# Multiresolution Languages for Portable yet Efficient Parallel Programming

*Bradford L. Chamberlain, Cray Inc.*
*411 First Ave S, Suite 600; Seattle, WA 98104*

*Abstract:* This position paper was hastily written at the end of October 2007 in response to DARPA RFI 08-03 on Efficient Compilation and Code Development (ECCD), a precursor to the Architecture-Aware Compiler Environment program (AACE). It describes a language-based approach to the problem of developing parallel programs that are easy to port between platforms, yet without sacrificing efficiency.

## I. Executive Summary

In this paper, we describe the idea of *multiresolution languages*, a concept that we have been exploring which relates to the programmer's ability to move from an extremely high-level specification of an algorithm to increasingly lower-level representations as their requirements dictate. The goal of such languages is to allow the programmer to ignore low-level details during initial development or for portions of a program that are not performance critical, while drilling down closer to the hardware for those portions of a program that require lower-level specification and optimization. An important aspect of a multiresolution language is that it support a good *separation of concerns* such that a programmer can incrementally evolve an algorithm's implementation over time without rewriting every function and loop from scratch. Good separation of concerns also allows a code to be written more portably since low-level implementation details can be changed on a platform-by-platform basis, either implicitly or explicitly.

We illustrate the concept of multiresolution languages in the context of our work on the Chapel parallel programming language where we have a desire to simplify parallel programming for many common cases without hindering the expert programmer's ability to tune for every last bit of performance. As a specific example, we describe Chapel's multiresolution support for arrays which allow the programmer to defer implementation decisions to the compiler, select from standard libraries of implementations, or author their own implementations depending on the level of control they want and require. This framework also allows different array implementations to be swapped in and out without modifying the program's high-level algorithmic description, supporting an appropriate separation of concerns.

Other aspects of this paper describe our proposed compilation structure, our belief that existing languages are insufficient for solving the large-scale parallel programming problem, and aspects of our previous experiences with machine representation through the CTA abstract machine model and programming languages such as ZPL.

This paper is organized in a question-and-answer format which begins on the page following. It starts by answering a number of rhetorical questions that describe the multiresolution language concept and how we believe it can help with the general problem of portable parallel programming for diverse architectures without sacrificing performance. Once the general framework has been described, it then moves on to answer the specific questions posed by the RFI (numbered and in bold text for emphasis), interspersed with a few supporting or clarifying questions along the way.

# II. Technical Response

**Q:** What do you mean by the term *multiresolution*?

**A:** In computer graphics and scientific computation, the term "multiresolution" is often used to describe a representation that is composed of multiple granularities, each appropriate for a given degree of required resolution. For example, a model of a 3D object that is to be graphically rendered may be represented using a series of increasingly coarse models. The coarsest models are the quickest to draw, and yet can be completely appropriate when the object is far enough in the distance that it does not consume much space on the screen. Finer representations take more time to draw and therefore become more appropriate as the object gets closer to the viewer and requires more screen resolution. In this paper, we use the term multiresolution to capture this notion of spending more effort where that effort is required and less where it is not.

**Q:** What then do you mean by *multiresolution language*?

**A:** This is a term that we have started to use informally to refer to programming languages that allow users to write code at multiple levels of detail. For example, in parts of a program that are not performance critical, the programmer should be able to express an algorithm using higher-level concepts and not be burdened with lower-level details. Conversely, in performance-critical sections of the code, the programmer should have the ability to code closer to the machine in order to obtain the performance that they require.

**Q:** What are the benefits of multiresolution languages?

**A:** First, multiresolution languages can be thought of as supporting the 90/10 rule, which suggests that 90% of a program's execution time is spent within 10% of its code. Programmers should be able to write the 90% portion of the program using rich abstractions, unburdened by low-level implementation details. Yet for the 10% of the code where performance is crucial, programmers should be able to roll up their sleeves and get as close to the machine as they need to.

