Parallel Processing Letters, Vol. 17, No. 1 (2007) 89–102

World Scientific
www.worldscientific.com

LANGUAGES FOR HIGH-PRODUCTIVITY COMPUTING: THE DARPA HPCS LANGUAGE PROJECT

EWING LUSK
Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue Chicago, Illinois 60439, USA

and

KATHERINE YELICK
University of California at Berkeley and Lawrence Berkeley National Laboratory Berkeley, California 94720-1776, USA

Received December 2006
Revised January 2007
Communicated by Guest Editors

ABSTRACT

We present a summary of the current state of DARPA's HPCS language project. We describe the challenges facing any new language for scalable parallel computing, including the strong competition presented by MPI and the existing Partitioned Global Address Space (PGAS) Languages. We identify some of the major features of the proposed languages, using MPI and the PGAS languages for comparison, and describe the opportunities for higher productivity along with the implementation challenges. Finally, we present the conclusions of a recent workshop in which a concrete plan for the next few years was proposed.

Keywords: High-Productivity Computing Systems, X10, Chapel, Fortress.

1. Introduction

In 2002 DARPA initiated the "High Productivity Computing Systems" project, with the goal of accelerating both the performance of the largest parallel computers and their usability. It was recognized that a significant barrier to the application of computing to science, engineering, and large-scale processing of data was the cumbersomeness of developing software that exploits the power of new architectures. As part of the HPCS project, three computer vendors—Cray, IBM, and Sun—have competed not only in the area of hardware design to address DARPA's performance goals, but also in language design to address the software development productivity goals. In this paper we present a snapshot of the language design efforts, comparing them with one another and also with the current non-HPCS major programming methodology, the message-passing model represented by the MPI standard.

In Section 2 we describe the current lay of the land in parallel programming and identify the challenges faced by ambitious language development projects. In Section 3 we outline some of the features of MPI that software developers have

found useful and that new approaches are likely to need in order to become widely adopted. In Section 4 we compare the three HPCS languages along several axes in order to facilitate an understanding of them as a group. In Section 5 we report the findings of a recent HPCS language workshop in which plans were made to initiate a new phase of the HPCS language development project, targeted at an eventual single high-productivity language. Our conclusions are presented in Section 6.

2. Background

In this section we present our view of the current state of scalable parallel programming. We discuss current practice, compare some early production languages with the HPCS languages, and comment on previous efforts to introduce new programming languages for productivity and parallelism.

2.1. Architectural Developments

Language development for productivity is taking place at a time when the architecture of large-scale machines is still an area of active change. Innovative network interfaces and multiprocessor nodes are challenging current programming model implementations to provide access to the best performance the hardware can provide, and multicore chips are providing another level of the processing hierarchy. The HPCS hardware efforts are at the leading edge of these innovations. By combining hardware and languages in one program, DARPA is allowing language designs that may take advantage of unique features of one system, although this design freedom is tempered by the desire for language ubiquity. New programming models and languages will be expected to exploit the full power of new architectures, while still providing reasonable performance on more conventional systems.

2.2. Current Practice

Most parallel programs for large-scale parallel machines are currently written in a conventional sequential language (Fortran-77, Fortran-90, C, or C++) with calls to the MPI message-passing library. The MPI standard [11,12] defines bindings for these languages. Bindings for other languages (particularly Java [13]) have been developed and are in occasional use but are not part of the MPI standard. MPI is a realization of the message-passing model, in which processes with completely separate address spaces communicate with explicit calls to `send` and `receive` functions. MPI-2 extended this model in several ways (parallel I/O, remote memory access, and dynamic process management), but the bulk of MPI programming utilizes only the MPI-1 functions. The use of the MPI-2 extensions is more limited, but usage is increasing, especially for parallel I/O.

2.3. The PGAS Languages

In contrast with the message passing model, the Partitioned Global Address Space (PGAS) languages provide each process direct access to a single globally addressable

space. Each process has local memory and access to the shared memory.[a] This model is distinguishable from a symmetric shared-memory model in that shared memory is logically partitioned, so there is a notion of near and far memory explicit in each of the languages. This allows programmers to control the layout of shared arrays and of more complex pointer-based structures.

