PARALLEL PROGRAMMABILITY AND THE CHAPEL LANGUAGE

B. L. Chamberlain[1]
D. Callahan[2]
H. P. Zima[3]

Abstract

In this paper we consider productivity challenges for parallel programmers and explore ways that parallel language design might help improve end-user productivity. We offer a candidate list of desirable qualities for a parallel programming language, and describe how these qualities are addressed in the design of the Chapel language. In doing so, we provide an overview of Chapel's features and how they help address parallel productivity. We also survey current techniques for parallel programming and describe ways in which we consider them to fall short of our idealized productive programming model.

Key words: parallel languages, productivity, parallel programming, programming models, Chapel

The International Journal of High Performance Computing Applications, Volume 21, No. 3, Fall 2007, pp. 291–312
DOI: 10.1177/1094342007078442
© 2007 SAGE Publications

1 Introduction

It is an increasingly common belief that the programmability of parallel machines is lacking, and that the high-end computing (HEC) community is suffering as a result of it. The population of users who can effectively program parallel machines comprises only a small fraction of those who can effectively program traditional sequential computers, and this gap seems only to be widening as time passes. The parallel computing community's inability to tap the skills of mainstream programmers prevents parallel programming from becoming more than an arcane skill, best avoided if possible. This has an unfortunate feedback effect, since our small community tends not to have the resources to nurture new languages and tools that might attract a larger community—a community that could then improve those languages and tools in a manner that is taken for granted by the open-source C and Java communities.

This gap between sequential and parallel programming is highlighted by frequent comments in the high-end user community along the lines of "Why isn't programming this machine more like Java/Matlab/my favorite sequential language?" Such comments cut to the heart of the parallel programmability problem. Current parallel programming languages are significantly different from those that a modern sequential programmer is accustomed to, and this makes parallel machines difficult to use and unattractive for many traditional programmers. To this end, developers of new parallel languages should ask what features from modern sequential languages they might effectively incorporate in their language design.

At the same time, one must concede that programming parallel machines is inherently different from sequential programming, in that the user must express parallelism, data distribution, and typically synchronization and communication. To this end, parallel language developers should attempt to develop features that ease the burdens of parallel programming by providing abstractions for these concepts and optimizing for common cases.

This article explores these two challenges by considering language features and characteristics that would make parallel programming easier while also bringing it closer to broad-market sequential computing. It surveys parallel languages that currently enjoy some degree of popularity in the HEC community and attempts to characterize them with respect to these features. And finally, it provides an introduction to the Chapel programming language, which is being developed as part of DARPA's High Productiv-

[1]CRAY INC., SEATTLE WA
(BRADC@CRAY.COM)

[2]MICROSOFT CORPORATION, REDMOND WA

[3]JPL, PASADENA CA AND UNIVERSITY OF VIENNA, AUSTRIA

ity Computing Systems (HPCS) program in order to try and improve the programmability and overall *productivity* of next-generation parallel machines. Productivity is defined by HPCS as a combination of performance, programmability, portability, and robustness. Chapel strives to positively impact all of these areas, focusing most heavily on programmability.

2 Principles for Productive Parallel Language Design

2.1 Programmer Responsibilities

Before describing the features that we believe productive parallel programming languages ought to provide, we begin by listing the responsibilities that we consider to be the programmer's rather than the compiler's. There are many research projects that take a different stance on these issues, and we list our assumptions here not to contradict those approaches, but rather to bound the space of languages that we consider in this paper and to clarify our starting assumptions in these matters.

Identification of parallelism While a holy grail of parallel computing has historically been to automatically transform good sequential codes into good parallel codes, our faith in compilers and runtime systems does not extend this far given present-day technology. As a result, we believe that it should be the programmer's responsibility to explicitly identify the subcomputations within their code that can and/or should execute in parallel. We also believe that a good language design can greatly help the programmer in this respect by including abstractions that make common forms of parallel computation simpler to express and less prone to simple errors. Such concepts should make the expression of parallel computation natural for the programmer and shift the compiler's efforts away from the detection of parallelism and toward its efficient implementation.

Synchronization To the extent that a language's abstractions for parallelism do not obviate the need for synchronization, the programmer will need to specify it explicitly. While it is possible to create abstractions for data parallel computation that require little or no user-level synchronization (High Performance Fortran Forum 1993; Chamberlain 2001), support for task parallelism tends to necessitate synchronization of some form. As with parallelism, good language design should result in high-level abstractions for synchronization rather than simply providing low-level locks and mutual exclusion.

Data distribution and locality As with detection of parallelism, we have little faith that compilers and runtime systems will automatically do a good job of allocating and distributing data to suit a user's computation and minimize communication. For this reason, we expect that the performance-minded programmer will ultimately need to specify how data aggregates should be distributed across the machine and to control the locality of interacting variables. Once again, it would seem that languages could provide abstractions and distribution libraries to ease the burden of specifying such distributions. Depending on the language semantics, one might also want a means of specifying where on the machine a specific subcomputation should be performed, potentially in a data-driven manner.

2.2 Productive Parallel Language Desiderata

In this section, we enumerate a number of qualities that we believe to be worth consideration in the design of a productivity-oriented parallel language. Since different programmers have differing goals and tastes, it is likely that readers will find some of these characteristics more crucial and others less so. Along these same lines, this list clearly reflects the preferences and biases of the authors, and may neglect characteristics that other language design philosophies might consider crucial. For these reasons, this section should be considered an exploration of themes and characteristics rather than a definitive list of requirements.

2.2.1 A global view of computation We call a programming model *fragmented* if it requires programmers to express their algorithms on a task-by-task basis, explicitly decomposing data structures and control flow into per-task chunks. One of the most prevalent fragmented programming models is the Single Program, Multiple Data (SPMD) model, in which a program is written with the assumption that multiple instances of it will be executed simultaneously. In contrast to fragmented models, a *global-view* programming model is one in which programmers express their algorithms and data structures as a whole, mapping them to the processor set in orthogonal sections of code, if at all. These models execute the program's entry point with a single logical thread, and the programmer introduces additional parallelism through language constructs.