In addition, a well-designed multiresolution language should support a *separation of concerns* in which lower-level implementation details can be added to the code without completely rewriting the existing algorithm. In this way, aspects of the algorithm that are independent of the lower-level data structures and loops (*e.g.*, "compute a matrix-vector multiplication") can remain unchanged while implementation details are being modified (*e.g.*, "convert the dense matrix format into a sparse block-diagonal format in which the blocks are stored in row-major order and traversed in a cache-optimal way"). Such separation of concerns not only makes the code easier to modify incrementally, it also preserves the readability, maintainability, and portability of the computation being performed.

**Q:** In what way would existing languages *not* be considered multiresolution?

**A:** Most languages tend to be considered either higher-level languages—such as Smalltalk, Lisp, or APL—or lower-level languages—such as C or Fortran. Higher-level languages have the advantages of providing more expressiveness and data abstraction, yet often tie the hands of programmers, making it difficult or impossible to code closer to the machine when desired. For example, in such languages, it is difficult to write a routine and have a good mental model for how the code will be mapped to the underlying processor (as in languages like C) in order to tune and make performance-critical decisions. Meanwhile, lower-level languages like C and Fortran

allow the programmer to code for performance, but provide little in the way of abstractions to avoid managing those details when that level of control is unnecessary or undesirable.

**Q:** What about object-oriented languages such as C++, C#, or Java?

**A:** Object-oriented languages tend to be closer to our vision of multiresolution languages in that they have good support for creating higher-level abstractions through classes. For languages like C++ whose base concepts have a reasonably clear implementation model on the underlying hardware, a reasonable mix of higher- and lower-level programming can be achieved. However, we believe that while such languages support the extreme points of programming at coarse/fine resolutions, they fail to support intermediate points that would provide useful tradeoffs between performance and expressiveness. For example, C++ users who are not satisfied with the static size restrictions of its multidimensional arrays might choose to implement richer arrays by creating their own array classes; yet in doing so, they have moved into the OOP segment of the language and thrown up barriers to the compiler's ability to understand the semantics of the operations on the arrays and optimize for those semantics.

**Q:** This tension between abstraction and compiler optimization seems tough. What can we do?

**A:** We agree that this is a difficult problem, and believe it worthy of further exploration. In our own work, one of the ways that we are striving to support this sort of multiresolution programming is to identify programming concepts that we believe are fundamental to the target community (HPC in our case) and then provide ways for the user to specify the implementation of those concepts to varying degrees.

**Q:** Up until now you've been fairly abstract; can you give a more concrete example?

**A:** Sure. In the Chapel language that we are developing as part of DARPA's HPCS program [1], we believe that arrays are crucial for the scientific communities that we are targeting. As such, we have included rich support for array programming in the language. "Rich support for arrays" implies multidimensional arrays and a rich set of operators on them, as in Fortran-90. But it also includes sparse arrays, associative arrays (indexed using values like strings or records), and even unstructured arrays that can be used to represent graph-based data structures. In addition to providing a wide palette of arrays, we support them in a multiresolution style so that users can specify the implementations of arrays in varying amounts of detail depending on the degree to which they want or need to do so. This allows them to choose from a spectrum of options ranging from abdicating decisions to the compiler and runtime to controlling all of the details explicitly.

**Q:** More precisely, what range of options would a user have for controlling array implementations in Chapel?

**A:** Consider an example in which a group of programmers want to compute using a large, distributed sparse matrix. In their initial version of the code they might use a dense array with a compiler-chosen implementation to represent the sparse matrix, in order to simply get the basic algorithm up and running. Once it is, they can then choose to switch the data structure from a dense array to a sparse array, but again defer the decision of representation to the compiler. They might then test the code for increasingly large problem sizes. Default array allocations in Chapel are typically limited to one node's local memory, so at some point they may exhaust the memory on the node upon which they are executing. At this point, they could change the declaration of the array to indicate that it should be distributed, yet leave the choice of the distributed

implementation up to the compiler once more. Now their program will execute with all of the machine's memory available to it.