The PGAS model is realized in three languages, each presented as an extension to a familiar base language: UPC (Unified Parallel C) [6] for C; Co-Array Fortran (CAF) [14] for Fortran, and Titanium [17] for Java. The three PGAS languages make references to shared memory explicit in the type system, which means that a pointer or reference to shared memory has a type that is distinct from references to local memory. These mechanisms differ across the languages in subtle ways, but in all three cases the ability to statically separate local and global references has proven important in performance tuning. On machines lacking hardware support for global memory, a global pointer encodes a node identifier along with a memory address, and when the pointer is dereferenced, the runtime must deconstruct this pointer representation and test whether the node is the local one. This overhead is significant for local references, and is avoided in all three languages by having expressions that are statically known to be local, which allows the compiler to generate code that uses a simpler (address-only) representation and avoids the test on dereference.

These three PGAS languages share with the strict message-passing model a number of processes fixed at job start time, with identifiers for each process. This results in a one-to-one mapping between processes and memory partitions and allows for very simple runtime support, since the runtime has only a fixed number of processes to manage and these typically correspond to the underlying hardware processors. The languages run on shared memory hardware, distributed memory clusters, and hybrid architectures.

Each of these languages is the focus of current compiler research and implementation activities, and a number of applications rely on them. All three languages continue to evolve based on application demand and implementation experience, a history that is useful in understanding requirements for the HPCS languages. UPC and Titanium have a set of collective communication operations that gang the processes together to perform reductions, broadcasts, and other global operations, and there is a proposal to add such support to CAF. UPC and Titanium do not allow collectives or barrier synchronization to be done on subsets of processes, but this feature is often requested by users. UPC has parallel I/O support modeled after MPI's, and Titanium has a bulk I/O facilities as well as support to checkpoint data structures based on Java's serialization interface. All three languages also have support for critical regions and there are experimental efforts to provide atomic operations.

The distributed array support in all three languages is fairly rigid, a reaction to

[a] Because they access shared memory, some languages use the term "thread" rather than "process."

the implementation challenges that plagued the High Performance Fortran (HPF) effort. In UPC distributed arrays may be blocked, but there is only a single blocking factor that must be a compile-time constant; in CAF the blocking factors appear in separate "co-dimensions;" Titanium does not have built-in support for distributed arrays, but they are programmed in libraries and applications using global pointers and a built-in all-to-all operation for exchanging pointers. There is an ongoing tension in this area of language design, most visible in the active UPC community, between the generality of distributed array support and the desire to avoid significant runtime overhead.

2.4. The HPCS Languages

As part of the Phase II of the DARPA HPCS Project, three vendors—Cray, IBM, and SUN—were commissioned to develop new languages that would optimize software development time as well as performance on each vendor's HPCS hardware being developed over the same time period. Each of the languages—Cray's Chapel [5], IBM's X10 [15], and Sun's Fortress [1]—provides a global view of data (similar to the PGAS languages), together with a more dynamic model of processes and sophisticated synchronization mechanisms. Salient features of the HPCS languages are considered in Section 4.

The original intent of these languages was to exploit the advanced hardware architectures being developed by the corresponding vendors, and in turn to be particularly well supported by these architectures. However, in order for these languages to be adopted by a broad sector of the community, they will also have to perform reasonably well on other parallel architectures, including the commodity clusters on which much parallel software development takes place. (And, in turn, the advanced architectures will have to run "legacy" MPI programs well in order to facilitate the migration of existing applications.)

Until recently, the HPCS languages were developed quite independently by the vendors. In the past year and a half, DARPA has also funded a small, academically based effort to consider the languages together, foster vendor cooperation, and perhaps eventually develop a framework for convergence to a single high-productivity language [9]. Recent activities on this front are described in Section 5.

2.5. Cautionary Experiences

The introduction of a new programming language for more than research purposes is a speculative activity. Much effort can be expended without creating a permanent impact. We mention two well-known cases.

In the late 1970s and early 1980s, an extensive community effort was mounted to produce a complete, general-purpose language expected to replace both Fortran and COBOL, the two languages in most widespread use at the time. The result, called Ada, was a large, full-featured language and even had constructs to support parallelism. It was required for many U.S. Department of Defense software con-

tracts, and a large community of programmers eventually developed. Today Ada is used within the defense and embedded systems community, but it did not supplant the established languages.

