.html`
- Labview `http://www.ni.com/`

AMPL and GAMS (`www.ampl.com` and `www.gams.com`) are popular high level domain specific languages for numerical optimization problems. These languages provide mechanisms for specifying the type of problem, the objective function to be minimized, and any constraints that must be satisfied [22].

TCE The Tensor Contraction Engine (`http://www.cse.ohio-state.edu/~gb/TCE/`, [6, 7]) targets a particular class of problems in quantum chemistry and nuclear physics, providing a domain-specific language and an optimizing compiler for the language. The compiler currently generates code in Fortran targeted to the Global Array Toolkit's [37, 36] global address space programming model. The language is designed to be very close to the way researchers in this domain would express their methods in scientific papers, and the optimizing compiler takes advantage of the restricted domain and the high-level view provided by the language to perform optimizations which could not be done by compilers for general-purpose languages operating on traditional hand implementations of the same methods.

SMS The Scalable Modeling System (`http://www-ad.fsl.noaa.gov/ac/sms.html`, [25]) is a tool for the automatic parallelization and optimization of weather/climate modeling codes. The language of the SMS is a directive-based extension to Fortran77 tailored to the needs of applications in the weather/climate domain. It was originally designed to facilitate parallelization of existing sequential codes, and seems to have been used primarily in that mode, rather than for the writing of new applications.

3.3.3 Domain-Specific Languages vs. General Purpose Languages: Pros and Cons

Because they are based on higher-level specifications and mechanical translation into code, DSL environments can be expected to produce code that is more reliable (in the sense of having fewer bugs) than direct hand implementations, assuming the DSL has been appropriately validated.

As with problem-solving environments, domain-specific languages can suffer from a small (but possibly active) community which limits support. The more that a domain-specific language exploits features of the domain, the more likely it is that an application may find itself constrained by the goals of the DSL. Extending the domain may also be difficult. Another issue is the high cost to develop and support domain-specific languages.

3.4 Other Tools and Packages

The categories of problem-solving environments and domain-specific languages are somewhat artificial. Rather than force all relevant systems into one or the other category, the working group chose to place some systems and environments into a separate category.

Matlab/IDL is a commercial high-level, full-featured, interactive programming language with built-in support for advanced graphics, matrix/array manipulations, numerical operations, statistics, image processing, and i/o formats.

Matlab (`www.mathworks.com`) is likely the most popular DSL in use today. Matlab started out as syntactic sugar for linear algebra operations. Today it features a highly advanced graphic system, complex structures like cell arrays and even object based programming, while being backwards compatible with the original language. See also `http://www.mathworks.com/company/newsletters/news_notes/pdf/spr95cleve.pdf`.

MathML (`http://www.w3.org/TR/MathML2`) is an XML language for specifying the content and layout of mathematical expressions. MathML 2.0 provides a rich set of content descriptors, including tags for arithmetic, algebra, logic, relations, calculus, sets, sequences, series, elementary classical functions, statistics, and linear algebra.

Autodiff (`www.autodiff.org`) Automatic differentiation tools generate code for computing derivatives from code for computing a nonlinear function [26]. This approach dramatically reduces the amount of code that must be written by scientific programmers while providing performance equal or superior to derivative code written by hand.

Gaussian (`http://www.gaussian.com`, [24]) is a widely used end-user application in computational chemistry. While neither a DSL nor a PSE itself, it is significant because in this domain there are a significant (and increasing) number of HPC users who are *not* software developers, but merely want to be able to run calculations using codes like Gaussian. Gaussian is a closed-source commercial code with a limited development community due to the cost of source licenses; other packages in this area may be more open, but do not have as large an end-user base. Productivity enhancements for those who are strictly *users* of scientific software often come from incorporation of such codes into PSEs. The closed-source nature of the application may make this challenging.

3.5 Code Generation

A variety of libraries use automatic code generation and tuning techniques at installation time to tailor the installed library to the target platform. Some significant systems are described below:

TAMPR (Transformation Assisted Multiple Program Realization) was one of the earliest general-purpose source-to-source transformation systems. It was used in the original production of the EISPACK numerical library to create the single- and double-precision, real and complex versions of many of the elementary function and linear algrebra subroutines in the library from an abstract algorithm description. It was also used to translate purely applicative LISP into the functional but awkward HEP Fortran language for the Denelcor Hep, thus creating a user-friendly programming environment for the first commercially available general-purpose parallel computer.

FFTW (`http://www.fftw.org`, [23]) is a library for discrete Fourier transforms in one or more dimensions. At installation time, highly-optimized high-composable C-language "codelets" are generated by a codelet generator written in Caml Lite (a dialect of ML). Optimizations include constant folding, algebraic identities, and loop unrolling. At execution time, depending on the inputs, a "plan" is created using a dynamic programming algorithm for a sequence of codelets which will minimize the execution time of the task; the plan is then carried out by the "executor".

ATLAS Automatically Tuned Linear Algebra Software (`http://math-atlas.sourceforge.net`, [44]) is an implementation of the BLAS library which is tuned at installation time. The approach is based on a parameterized matrix multiplication algorithm, where the parameters are determined empirically (at installation time) to allow the entire operation to take place in the L1 cache of the target system.

PHiPAC Portable High-Performance ANSI C BLAS3 (`http://www.icsi.berkeley.edu/~bilmes/phipac/`, [11]) is another empirically-tuned BLAS3 library. PHiPAC is based on a conceptual model of modern microprocessors and their associated ANSI C compilers which provides a framework for reasoning about how to produce high-performance algorithms for multi-level cache environments. Based on this model, parameterized algorithms are written, and, as with ATLAS, the actual parameters are determined at installation time by emirical means.

FLAME Formal Linear Algebra Methods Environment (`http://www.cs.utexas.edu/users/flame/`, [27]) is a library, which, coupled with a formal methodology, provides an environment for the implementation of linear algebra algorithms. Interfaces are provided for C and Matlab. It is used to implement the PLAPACK parallel linear algebra package (`http://www.cs.utexas.edu/users/plapack/`).

Tools for automatic generation of application code from user-specified tables can provide significant improvements in development and maintenance of large applications. An example is the Weather Research and Forecast (WRF) model software. Some 30,000 lines of WRF code are automatically generated from a user-edited table, called the Registry [34]. The Registry lists state data fields and their attributes: dimensionality, binding to particular solvers, association with WRF I/O streams, communication operations, and run time configuration options (namelist elements and their bindings to model control structures). The Registry provides a high-level single-point-of-control over the fundamental structure of the model data, and thus provides considerable utility for developers and maintainers. Adding or modifying a state variable to a model involves modifying a single line of a single file; this single change is automatically propagated to scores of locations in the source code the next time the code is compiled.

There is related work in the area of application specific code generation [19].

3.6 PSEs and Domain-Specific Languages in Perspective

Because PSEs and DSLs provide high levels of abstraction, complex algorithms and applications can be specified with significantly less code than an equivalent formulation in a general-purpose programming language would require. Code generation techniques such as automatic differentiation or domain-specific compilation can further reduce the amount of user-level code.