At this point, the programmers may start to want to tune for performance, and the default array implementations made by the compiler may not be optimal for their problem. As a first step towards improving the performance, the programmers may select between array implementations defined in a standard library that comes with Chapel, or by selecting a distribution that some other group has written and made available on the web. But perhaps there are characteristics of their algorithm that no existing array implementation can take advantage of. In this case, the programmers might be motivated to modify and customize an existing distribution to best suit their problem, or they might write a distribution from scratch.

This sequence of actions illustrates Chapel's support for multiresolution array programming, since the programmers could have stopped at any of these intermediate points—or skipped right past them—depending on the degree of control that they needed or wanted for their algorithm. For parts of their code requiring more care, they would dive deeper into the array distribution; for parts that are less performance-critical, they would rely more on the compiler and runtime or on the standard array implementations. Moreover, Chapel's separation of concerns implies that the top-level algorithm would remain relatively unchanged during this evolutionary process. Changing an array's implementation requires changing its declaration, and the array implementation code itself, but not the high-level operations on that array—the accesses, slices, and iterations that comprise the algorithm.

**Q:** Again, in what way does object-oriented programming not support a similar story?

**A:** The main difference between these Chapel arrays and a pure object-oriented approach is that the Chapel compiler and runtime understand what an array is and what the interface provided by an array implementation is like. Having this semantic and structural knowledge allows the compiler to optimize computations on the array by understanding the high-level operations specified by the algorithm and mapping them to the lower-level concepts described by the array implementation. To be clear, objects with state and methods are still used to implement Chapel arrays, so we retain many of the benefits of OOP. Yet, because the concepts being implemented by the class interfaces are understood by the compiler, it does not need to treat those classes as black boxes, nor require the user to express their algorithm in terms of lower-level method calls.

**Q:** To round out this example, what do Chapel programmers have to do to create their own array implementation?

**A:** They essentially need to implement a set of classes that support a standard interface that the compiler will use when rewriting operations on that array. We tend to think of this array interface as having three sub-interfaces. The first is a required interface that the array implementation must provide in order to work at all. For example a method for accessing the array element corresponding to a specific index would be part of the required interface. The second is an optional interface that need not be implemented, yet by providing some or all of it, the implementer may enable additional compiler optimizations. For example, if the array format supports some sort of ghost cell or halo format, supplying the part of the interface that allocates and refreshes those cells would support the compiler's ability to generate more optimal stencil computations. The third part of the interface is a portion that the compiler is completely ignorant of. It is designed for the user to call into explicitly, as a means of performing operations on the array implementation that are not supported by the compiler rewrites. Using this third interface gives the user full control over their array implementation, yet also makes code utilizing that portion of the interface less flexible since other array implementations are not guaranteed to

support those same routines. It is worth noting that standard array implementations supported by the Chapel compiler will be written using this same mechanism to ensure that there isn't a performance cliff when a user switches between a standard implementation and a custom one.

**Q:** This is one example of multiresolution programming in Chapel. Are there others?

**A:** A similar theme shows up in other areas of Chapel where a programmer may want more or less control over some mechanism. Here are some examples:

First, programmers may be more or less explicit about the types of arguments accepted by a function call, from saying nothing about an argument's type, to putting constraints on it, to fully specifying it and possibly optimizing specific overloads. For the more generic versions, the compiler will create specialized versions of the function as are required by its calls.

As a second example, we expect to support a variety of policies for managing, scheduling, and load balancing a user's tasks. Users who want the least control over task scheduling may request a very dynamic task management scheme in which idle cores may "steal" queued up tasks from their neighbors or from remote processors. Users who want a great deal of control over tasks may work in a more explicit model in which each task results in a new thread being created and work is never queued up or migrated (unless they manage it manually). Intermediate policies would support mid-resolution options that would trade off programmer control for effort.