A project with several similarities to the DARPA HPCS program was the Japanese "5th Generation" project of the 1980s. Like the DARPA program, it was a ten-year project involving both a new programming model, presented as a more productive approach to software development, and new hardware architectures designed by multiple vendors to execute the programming model efficiently and in parallel. The language realizing the model, called CLP (Concurrent Logic Programming), was a dialect of Prolog specifically engineered for parallelism and high performance. The project was a success in training a generation of young Japanese computer scientists, but it has had no influence on the current parallel computing landscape.

2.6. Lessons

Programmers *do* value productivity, but reserve the right to define it. Portability, performance, and incrementality seem to have mattered more in the recent past than elegance of language design, power of expression, or even ease of use, at least when it comes to programming large scientific applications. Successful new sequential languages have been adopted in the past twenty-five years, but each has been a modest step beyond an already established language (from C to C++, from C++ to Java). While the differences between each successful language have been sigificant, both timing of the language introduction and judicious use of familiar syntax and semantics were important. New "productivity" languages have also emerged (Perl, Python, and Ruby); but some of their productivity comes from their interpreted nature, and they are neither high-performance nor particularly suited to parallelism.

The PGAS languages, being smaller steps beyond established languages, thus present serious competition for the HPCS languages, despite the advanced, and even elegant, features provided in Chapel, Fortress, and X10. The most serious competition, however, comes from the fact that the current status quo based on MPI appears adequately productive for a wide range of applications. In the next section we describe some of the "productive" features of MPI, as a way of setting the bar for the HPCS languages. That is, new approaches to scalable parallel programming must be at least as good as MPI + Fortran, so in looking at MPI we are reviewing some of the features that the HPCS languages must eventually have.

3. MPI and Productivity

MPI [11,12] is a standard definition of a library interface implementing the message-passing model and (in MPI-2) extending that model to include parallel I/O, remote memory access, and dynamic process management. It was defined by the MPI Forum, a voluntary collection of parallel computer vendors, computer scientists, and application developers who participated in an intensive sequence of meetings,

first in 1992-93 (for the original MPI) and again in 1995-97 (for MPI-2). Once the standard was defined, multiple implementations appeared quickly, both from the vendors who had participated (and some who had not) and from open source research and development groups.

MPI has the advantage of representing a complete definition of a well-understood model, and it was not conceptually difficult for multiple implementations to appear. As a library rather than a language, it is orthogonal to compiler issues and works immediately with existing optimizing compilers for its sequential host languages, performance monitors, and debuggers.

MPI was not designed with convenience in mind; rather, it was intended to provide the portability layer for its message-passing programming model. It also provided explicit support for parallel libraries through its communicator feature, with the idea that convenience would be provided by application-oriented libraries, whose development was encouraged by MPI's portability.

Despite this intention, which does make MPI programming cumbersome at times, MPI provides a number of productivity-oriented features. It was designed for portability, even to heterogeneous networks of machines, and this portability not only has provided leverage to applications, which can run on multiple large-scale machines, but also has enabled a development path from laptops to commodity clusters to the largest computers available. MPI provides performance transparency in the sense that the execution model for an MPI program, in which transmission of data from one address space to another is a relatively expensive operation, is easy to understand and is likely to reflect the hardware architecture of current scalable machines. MPI's collective operations provide an opportunity for implementations to supply sophisticated scalable algorithms for a wide variety of global communication and computation patterns.

MPI may appear complicated, but its core is essentially simple. Its roughly 275 functions (including all of MPI-2) implement variations on a relatively small number of concepts, and serious applications can be written with as few as six functions. The message passing model also has a semantic simplicity because all intereactions between processes are visible with explicit communication.

MPI provides integrated parallel I/O, building on concepts that it shares with message passing for describing noncontiguous data both in memory and in files, nonblocking operations, and collective calls to enable scalable performance.

All of these properties of MPI contribute to its current popularity as a way to develop large-scale applications for scalable supercomputers. This is not to say that a new model and language is impossible or even undesirable, but to point out features that a new language must have in order to lure application developers away from MPI.