As a simple data-parallel example, consider the expression of a three-point stencil on a vector of values. Figure 1 shows pseudocode for how this computation might appear in both global-view and fragmented programming models. In the global-view version, the problem size is defined on line 1 and used to declare two vectors on line 2. Lines 3–4 express the computation itself, using the global problem size to express the loop bounds and indices.

In the fragmented version, the global problem size (defined on line 1) is divided into a per-task problem size

```
1 var n: int = 1000;
2 var A, B: [1..n] float;
3 forall i in 2..n-1
4   B(i) = (A(i-1) + A(i+1)) / 2;
```

(a)

```
1  var n: int = 1000;
2  var locN: int = n/numTasks;
3  var A, B: [0..locN+1] float;
4  var myItLo: int = 1;
5  var myItHi: int = locN;
6  if (iHaveLeftNeighbor) then
7    send(left, A(1));
8  else
9    myItLo = 2;
10 if (iHaveRightNeighbor) {
11   send(right, A(locN));
12   recv(right, A(locN+1));
13 } else
14   myItHi = locN-1;
15 if (iHaveLeftNeighbor) then
16   recv(left, A(0));
17 forall i in myItLo..myItHi do
18   B(i) = (A(i-1) + A(i+1)) / 2;
```

(b)

Fig. 1 Pseudocode fragments illustrating a data parallel three-point stencil written in (a) global-view and (b) fragmented styles. The global-view code starts with a single logical thread and introduces additional parallelism via the *forall* statement. It also allocates and accesses its arrays holistically. In contrast, the fragmented code assumes that *numTasks* threads are executing the code concurrently, and requires the programmer to divide data into per-processor chunks and manage communication explicitly (illustrated here using message passing, though other communication schemes could also be substituted).

on line 2. This local problem size is then used to allocate the vectors, including extra elements to cache values owned by neighboring tasks (line 3). Lines 4–5 set up default local bounds for the iteration space. The conditionals in lines 6, 10, and 15 express the communication required to exchange boundary values with neighbors. They also modify the local iteration bounds for tasks without neighbors. The computation itself is expressed on lines 17–18 using the local view of the problem size for looping and indices. Note that as written, this code is only correct when the global problem size divides evenly between the number of tasks—more effort would be required to write a general implementation of the algorithm that relaxes this assumption.

As a second example of global-view and fragmented models, consider a task-parallel divide-and-conquer algorithm like Quicksort. Figure 2 shows this computation as it might appear in each programming model. In the global-view version, the code computes the pivot in line 1 and then uses a *cobegin* statement in line 2 to indicate that the two “conquer” steps in lines 3 and 4 can be executed in parallel.

```
1 var pivot = computePivot(lo, hi, data);
2 cobegin {
3   Quicksort(lo, pivot, data);
4   Quicksort(pivot, hi, data);
5 }
```

(a)

```
1  if (iHaveParent) then
2    recv(parent, lo, hi, data);
3  if (iHaveChild) {
4    var pivot = computePivot(lo, hi, data);
5    send(child, lo, pivot, data);
6    Quicksort(pivot, hi, data);
7    recv(child, data);
8  } else
9    LocalSort(lo, hi, data);
10 if (iHaveParent) then
11   send(parent, data);
```

(b)

Fig. 2 Pseudocode fragments illustrating a task parallel Quicksort algorithm written in (a) global-view and (b) fragmented styles. As before, the global-view code starts with a single logical thread, and introduces parallelism using the *cobegin* statement. It operates on its data array as a whole. In contrast, the fragmented code again assumes that *numTasks* threads are executing the code, and requires the user to explicitly embed the divide and conquer task tree into the multiple program images.

In contrast, the fragmented expression of the algorithm describes Quicksort from a single task's point of view, overlaying the virtual tree of recursive calls onto the available tasks. Thus, each task begins on line 2 by receiving the portion of the data for which it is responsible from its parent in the task tree, if it has one. If it has a child at this level of the task tree, it computes the pivot (line 4), and sends half of the work to the child (line 5). It then makes a recursive call to sort the second half of the data (line 6) and receives the sorted data from its child task for the initial half (line 7). If, on the other hand, it has no child at this level of the task tree, it simply sorts the data locally (line 9). In either case, if it has a parent task, it sends the sorted result back to it (line 11).

Note that while the pseudocode presented in these two examples uses a 2-sided message passing style, other fragmented models might use alternative communication paradigms such as 1-sided puts and gets, co-arrays, shared memory, synchronization, etc. For this discussion the important point is not what style of communication is used, but rather that any fragmented model will require the programmer to explicitly specify and manage communication and synchronization due to its focus on coding at a task-by-task level.

While these examples are merely pseudocode, intended to introduce the concepts of global-view and fragmented programming models, they capture some of the differences that exist in real languages which implement these models: global-view codes tend to be shorter and tend to express the overall parallel algorithm more clearly. Fragmented codes are typically longer and tend to be cluttered with the management of per-task details such as local bounds and indices, communication, and synchronization.

The global view is not without some cost, however. Typically, compilers and runtimes for global-view languages must be more sophisticated, since they are ultimately responsible for breaking the algorithm into the per-processor pieces that will allow it to be implemented on a parallel machine. In contrast, compilers for fragmented languages primarily need to be good sequential compilers, although building an understanding of the specific fragmented programming language into the compiler can lead to additional optimization opportunities such as communication pipelining.