We are also exploring other ways of supporting multiresolution programming, such as providing a number of memory management schemes and allowing programmers to use the one that is appropriate for their performance and level-of-control requirements. For parts of the code least concerned about performance, memory management could be performed using garbage collection, while for the most performance-critical parts, the user might use explicit allocations and frees. In the middle ground, the programmer might choose region-based memory management, which provides many of the ease-of-use benefits of garbage collection, while also controlling the overheads associated with collecting and freeing memory, thereby making performance impacts easier to corral.

**Q:** How do multiresolution languages help with this RFI's goal of obtaining portability and performance?

**A:** The main aspect relates to the concept of a multiresolution language supporting an appropriate separation of concerns. Taking our Chapel array example, the fact that a given array implementation can be changed without modifying the high-level algorithm not only means that array implementations can easily be evolved over time, but also that different implementations can be plugged in for different hardware platforms. This could either be done implicitly, by providing different implementations of standard array implementations for different platforms, much as current library routines tend to be optimized for each platform. Or it can be done explicitly, by having the user swap between different array implementations when they move between platforms. In either case, the important thing is that any change would affect only the declaration of the arrays, not the expression of the computation.

**Q:** Would such a scheme result in a multiplicity of platform-specific array implementations? (and a corresponding software engineering nightmare?)

**A:** If done naively, it could. However, we believe that by characterizing similar classes of architectures with appropriate parameters, one should be able to share a single array implementation across a large number of distinct hardware systems. For example, in Chapel, we can refer to parameters like the number of cores in a processor in order to generate an appropriate

amount of parallelism for a given loop. Using such parameters allows us to create code that is appropriate for single- to multi- to many-core processors. Yet for architectures that are significantly different, such as a massively multithreaded shared-memory platform, we should expect that a version more specialized to that architecture might be better than using a generic implementation for a cluster of multicore processors.

**Q1: What characteristics of a computational system are required to sufficiently represent a computer's capabilities and could be used by a self-assembling or re-targetable compiler environment?**

**A:** Useful characteristics for writing multiresolution code that performs well across platforms include the number of cores (or degree of parallelism) per node; the amount of memory per node; the sizes of the caches and cache lines; and the hierarchical structure of the machine's resources, particularly the processors and memory. Note that in order to take advantage of these facts in the source code, it is not as important to know the precise values as it is to be able to refer to them symbolically. For example, to write a loop that spawns a thread per core and blocks work among them, users don't need to know the number of cores statically, they merely need a symbolic way of representing that fact. To the extent that these values are then known at compile-time, the values can be expanded and the code optimized further by the compiler.

The biggest challenge in this area is anticipating the complete class of machines that one plans to target and the characteristics that are most important to them. For example, having gathered all of the parameters that are important for targeting multicore clusters, a model could fall apart when the user suddenly wants to target a long-vector processor or one with hardware support for multithreading. If these characteristics were not anticipated by the model, the programmer and compiler cannot make good use of them. As an example of such a machine model, consider the CTA (Candidate Type Architecture) model developed by Dr. Larry Snyder at the University of Washington [2]. In it, he characterized the aspects of parallel machines that he believed were important to performance while ignoring those that were not. The ZPL language was designed using the CTA, allowing the programmer and compiler to make intelligent decisions for a wide class of machines resulting in portable performance throughout the 1990's [3,4].

**Q:** You seem to be focusing primarily on higher-level architectural characteristics and ignoring lower-level ones like instruction sets, memory latencies, and so forth. Aren't such factors also important?

**A:** They are. We are purposefully ignoring such details because for multiresolution languages we advocate a hierarchical approach to compilation in which the language is implemented using a lower-level language like C. We then expect that an optimizing compiler for that lower-level language will be used to generate the machine code for a particular architecture, and expect its compiler to be concerned with the lower-level machine characteristics. Such an approach results in an appropriate division of labor between the lowering of higher-level language concepts and their lower-level implementation and optimization for the physical hardware. We expect that the lower-level compiler will need to know all sorts of additional facts such as instruction set details, memory latencies, and so on, and we refer the reader to the companion paper from Cray Inc., *Transparent Compilation and Automatic Parallelization in Hybrid Architecture Environments* for more details at this level. Ideally, we would want to build our multiresolution language compiler on top of a lower-level compiler such as the one described in this paper.