4. The HPCS Languages as a Group

The detailed, separate specifications for the HPCS languages can be found at [8].

In this section we consider the languages together and compare them along several axes in order to present a coherent view of them as a group.

4.1. Base Language

The HPCS languages use different sequential bases. X10 uses an existing object-oriented language, Java, inheriting both good and bad features. It adds to Java support for multimensional arrays, value types, and parallelism, and it gains tool support from IBM's extensive Java environment. Chapel and Fortress use their own, new object-oriented languages. An advantage of this approach is that the language can be tailored to science (Fortress even explores new, mathematical character sets), but the fact that a large intellectual effort is required in order to get the base language right has slowed development and may deter users.

4.2. Creating Parallelism

Any parallel programming model must specify how the parallelism is initiated. All three HPCS languages have parallel semantics; that is, there is no reliance on automatic parallelism, nor are the languages purely data parallel with serial semantics, like the core of HPF. All of them have dynamic parallelism for loops as well as tasks, and encouragement for the programmer to express as much parallelism as possible, with the idea that the compiler and runtime system will control how much is actually executed in parallel. There are a variety of mechanisms for expressing different forms of task parallelism, including explicit spawn, asynchronous method invocation, and futures. Fortress is unusual in making parallelism the default semantics for both loops and for argument evaluation; this encourages programmers to "think in parallel" which may result in more highly parallel code, but could also prove to be surprising to programmers.

The use of dynamic parallelism is the most significant semantic difference between the HPCS language and the existing PGAS languages with their static parallelism model. It presents the biggest opportunity to improve performance and ease of use relative to these PGAS languages and MPI. Having dynamic thread support along with data parallel operators may encourage a higher degree of parallelism in the applications, and allows the simplicity of expressing this parallelism directly rather than mapping it to a fixed process model in the application. The fine-grained parallelism can be used to mask communition latency, reduce stalls at synchronization points, and take advantage of hardware extensions such as SIMD units and hyperthreading within a processor.

The dynamic parallelism is also the largest implementation challenge for the languages, since it requires significant runtime support to manage the parallelism. The experience with Charm++ shows the feasibility of such runtime support for a class of very dynamic applications with limited dependencies [10]. A recent UPC project to apply multithreading to a matrix factorization problem reveals some of the challenges that arise with more complex dependences between tasks. In that UPC code,

the application level scheduler manages user level threads on top of UPC's static process model: it must select tasks on the critical path to avoid idle time, delay allocating memory for noncritical tasks to avoid running out of memory, and ensure that tasks run long enough to gain locality benefits in the memory hierarchy. The scheduler uses application-level knowledge to meet all of these constraints, and performance depends critically on the quality of that information; it is not clear how such information would be communicated to one of the HPCS language runtimes.

4.3. Communication and/or Data Sharing

All three of the HPCS languages use a global address space for sharing state rather than an explicit message passing model. They all support shared multidimensional arrays as well as pointer-based data structures. X10 originally allowed only remote method invocations rather than direct reads and writes to remote values, but this restriction has been relaxed with the introduction of "implicit syntax," which is syntactic sugar for a remote read or write method invocation.

Global operations such as reductions and broadcasts are common in scientific codes and while their functionality can be expressed easily in shared memory using a loop, they are often provided as libraries or intrinsics in parallel programming models. This allows for tree-based implementations and the use of specialized hardware that exists on some machines. In MPI and some the existing PGAS languages, these operations are performed as "collectives": all processes invoke the global operation together so that each process can perform the local work associated with the operation. In data parallel languages global operations may be converted to collective operations by the compiler. The HPCS languages provide global reductions without explicitly involving any of the other threads as a collective: a single thread can execute a reduction on a shared array. This type of one-sided global operation fits nicely in the PGAS semantics, as it avoids some of the issues related to processes modifying the data involved in a collective while others are performing the collective [6]. However, the performance implications are not clear. To provide tree-based implementation and allow work to be performed locally, a likely implementation will be to spawn a remote thread to reduce the values associated with each process. Since that thread may not run immediately, there could be a substantial delay in waiting for such global operations to complete.