The observation that fragmented languages tend to impose less of a burden on their compilers is almost certainly the reason that today's most prevalent parallel languages[1]—MPI, SHMEM, Co-array Fortran, Unified Parallel C (UPC), and Titanium—are based on fragmented or SPMD programming models. In contrast, global-view languages such as High Performance Fortran (HPF), Sisal, NESL, and ZPL have typically not found a solid foothold outside of the academic arena.[2] One exception to this generalization is OpenMP, which provides a global-view approach in its typical usage. However, it also provides a shared-memory abstract machine model, which tends to present challenges to optimizing for locality and scaling to large numbers of processors on current architectures.

We believe that the dominance of the fragmented programming model is the primary inhibitor of parallel programmability today, and therefore recommend that new productivity-oriented languages focus on supporting a global view of parallel programming. Note that support for a global view does not necessarily prevent a programmer from coding on a task-by-task basis; however, it does save them the trouble of doing so in sections of their code that do not require such low-level detail management.

2.2.2 Support for general parallelism Parallel algorithms are likely to contain opportunities for parallel execution at arbitrary points within the program. Yet today's parallel languages tend to support only a single level of parallelism cleanly, after which additional parallelism must either be ignored and expressed sequentially or expressed by resorting to a secondary parallel language. The most common example of this in current practice is the use of MPI to express a coarse level of algorithmic parallelism, combined with OpenMP to specify a second, finer level of parallelism within each MPI task. To a certain extent, this "single level of parallelism" characteristic results from the prevalence of the SPMD model in the programming models, execution models, and implementation strategies of today's parallel languages. Parallelism in the SPMD model is only expressed through the cooperating program instances, making it difficult to express additional, nested parallelism.

In addition, algorithms tend to contain sub-computations that are both data- and task-parallel by nature. Yet most of today's parallel programming languages cleanly support only a single type of parallelism, essentially ignoring the other. Again, due to the prevalence of the SPMD model, data parallelism tends to be the more common model. In order for a parallel language to be general, it should cleanly support both data and task parallelism.

We believe that when languages make it difficult to express nested instances of data and task parallelism, they limit their general applicability within the space of parallel computations and will eventually leave the programmer stuck and frustrated. To this end, we recommend that productivity-oriented parallel languages focus on supporting general forms of parallelism—both data and task parallelism, as well as the composition of parallel code sections.

2.2.3 Separation of algorithm and implementation A great majority of today's languages—sequential as well as parallel—do a poor job of expressing algorithms in a manner that is independent of their data structures'

```
var n: int;
var M: [1..n, 1..n] float;
var V, S: [1..n] float;
for i in 1..n {
  S(i) = 0.0;
  for j in 1..n do
    S(i) += M(i,j)*V(j);
}
```

(a)

```
var n, nnz: int;
var Mvals: [1..nnz] float;
var col: [1..nnz] int;
var rptr: [1..n+1] int;
var V, S: [1..n] float;
for i in 1..n {
  S(i) = 0.0;
  for j in rptr(i)..rptr(i+1)-1 do
    S(i) += Mvals(j) * V(col(j));
}
```

(b)

Fig. 3 Pseudocode fragments illustrating how sensitive algorithms in most languages are to data layout—in this case (a) a dense matrix layout versus (b) a sparse compressed row layout. While the two codes express the same logical matrix–vector multiplication operation, the lack of separation between algorithms and data structures causes the codes to be expressed very differently.

implementation in memory. For example, consider the changes required to convert a typical sequential matrix–vector multiplication algorithm from a dense array to a sparse array using a compressed row storage (CRS) format, as illustrated in Figure 3. To perform the conversion, a new scalar, *nnz* is added to describe the number of non-zeros in the array, while the 2D array *M* is converted into three 1D arrays of varying sizes: *Mvals*, *col*, and *rowptr*. In addition, the inner loop and accesses to *M* and *V* have changed to the point that they are virtually unrecognizable as matrix–vector multiplication to anyone but the experienced CRS programmer. As a result, we have failed in our goal of writing the algorithm independently of data structure. In contrast, languages such as Matlab which do a better job of separating data representation issues from the expression of computation result in codes that are less sensitive to their data structures' implementation details (Gilbert, Moler, and Schreiber 1992).

Note that one need not rely on an example as advanced as converting a dense algorithm into a sparse one to run afoul of such sensitivities. Even when dealing with simple dense arrays, the loop nests in most HEC codes are influenced by factors such as whether a matrix is stored in row- or column-major order, what the architecture's cache-line size is, or whether the processor is a vector or scalar processor. Yet, the algorithm being expressed is semantically independent of these factors. Most novice programmers are taught that algorithms and data structures ought to be orthogonal to one another, yet few of our languages support such separation cleanly without sacrificing performance.

In the realm of parallel computing, these problems are compounded since a computation's expression is sensitive not only to the traditional sequential concerns, but also to the distribution of data aggregates between multiple processors and the communication that may be required to fetch non-local elements to a processor's local memory. Consider how sensitive today's fragmented parallel codes tend to be to such decisions as: whether they use a block or block-cyclic distribution; whether the distribution is applied to a single dimension or multiple dimensions; or whether or not the problem sizes divide evenly between the processors. Altering these decisions tends to affect so much code that making such changes occurs only when absolutely necessary, rather than as part of the normal experimentation that ought to take place while developing new codes. And yet, much of it is orthogonal to the numerical computation that is being expressed. Unfortunately, fragmented programming models have limited means for addressing this problem since data is allocated by each program instance, limiting the user's ability to express computation over a distributed data structure in a manner that is orthogonal to its distribution.

In order to support such a separation between algorithm and data structure implementation, it would seem that parallel languages would minimally need to support a means for defining a data aggregate's distribution, local storage, and iteration methods independently of the computations that operate on the data structure. In such an approach, the compiler's job would be to convert the global-view expression of an algorithm into the appropriate local allocations, iterations, and communication to efficiently implement the code. In order to achieve ideal performance, it seems likely that the language and compiler would need to have knowledge about the mechanisms used for expressing storage and iteration.