**Q:** Using C as an intermediate format? But isn't C known to be fraught with performance problems due to its pointer math and aliasing issues?

**A:** When used naively, C can create problems for the compiler. However, when used in an informed manner, a compiler back-end can use C very effectively as a portable assembly language. This has become particularly true given C's recent support for *restrict pointers* which can help the compiler avoid the alias analysis problem for most common cases. In earlier work on ZPL, members of our team targeted C in the ZPL compiler's back-end using these techniques and generated code that competed with or outperformed hand-coded Fortran. At times, we have proposed the creation of a modern C-level "portable assembly language" that would sidestep a number of the problems inherent in C to be used for the creation of new higher-level languages, but given the inertia that C has, perhaps a better use of that effort would simply be to continue to identify C subsets, concepts, and idioms that help avoid the most egregious performance problems.

**Q2: What are the causes of the performance gaps between what current compilers can provide and what computational systems should be able to achieve?**

**A:** At the level that we are concerned about—that of parallel programming on large-scale machines—the primary performance gaps are due to inter-processor communication and data transfers. Analogously to our previous characterization of languages, programmers are either forced to manage communication details using low-level notations such as message passing, or they are restricted to high-level concepts that thwart user-level optimization and reasoning about performance.

The low-level message passing formats are problematic in a few ways. First of all, they place the full burden of optimizing communication on the programmer, throwing away useful compiler knowledge such as the def-use chains of the variables to be communicated. This requires users to manually overlap communication and computation, to combine communications to reduce overhead, and to eliminate redundant communications manually. Secondly, they tend to encode too many properties of the data transfer, forcing the programmer to make decisions that may not be appropriate for all architectures. For example, by encoding the synchronization and buffering semantics of a given message send into the interface, users who want to write portable code are required to choose a specific mechanism rather than relying on the compiler to make the best choice for a given platform.

At the other end of the spectrum are high-level languages such as High Performance Fortran (HPF) that put the burden of implementing communication on the compiler. While this sounds attractive, in many cases the compiler must make conservative decisions and insert suboptimal or redundant communication, hurting performance. Worse, the user has no lower-level of the language to drop down to and control such details themselves.

The ZPL language, mentioned previously, represents an interesting midpoint which retains many of the high-level advantages of languages like HPF (particularly, the compiler management of communication), yet using a performance model that allows the user and compiler to reason about a code's communication needs much as a C programmer can reason about the implementation of their code on a standard processor. As a result, the ZPL team was able to generate code that competed with or outperformed hand-coded Fortran with MPI on a variety of architectures. The main limiter to ZPL's approach was that it constrained the user to a single level of parallelism and the execution model to be SPMD (single program, multiple data). These restrictions allow for a large number of interesting data parallel codes to be written, but thwart the expression of more general parallel programs and algorithms with nested parallelism.

In our work on Chapel, we are striving to remove these restrictions by supporting task parallelism and nested parallelism without sacrificing the performance obtained by ZPL for data parallel programs. We share some important commonalities with ZPL such as language concepts for

describing distributed index sets and arrays, yet were forced to abandon its syntax-based performance model, making communication as difficult to identify as it is in HPF. A research question for our group is whether the multiresolution theme could allow a programmer to code in a communication-oblivious way for large portions of the program, but then choose to move into a more restricted mode for performance-critical kernels, where ZPL-style operators and syntax-based performance could be supported. This idea remains an open question, but fits our multiresolution theme by allowing the user to choose the level of expressiveness or control that is needed for a given code fragment.