4.4. Locality

The languages use a variation of the PGAS model to support locality optimizations in shared data structures. X10's "places" and Chapel's "locales" provide a logical notion of memory partitions. A typical scenario maps each memory partition at program startup to a given physical compute node with one or more processors and its own shared memory. Other mappings are possible, such as one partition per processor or per core. Extensions to allow for dynamic creation of logical memory partitions has been discussed, although the idea is not fully developed. Fortress

has a similar notion of a "region", but regions are explicitly tied to the machine structure rather than virtualized, and regions are hierarchical to reflect the design of many current machines.

All three languages support distributed data structures, and in particular distributed arrays that include user-defined distributions. These are much more general than in the existing PGAS languages. In Fortress the distribution support is based on the machine-dependent region hierarchy and is delegated to libraries rather than being in the language itself.

4.5. Synchronization among Threads and Processes

The most common synchronization primitives used in parallel applications today are locks and barriers. Barriers are incompatible with the dynamic parallelism model in the HPCS languages, although their effect can be obtained by waiting for a set of threads to complete. X10 has a sophistcated synchronization mechanism called "clocks," which can be thought of as barriers with attached tasks. Clocks provide a very general form of global synchronization that can be applied to subsets of threads.

In place of locks, which are viewed by many as cumbersome and error prone, all three languages support atomic blocks. Atomic blocks are semantically more elegant than locks, because the syntactic structure forces a matching begin and end to each critical region, and the block of code is guaranteed to be atomic with respect to all other operations in the program (avoiding problems of acquiring the wrong lock or with deadlock). Atomic blocks place a larger burden on runtime support: one simple legal implementation involves a single lock to protect all atomic blocks[b] but the performance of such an implementation is probably unacceptable. More aggressive implementations will use speculative execution and rollback, possibly relying on hardware support within shared memory systems. The challenge comes from the use of a single global notion of atomicity, whereas locks may provide atomicity on two seperate data structure using two seperate locks. The information that the two data structures are unaliased must be discovered dynamically in a setting that relies on atomics. The support for atomics is not the same across the three HPCS languages. Fortress has abortable atomic sections, and X10 limits atomic blocks to a single place, which allows for a lock per place implementation.

The languages also have some form or a "future" construct that can be used for producer-consumer parallelism. In Fortress if one thread tries to access the result of another spawned thread, it will automatically stall until the value is available. In X10 there is a distinct type for the variable on which one waits and the contents, so the point of potential stalls is more explicit. Chapel has the capability to declare variables as "single" (single writer), and "sync" (multiple readers and writers).

[b]This assumes atomic blocks are atomic only with respect to each other, not with respect to individual reads and writes performed outside and atomic block.

5. Moving Forward

In July 2006 a workshop was held at Oak Ridge National Laboratory bringing together the three HPCS language vendors, computer science researchers representing the PGAS languages and MPI, potential users from the application community, and program managers from DARPA and the Department of Energy's Office of Advanced Scientific Computing. In this section we describe some of the findings of the workshop at a high level and present the tentative plan for further progress that evolved there. A workshop report will soon be available.

The workshop was organized in order to explore the topic of converging the HPCS languages to a single language. Briefings were presented on the status of each of the three languages and an effort was made to identify the common issues involved in completing the specifications and initiating the implementations. Potential users offered requirements for adoption, and computer science researchers described recent work in compilation and runtime issues for PGAS languages. One high-level finding of the workshop was the considerable diversity in the overall approaches being taken by the vendors, in the computer science research being undertaken relevant to the HPCS language development, and in the application requirements.

5.1. Diversity in Vendor Approaches

Although the three vendors are all well along the path to complete designs and prototype implementations of languages intended to increase the productivity of software developers, they are not designing three versions of the same type of object. X10, for example, is clearly intended to fit into IBM's extensive Java programming environment. As described in Section 1, it uses Java as a base language, allowing reuse of multiple existing tools for parsing, compiling, and debugging to be extended to encompass the new parallel language. Cray's approach is more revolutionary, with the attendant risks and potential benefits; Chapel is an attempt to design a parallel programming language from the ground up. Sun is taking a third approach, providing a framework for experimentation with parallel language design, in which many aspects of the language are to be defined by the user and many of the features are expected to be provided by libraries instead of by the language itself. One novel feature is the option of writing code with symbols that, when displayed, can be typeset as classical mathematical symbols, improving the readability of the "code."