2.2.4 Broad-market language features The gap between parallel languages and popular broad-market languages is wide and growing every year. Consider the newly-trained programmer emerging from college with experience using Java, Matlab, or Perl in a sequential computing setting and being asked to implement parallel code in Fortran/C + MPI. In addition to the general challenge of writing correct parallel programs, these programmers are forced to contend with languages that seem primitive compared to those in which they have been trained, containing features whose characteristics were determined by compiler technology that is now decades old.

In order to bring new programmers into the parallel computing community, we believe that new parallel languages need to bridge this gap by providing broad-market features that do not undermine the goals of parallel computing. These features might include: object-oriented programming; function and operator overloading; garbage collection; *generic programming* (the ability to apply a piece of code to a variety of types for which it is suitable without explicitly rewriting it); *latent types* (the ability to elide a variable's type when it can be determined by the compiler); support for *programming in-the-large* (composing large bodies of code in modular ways); and a rich set of support routines in the form of standardized libraries.

2.2.5 Data abstractions Scientific programmers write algorithms that tend to call for a wide variety of data structures, such as multidimensional arrays, strided arrays, hierarchical arrays, sparse arrays, sets, graphs, or hash tables. Yet current parallel languages tend to support only a minimal set of data structures, typically limited to traditional dense arrays (and C-based languages fail to even support much in the way of a flexible multidimensional array). To the extent that these languages fail to support objects, more complex data structures tend to be implemented in terms of simpler arrays, as exhibited by the sparse CRS example of Figure 3b.

Despite the fact that HEC computations almost always operate on large data sets by definition, parallel languages typically fail to provide a richer set of built-in data structures from which the programmer can select. Having such data structures available by default, whether through a library or the language definition itself, not only saves the programmer the task of creating them manually, it also improves code readability by encouraging programmers to use a standard and familiar set of data structures. Such support can also create opportunities for compiler optimizations that hand-coded data structures tend to obscure. As a specific example, the accesses to array *V* in the CRS matrix–vector multiplication of Figure 3b are an example of indirect indexing, which tends to thwart compiler optimization. Yet in the context of CRS, this indexing has specific semantics that would have been valuable to the compiler if the language had supported a CRS sparse matrix format directly. In past work, we have demonstrated how compiler familiarity with a distributed sparse data structure can improve code clarity and provide opportunities for optimization (Ujaldon et al. 1997; Chamberlain 2001; Chamberlain and Snyder 2001). We believe that doing the same for other common data structures would benefit parallel productivity.

2.2.6 Performance Since performance is typically the bottom line in high-performance computing, it stands to reason that support for the previous features should not ultimately prevent users from achieving the performance they require. Yet, this represents a serious challenge since there tends to be an unavoidable tension between performance and programmability that typically complicates programming language design.

Our guiding philosophy here is that languages should provide a spectrum of features at various levels so that code which is less performance-oriented can be written more easily, potentially resulting in less-than-optimal performance. As additional performance is required for a section of code, programmers should be able to rewrite it using increasingly low-level performance-oriented features until they are able to obtain their target performance. This philosophy follows the 90/10 rule: if 90% of the program's time is spent in 10% of the code, the remaining code can afford to be relatively simple to write, read, and maintain, even if it impacts performance. Meanwhile, the crucial 10% should be expressible using a separation of concerns so that distributions, data structures, and communication algorithms can be modified with minimal impact on the original computation. Such a separation should ideally result in increased readability, writability, and maintainability of the performance-oriented code as compared with current approaches.

2.2.7 Execution model transparency A quality that has made languages like C popular is the ability for architecture-savvy programmers to understand roughly how their source code will map down to the target architecture. This gives programmers the ability to make informed choices between different implementation approaches by considering and estimating the performance impacts that they will have on the execution. While such mental mappings are typically imperfect given the complexity of modern architectures and compiler optimizations, having even a coarse model of the code's execution is far better than working in a language which is so abstract that the mapping to the hardware is a mystery.

This would seem to be an important quality for parallel languages to have as well. While sequential programmers will typically be most interested in memory accesses, caches, and registers, parallel programmers will also be concerned with how parallelism is implemented, where code is executed, which data accesses are local and remote, and where communication takes place. A secondary benefit for creating parallel languages with transparent execution models is that it often makes writing compilers for such languages easier by narrowing the space of implementation options. This can result in making code perform more portably since different compilers will need to obey similar implementation strategies at a coarse level, even if their optimizations and implementation details vary.

2.2.8 Portability Given the evolution and diversity of parallel architectures during the past several decades, it seems crucial to make sure that new parallel languages are portable to a diverse set of architectures. This represents a unique challenge since the simplest way to achieve this is to have languages target a machine model that represents a lowest common denominator for the union of parallel architectures—for example, one with a single thread per processor and distributed memory. Yet making such a choice can limit the expressiveness of a language, as exhibited by many current parallel languages. It can also limit the language's ability to take advantage of architectures that offer a richer set of features. Another approach, and the one we espouse, is to target a more sophisticated machine model that can be emulated in software on architectures that fail to support it. For example, a language that assumes each processor can execute multiple threads may be implemented on a single-threaded processor by managing the threads in software. In addition, we believe that languages should be usable in a mode that is simpler and more compatible with a less-capable architecture when the best performance is required. Continuing our example, programmers of a single-threaded architecture ought to be able to restrict their code to use a single thread per processor in order to avoid the overhead of software multithreading.

2.2.9 Interoperability with existing codes When designing a new parallel language, it is often tempting to fantasize about how much cleaner everyone's existing codes would be if they were rewritten in your language. However, the reality of the situation is that there is so much code in existence, so much of which is obfuscated by struggling with many of the challenges listed above, that it is simply unrealistic to believe that this will happen in most cases. For this reason, making new languages interoperate with existing languages seems crucial.

