Research Agenda for

High Productivity Language Systems

1 Introduction

High performance computing is a strategic tool for leadership in science and technology, providing the superior computational capability required for dramatic advances in national security applications as well as in fields such as DNA analysis, drug design, data mining, and structural engineering. Over the past decade, emerging technology problems have posed serious challenges for continued advances in this field. One of the key problems is the lack of adequate language and tool support for high performance computing. In today's dominating programming paradigm users are forced to adopt a low level programming style 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.

With the emergence of **High Productivity Computing Systems (HPCS)** architectures delivering peta-scale performance in the near future this problem will become even more acute: it is clear that state-of-the-art software technology cannot adequately support such architectures. In this document, we address this issue by describing a research agenda for **High Productivity Language Systems (HPLS)**. The goal is to make scientists and engineers more productive by increasing programming language usability and time-to-solution, without sacrificing performance. Reaching this goal requires an integrated research effort including the design of general-purpose languages together with the supporting compiler, runtime, and tool infrastructure as well as domain-specific languages and problem-solving environments. The purpose of the research agenda is to provide guidance for the planning of future research programs in this field. The results of the work in Phase 2 of the HPCS program will provide an infrastructure for the research effort.

2 State-of-the-Art

Since the 1990s, the integration of commercial off-the-shelf processing and memory components allowed the synthesis of large computing systems at affordable cost, either through custom MPPs or commodity clusters. As a consequence, hundreds of clusters of SMPs have been installed worldwide in government laboratories, industry, and academic sites, with the largest systems reaching theoretical peaks on the order of tens of teraflops.

However, despite their impressive peak capabilities these systems have not delivered satisfactory performance for many important applications, and their support for program development, debugging, and performance tuning is far below the standard for commercial desktop systems. The key issues can be summarized as follows:

- The **hardware infrastructure** of most current high end computing systems is built for the mass market, providing little specific support for parallel computation and limiting the scalability of applications as a result of severe constraints in latency and bandwidth.

- The dominating **programming paradigm** is based upon standard sequential programming languages (Fortran and C/C++), augmented with message passing constructs. In this model the fragmented nature of memory is explicit, enforcing a local view of data and communication intertwined with algorithms. 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.

- **Programming environments and tools** for high end systems 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.

3 New Challenges and Requirements

The rising scale and complexity of the emerging peta-scale 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 will have tens to hundreds of thousands processing components, 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 more than 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. Furthermore, the bandwidths available at different levels of the memory/interconnect hierarchies will differ dramatically.
- **Applications: Scale, Complexity and Time-Scale Mismatch**
  High end 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. Furthermore, applications are often developed and used over decades, whereas the hardware and software systems on which they run may change every few years. As a consequence, the efficient porting of legacy codes to new architectures is a highly important problem for which little automatic support exists today.

4 Research Agenda 2005 – 2010

This is the main section of the document, outlining guiding principles for HPLS as well as the process leading to an HPLS, and providing a short overview of key research topics.

4.1 Guiding Principles

There are two sets of **guiding principles**: the guiding principles for a High Productivity Language System (HPLS) and the guiding principles for the *development process* leading to a HPLS.
The guiding principles specific to HPLS are a set of properties which the language system must embody:

1. **Parallelism and Locality**: Formulation and efficient control of parallelism which scales to tens of thousands of processors/processes/threads must be combined with data and program locality which yields efficient execution at the single thread level.
2. **Programming in the Large**: Composability of programs from components at multiple levels of abstraction and functionality ranging from elemental operations across adaptible libraries to complete applications written in other languages.
3. **Abstraction and Analysis**: Programs and systems should be analyzable for structural, correctness and performance properties from design through production runtime.

4. **Extensibility**: The language system should be extensible at the user level to enable customization for efficient use and efficient execution in different application domains and execution environments.
5. **Fault and Uncertainty Management**: Provision for the specification, detection, and recovery from faults must be intrinsic in the language.

This set of properties infers features and mechanisms for the language currently thought to include: **global name space**, **object-orientedness**, and **generic programming**. The related set of mechanisms and insuring consistency among them is a portion of the research agenda.