3.7 Research Roadmap 2005–2009

3.7.1 Support for Programming-in-the-Large

Current problem-solving environments, domain-specific languages, and general purpose high-productivity languages are subject to one common problem. Each is likely to either require that an entire application be expressed in its own single paradigm or at least require that it be the environment "on top". For example, it may be difficult for a subroutine written in a domain-specific language to be linked to a standard Fortran main program. (This problem is not even conveniently solved at the moment with respect to C and Fortran.) A challenging and useful research project would be to define an impartial formalism to describe aspects of interoperability, to guide language designers and compiler writers in creating components that could be linked together or otherwise combined in applications. Current work in the Common Component Architecture SciDAC projct could be used as a starting point for such a formalism.

A software environment that maximizes the potential of component architectures to deliver their productivity benefits should include:

- Programming languages and programming models that can be composed in well-defined ways, under clearly defined performance constraints.

- Debugging and performance tools capable of supporting the breadth of languages and programming models on the system.
- Dynamic linking and loading in a standardized way across platforms. (Though much can be done with statically linked executables, the full benefits of component environments cannot be realized without the possibility of more dynamic interactions among components.)

3.7.2 Legacy Code Migration

Legacy codes are often the only repository of computational science knowledge accumulated over the years. These codes were developed for many different architectures, ranging from vector computers to distributed memory machines, symmetric multiprocessors, NUMA shared address space architectures, and clusters of SMPs. Likewise, they cover a broad class of languages from Fortran 77, C, and C++ to message-passing dialects and, more recently, CoArray Fortran and UPC. These codes represent a major investment which must be protected by providing a transition path to new architectures.

Migrating such applications to future HPCS architectures is a difficult task if performance portability is to be achieved. This will require the development of a set of sophisticated semi-automatic or automatic tools dealing with

- reverse engineering of the original program,
- static analysis, possibly using domain-specific information,
- applying optimizing transformations to the program tree,
- recognition of patterns and idioms that can be efficiently implemented on the target architecture, and
- optimizing code generation guided by the target architecture.

User interaction (possibly via directives) will be needed to guide the system in situations where static analysis is incapable of deriving sufficient information for the creation of highly efficient target code. Alternatively, feedback-oriented optimization can be used in such a situation.

The research issues outlined here are closely related to a number of tasks discussed in Sections 2 and 5.

3.7.3 Telescoping Languages

Telescoping languages form a particularly practical approach to address the specialized productivity and optimization requirements for scientific computing. They refer to a strategy for compiling high-level programs with calls to libraries. The telescoping language compiler seeks to automatically generate optimizing compilers for domain-specific languages defined by libraries. It achieves this in a number of ways. First, it envisions an extensive pre-compilation phase in which the library will be specialized for several possible uses. Several variants will be generated based on the different possible calling contexts including the types of the input parameters, etc. Furthermore, the library routines will be specialized for interactions with each other. Annotations provided by the library writer will aid the compiler by hinting at what optimizations would be useful and how the library writer expects the library to be used. Compiling the script then just involves replacing the library calls (and combinations thereof) with the appropriate variants, alleviating the need to recompile the library subroutines to optimize for the given calling context.

The efficient optimization of domain specific libraries provides an approach to tailor a base language's features to arbitrarily small economic markets within scientific computing. Significant work is required to permit library developers to optimize the abstractions that are developed for application developers (users).

3.7.4 AST: Standard Representations and Transformations

Ubiquitous standard AST representations. We propose the definition of a standard Intermediate Representation (IR) for the major languages used in scientific computing, with the goal of promoting an effort for broad-based tool development for these languages.

This is not an attempt to build a standard IR for *all* languages, which has been tried previously several times and failed each time for well understood reasons. Rather, the IR should be defined specifically for each language, in tandem with the formal language standard. It should be at a sufficiently high level to permit source-to-source translation. The result would be a standardized Abstract Syntax Tree (AST). This would encourage the development of standardized tools addressing the complexity of scientific applications developed with common languages such as C, C++, and Fortran 90. A further enhancement of this approach would provide a capability for extending the IR with domain-specific features.

Transformation Languages. Based on a standardized IR, standardized operators for the transformation of the AST can be introduced. The generalization of this idea leads to *transformation languages*, which allow the user the formulation of sophisticated transformation strategies at an abstract level, without the need to deal with the intricacies of a compiler. Such languages could provide support for annotations specifying user-defined abstractions guiding the transformation process.

3.7.5 Code Generation

Tools for the development of application specific code generators. As has been noted in the example of the WRF model [34], table-based application specific source code generation provides significant improvements for developers, maintainers, and users of large scientific applications. Much of the WRF code — some 30,000 lines — is seen only by the compiler. A current shortcoming is that the only interface for controlling the back-end output is the tool's own source code, although the front-end user semantics are well defined. Adding or modifying a particular code generating capability in the tool involves modifying the tool itself. A key area for research and development into table-driven code generation is devising and implementing a generalized and abstract notation for specifying code to be generated from the table.

Feedback directed static compilation. As a technique for the optimization of scientific applications, several projects, such as FFTW and ATLAS, have demonstrated the value of empirical testing of a range of optimizations as a robust way to address the peculiarities of modern architectures. Such techniques should be expanded to be incorporated into compilers as an approach to optimize key computational parts of application codes. Empirical modeling of that kind uses exhaustive searches based on parameters such as degree of loop fusion, degree of unrolling, and tile sizes for blocking. Dynamic optimizations can identify and provide information to specialize code for more aggressive optimizations.

3.8 Research Roadmap 2010–2015

3.8.1 Advanced Analysis Tools

Standard analysis tools can be generalized in a number of ways to support correctness debugging, error recovery, and optimization. Some relevant research aspects in this context include:

- support of debugging through automatic verification, model-checking, or consistency analysis,
- improving standard analysis algorithms by including domain-specific knowledge,
- applying linearity analysis to numerical optimization, or
- combining complex domain-specific analyses (such as data flow information) with dependence analysis or iteration space slicing.

3.8.2 Defensive Programming Productivity Tools

Unlike many engineering efforts, programs are rarely written with adequate attention to early detection and recovery from errors, whether they are due to errors in input data, other parts of the program, or hardware faults. Tools, built on a solid understanding of sources of errors and effective techiques for detecting and recovering from those errors, are needed to improve the robustness of major programming projects. These tools must include a component that is cognizant of the problem domain and hence must be developed in the context of high-performance computing.

3.8.3 Automatic Generation of DSLs

User-accessible Language Generators. Constructing a high-quality, robust, high-performance DSL environment is far from an easy task. While small, simple DSLs are fairly common, more sophisticated efforts are typically undertaken only by the most "computer-savvy" computational scientists or by computer scientists in direct collaboration with the computational scientists. However one of the key advantages of DSLs is that they provide users a very high-level *domain-specific* environment that matches their problems closely, and there are obviously a great many different domains which could benefit from DSLs. Radical new approaches are needed to simplify the construction of high-quality DSLs in order to make them more accessible to computational scientists. Some ideas in this vein include:

- Simplifying the task of generating the language itself (parser), such as example-driven approaches, where user/developers provide examples of the syntax they desire which are then interpreted by a tool to produce a formal grammar.
- Tools for knowledge and concept management to help factor the problem into elements of various degrees of generality. More general concepts (i.e., fundamental mathematics) and DSL features (linguistic, optimization, code generation, etc.) associated with them may be reusable across multiple DSLs, while more specific characteristics can only be reused in narrower contexts.
- General, standardized, pluggable frameworks for the construction of DSL environments. This logically relates to ideas about opening up the compilation environments for general-purpose languages.