For our purposes, interoperation has two main goals. The first is to deal with the mechanics of invoking functions from an existing language to the new language and back again, passing arguments of various standard types. The second is to have the two languages interoperate on each others' data structures in-place, in order to avoid making redundant copies of large distributed data structures, or redistributing them across the machine. While such approaches may be reasonable for exploratory programming, if production codes switch between the two languages during their core computations, such overheads are likely to impact performance too much. It is unrealistic to assume that any two languages will be able to compute on shared data structures of arbitrary types without problems, but new languages should strive to achieve this for common data structures and their implementations within HEC codes.

2.2.10 Bells and whistles There are a number of other features that one may want in a language that have less to do with the language and more to do with its implementation, libraries, and associated tools. For example, attractive features might include built-in support for visualizations, interpreters to support interactive exploratory programming, good debuggers and performance monitors, and clear error messages. While these features are not as "deep" from a language-design perspective as many of the others that have been covered in this section, one only needs to interact with a few users (or reflect on their own experiences) to realize how valuable such ancillary features can be for productivity.

3 Parallel Language Survey

In this section, we give a brief overview of several parallel languages and describe how they meet or fall short of the design principles in the previous section. Given the large number of parallel languages that have been developed in recent decades, we focus primarily on languages that are in current use and that have influenced our design.

Observations on SPMD languages Since many of the languages described in this section provide the programmer with an SPMD view of computation, we begin by making some general observations about the SPMD programming model. As described in the previous section, SPMD programming gives the user a fairly blunt mechanism for specifying parallel computation and distributed data structures: parallelism is only available via the multiple cooperating instances of the program while distributed data aggregates must be created manually by having each program instance allocate its local piece of the data structure independently. Operations on remote chunks of the data structure typically require some form of communication or synchronization to keep the cooperating program instances in step.

The main benefits of languages supporting an SPMD programming model relate to their simplicity. The restricted execution model of these languages tends to make them fairly simple to understand and reason about, resulting in a high degree of transparency in their execution model. This simplicity can also result in a high degree of portability due to the lack of reliance on sophisticated architectural features or compilers. Yet the simplicity of this model also requires the user to shoulder a greater burden by managing all of the details related to manually fragmenting data structures as well as communicating and synchronizing between program instances. This explicit detail management obfuscates algorithms and creates rich opportunities for introducing bugs that are notoriously difficult to track down.

3.1 Communication Libraries

MPI MPI (Message Passing Interface; Message Passing Interface Forum 1994) (Snir et al. 1998) is the most prevalent HEC programming paradigm, designed by a broad consortium effort between dozens of organizations. MPI supports a fragmented programming model in which multiple program instances execute simultaneously, communicating by making calls to the MPI library. These routines support various communication paradigms including two-sided communication, broadcasts, reductions, and all-to-all data transfers. MPI-2 is an extension to the original MPI standard that introduces a form of single-sided communication as well as dynamic task creation which allows programs to be spawned at arbitrary program points rather than just at load-time (Message Passing Interface Forum 1997; Gropp et al. 1998).

MPI has been a great success in the HEC community due to the fact that it has a well-specified standard, freely available implementations, and a high degree of portability and performance consistency between architectures. As a result, MPI has become the *de facto* standard for parallel programming.

MPI is most often used in an SPMD style because of its simplicity, yet it does not force the user into an SPMD programming model since instances of distinct programs can be run cooperatively. However, even when used in this mode, MPI shares the general characteristics of SPMD languages as described above, because of the fact that its parallelism is expressed at the granularity of a program rather than a function, statement, or expression.

The MPI interface is supported for C and Fortran, and MPI-2 adds support for C++. To a large extent, MPI's ability to support the separation of an algorithm from its implementation, broad-market features, data abstractions, and interoperability with existing codes is strongly tied to the base languages with which it is used, as impacted by the introduction of a fragmented programming model. MPI tends to serve as a favorite target of criticism in the HEC community, due in part to its failure to satisfy many of the design principles from Section 2.2. However, the fact that so many sophisticated parallel codes have been written using MPI is testament to its success in spite of its shortcomings.

PVM PVM (Parallel Virtual Machine) (Geist et al. 1994a, 1994b) is another two-sided message passing interface that was developed during the same period as MPI, but which has not enjoyed the same degree of adoption in the community. PVM's interface is somewhat simpler than MPI's, and supports the ability to dynamically spawn new parallel tasks at the program level, as in MPI-2. PVM is supported for C, C++, and Fortran. Despite numerous differences between PVM and MPI, our analysis of PVM with respect to our desiderata is much the same: ultimately its fragmented programming model stands in the way of making it a highly productive parallel language.

SHMEM SHMEM (Barriuso and Knies 1994) is a single-sided communication interface that was developed by Cray in the 1990s to support a single processor's ability to *put* data into, and *get* data from, another processor's memory without that processor's code being involved. Such an interface maps well to Cray architectures and supports faster data transfers by removing much of the synchronization and buffering that is necessitated by the semantics of two-sided message passing. This one-sided communication style is arguably easier to use since programmers do not have to write their program instances to know about the communication that the other is performing, yet in practice some amount of synchronization between the programs tends to be required to know when remote values are ready to be read or written. Since its initial development, a portable version of SHMEM has been implemented named GPSHMEM (Generalized Portable SHMEM) and is designed to support portability of the interface to a broader range of platforms (Parzyszek, Nieplocha, and Kendall 2000).

Programmers debate whether two-sided or single-sided communication is easier because of the tradeoff between having both program instances involved in the communication versus the subtle race conditions that can occur as a result of incorrect synchronization in the one-sided model. However, from our perspective, the SHMEM interface still relies on an SPMD programming model, and as such does little to address our wishlist for a productive language.