The guiding principles for the research and development *process* leading to HPLS include:

1. **User Involvement**: The community of customers for the HPLS must be involved at all stages of development, from design to benchmarking.
2. **Evaluation**: Benchmarking and evaluation of the language system for adherence to the guiding principles and consistency of features and mechanisms should be conducted regularly during the research and development process.
3. **Education, Training, and Marketing**: Benefit will only accrue if the HPLS is widely adopted and used. Education, training, and marketing must be begun early with demonstration of quantifiable benefits.

4.2 Research Topics

This section sketches research topics and the associated research challenges in some critical areas.

4.2.1 Explicit Concurrency and Locality Awareness

Single thread effectiveness and scalable parallelism must be realized together. The effective exploitation of complex parallel and distributed architectures requires integration of concurrency and locality specification and management. New generations of adaptive applications require parallelism and locality to be explicitly adapted during execution. These requirements cannot be realized from conventional specifications by current compiler technology; therefore (1) explicit representations for scalable, dynamic and adaptive concurrency covering a broad range of granularities and supporting different parallel programming paradigms must be developed; (2) new methods must be provided to express locality at a high level of abstraction, including features for user-defined data distribution, alignment, and data/thread affinity, and covering both dense and sparse data structures; and (3) the tradeoffs between compilation time and runtime must be explored.

4.2.2 Management of Distributed Collections in a Global Address Space

The language should support general types of collections, such as arrays, sequences, sets, and graph-based data structures, and provide efficient implementations of aggregate operations such as reduction and parallel prefix. The control of large-scale collections of threads distributed across collections of distributed resources leads to coordination problems of unprecedented scale and complexity, requiring the development of new abstractions for coordination based on distributed control.

4.2.3 Programming in the Large

Multidisciplinary applications, which utilize libraries and components from multiple disciplines and/or couple several discipline-specific applications into a single focused application are becoming an important paradigm. Applications will typically incorporate and integrate both computationally and data intensive phases. Languages must provide support for composition of systems from components with high levels of semantic complexity. These challenges have not been previously faced at the language level; they include the difficult problem of specifying mappings across semantically different interfaces.

4.2.4 Compilation and Runtime Technology

Effective realization of the specifications for concurrency, locality, coordination, composition and dynamic structuring outlined above will require compilers which utilize much greater depth of analysis than current compilers. Runtime systems, which are currently largely passive providers of functionality, will need more measurement and analysis capabilities and will become active providers of feedback to executing applications. A comprehensive level of coordination and integration between compilers and runtime systems will be required. Specific research efforts are needed in the areas of scheduling massively parallel threads across a hierarchy of processors and optimizing communication for irregular and adaptive comuputations.

Compilers and runtime systems must provide more powerful capabilities for the analysis of program correctness. Safety with respect to all forms of errors (including type, pointer, and initialization errors) which can be detected by static analysis should be provided by the compiler. Runtime system capabilities for error detection should be coordinated with compiler validation.

4.2.5 Tools for High-Level Programming Support

Programming environments and tools encompass a wide range of software, from software development environments to debugging systems and performance analysis tools. The emergence of very large systems, higher probabilities of partial system failures and rapidly rising code complexity lead to new challenges for programming environments. The key research topics in this area include:

1. **Fault Tolerance and Resilience**
   We must develop new software that embodies the two important realities of future large-scale systems: (i) frequent hardware component failures are to be expected as part of normal operation and (ii) 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, 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 be explored. Redundancy based on reliability models would allow application developers and runtime systems to minimize the effect of failures.

2. **Scalability and Intelligence**
   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. "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, 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, standard information sharing interfaces across compilers, libraries and runtime systems need to be defined.

3. **Tool Integration and Support**
   For future peta-scale environments, users will want fully integrated programming environments with tools that are dynamically adaptive. 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 be included.

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 are needed across all levels. This will require collaborative co-design – when programming languages and models are designed, the design of programming environment and tools should be included in the process.

To allow tool composability and reuse, interoperable frameworks must 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.

4.2.6 Performance-Guided Re-Engineering of Legacy Codes

Existing application codes are a rich legacy of functionality, knowledge and intellectual property. There is no possibility of user acceptance of a new language which does not provide powerful support for migration and integration of legacy codes. Re-engineering of these codes for composition into system-level applications and the extraction of components for new applications must be fully supported in a new language. This requires the integration of tool support for the re-engineering of programs in the native language of the application, including support for mapping, composition, and automated source-to-source transformations to generate interfaces.