3.8.4 Nonlinear PDEs

While domain specific languages have proven effective in some regimes, including numerical optimization, statistics, and signal processing, they remain an elusive goal for general, nonlinear partial differential equations (PDEs). An important research goal is the development of effective domain specific languages for such problems or the inclusion of sufficient support in a general purpose programming language so that nonlinear PDEs can be easily specified.

4 Program Development Support

Chair: William Carlson, IDA/CCS

Many mechanisms are necessary to support the production of HPCS software systems in addition to high productivity languages. This includes tools to build complex software systems, configuration management systems to chronicle application history, library infrastructures to facilitate the reuse of intellectual contributions from many sources; test harnesses to ensure that modifications are enhancements, and program distribution mechanisms to publish the work for end use.

4.1 Key Challenges in Developing Advanced Applications

4.1.1 Collaborative Development by Distributed Groups

An application programming interface[4] (API) is a set of definitions of the ways in which one piece of computer software communicates with another. It is a method of achieving abstraction, usually (but not necessarily) between lower-level and higher-level software. One of the primary purposes of an API is to provide a set of commonly-used functions – for example, to draw windows on the screen or solve a certain type of equation system. Programmers can then take advantage of the API by making use of its functionality, saving them the task of programming everything from scratch. APIs themselves are abstract: software that provides a certain API is often called the implementation of that API.

An API can be thought of as a contract for technology transfer between producer and consumer for software. Ideally, potential consumers do not concern themselves with the internal details of the software but rather use the API that defines its interface. In practice, the way in which APIs are currently specified is far from ideal. Typically they are specified only syntactically, while critical information such as semantics, development history, failure modes, and dependencies is not defined rigorously and may not even appear in documentation. This puts the burden of their correct use on programmers and is a source of repeated bugs, unnecessarily steep learning curves, and retarded productivity.

New tools and techniques are needed for specifying APIs so that developers can be more explicit about their intent and consumers have benefits of automation to verify and validate their use. For the issues of language specifics, semantics, evolution history, failure modes, and API dependencies, we will explain the topic and related problems, give an example solution that addresses these problems, and conclude with suggestions on what more needs to be done.

The language details of APIs are commonly handled by writing wrappers for other languages to connect to the core library and publishing separate APIs for each of these wrappers. Wrappers are currently generated by hand and can be labor intensive with their quality varying with the skill of the programmer. Moreover, this approach is non-portable and tends to get cumbersome when three or more languages are involved in a single application. Interface Definition Languages (IDLs) are very effective at creating language independent APIs that can then be used by tools such as Babel [14] to automatically generate wrappers which provide increased portability and uniform quality that one would expect with automation.

Since software and APIs can change, it is not enough to have the API, but the correct version of the API is needed as well. There is no clear way to manage the API as it evolves over time. In Babel, an API cannot be created without a version number, but there is no mechanism for managing the history of the APIs. More work needs to be done here to make software more trackable.

[4]Originally called "Advanced Programming Interface" but since has been made generic and is now called "Application Programming Interface"

A failure mode could provide an error flag to be returned from the API enabling users to lookup the meaning of the error in a table of documentation. This is another example of semantics not being reflected in the syntax. The situation is slightly better with the exception idiom, where exceptions can be generated in a subroutine and caught by any one of its parents. There are still limitations in that the syntax carries only the type of the exception, not the specific circumstances. In addition, the originator of the exception is not always clear. For instance, was it raised in the library from which it was called directly, or by some tertiary library by another party down deep in the call stack. Another issue is that exceptions can be intercepted unintentionally by intermediate libraries and never get caught by their intended handler. Research into better error handling, including a more event-driven approach with a publish/subscribe model would be better than the present catch/throw paradigm.

Finally, libraries implementing APIs often have other APIs and other libraries they themselves depend on. This causes problems in downloading all the pieces of code needed to run the software, but more subtly it can be a source of bugs. Tools that investigate and display dependency graphs of APIs would improve the productivity of developers as they integrate multiple modules. Often libraries will have circular dependencies, which means they need to be listed repetitively for most single-pass linkers to resolve all the symbols and generate valid executables. Conflicts where two libraries have the same name but different signatures are actually the beneficial kind because the linker will quickly generate an error message. More insidious is the case where the two have the same name and same signature, but different semantics. Since linkers can check semantics, one subroutine is used everywhere and the other ignored. This often creates subtle and time consuming bugs.

Software is always used in ways that the original author(s) never intended. Sometimes this is simply a result of innovation and even though the use was not intended, it is still valid. The problem we are trying to address here is identifying when the unintentional use is also invalid. There are a great many cases where invalid use is not checked for or protected against by any formal mechanism or technology, but only through documentation or word-of-mouth. Reusing existing software is one of the most productive and effective things software developers can do, but we must better facilitate and expand this practice.

4.1.2 Application Lifecycle Support

One key challenge to the development of high productivity programs is the lack of appropriate infrastructures to support the complete lifecycle of an application. This area includes everything from systems to allow the integration of programs written in many different languages and styles, testing harnesses to ensure that program changes do not result in unexpected results, and systems to control the actual build cycle of the program. One recent study at LLNL ("The Build") [31] found that up to 20% of human resources on large projects are spent on the process of building software and maintaining the configuration of software systems. This work is done almost entirely by hand or with custom-built tools.

4.2 Vision

4.2.1 Versioning and Configuration Management

Change is essential in the development of high productivity applications for many reasons. First and foremost, applications development in this area is often an experimental process. Running an application will often lead to new discoveries which must be quickly folded back into either the same or a new application. But change is also essential in code that gets reused from one application to another, both by a single developer, between wide-ranging collaborative teams, and by the reuse of code written for entirely different purposes. We propose a vision that expects change in all software elements and can lead to dramatic increases in productivity.

This vision starts by presuming that every element has associated with it version designators and a history of which designators provide what functionality. The vision also requires that any use of a software element by another must specify not only the element but also the versions of the element which it uses. In addition, all systems must allow for the simultaneous existence of many versions of the same software elements.

4.2.2 Refactoring

In the world of "business computing", modern development environments include tools for performing significant changes to large code bases in single steps. For example, it is possible to select a program identifier and contextually replace all occurrences of that identifier automatically while maintaining the scoping semantics of the programming language. It is also possible to select a block of code and automatically turn it into a call to a new function whose body contains the selected code. In object-oriented languages, method definitions can be automatically moved into superclasses or subclasses on demand. These sorts of programming transformations are referred to as "refactorings". Development environments providing support for them are referred to as "refactoring browsers". Refactorings provide a significant boost to productivity because they allow a programmer to maintain and transform code at a higher level of abstraction than that of text-based approaches.

In the world of HPC, refactoring tools could help scientists build and maintain larger code bases more easily. Although significant academic research on such tools has been conducted for a long time, particularly in the area of automatic support for the parallelization of sequential Fortran programs [48], no production-quality tools are currently available.

Because refactoring browsers are necessarily language-specific, we would have to build new browsers for languages such as Fortran 90 that are commonly used in HPC. But there are also opportunities for "performance-oriented" refactorings specifically suited to the HPC domain. For example, a refactoring tool that automatically transforms a sequential looping idiom into parallelized code could allow us to tune code for parallelism much more quickly. Refactoring tools that alter numeric code based on properties of the numeric idiosyncrasies of specific architectures could be quite useful as well. Refactoring for converting parts of a program to software components in a component based development environment, or refactoring to separate different concerns in a complex multi-platform build script and preprocessor statements (using concepts from aspect oriented programming) are interesting research challenges. These examples only scratch the surface; no doubt, there are many useful refactorings (yet to be discovered) that could be explored for HPC performance tuning and for tailoring to particular architectures.