ARMCI and GASNet Two other single-sided communication libraries that have recently grown in popularity are ARMCI (Aggregate Remote Memory Copy Interface) (Nieplocha and Carpenter 1999) and GASNet (Global Address Space Networking) (Bonachea 2002). Both of these interfaces seek to support portable single-sided communication by generalizing the concepts established by the SHMEM library. ARMCI was developed at Pacific Northwest National Laboratory and is built on native network communication interfaces and system resources. It has been used to implement GPSHMEM, Co-array Fortran (Dotsenko, Coarfa, and Mellor-Crummey 2004), and the Global Arrays library. GASNet was developed at Berkeley. It is constructed around a core API that is based on Active Messages (von Eicken et al. 1992) and is implemented on native network communication interfaces. GASNet has been used to implement UPC (Chen et al. 2003; Chen, Iancu, and Yelick 2005) and Titanium (Su and Yelick 2005). Both ARMCI and GASNet are primarily used as implementation layers for libraries and languages. Their use as stand-alone parallel programming languages has similar productivity limiters as the previous communication libraries described in this section.

3.2 PGAS Languages

A group of parallel programming languages that are currently receiving a great deal of attention are the *Partitioned Global Address Space (PGAS) languages* (Carlson et al. 2003). These languages are designed around a memory model in which a global address space is logically partitioned such that a portion of it is local to each processor. PGAS languages are typically implemented on distributed memory machines by implementing this virtual address space using one-sided communication libraries like ARMCI or GASNet.

PGAS languages are a welcome improvement over one-sided and two-sided message passing libraries in that they provide abstractions for building distributed data structures and communicating between cooperating program instances. In spite of these improvements, however, users still program with the SPMD model in mind, writing code with the understanding that multiple instances of it will be executing cooperatively. Thus, while the PGAS languages improve productivity, they continue to fall short of our goals of providing a global view of parallel computation and general parallelism.

The three main PGAS languages are Co-array Fortran, UPC, and Titanium. While these are often characterized as Fortran, C, and Java dialects of the same programming model, this is an over-generalization. In the paragraphs below, we highlight some of their features and differences.

Co-array Fortran Co-array Fortran (CAF; formerly known as F—) is an elegant extension to Fortran to support SPMD programming (Numerich and Reid 1998). Its success has been such that its features will be included in the next Fortran standard (Numerich and Reid 2005). CAF supports the ability to refer to the multiple cooperating instances of an SPMD program (known as *images*) through a new type of array dimension called a *co-array*. By declaring a variable with a co-array dimension, the user specifies that each program image will allocate a copy of the variable. Each image can then access remote instances of the variable by indexing into the co-array dimensions using indices that refer to the logical image space. Co-arrays are expressed using square brackets which make them stand out syntactically from traditional Fortran arrays and array references. Synchronization routines are also provided to coordinate between the cooperating images.

While PGAS languages like CAF tend to provide a more coherent view of distributed data structures than message-passing libraries, they still require users to fragment arrays into per-processor chunks. For example, in Co-array Fortran an *n*-element array would typically be allocated as a co-array of size *n*/*numImages*, causing the user to deal with many of the same data structure fragmentation details as in MPI programming. The greatest programmability improvement comes in the area of simplifying communication, since co-arrays provide a truly elegant abstraction for data transfer as compared to one-sided and two-sided communication. Moreover, the syntactic use of square brackets provides the user with good insight into the program's execution model.

UPC UPC is similar to Co-array Fortran in that it extends C to support PGAS-style computation (Carlson et al. 1999; El-Ghazawi et al. 2005; UPC Consortium 2005). However, UPC supports a very different model for distributing arrays in which declaring an array variable with the *shared* keyword causes the linearly-ordered array elements to be distributed between the program instances (or *threads*) in a cyclic or block-cyclic manner. This mechanism provides a more global view of the user's arrays, yet it is restricted enough that locality-minded programmers will still tend to break arrays into *THREADS* arrays of *n*/*THREADS* elements each. The biggest downside to UPC's distributed arrays is that since they are based on C arrays, they inherit many of the same limitations. In particular, performing a 2D blocked decomposition of a 2D array is non-trivial, even if the problem size and number of threads are statically known.

UPC also supports a slightly more global view of control by introducing a new loop structure, the *upc_forall loop*, in which global iterations of a C-style "for loop" are assigned to threads using an *affinity expression*. While this support for a global iteration space is useful, it also tends to be somewhat at odds with the general UPC execution model which is SPMD by nature until a upc_forall loop is encountered.

UPC's most useful feature is perhaps its support for pointers into the partitioned global shared address space. Pointers may be declared to be *private* (local to a thread) or *shared*, and may point to data that is also either private or shared. This results in a rich abstraction for shared address space programming that successfully maintains a notion of affinity since a symbol's shared/private characteristic is part of the type system and therefore visible to the programmer and compiler.

Titanium Titanium is a PGAS language that was developed at Berkeley as an SPMD dialect of Java (Yelick et al. 1998; Hilfinger et al. 2005). Titanium adds several features to Java in order to make it more suited for HEC, including: multidimensional arrays supporting iterators, subarrays, and copying; immutable "value" classes; operator overloading; and *regions* that support safe, performance-oriented memory management as an alternative to garbage collection. To support coordination between the SPMD program instances, Titanium supports: a number of synchronization and communication primitives; *single* methods and variables which give the compiler and programmer the ability to reason about synchronization in a type-safe manner; and a notion of private/shared references and data similar to that of UPC.

In many respects, Titanium is the most promising of the PGAS languages in terms of our productivity wishlist. It has the most support for broad-market features that current sequential programmers—particularly those from the Java community—would come to expect. The fact that it is an object-oriented language gives it better capabilities for separating algorithms from implementations and providing mechanisms for creating data abstractions. Its support for single annotations and private/shared distinctions help it achieve performance and to expose its execution model to the user. The chief disadvantage to Titanium is that, like other PGAS languages, it supports an SPMD programming model which thwarts its ability to support a global view of data structures and control, as well as to express general parallelism.