4.2.7 Domain-Specific Languages

Domain-specific languages (DSLs) build upon techniques used for general-purpose languages and do not require completely new research directions. However, DSLs can benefit from specializations of approaches used for general-purpose languages. Since DSLs provide a powerful and relatively low-cost way of raising the level of abstraction for a problem domain, improving the quality of and the speed with which domain-specific languages are developed is an important component in improving the productivity of high-performance computing systems.

Domain-specific languages emphasize a high-level of abstraction, combined with detailed knowledge of the problem area to provide one, albeit special purpose, solution to the productivity problem. This allows DSLs to address some special needs that may be difficult to integrate into general-purpose languages. Further, many DSLs are implemented as source-to-source translators and can provide valuable information on the capabilities that are needed in a general-purpose, high-productivity language.

The following research areas – which partially overlap with research in general-purpose languages and tools – are of special interest for DSLs:

1. **Programming in the Large** (see Section 4.2.3)
2. **Library Management** – techniques to integrate modules from domain-specific languages into code libraries, including dynamic generation of code
3. **Telescoping** – a strategy for the optimized compilation of programs in very high-level (scripting) languages with calls to library routines
4. **Knowledge-based Approaches**
   The domain knowledge inherent in a domain-specific language can be used for the improvement of static program analysis, verification and validation, fault tolerance, and the detection and optimization of domain-specific patterns in serial legacy codes. Furthermore, domain knowledge may contribute to the selection of specialized algorithms or code generation techniques.

In addition, research in source-to-source transformations, such as those intended for legacy code migration or for the application of performance enhancement can provide valuable tools for aiding in the development of domain-specific languages, particularly for niche areas that cannot support the development of the basic language infrastructure.

4.3 Language Benchmarking

The above discussion was dominated by the guiding principles of HPLS and the required properties of such languages. In this section, we turn to the issue of evaluating and measuring the productivity of HPLS.

An increase in productivity can be defined as producing "better" software with smaller turn-around times at lower cost. In turn, "better" can be defined in terms of both software quality, such as maintainability, composability, and reusability, and the performance of the resulting execution code. Most benchmarking work in the past has considered only the latter. What is needed are research efforts that focus on producing metrics and benchmarks to measure the effect of language systems on the overall productivity of high-end computing systems.

The key issue here is defining the right metrics, dealing with the tradeoff between programming effort and target code performance, and including the amount of time to develop a solution and its cost. Along with the metrics, it is necessary to develop benchmark programs and methodologies that can be utilized to determine the overall effectiveness and efficiency of new language systems beyond the state-of-the-art. The focus of such benchmark programs could range from measuring the language power of key constructs in isolation to full programming environments encompassing the complete support system. One major problem here is the subjectivity of some of the measures. For example, ease of use may be critically dependent on the experience of the user. The challenge is to develop the metrics and experimental methodologies that take subjectivity into account to produce maningful results measuring the relative power and efficiency of language systems.

5 Outline of a Research Agenda 2010 – 2015

Below we list research topics that are considered relevant in the longer term. These topics may change depending on the features of emerging architectures and shifting trends in application development strategies. They will also be influenced by the results delivered during the first phase of the research agenda.

1. automatic support for data distribution, alignment, and data/thread affinity
2. advanced techniques for language extensibility
3. advanced study of type inference in the context of cloning, data structure specialization, and telescoping
4. extending the use of functional and declarative language features for high performance
5. use of transactional programming models for management of shared data structures
6. intelligent support systems for performance-guided legacy code migration
7. increased support for integrated software design that deeply couples programming models and tools,
8. 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,
9. asynchronous agent systems for introspection,
10. advanced testing and exploration of intelligent introspection and adaptation autonomic behavior, and
11. exploration of very high level abstraction mechanisms that reason about execution aggregates.

6 Research Infrastructure Provided by the HPCS Project

In this section we outline the expected contributions of the vendors in the HPCS project to a language research program by the time Phase 2 ends (July 2006).

6.1 Cray

1. Chapel language specification
2. Chapel reference implementation, using a source-to-source translation from Chapel to C extended by a parallel library.
3. Key application benchmarks and test cases written in Chapel
4. Performance and debugging tools supporting Chapel

An outline on Cray's language research direction is prsented in the Appendix (A1).