4.2.3 Testing

Support for application and library testing are essential for high productivity. Beyond the normal challenges that testing presents (e.g., managing many test cases, automatic comparison of "good" results), the high productivity area has special needs. First the scale of systems used presents problems due to the sheer volume of testing required. Also, many high productivity applications appear to be (or are by design) fundamentally non-deterministic at the level of detail required for repeatable testing. Our vision is to develop systems that support large-scale automatic testing of applications and libraries in the presence of non-determinism. Research directions in hardware and software support for efficient deterministic replay on petascale systems will be a key component for the future of testing. Efficient deterministic replay, if successful, can in essence provide arbitrarily detailed information about the history of events that led to the error, isolate the error (with source line information) and present this information to the developers after each run. Efficient deterministic replay can also be invaluable in helping to identify the root causes of failures in the field. With this vision in place, significant progress can be made in achieving true high productivity.

4.3 Requirements Catalog

To achieve the previously outlined vision for program development, the support of other components of a language system will be needed. The major requirement for language, compiler, and runtime systems is support for a seamless and efficient connection of program components written in different programming languages while enforcing the highest level of safety.

HPC programs will increase significantly in complexity over the next decade as researchers develop full-system simulations encompassing a variety of scientific systems at possibly different time scales. Different sections of simulations will require different classes of algorithms, math methods, data structures, and computational techniques. It is unlikely that any one programming language or paradigm will support equally well all needs. Thus, it is imperative that programming languages and techniques interact seamlessly and are supported efficiently by runtime systems.

Today, it is straightforward to mix source code written in FORTRAN, C, and C++ that uses array structures, however it is more difficult for these same programs to use lists or pointer data structures. This situation is likely to become worse in the future as new languages emerge that promote more dynamic, irregular, and sparse data structures to first-order objects. Mixed-language programming, compiler optimization, and debugging can benefit from a common intermediate representation with well-defined semantics that serves as a target for the major high-level languages and implies an execution model for runtime systems. Since most new high-level programming languages compile to C and assume a threaded execution model, establishing a lower level intermediate form more appropriate for optimization should not be difficult.

All tools of a programming environment need to support an infrastructure which is aware of both the version-based APIs we envision and the fact that most program development will exist in a multi-lingual environment. Future high productivity debugging aids need to provide feedback to automatic test generation systems and automatically document which program changes fixed which program errors.

4.4 Cost vs. Utility of Program Development Tools

In software development there is generally an economics trade-off between utility and cost. Program development tools are usually considered a cost to be minimized. But expending resources on program development infrastructure is a solid investment with a return in utility that can more than offset their original costs.

Below we discuss some examples of cost versus utility in software development tools:

- Extending programming languages with well-designed parallelism features (see Section 2.2.1). Although the introduction of such features may be expensive, they increase the utility of the programming language.
- The utility of being able to change applications more rapidly with good tools is particularly useful in science and engineering where an easy adaptation of applications contributes to invigorating the scientific discovery process.
- Investment in better software testing, a cost, leads to greater application robustness, a utility.
- The cost of porting applications to new environments brings with it an obvious utility. But we also know that porting frequently makes the software more robust; we find subtle bugs due to the differences in compilers and runtime environment.

There is also an interplay between the initial cost of software and its operating, or maintaining, utility. In order to increase operating utility and decrease operating cost, more investment in these tools that have an immediate and demonstrated utility will increase productivity.

4.5 Research and Engineering Roadmap 2005-2009

In the development of a roadmap to support programming in the high productivity arena, it is essential that both research *and* engineering be included. Engineering activities are essential to both the testing of research ideas and their appropriate adoption by a broader community of applications builders and system builders. In the near term research area, the following items should be considered:

- Conveying semantics of an API in a form that can be verified and validated by tools has been subject to some research, but more needs to be done to make this widespread. Most work in semantic checking has either been in niche languages (e.g. Eiffel) or separate libraries. IDLs and telescoping languages pose new opportunities to get semantic information more rigorously encoded into mainstream programming languages.
- The development of a language to specify application program interfaces in a multi-lingual environment.

In the engineering area we recommend:

- The design and production of reference implementations for all tools mentioned here. By focusing on design and reference implementation, the full range of options for production quality tools will be enabled: commercial tools developed by ISVs; system vendor products tailored to a specific platform; and community-based open source tools.
- Refactoring tools which support multi-language application development.

4.6 Research and Engineering Roadmap 2010-2015

In the longer term research arena, we recommend the following items:

- Allow for the deterministic testing of non-deterministic programs.
- We also recommend the continuing production of engineering designs and reference implementations of the concepts developed by research in both the first phase of the roadmap and this one.

4.7 Acceptance of New Technology – A Social Issue

A frequent occurrence in technology development is that a new technology, with obvious technical benefits over the old technology, is introduced to a market, but fails to be adopted. There are many reasons given for this, including lack of familiarity/portability, poor design, novelty, tradition, economics, and culture. The actual causes may be of a more human nature, such as the unwillingness to accept a change to long-established habits, or a lack of trust in new methodologies or procedures. They can present real impediments to the adoption of new techniques.

One of the best known solutions to this problem was stated by industrial designer Raymond Loewy (1893-1987) using the term MAYA, defined as "Most Advanced Yet Acceptable". Historic examples of this concept are TVs initially placed in wooden radio cabinets and email which was developed along the lines of postal mail, but easier/faster/cheaper.

In order for new programming development tools to be accepted, all these reasons and excuses need to be considered. Programmers (even HPCS programmers) tend to be generally conservative, but willing to accept small changes, particularly given some successful test experience with a new technology.

This implies that we need new ways of developing programs that can be incrementally applied to the current development environment. These methods must allow programmers to retain ownership of the programming process. (Programming must remain programmer-centric, not tool-centric.) We need a set of adventuresome programmers who can be encouraged to try these incremental advances, and who will advocate them once they have used them successfully and seen some payoff. We need an organizational structure that recognizes the need to support these incremental changes, and encourages the programmers to take the time needed to learn and try the new technologies. One way of doing this would be to add some small overhead to common projects where the programmers are encouraged to try some parts of the development using multiple tools. Another possibility is to perform a study where the programmers are asked to use multiple tools (one current tool and one new tool) and to track the advantages and disadvantages of each tool.

5 Programming Environments and Tools

Chair: Daniel A. Reed, Institute for Renaissance Computing, University of North Carolina

Programming environments and tools encompass a wide range of software, from software development environments to debugging systems and performance analysis tools. Although integrated software development and testing environments are almost universally used to create desktop applications, and application scripting languages are now commonplace for rapid prototyping and component integration, the tools used to develop, debug and optimize high-performance computing (HPC) applications have changed little in the past decade.

In the 1990s, the U.S. high-performance computing and communications (HPCC) program supported the development of several new computer systems. In retrospect, we did not recognize the critical importance of long-term, balanced investment, particularly in software and tools. As a result, users are now struggling to develop and maintain applications for systems containing thousands of processors. With even larger systems on the horizon, we must rethink current approaches if we are to broaden the base of application developers and lower the barriers to productive use of parallel computing.