5.2. Diversity in Application Requirements

Different application communities have different expectations and requirements for a new language. Although only a small fraction of potential HPC applications were represented at the workshop, there was sufficient diversity to represent a wide range of positions with respect to the new languages.

One extreme is represented by those applications for which the current programming model—MPI together with a conventional sequential language—is working well. In many cases MPI is being used as the MPI Forum intended: the MPI calls

are in libraries written by specialists, and the application programmer sees these library interfaces rather than MPI itself, thus bypassing MPI's ease-of-use issues. In many of the applications content with the status quo, the fact that the application may have a long life (measured in decades) amortizes the code development effort and makes development convenience less of an issue.

The opposite extreme is represented by those for whom rapidity of application development is *the* critical issue. Some of these codes are written in a day or two for a special purpose and then discarded. For such applications the MPI model is too cumbersome and error prone, and the lack of a better model is a genuine barrier to development. Such applications cannot be deployed at all without a significant advance in the productivity of programmers.

Between these extremes is, of course, a continuously variable range of applications. Such applications would welcome progress in the ease of application development and would adopt a new language in order to obtain it, but any new approach must not come at the expense of qualities they find essential in the status quo: portability, completeness, support for modularity, and at least some degree of performance transparency.

5.3. Diversity in Relevant Computer Science Research

The computer science research most relevant to the HPCS language development is that being carried out in the various PGAS language efforts. These languages also offer a global view of data, with explicit control of locality in order to provide performance. Their successful implementation has involved research into compilation techniques that are likely to be useful as the HPCS languages finalize language design and begin developing production compilers, or at least serious prototypes. The PGAS languages, and associated research, also share with the HPCS languages the need for runtime systems that support lightweight communication of small amounts of data. Such portable runtime libraries are being developed in both the PGAS and MPI implementation research projects [4,3].

The PGAS languages are being used in applications to a limited extent, while the HPCS languages are still being tested for expressibility on a number of example kernels and benchmarks.

The issue of the runtime system is of particular interest, because standardizing it would bring multiple benefits. A standard syntax and semantics for a low-level communication library that could be efficiently implemented on a variety of current communication hardware and firmware would benefit the entire programming model implementation community: HPCS languages, PGAS languages, MPI implementations, and others. A number of such portable communications exist now (GASNet [2], ADI-3, ARMCI), although most have been developed with a particular language or library implementation in mind.

5.4. Immediate Needs

Despite the diversity in approaches to the languages being taken by the vendors, some common deficiencies were identified. These are areas that have been postponed while the initial language designs have been formulated, but now really need to be addressed if the HPCS languages are to catch the attention of the application community.

5.4.1. Performance

The Fortress [7] and X10 [16] implementations are publicly available and an implementation of Chapel exists, but is not yet released. So far these prototype implementations have focused on expressivity rather than performance. This direction of effort has been appropriate up to this point, but now that one can see how various benchmarks and kernels can be expressed in these languages, one wants to see how they can be compiled for performance competitive with existing approaches, especially for scalable machines. While the languages may not reach their full potential without the HPCS hardware being developed by the same vendors, the community needs some assurance that the elegant language constructs, such as those used to express data distributions, can indeed be compiled into efficient programs for current scalable architectures.

5.4.2. Completeness

The second deficiency involves completeness of the models being presented. At this point none of the three languages has a parallel I/O model embedded in the languages. At the very least the languages should define how to read and write distributed data structures from and into single files, and the syntax for doing so should enable an efficient implementation that can take advantage of parallel file systems.

Another feature that is important in multiphysics applications is modularity in the parallelism model, which allows subsets of processors to act independently. MPI has "communicators" to provide isolation among separate libraries or separate physics models. The PGAS languages are in the process of introducing "teams" of processes to accomplish the same goals. Because the HPCS languages have a dynamic parallelism model combined with data parallel operators, this form of parallelism should be expressible, but more work is needed to understand the interactions between the abstraction mechanisms used to create library interfaces and the parallelism features.