6.2 IBM

1. X10 language specification and tutorial documents
2. X10 reference implementation (for correctness, not performance), which includes:
   (a) X10-to-Java source-to-source translator
   (b) X10 runtime system and API implemented in Java
   (c) Eclipse plug-in for X10
3. Set of test cases and benchmarks written in X10
4. Performance Explorer tool for visualizing abstract/idealized parallel performance metrics in an X10 program execution

6.3 Sun

Sun expects to contribute certain pieces of software, such as portions of an experimental compiler (for example, a parser or type analysis phase) or an experimental IDE. However, at this point in time it is considered too early to make specific commitments.

An outline of Sun's priorities in the language research agenda is presented in the Appendix (A2).

Acknowledgment

This research agenda is the result of activities that were initiated by the *Workshop on High Productivity Languages and Models* in Santa Monica, California, in May 2004, and continued during the following year, guided by an Advisory Committee chaired by Bob Graybill from DARPA. Many individuals have contributed to this document, including (in alphabetic order) Stan Ahalt, James Browne, David Callahan, William Carlson, Frederica Darema, Elmootazbellah Elnozahy, Howard Gordon, Robert Graybill, William Gropp, Frederick Johnson, Robert Lucas, Ken Kennedy, Jeremy Kepner, Ewing (Rusty) Lusk, Piyush Mehrotra, John Mellor-Crummey, Daniel Reed, Burton Smith, Lauren L.Smith, Guy Steele, Vivek Sarkar, and Thomas Sterling.

Hans Zima, Editor
April 2005

7 Appendix

A1. Directions for Research in High Productivity Programming Languages

Burton Smith, Cray Inc.

History

Although the most popular parallel programming languages were and are motivated by pure performance rather than productivity, there have been many attempts to achieve programming ease without losing too much performance. Functional languages such as Sisal, Id, and Nesl were among the most successful of these alternatives. Languages that adopted a global rather than processor-centric view of the computation also represented a step forward; High Performance Fortran and ZPL are examples. These alternative languages required more serious investments in compiler technology than their low-level competition (OpenMP, PVM, MPI). More recently, Partitioned Global Address Space (PGAS) languages like UPC and Co-array Fortran introduce a global address space and fine-grained single-sided access to remote memory at moderate cost in compiler effort.

At the same time, scientists interested in getting work done have increasingly been turning to much higher level languaqges (e.g. Matlab) to design their programs. Due to the poor performance that results, a staff of "coders" are typically employed to manually translate the high-level program to C++ or Fortran and MPI.

Language Ideas

There are undoubted possibilities to improve the performance of high-level languages like Matlab, but additional potential could be realized by rethinking language design to address the basic issues of parallel programming in a unified way. Some of the concepts that should be explored are:

- Explicit parallel syntax
- Generalized data aggregates
- User-defined data distributions
- Affinity between computations and data
- Global view of computations
- Separation of concerns
- Implicit data types
- Support for genericity
- Object orientation
- Fine-grain atomic transactions
- Race-free programming
- Performance transparency
- Succinctness
- Language interoperability
- Breadth of applicability
- Interval arithmetic

Strategy

A single high-level language for the whole parallel computing community is a desire of many. This objective is achievable eventually, but it will not be easy because language coherence is not ensured by language standardization. A well-designed language is much more than a melange of ideas, some from here and others from there. There are always alternative and competing approaches to accomplish the same objective; deciding among them is challenging because the components of a language mutually interact, often subtly and synergistically.

There is another reason to pursue multiple languages at least in the short run. The dearth of funding for language and compiler research since 1990 has greatly depleted the pool of new ideas. Worse yet, many of us believe that the field has forgotten much of what it knew in the 1980's. At least for the near future, we need more quantity and diversity in language research and development rather than less.

A2. Research Directions at Sun

Guy Steele, Sun Microsystems, Inc.

Sun's contributions and/or proposed research topics for High-Productivity Language Systems are likely to fall in the following areas:

1. The melding of programming-language syntax (and particularly object-oriented programming-language syntax) with mathematical notation, and the development of appropriate parsing techniques. Initial investigations indicate that purely syntactic parsing strategies are inadequate for this task; a certain amount of global information (type information) is required to disambiguate certain cases. For example, the phrase "x y z" is typically to be interpreted as the product of three variables x, y, and z, but the syntactically similar phrase "x log n" is to be interpreted as the product of a variable x and a function call log(n). As another example, superscripts are used in various ways in mathematics: as exponents, as tensor indices, and as arbitrary tags. The different uses must be distinguished by context. Research is needed on the best ways to do this.