At the workshop, the programming environments and tools group discussed the current state of the art, the challenges and opportunities associated with high-productive computing, and possible solutions to current problems. Below, we summarize the conclusions of the group and present a roadmap for possible resolution of current problems.

5.1 Current State of the Art

Software tools for HPC systems remain rooted in their historical origins and are slightly modified versions of those used on sequential systems, typically variants of Unix or Linux. These implementation choices are driven both by the cost to develop software tailored to parallel systems and the desire to maintain compatibility and user familiarity with sequential systems.

Today's programming tools also remain focused on programming in the small, emphasizing the debugging and tuning of code fragments on individual processors or execution contexts. As systems grow to contain thousands of processors, understanding aggregate system behavior becomes increasingly complex. This is especially true given the increasing probability of component failure. As processor counts for multi-teraflops systems grow to thousands and with proposed petaflops system likely to contain tens of thousands of nodes, the standard assumption that system hardware and software are fully reliable becomes much less credible. This will necessitate development of software and tools to accommodate resilient operation and adaptation to failures.

In addition, current HPC tools have limited interoperability, constraining information sharing and user customization to meet specific needs. Moreover, because debugging and performance tuning tools are often used only when simpler remedies have failed, such tools must be simple and intuitive to use, lest they be eschewed for less arcane alternatives. As a consequence, users often adopt *ad hoc* schemes based on `write` or `printf` for debugging and hardware counter sampling and profiling for performance tuning *because they are familiar* – despite their obvious lack of scalability and global perspective.

Finally, as applications become increasingly complex and interdisciplinary, coupling software components and libraries developed by disparate groups, understanding the interplay of components and the implications of this interplay will become increasingly critical. The balance between scripting and orchestration systems for rapid customization and high productivity programming and measurement for correctness and performance assessment will require higher level mechanisms than today's source code centric tools.

Together, the emergence of very large systems, higher probabilities of partial system failures and rapidly rising code complexity mean that we face a mounting crisis in software tool research, development and support. Simply put, tools have not kept pace with rising scale and complexity.

Experience has shown that effective software tools are developed over periods of a decade or more, as experience with applications and architectures is used to rectify software shortcomings and enhance software strengths. Applying these lessons requires a long-term commitment to tool development and user assessment, with concomitant funding and support. In addition, tool developers, whether academic or commercial, need access to testbeds to develop and test tools at scale.

5.2 HPCS Challenges and New Approaches

The workshop discussions identified three major tool challenges for high productive computing:

- fault tolerance and resilience to component failures for large-scale systems,
- dynamic adaptation and application behaviorial resilience for complex applications, and
- tool integration and support for high productivity.

Changing current models will require a long-term commitment to tool development, assessment and refinement, with access to large-scale testbeds for tool development and assessment. Below, we describe these three challenges and possible approaches in greater detail.

5.2.1 Fault Tolerance and Resilience

Today's terascale systems contain thousands of processors, with new systems (e.g., Blue Gene/L and Red Storm) projected to contain tens of thousands of processors. Although the mean time before failure (MTBF) for the individual components (i.e., processors, disks, memories, power supplies, fans and networks) is high, the large overall component count means large-scale systems will often operate in degraded mode, with some components inoperative.[5]

Unlike today's *deus ex machina* application and system software for uniprocessor systems, which presumes centralized control, system size and communication costs for data acquisition on multi-teraflops and petaflops systems will likely preclude global knowledge and control. This has profound implications for how we manage and coordinate large numbers of parallel tasks. Indeed, the number of nodes in terascale and petascale systems rivals or exceeds that found in many wide-area networks, suggesting that similar software approaches might be appropriate for large-scale parallel systems. Wide area networking software presumes unreliable components; packets are lost, nodes are unreachable, and both observed communication latency and bandwidth can vary widely due to incomplete or inaccurate global information.

For larger multi-teraflops and petaflops systems to be useable, we must develop new software that embodies the two important realities of large-scale systems: (a) frequent hardware component failures are to be expected as part of normal operation and (b) very large system sizes will limit fully quantitative knowledge of global behavior.

New, homoeostatic software models would allow applications to both recognize and recover from transient failures and to adapt to permanent failures by continuing operation, albeit in a degraded mode. Implicit in such an approach is the need for fault prognostication, detection and recovery interfaces and abstractions.

Abstraction-based application assertions, application source code correlation, runtime support and management, measurement and prediction and controlled fault injection will also be needed. Finally, transactional/functional execution models that enable rollback and retry rather than traditional dump and restart mechanisms must also be explored. Finally, redundancy based on reliability models would allow application developers and runtime systems to optimize redundancy to minimize execution time given failures.

5.2.2 Scalability and Intelligence

As noted elsewhere in this report, one of the keys to raising software development productivity is use of high level abstractions and language models that hide many of the details and idiosyncrasies of large-scale, high-performance systems. In addition, emerging applications are increasingly interdisciplinary, requiring

[5]Rigorous engineering and testing can ameliorate but not eliminate failures.

the coupling of models and libraries from multiple disciplines to understand complex phenomena, whether natural or artificial.

A new generation of software tools will be required that allow developers to reason about application behavior within the context of the application specification abstractions. Without such high-level tools, one must expose the abstraction implementations to debug and tune behavior, negating many of the advantages of the abstraction.

"Smarter" tools would recognize common abstraction idioms and also learn from experience across checkpoints or executions. Coupled with audit trails and data mining techniques, such tools would learn common behaviors and mistakes, providing high level guidance based on embedded implication knowledge bases.

Finally, because complex, multidisciplinary applications will have multiple behavioral regimes, particularly when executing atop systems with possible component failures, automatic adaptation and recovery are the next steps beyond semi-automated advice systems. Such closed loop adaptation will require deep integration and multilevel data correlation across application code, hierarchical libraries and runtime systems, together with decision procedures and actuation mechanisms.

To enable such smart tools, we must define standard information sharing interfaces across compilers, libraries and runtime systems. These standard access points for data retrieval and control would increase component interoperability and enable retention of historical context in behavioral databases. In turn, this would provide the raw data needed by data mining and reasoning systems.

5.2.3 Tool Integration and Support

Users often turn to programming environment tools only at the end of the cycle of design, coding, debugging, tuning and execution. When they finally use a debugger, performance tool or run-time monitoring tool, they really want a fully integrated programming environment where everything the tool needs is available to it. Users do not like recompling with special options, inserting probes, or making significant changes to their everyday practice simply to enable a tool to function properly.

For future petascale environments, users will want fully integrated programming environments with tools that are dynamically adaptive. Such an integrated "developer's notebook" would maintain the code history, the code, the documentation, the suite of application tests and associated experiment management, plus multiple perspectives and levels of detail. In addition, library management encouraging the reuse of tools components and capturing institutional memory on the use of the components should also be included.

Today, programming environments for HPC systems are very nearly non-existent, and they often consist nothing more than a collection of disparate tools with limited interoperability. Source code is the only truly programmable tool. Tools are not reconfigurable, and often users must create their own interfaces to obtain even limited interoperability. Tools such as debuggers and performance analysis tools often only have access only to limited symbol information from compilation systems.

Creating more effective programming environment will require significant changes in approach. First, the separation among development, compilation, execution and testing will continue to blur. There must be much deeper integration and interoperability, from language to compiler to run-time system to tools and to hardware. Well-defined, shared interfaces and shared information is needed across all levels.