3.3 OpenMP

OpenMP is a set of directives and library routines that are used to specify parallel computation in a shared memory style for C, C++, and Fortran (Dagum and Menon 1998; Chandra et al. 2000). OpenMP is a drastic departure from the communication libraries and PGAS languages described above in that it is the first programming model described in this survey that can be used in a non-fragmented manner. OpenMP is typically used to annotate instances of parallelism within a sequential program—most notably, by identifying loops in which parallel computation should occur. The OpenMP compiler and runtime implement this parallelism using a team of cooperating threads. While OpenMP can be used in a fragmented manner, querying the identity of a thread within a parallel region and taking actions based on that identity, it is more often used in a global-view manner, identifying the opportunity for parallelism and letting the the compiler and runtime manage the thread-level details. This ability to inject parallelism incrementally into a sequential program is considered OpenMP's greatest strength and productivity gain.

While the OpenMP standard supports nested parallelism, most implementations only handle a single level of parallelism at a time. OpenMP is currently not suited for task parallelism, though there is interest in evolving it to handle such problems. The biggest downside to OpenMP is its reliance on a shared-memory programming model, which has generally not proven to be scalable to large numbers of processors. For this reason, OpenMP is typically used within shared memory portions of a larger distributed memory machine—for example, to express thread level parallelism within an MPI program running on a cluster of SMPs. OpenMP's other downside is that, like MPI, it is used in combination with C, C++, and Fortran, limiting its ability to support higher-level data abstractions and broad-market features from more modern languages.

3.4 HPF

High Performance Fortran (HPF; High Performance Fortran Forum 1993, 1997) (Koelbel et al. 1996) is a parallel extension to Fortran 90 that was developed by the High Performance Fortran Forum, a coalition of academic and industrial language experts. HPF is an evolution of earlier parallel Fortran dialects such as Fortran-D (Fox et al. 1990), Vienna Fortran (Chapman, Mehrotra, and Zima 1992; Zima et al. 1992), and Connection Machine Fortran (Albert et al. 1988). HPF is a global-view language supporting distributed arrays and a single logical thread of computation. It supports a number of directives that allow users to provide hints for array distribution and alignment, loop scheduling, and other details relevant to parallel computation. HPF compilers implement the user's code by generating an SPMD program in which the compiler-generated code and runtime manage the details of implementing distributed arrays and interprocessor communication.

HPF meets our productivity goals by providing a global view of computation, but does not provide for general parallelism. Because of its SPMD execution model and single logical thread of execution, it is good at expressing a single level of data parallelism cleanly, but not at expressing task or nested parallelism. HPF also suffers from a lack of transparency in its execution model—it is difficult for both users and compilers to reason about how a code will/should be implemented (Ngo, Snyder, and Chamberlain 1997). Depending on who you talk to, HPF's lack of success is attributed to some combination of: this lack of a transparent execution model; its inability to achieve good performance quickly enough (alternatively, the impatience of its user community); its lack of support for higher-level data abstractions such as distributed sparse arrays, graphs, and hash tables; and a number of other theories. In our work on Chapel, we build on HPF's general approach for global-view programming on distributed arrays while adding richer support for general parallelism and data structures.

3.5 ZPL

ZPL is an array-based parallel language developed at the University of Washington (Snyder 1999; Chamberlain 2001; Deitz 2005). Like HPF, ZPL supports global-view parallel computation on distributed arrays where the management details are implemented by the compiler and runtime using an SPMD implementation. As with HPF, ZPL's execution model only supports programs with a single level of data parallelism at a time, making it similarly limited in terms of generality.

ZPL is unique among global-view languages in that it supports execution model transparency in its syntax via a concept known as the WYSIWYG performance model (Chamberlain et al. 1998). Traditional operations are semantically restricted in ZPL to only be applicable to array expressions that are distributed in an aligned manner. To operate on arrays that are not aligned, a series of *array operators* are used to express different access patterns including translations, broadcasts, reductions, parallel prefix operations, and gathers/scatters. These stylized operators show up clearly in the syntax and can be used to reason about where programs require communication and what type it is. In this way, programmers are able to make coarse algorithmic tradeoffs by inspecting the syntax of their programs. In practice, ZPL achieves performance that is competitive with hand-coded Fortran+MPI, and is portable to most platforms that support MPI. ZPL supports array computations using a language concept known as a *region*[3] to represent distributed index sets, including sparse arrays (Chamberlain et al. 1999; Chamberlain 2001; Chamberlain and Snyder 2001). In our Chapel work, we expand upon the region concept to support distributed sets, graphs, and hash tables. We also strive to implement the general parallelism and broad-market features that ZPL fails to provide.

3.6 Cilk

Cilk is a global-view parallel language that supports a multithreaded execution model (Randall 1998; Supercomputing Technologies Group 2001). To the programmer, Cilk codes look like C programs that have been annotated with operations to spawn and synchronize threads. Moreover, the elision of Cilk's keywords result in a C program that validly implements Cilk's semantics. Cilk's runtime system is in charge of scheduling the parallel computation so that it will run efficiently on a platform, and it utilizes aggressive work sharing and stealing techniques to balance the computational load and avoid swamping the system with too much parallelism.

Cilk is attractive with respect to our productivity desiderata in that it provides a global view of nestable parallelism. While Cilk threads can be used to implement data parallelism, the language provides little in the way of abstractions to make operating on distributed arrays trivial—the user would have to manage such data structures manually. As a result, Cilk seems best-suited for nested task parallelism—an important area since so few languages support it. The current version of Cilk is also limited to shared memory platforms, resulting in potential scaling and portability problems as in OpenMP. While previous versions of Cilk ran on clusters of workstations (Joerg 1996), this capability has not been carried forward. A final point of interest about Cilk is that while its execution model is not transparent, it does have an analytic performance model that allows users to predict asymptotic program performance.

4 Chapel Overview

In this section, we describe a number of key Chapel language features, relating them to the goals that we established in Section 2.2. Because of the space limitations of this paper, we provide a brief introduction to a number of Chapel concepts without covering them in exhaustive detail. For a more complete introduction to the language, the reader is referred to the Chapel Language Specification (Cray Inc. 2005).