**Q3: What compiler approaches are possible to enable a compiler system or environment that would be self-assembling and optimize capabilities to efficiently address a broad range of computational systems?**

**A:** Our main thought here is the notion of hierarchical compilation as mentioned previously in order to support an appropriate division of labor. Since multiresolution languages are intended to raise the level of computational expression, they must necessarily be concerned with higher-level features than a typical C or Fortran compiler would be. Yet to have them map those higher-level features all the way down to a multitude of hardware platforms without sacrificing performance seems unlikely. Instead, we propose a division of labor in which the multiresolution language compiler lowers to a portable assembly code such as C, and then a lower-level C compiler worries about the optimal mapping of that code to the target hardware architecture. It should be pointed out that even C compilation is hierarchical in nature, lowering the source code to assembly code that is then compiled by an assembler. In a sense, we are simply proposing an additional layer to this mode of compilation.

**Q4: What are the characteristics of a compiler environment that will significantly improve development productivity and minimize execution time for large-scale parallel computers?**

**A:** As should be clear by now, we believe that the most important concept for improving productivity while minimizing execution time for parallel computers is the use of multiresolution languages that allow the user to program at a variety of levels of abstraction and control. Designing such languages with an appropriate separation of concerns allows code to be evolved, ported, and tuned without requiring the computation itself to be changed again and again.

There are several fundamental language concepts that we believe aid in the implementation and use of multiresolution languages due to their support for abstraction and code reuse. Chief among these are:

- **generators**: the concept of a function that yields a stream of values rather than returning a single value. These are useful for abstracting loop structure away from code in a modular, reusable way, just as functions serve as an abstraction for straight-line code. Generators are amenable to compiler optimization and support the user's ability to create implementations of data structures like arrays in a clean and easily-reasoned about way.
- **polymorphic/generic functions and classes**: this is the notion of creating functions or classes that can be used with a variety of different types. Given the choice between forcing the user to create versions of routines for each scalar type/bit-width representation or having the compiler create those versions, it is clearly preferable to have the compiler shoulder this burden. Compilers are good at redundant mechanical transformations, users are error-prone and have better things to do.
- **object-orientation:** though the common user's view of a multiresolution language need not be object-oriented, support for object-oriented programming is important within the

lower-levels of a multiresolution language in order to support specification of standard interfaces, encapsulation of data structures, reuse of common methods, and specialization of existing code.

**Q5: What is the structure of the ideal, self-assembling or re-targetable compiler environment that will efficiently recognize the computational capabilities and utilize the optimal compiler algorithms for a large-scale parallel computer?**

**A:** See the answers to questions 3 and 4 and the companion paper *Transparent Compilation and Automatic Parallelization in Hybrid Architecture Environments.*

**Q:** In developing Chapel, your multiresolution parallel language, you are obviously creating a new language rather than modifying an existing one. Why is this?

**A:** There are a few reasons:

1. Most mainstream languages were designed for sequential computing, or in more recent years for fork-join threaded computing within a shared-memory environment. General parallel computing for large-scale distributed memory machines imposes different concerns than these mainstream languages have, and therefore requires new concepts such as high-level identification of parallelism and locality. By creating a new language, we liberate ourselves from traditional concepts that are burdensome or irrelevant for parallel computing and allow ourselves to create new concepts that are appropriate for this problem domain.
2. Parallel algorithms are typically very different than sequential algorithms. For example, a tightly-coded sequential matrix multiplication algorithm looks vastly different than a tightly-coded parallel matrix multiplication algorithm for distributed memory machines. To this end, the notion that extending an existing language will make it easier to create parallel algorithms is an illusion. The more important concept is to create a language that can interoperate with existing languages so that pre-existing code can be reused without being rewritten. This in itself is an extension of the multiresolution concept—reuse code when it makes sense and allow the programmer to rewrite it for parallelism when it doesn't.
3. The distinction between creating a new language versus adding concepts to an existing language is a minor one. If we take "language" to mean "the means by which one conveys an algorithm to a computer" rather than strictly "programming language", then we must admit that whether we extend a language through a library (as with MPI) or pragmas (as with OpenMP) or new concepts (as with Co-Array Fortran), we are also creating new languages. Programmers still have to learn new things, adjust to a new execution model, and reconsider how they will write their sequential code. That said, to the extent that a language can re-use familiar syntax, operators, and concepts, the burden of learning the new language can be minimized. Java and Perl are examples of this principle since they are not extensions of the C language per se, but similar enough to it that an experienced C programmer can make reasonable inferences about their concepts and learn the language correctly.
4. Frankly, given the multitude of supporters for any given language, be it C, Fortran, or Java, one cannot pick a single language to extend and make all users happy. One might view our creation of a new language as a neutral act in that it equally burdens all users with the need to learn the new language. Over time, one could imagine taking successful concepts in Chapel and rolling them into a dialect of C or Fortran or Java. However, we