As we noted earlier, this information sharing will lead to deeper integration. Even the user interfaces should develop shared "look and feel" with usability studies determining the most effective interfaces for large-scale computation environments. This will require collaborative co-design – when programming languages and models are designed, the programming environment and tools designers should be included in the process.

This is a radical shift from developing tools in isolation and after the fact to one where tool developers are an integral part of the systems architectural team. A co-design approach would lead to better balance in the tool chain, common interfaces and integrated environments.

To allow tool composability and reuse, interoperable frameworks must also be developed. Such frameworks will have standard APIs for component information sharing, allowing component plugins and extensibility and composability. In addition, the information flow among tool chain components should be invertible. This bi-directional information flow will enable static and dynamic introspection. As an example, compilers have a wealth of information, including code transformations, user directives and analysis - much of this would be useful for rest of the tool chain and should be made available in standardized formats.

To support such integrated programming environments with well-defined interfaces among different components, longer-term infrastructure must be built and supported. Currently, because the HPC community is too small, infrastructure is not easily funded - it rarely meets the definition of research, and few government agencies or research institutions view it as their charter to fund and support programming environment infrastructure that others can use to research and develop new tools for petascale architectures. Current HPC software infrastructure lacks long-term stability. Over time, the maintenance and support of such infrastructure fades away or no new development occurs.

5.3 Research Roadmap 2005-2009

Software tool research and development has both short term and long term components. In the next five years, the tools group identified the following activities:

- specification of tool architectures and interface standards, including information sharing for deeper integration and creation of a shared "look and feel" driven by usability studies
- support for multiple cycles of experience and learning, with rapid tool prototyping based on standards to gain experience with user needs and common programming idioms,
- scalable testbed access for software development, testing and validation at the scales used for application execution,
- exploration of the techniques needed to build smarter tools and automation, emphasizing learning, historical context and evolutionary behavior,
- creation of tool designs that support scalability, including reasoning and abstraction about system behavior,
- development of presentation metaphors and measurement techniques that support systems containing thousands or tens of thousands of concurrently interacting entities, and
- focus on fault monitoring and self awareness mechanisms that will allow applications and systems to operate in degraded system states when failures occur, together with recovery mechanisms that allow execution.

5.4 Research Roadmap 2010-2015

Longer term research and development should consider the following issues:

- increased support for integrated software design that deeply couples programming models and tools,
- development of multiple tool classes to support different behavioral models, based on the evolution of high productivity application specification systems, ranging from domain-specific languages and toolkits through scripting languages to parallel programming models,
- advanced testing and exploration of intelligent introspection and adaptation autonomic behavior, and
- exploration of very high level abstraction mechanisms that reason about execution aggregates, rather than individual tasks or groups of tasks.

References

[1] Gabrielle Allen, Werner Benger, Tom Goodale, Hans-Christian Hege, Gerd Lanfermann, Andre Merzky, Thomas Radke, Edward Seidel, and John Shalf. The Cactus code: A problem solving environment for the Grid. In *High Performance Distributed Computing (HPDC)*, pages 253–260. IEEE Computer Society, 2000.

[2] P. Andrews, G. Cats, D. Dent, M. Gertz, and J.L. Ricard. European standards for writing and documenting exchangeable Fortran 90 code, 1.1, 1995. `www.meto.gov.uk/research/nwp/numerical/fortran90/f90_standards.html`.

[3] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. C. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of High Performance Distributed Computing*, pages 115–124, 1999.

[4] ARPS version 5 coding standard in Fortran 90. `http://www.caps.ou.edu/ARPS/coding/arps50_codingstandard.html`.

[5] M. Ashworth. *Towards Teracomputing*, chapter Optimization for vector and RISC processors, pages 353–359. World Scientific, River Edge, New Jersey, 1999.

[6] G. Baumgartner, D. Bernholdt, D. Cociorva, R. Harrison, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. A high-level approach to synthesis of high-performance codes for quantum chemistry. In *SC002 Conference*. Institute of Electrical and Electronics Engineers and Association for Computing Machinery, November 2002.

[7] Gerald Baumgartner, Alexander Auer, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, So Hirata, Sriram Krishanmoorthy, Sandhya Krishnan, Chi-Chung Lam, Marcel Nooijen, Russell Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. Synthesis if high-performance parallel programs for a class of ab initio quantum chemistry models. *Proc. of the IEEE*, accepted.

[8] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The New S Language*. Chapman & Hall, London, 1988.

[9] S. Benkner, P. Mehrotra, J. Van Rosendale, and H. Zima. High-Level Management of Communication Schedules in HPF-like Languages. In *Proc. International Conference on Supercomputing 1998 (ICS'98), Melbourne (July 1998)*, Melbourne, July 1997. Also: Technical Report TR 97-5, Institute for Software Technology and Parallel Systems, University of Vienna (April 1997).

[10] David E. Bernholdt, Benjamin A. Allan, Robert Armstrong, Felipe Bertrand, Kenneth Chiu, Tamara L. Dahlgren, Kostadin Damevski, Wael R. Elwasif, Thomas G. W. Epperly, Madhusudhan Govindaraju, Daniel S. Katz, James A. Kohl, Manoj Krishnan, Gary Kumfert, J. Walter Larson, Sophia Lefantzi, Michael J. Lewis, Allen D. Malony, Lois C. McInnes, Jarek Nieplocha, Boyana Norris, Steven G. Parker, Jaideep Ray, Sameer Shende, Theresa L. Windus, and Shujia Zhou. A component architecture for high-performance scientific computing. *Intl. J. High-Perf. Computing Appl.*, 2004. Submitted to ACTS Collection Special Issue.

[11] J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C.W. Chin. PHiPAC: A portable, high-performance, ANSI C coding methodology and its application to matrix multiply. LAPACK working note 111, University of Tennessee, 1996.

[12] Coding standard for Community Climate Model 4 (CCM4). `http://www.cgd.ucar.edu/cms/ccm4/codingstandard.shtml`.

[13] G. Chin, L. R. Leung, K. Schuchardt, and D. Gracio. Conceptualizing a collaborative problem solving environment for regional climate modeling and assessment of climate impacts. Technical report, PNL, 2001. Preprints, 2001 International Conference on Computational Science, San Francisco, CA, 28-30 May 2001. (Available as PNL Preprint PNL-SA-34489).

[14] T. Dahlgren, T. Epperly, and G. Kumfert. Babel users' guide. Technical report, Lawrence Livermore National Laboratory, UCRL-MA-145991, January 2004.

[15] Dan Reed. Big systems and big challenges. `http://rparco.urz.tu-dresden.de/Parco2003/up/ma-1.pdf`, 2003.

[16] D. Dent and M. Hamrud. Fortran developments in IFS. ECMWRF Newsletter, No. 85, 1999. `http://www.ecmwf.int/publications/newsletters/pdf/85.pdf`.

[17] DC Department of Defense, Washington. Dod research and development agenda for high productivity computing systems. white paper., June 2001.

[18] Daniel A. Reed (Editor). Workshop on the roadmap for the revitalization of high-end computing, June 2003.

[19] Martin Erwig and Zhe Fu. Parametric Fortran — A program generator for customized generic Fortran extensions. In *Submitted for publication: 6th Int. Symp. on Practical Aspects of Declarative Languages (PADL'04)*, 2004. `http://web.engr.oregonstate.edu/~erwig/papers/abstracts.html\#PADL04`.