5.5. A Plan for Convergence

On the last day of the workshop a plan for the near future emerged. It was considered too early to force a convergence on one language in the near term, given that the current level of diversity seemed to be beneficial to the long term goals of the HPCS project rather than harmful. The level of community and research involvement could be increased by holding a number of workshops over the next few

years in specific areas, such as memory models, data distributions, task parallelism, parallel I/O, types, tools, and interactions with other libraries. Preliminary plans were made to initiate a series of meetings, loosely modelled on the MPI process, to explore the creation of a common runtime library specification.

An approximate schedule was proposed at the workshop. In the next eighteen months (i.e., by the end of the calendar year 2007) the vendors should be able to freeze the syntax of their respective languages. In the same time frame, workshops should be held to address the research issues described above. Vendors should be encouraged to establish "performance credibility" by demonstrating competitive performance of some benchmark on some high-performance architecture. This would not necessarily involve the entire language, nor would it necessarily demand better performance of current versions of the benchmark. The intent would be to put to rest the notion that a high-productivity language precludes high performance. Also during this time, a series of meetings should be held to determine whether a common communication subsystem specification can be agreed upon.

The following three years should see the vendors (at least those funded in Phase III) improve performance of all parts of their languages. Inevitably, during this period the languages will continue to evolve independently as experience is gained. At the same time, aggressive applications should get some experience with the languages. After this period, when the languages have had an opportunity to evolve in both design and implementations while applications have had the chance to identify strengths and weaknesses of each, would come the consolidation period. At this point (2010-2011) an MPI forum like activity could be organized to take advantage of the experience now gained, in order to cooperatively design a single HPCS language with the DARPA HPCS languages as input (much as the MPI Forum built on, but did not adopt any of, the message-passing library interfaces of its time).

By 2013, then, we could have a new language, well vetted by the application community, well implemented by HPCS vendors and even open-source developers, that could truly accelerate productivity in the development of scientific applications.

6. Conclusion

The DARPA HPCS language project is active, healthy, and on schedule. Excellent work is being carried out by each of the vendor language teams, and it is to be hoped that Sun's language effort will not suffer from the end of Sun's hardware development contract with DARPA. Now is the time to get the larger community involved in the high-productivity programming model and language development effort, through workshops targeted at outstanding relevant research issues and through experimentation with early implementations of all the "productivity" languages. In the long run, acceptance of any new language for HPC is a speculative proposition, but there is much energy and enthusiasm for the project, and a reasonable plan is in place by which progress can be made.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contracts DE-AC02-06CH11357 and DE-AC03-76SF00098. It was also supported by the Defense Advanced Research Program Agency.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The Fortress language specification. Available from `http://research.sun.com/projects/plrg/`.

[2] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

[3] Darius Buntinas and William Gropp. Designing a common communication subsystem. In Beniamino Di Martino, Dieter Kranzluüller, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume LNCS 3666 of *Lecture Notes in Computer Science*, pages 156–166. Springer, September 2005. 12th European PVM/MPI User's Group Meeting, Sorrento, Italy.

[4] Darius Buntinas and William Gropp. Understanding the requirements imposed by programming model middleware on a common communication subsystem. Technical Report ANL/MCS-TM-284, Argonne National Laboratory, 2005.

[5] Chapel: The Cascade high productivity language. `http://chapel.cs.washington.edu/`.

[6] UPC Consortium. UPC language specifications v1.2. Technical report, Lawrence Berkeley National Lab, 2005.

[7] Project Fortress code.

[8] HPCS Language Project Web Site. `http://hpls.lbl.gov/`.

[9] HPLS. `http://hpls.lbl.gov`.

[10] L.V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, Sep-Oct 1993. ACM Sigplan Notes, Vol. 28, No. 10, pp. 91-108.

[11] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[12] Message Passing Interface Forum. MPI2: A Message Passing Interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–299, 1998.

[13] mpiJava home page. `http://www.hpjava.org/mpiJava.html`.

[14] R. Numrich and J. Reid. Co-Array Fortran for parallel programming. In *ACM Fortran Forum 17, 2, 1-31.*, 1998.

[15] The X10 programming language. `http://www.research.ibm.com/x10`.

[16] The X10 compiler. `http://x10.sf.net`.

[17] Katherine Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10:825–836, 1998.