2. Research on how to allow programmers to extend the core language syntax in controlled ways, in order to support domain-specific languages and syntax tailored for particular libraries. We have in mind something much more disciplined than traditional macro mechanisms.

3. Appropriate notations for keyboarding and display of mathematical programs. One possibility is TeX-like or Wiki-like notations, where the intent is that an ASCII-based syntax is convenient for keyboard input and reasonably readable by itself, but is intended to indicate greater typographical variety in a fairly direct way. For example, "x_max" might indicate a variable named "x" tagged with a fixed subscripted label "max". Similarly, if such a notation were to use " ˆ" to indicate superscripts, then "x$\hat{2}$" would indicate the squaring of "x" whereas "x$\hat{*}$" might indicate the complex conjugate of "x" (a superscripted asterisk as a unary postfix operator). Such keyboarding techniques have implications not only for language design but for the design of tools such as IDEs.

4. Object-oriented type systems. We are interested in introducing a more powerful and more rigorous parametric polymorphism than is available in, say, the C++ or Java programming languages. Dynamic dispatch on multiple arguments of overloaded methods may be more convenient than the hybrid dyanamic/static dispatch now provided by Java. We view this as a mechanism likely to be used more heavily by library designers than by authors of application code. We conjecture that a certain amount of automatic type inference is desirable to avoid requiring application programmers to clutter their code with elaborate type annotations. Research is needed (1) on how best to design a type system to support the appropriate amount of type inference, and (2) how to design error reporting mechanism that report problems clearly to programmers when type inference fails.

5. Compile-time analysis of dimensional units (such as meters, seconds, and kilograms). Appropriate syntax and semantics for dimensional units and their interaction with other aspects of a type system. We are particularly interested in knowing how much needs to be built into a language specifically to support dimensional units and how much can be provided by more general type mechanisms. Ideally, the language itself makes no commitment to a particular system of units, but provides the necessary infrastructure for defining one or more appropriate systems of units in libraries. Research is needed in particular to determine how units need to interact with I/O and serialization facilities.

6. Better support for programming in the large. Support for organizing code into mix-and-match components. Support for simultaneous installation of different versions of the same component. Explicit control of components, APIs, and linking expressed within the language rather than through an external mechanism such as Unix "make", for example. Support for unit testing and verification of declared contracts and invariants. In addition to providing more robust behavior for deployed programs, and allowing for controlled upgrades, encapsulated upgradable components also facilitate better performance, in several ways. First, they improve start-up time for a runtime, since only libraries used by an application are loaded. Second, they allow for cross-component and "telescoping languages"-style optimizations.

7. Exploring the division of compilation labor between static (performed prior to program execution) and dynamic (performed during program execution) that yields best performance. We propose to build on lessons learned about dynamic complication within various implementations of the Java Virtual Machine, such as HotSpot. We also propose to systematically use performance data gathered from previous executions to guide dynamic compilation decisions.

8. How to design a language to support an appropriate division of labor between domain experts and computational experts (where the former are responsible for correctness and the latter for performance). How to design a language so that a working code may be effectively tuned for execution on a large parallel machine without requiring extensive reorganization of the source code.

9. How to design a language to support an appropriate division of labor between application-specific coders and library designers. We conjecture that the coding of general-purpose libraries is a rather different process from that of coding specific applications and requires rather different features from the language. How can a language provide features to support the coding of libraries without causing the application programmer to trip over them? We conjecture, for example, that a library writer needs fairly powerful facilities for exploiting parametric polymorphism, and that if such facilities are correctly designed then the application programmer using such a library need be barely aware of their existence.

10. Facilities for describing the distribution of execution labor among processors and the distribution of data among memories. This is a prime example of what ought to be done in a library rather than wired into the language. We envision the basic language providing bare-bones facilities for controlling distribution, then providing higher-level facilities that define policy and provide convenient notations; it should be possible to code such higher-level facilities as libraries. In particular, we envision a design in which the specific strategies supported by High Performance Fortran or Chapel can be expressed as libraries.

11. Facilities for better interprocess and/or interthread communication and synchronization based on the paradigm of a global shared memory address space.