have chosen not to do so. Better to work on solving the unique problems of large-scale parallel computing with a blank slate and leave the question of hybrid languages to the enthusiasts of those languages and their standards committees.

To this end, we have designed Chapel to be a new language that is similar to familiar HPC languages in that it is block-structured, imperative, and utilizes familiar concepts and syntax. We have also designed it to be interoperable with existing languages in order to support reuse of existing code.

**Q6: Which (preferably existing) computer language will enable a compiler environment to generate executables for application codes with minimum execution time when executed on a large-scale parallel computer system?**

**A:** For the reasons given in the previous question, we do not believe that an existing language can do this satisfactorily. Chapel is the language that we believe has the best shot of becoming such a language due to its support for general parallel programming, a general execution model (in particular, not constrained to SPMD), a global namespace, user-specification of locality, and multiresolution language features that support both abstraction and low-level optimization.

**Q7: What development feedback or closed loop capability is needed for a computational system, compiler, and application environment?**

**A:** We certainly believe that when fed back to a compiler, runtime profiling and performance data can be very useful in helping a compiler optimize code. However, we have very limited experience in this area within our immediate group, simply a desire to enable such a loop. To help with this feedback process, we believe that it would be helpful for compilers and languages to support a profiling interface similar to PMPI or POMP so that third-parties can instrument code and collect performance data without being intimately familiar with the compiler's structure. To this end, we are exploring the concept of supporting such a profiling interface within Chapel's generated code and would hope to use statistics gathered by that layer to optimize subsequent compilations of a program.

**Q8: What is the collective expertise required for the optimal team capable of developing a high quality self-assembling or re-targetable compiler environment for large-scale parallel computers that can satisfy the development and performance requirements for DoD application code developers?**

**A:** At a minimum, such a team would need to consist of language developers, compiler experts, parallel applications and algorithms experts, computer architects, performance analysis experts, and a significant team of engineering staff. To make the end result truly high-quality, the team should also incorporate members of the user community, quality assurance personnel and testers, a documentation team, debugger support, and possibly IDE (integrated development environment) support. In addition, quality management would be required to help coordinate the efforts of such a large group, manage communications, and ensure that the project ran smoothly.

**Acknowledgements:** Thanks to the members of the Chapel team, past and present, who helped develop the themes and ideas described within this paper, particularly Steve Deitz and David Callahan. Thanks also to Larry Snyder and the members of the ZPL team who helped establish several of the concepts that influenced Chapel and our notion of multiresolution programming languages. And finally, thanks to the programming environments group at Cray Inc., and the company's technical leadership for many conversations that helped these ideas evolve over time.

# III. Bibliography

[1] **Parallel Programmability and the Chapel Language**. Bradford L. Chamberlain, David Callahan, and Hans P. Zima. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.

[2] **Type Architectures, Shared Memory, and the Corollary of Modest Potential**. Lawrence Snyder. *Annual Review of Computer Science*, 1:289–317 (1986).

[3] **The High-Level Parallel Language ZPL Improves Productivity and Performance**. Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. In *Proceedings of the 1st Workshop on Productivity and Performance in High-End Computing (PPHEC-04)*, pages 66–75, February 2004.

[4] **The Design and Development of ZPL**. Lawrence Snyder. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HoPL)*, pages 8–37, May 2007.