[20] ESMF home page. `http://www.esmf.ucar.edu`.

[21] S. J. Fink, S. R. Kohn, and S. B. Baden. Flexible communication mechanisms for dynamic structured applications. In *Third International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR '96), Santa Barbara, CA*, pages 2030–215, August 1996. Also CSD CSE Dept. Tech. Rep. CS96-490, Aug. 1996.

[22] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press/Brooks/Cole Publishing Co., second edition, 2003.

[23] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[24] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, J. A. Montgomery, Jr., T. Vreven, K. N. Kudin, J. C. Burant, J. M. Millam, S. S. Iyengar, J. Tomasi, V. Barone, B. Mennucci, M. Cossi, G. Scalmani, N. Rega, G. A. Petersson, H. Nakatsuji, M. Hada, M. Ehara, K. Toyota, R. Fukuda, J. Hasegawa, M. Ishida, T. Nakajima, Y. Honda, O. Kitao, H. Nakai, M. Klene, X. Li, J. E. Knox, H. P. Hratchian, J. B. Cross, C. Adamo, J. Jaramillo, R. Gomperts, R. E. Stratmann, O. Yazyev, A. J. Austin, R. Cammi, C. Pomelli, J. W. Ochterski, P. Y. Ayala, K. Morokuma, G. A. Voth, P. Salvador, J. J. Dannenberg, V. G. Zakrzewski, S. Dapprich, A. D. Daniels, M. C. Strain, O. Farkas, D. K. Malick, A. D. Rabuck, K. Raghavachari, J. B. Foresman, J. V. Ortiz, Q. Cui, A. G. Baboul, S. Clifford, J. Cioslowski, B. B. Stefanov, G. Liu, A. Liashenko, P. Piskorz, I. Komaromi, R. L. Martin, D. J. Fox, T. Keith, M. A. Al-Laham, C. Y. Peng, A. Nanayakkara, M. Challacombe, P. M. W. Gill, B. Johnson, W. Chen, M. W. Wong, C. Gonzalez, and J. A. Pople. Gaussian 03, revision a.1. Gaussian, Inc., Pittsburgh, PA, 2003.

[25] M. Govett, L. Hart, T. Henderson, J. Middlecoff, and D. Schaffer. The scalable modeling system: Directive-based code parallelization for distributed and shared memory computers. *J. Parallel Computing*, 29(8):995–1020, 2003.

[26] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.

[27] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *TOMS*, 27(4):422–455, December 2001.

[28] High Performance Fortran Forum. High Performance Fortran language specification, version 2.0. Technical report, January 1997.

[29] Ken Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000 (IPDPS 2000)*, pages 297–304, May 2000.

[30] James A. Kohl, Torsten Wilde, and David E. Bernholdt. CUMULVS: Interacting with high-performance scientific simulations, for visualization, steering and fault tolerance. *Intl. J. High-Perf. Computing Appl.*, 2004. Submitted to ACTS Collection Special Issue.

[31] G. Kumfert and T. Epperly. Software in the DOE: The hidden overhead of "the build". Technical report, Lawrence Livermore National Laboratory, UCRL-ID-147343, February 2002.

[32] P. Mehrotra, J. Van Rosendale, and H. Zima. High performance fortran: History, status and future. *Zapata,E. and Padua,D.(Eds.): Parallel Computing, Special Issue on Languages and Compilers for Parallel Computers, Vol.24, No.3-4,pp.325–354 (1998)*, 1998.

[33] J. Michalakes. The same-source parallel MM5. *Scientific Programming*, 8(1):5–12, 2000.

[34] J. G. Michalakes. WRF design and implementation document (Sec. 5). `http://www.mmm.ucar.edu/wrf/users/WRF_arch_03.doc`.

[35] MM5 home page. `http://www.mmm.ucar.edu/mm5`.

[36] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A non-uniform-memory-access programming model for high-performance computers. *J. Supercomputing*, 10(2):169, 1996.

[37] Pacific Northwest National Laboratory. Global Array Toolkit homepage. `http://www.emsl.pnl.gov:2080/docs/global/`, 2004.

[38] D. Quinlan, B. Miller, B. Philip, and M. Schordan. Treating a user-defined parallel library as a domain-specific language. In *7th International Workshop on High-Level Parallel Programming Models and Supportive Environments, part of the 17th International Parallel and Distributed Processing Symposium (IPPS), Ft. Lauderdale*, 2002. April 15–19.

[39] J. R. Rice and R. F. Boisvert. *Solving Elliptic Problems using Ellpack.* Springer-Verlag, 1984.

[40] L. Rivier, R. Loft, and L.M. Polvani. An efficient spectral dynamical core for distributed memory parallel computers. *Monthly Weather Review*, 130(5):1384–1386, 2002.

[41] S.Kothari, G.Daugherty, L.Bishop, and J.Sauceda. A pattern-based framework for detecting software anomalies. *Software Quality Journal, Vol.12, No.2*, pages 99–120, 2004.

[42] L. Tierney. *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics.* Wiley, New York, 1990.

[43] William N. Venables and Brian D. Ripley. *Modern Applied Statistics with S. Fourth Edition.* Springer, 2002. ISBN 0-387-95457-0.

[44] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[45] Weather Research and Forecast (WRF) model coding conventions. `http://www.mmm.ucar.edu/wrf/WG2/WRF_conventions.html`.

[46] WRF home page. `http://www.wrf-model.org`.

[47] Takashi Yanagawa and Kenji Suehiro. Software system of the earth simulator. *Parallel Computing (in print)*, 2004. Special Issue on the Earth Simulator (Masaaki Shimasaki and Hans P. Zima, Editors).

[48] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB - A Tool For Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, pages 1–18, 1988.

6 APPENDIX

6.1 Committee Members

- Stan Ahalt (Ohio Supercomputer Center)
- David Callahan (Cray Inc.)
- William Carlson (IDA/CCS)
- Frederica Darema (National Science Foundation)
- Elmootazbellah Elnozahy (IBM Austin Research Lab)
- Robert Graybill (DARPA/IPTO)
- William Gropp (Argonne National Laboratory)
- Frederick Johnson (Department of Energy)
- Ken Kennedy (Rice University)
- Jeremy Kepner (MIT Lincoln Lab)
- Ewing (Rusty) Lusk (Argonne National Laboratory)
- Piyush Mehrotra (NASA AMES)
- Daniel A. Reed (University of North Carolina at Chapel Hill)
- Vivek Sarkar (IBM T.J. Watson Research Center)
- Lauren L. Smith (NSA)
- Guy Steele (SUN Microsystems Inc.)
- Thomas Sterling (California Institute for Technology and JPL)

6.2 Working Group Charters and Participants

6.2.1 Working Group 1: Core Language and Compilation Issues

Co-Chairs: John Mellor-Crummey (Rice University) and Vivek Sarkar (IBM T.J.Watson Research Center)

Charter

Determine the key issues in languages, compilers, and runtime systems that will decide the success of future high-productivity language systems. Identify the constraints on high-level abstraction in view of the necessity to ensure high-performance target code. Relevant technical topics include multi-threading in massively parallel systems, locality awareness, reuse and modularity. Identify the critical interfaces between language and compiler on the one hand, and operating systems, programming environments, and input/output systems on the other hand. Outline a research roadmap addressing the major language design and implementation problems.

Questions

- What are the key requirements of applications that should be addressed by a high productivity general-purpose programming language?
- What is the right compromise between the goals of high-level language abstraction and target code efficiency?
- Which features of programming environments can support high productivity languages?
- Which innovative ideas in programming models and paradigms promise a dramatic improvement in programming productivity?

Participants

- John Amuedo
- David Bacon
- Robert Blainey
- Jay Brockman
- David Callahan
- Brad Chamberlain
- Guang Gao
- Mary Hall
- Laxmikant Kale
- Bradley Kuszmaul
- Jan-Willem Maessen
- Robert Numrich
- Michael Paleczny
- Rodric Rabbah
- Vijay Saraswat

6.2.2 Working Group 2: Problem-Solving Environments and Domain-Specific Languages

Co-Chairs: William Gropp and Ewing (Rusty) Lusk (Argonne National Laboratory)

Charter

Identify the challenges faced by the development of large-scale applications over a long period of time. Determine the similarities and differences in the approaches represented by problem-solving environments (PSEs) and domain-specific languages (DSLs), and compare these approaches to general-purpose languages. Identify the issues involved in transforming applications from a prototyping to a production environment. Determine the issues involved in porting legacy codes. Outline a research roadmap addressing the major problems to be pursued in the areas of PSEs and DSLs.

Questions

- What are the characteristics of applications supported best by PSEs or DSLs?
- What kind of support should languages, compilers, and programming environments provide for large-scale application development?
- What are the major pros and cons in terms of high-productivity programming when comparing PSEs and DSLs with general-purpose languages?
- What are the issues involved in the transition of legacy codes? Should they be rewritten, and if so, how? (automatic support?) What can be done to lower the cost of porting existing applications to new programming languages and environments?

Participants

- Stan Ahalt
- David Bernholdt
- Robert Harrison
- Bill Harrod
- Paul Hovland
- Fred Johnson
- Ken Kennedy
- Cheryl McCosh
- John Michalakes
- Cleve Moler
- Dan Quinlan
- P Sadayappan
- Andrew Siegel
- Carl Shapiro

6.2.3 Working Group 3: Program Development Support Systems

Chair: William Carlson (IDA/CCS)

Charter

Identify the key problems facing the developers of advanced application codes. Relevant issues include scalability support, multiple programming paradigms, seamless integration of legacy codes, collaborative development by geographically distributed groups, and the management of application lifecycles. Establish a catalog of methods required to deal with these issues and identify the support required from languages, compilers, and programming environments. Outline a research roadmap addressing the major problems to be dealt with in this area.

Questions

- What are the major challenges for program development support systems in the context of future high productivity languages and programming paradigms?
- What kind of support should languages, compilers, and programming environments provide for dealing with these issues?
- How can dynamically evolving libraries be successfully used by program development support systems?
- What are the major language and compiler-related research issues that need to be addressed?

Participants

- Eric Allen
- Larry Bergman
- Kemal Ebcioglu
- Dan Katz
- Bob Kuhn
- Gary Kumfert
- Jose Munoz
- Jose Moreira
- Rod Oldehoeft
- Jarek Nieplocha

6.2.4 Working Group 4: Programming Environments and Tools

Chair: Daniel Reed (University of North Carolina)

Charter

Address the components of programming environments essential for high-productivity language systems. Major topics in this context include support for fault tolerance, high-level debugging, performance analysis and tuning, and autonomy. Identify the relationship of these components to language features and advanced compilation and runtime system approaches. Outline a research roadmap that addresses the major open problems that need to be addressed.

Questions

- What are the limitations of current programming environments and tools from the viewpoint of high productivity programming? Which new features and technlogies are required?
- What kind of support can the programming environment provide for dealing transparently with faults in systems with million-way parallelism?
- What kind of broad issues need to be addressed to deal effectively with automatic or semi-automatic performance tuning?
- How will the interfaces between the programming environment and the language, compiler, and runtime system evolve in high productivity language systems? What are the changes that compilers and runtime systems will undergo?

Participants

- David Chase
- Bob Hood
- Mark James
- Alan Kielstra
- Ashok Krishnamurthy
- Pat Miller
- Lawrence Rauchwerger
- Dolores Shaffer
- Keith Shields

6.3 List of Attendees

- Stan Ahalt (Ohio State University)
- Eric Allen (Sun Microsystems Inc.)
- George Almasi (IBM T.J.Watson Research Center)
- John Amuedo (Signal Inference Corp.)
- David Bacon (IBM T.J. Watson Research Center)
- Larry Bergman (JPL)
- David Bernholdt(Oak Ridge National Laboratory)
- Robert Blainey (IBM)
- David Callahan (Cray Inc.)
- Bill Carlson (IDA/CCS)
- Brad Chamberlain (Cray Inc.)
- David Chase (Sun Microsystems Inc.)
- Elmootazbellah Elnozahy (IBM Austin Research Laboratory)
- John Feo (Cray Inc.)
- Guang Gao (University of Delaware)
- Robert Graybill (DARPA)
- William Gropp (Argonne National Laboratory)
- Mary Hall (University of Southern California)
- Robert Harrison (Oak Ridge National Laboratory)
- Bill Harrod (SGI)
- Bob Hood (NASA AMES)
- Paul Hovland (Argonne National Laboratory)
- Mark James (JPL)
- Fred Johnson (U.S. Department of Energy)
- Laxmikant Kale (University of Illinois)
- Dan Katz (JPL)
- Ken Kennedy (Rice University)
- Allen Kielstra (IBM)
- Peter Kogge (University of Notre Dame)
- Ashok Krishnamurthy (Ohio Super Computer Center)

- **Bob Kuhn (Intel Corp.)**
- **Gary Kumfert (Lawrence Livermore National Laboratory)**
- **Bradley Kuszmaul (MIT-CSAIL)**
- **Ewing (Rusty) Lusk (Argonne National Laboratory)**
- **Jan-Willem Maessen (Sun Microsystems Inc.)**
- **Cheryl McCosh (Rice University)**
- **John Mellor-Crummey (Rice University)**
- **John Michalakes (NCAR)**
- **Pat Miller (Lawrence Livermore National Laboratory)**
- **Cleve Moler (MathWorks)**
- **Jose Moreira (IBM T.J.Watson Research Center)**
- **Jose Munoz (National Science Foundation)**
- **Jarek Nieplocha (PNNL)**
- **Robert Numrich (University of Minnesota)**
- **Rod Oldehoeft (Los Alamos National Laboratory)**
- **Michael Paleczny (Sun Microsystems Inc.)**
- **Dan Quinlan (Lawrence Livermore National Laboratory)**
- **Rodric Rabbah (MIT)**
- **Lawrence Rauchwerger (Texas A&M University)**
- **Daniel Reed (University of North Carolina)**
- **P Sadayappan (Ohio State University)**
- **Vijay Saraswat (IBM)**
- **Vivek Sarkar (IBM T.J.Watson Research Center)**
- **Dolores Shaffer (Science & Technology Associates Inc.)**
- **Keith Shields (Cray Inc.)**
- **Andrew Siegel (Argonne National Laboratory)**
- **Lauren Smith (NSA)**
- **Larry Snyder (University of Washington)**
- **Guy Steele (Sun Microsystems Inc.)**
- **Thomas Sterling (Caltech/JPL)**
- **Ed Upchurch (Caltech/JPL)**
- **Kathy Yelick (University of California Berkeley)**
- **Hans Zima (University of Vienna/JPL)**