4.1 Base Language Concepts in Chapel

Chapel is a global-view parallel language that supports a block-imperative programming style. In fact, the "pseudocode" fragments presented in Figures 1a, 2a, and 3 are written in Chapel. Syntactically, Chapel diverges from other block-imperative HEC languages like C and Fortran. At times, this is because of our differing goals (e.g. support for generic programming and a better separation of algorithm and data structures), and at times because of differing sensibilities. The decision to avoid building directly on C or Fortran also reflects our belief that looking too similar to an existing language can cause users to fall back on their sequential programming styles and assumptions rather than considering their algorithm afresh, as is often necessary when writing parallel programs. We believe that interoperability with existing languages is far more important than extending an existing syntax. That said, elements of Chapel's syntax will be familiar to users of C, Fortran, Java, Modula, and Ada.

Basic types and variables Chapel supports a number of built-in types that are standard for HEC: floating point and integer types of varying widths, complex values, boolean values, and strings. Variable declarations in Chapel take the general form:

```
var <name> [ : <definition> ] [ = <initializer> ] ;
```

where *name* gives the name of the variable, *definition* indicates its type and structural properties, and *initializer* supplies its initial value. For code robustness reasons, each type in Chapel has a default initial value used for uninitialized variable declarations, though these may be overridden or suppressed by the user. Chapel's declaration syntax differs from the more common "type first" style used in C and Fortran primarily due to our goal of supporting generic programming by eliding a variable's type (described below). In addition, we have a preference for variable definitions that can be read left-to-right, and for array specifications that can define multiple variables and that need not be split by a variable's name.

Locales In Chapel, we use the term *locale* to refer to the unit of a parallel architecture that is capable of performing computation and has uniform access to the machine's memory. For example, on a cluster architecture, each node and its associated local memory would be considered a locale. Chapel supports a *locale type* and provides every program with a built-in array of locales to represent the portion of the machine on which the program is executing. Effectively, the following variables are provided by the system:

```
const numLocales: int = ...;
const Locales: [1..numLocales] locale = ...;
```

Programmers may reshape or partition this array of locales in order to logically represent the locale set as their algorithm prefers. Locales are used for specifying the mapping of Chapel data and computation to the physical machine using features described below.

Other basic types Chapel supports *tuples* of homogeneous and heterogeneous types as a means of bundling several values together in a lightweight fashion. It also supports a *sequence type* that is used to represent ordered homogeneous collections in cases where richer data aggregates such as arrays are overkill. Chapel's semantics for operations on arrays are defined in terms of sequence semantics.

Chapel also supports *type unions* which are similar to C's union types, except that they must be used in a type-safe manner. For example, having written to a union variable via member x, the variable cannot then be accessed via member y.

Control flow and functions Chapel supports a variety of standard control flow statements as in most block-imperative languages—*loops*, *conditionals*, *select statements* (similar to "switch" or "case" statements), *breaks*, *continues*, *gotos*, and *returns*. Chapel supports function definitions that support *default argument values*, *argument matching by name*, and *argument intents*. Functions can be *overloaded*, as can most of the standard operators. Function resolution is performed in a *multiple-dispatch* manner.

4.2 Parallelism in Chapel

Chapel is designed around a *multithreaded execution model* in which parallelism is not described using a processor- or task-based model, but in terms of independent computations implemented using threads. Rather than giving the programmer access to threads via low-level fork/join mechanisms and naming, Chapel provides higher-level abstractions for parallelism using anonymous threads that are implemented by the compiler and runtime system. We choose this model both to relieve users of the burden of thread management and due to its generality. Although most architectures do not currently have hardware support for multithreading, we believe that the benefits gained in programmability and generality will outweigh the potential performance impacts of managing these threads in software. Moreover, as multicore processors become more readily available, we anticipate that support for multithreading will become more commonplace.

4.2.1 Data parallelism

Domains and arrays Chapel supports data parallelism using a language construct known as a *domain*—a named, first-class set of indices that is used to define the size and shape of arrays and to support parallel iteration. Note that we use the term *array* in Chapel to represent a general mapping from an index set of arbitrary type to a set of variables. This results in a richer array concept than the rectilinear blocks of data supported by most HEC languages. Chapel's domains are an evolution of the region concept that was developed in ZPL, generalized here to support sets, graphs, and associative arrays. Chapel has three main classes of domains, described in the following paragraphs.

Arithmetic domains are those in which indices are represented using multidimensional integer coordinates, to support rectilinear index sets as in traditional array languages. Arithmetic domains may have dynamic sizes and bounds, and they may be dense, strided, or sparse in each dimension. The following declarations create a simple 2D arithmetic domain and array:

```
var D: domain(2) = [1..m, 1..n];
// a 2D arithmetic domain storing
// indices (1,1)...(m,n)
var A: [D] float;
// an m × n array of floating point values
```

Indefinite domains represent an index set of arbitrary type, as specified by the user. For example, a programmer might choose to use floating point values, strings, or object references as an indefinite domain's index type.

Indefinite domains are used to implement sets or associative arrays, and can be thought of as providing hash table-like functionality. The following example creates an array of integers indexed using strings:

```
var People: domain(string);
// a domain storing string indices,
// initially empty
var Age: [People] int;
// an array of integers indexed via strings
// in the People domain
People += "John";
// add string "John" to the People domain
Age("John") = 62;          // set John's age
```

Opaque domains are those in which the indices have no values or algebra in relation to one another—they are simply anonymous indices, each of which is unique. Opaque domains can be used to represent graphs or link-based data structures in which there is no requirement to impose an index value or ordering on the nodes that comprise the data structure. Opaque domains and arrays are declared as follows:

```
var Points: domain(opaque);
// a domain storing opaque indices
var Weight: [Points] int;
// an integer array indexed via "Points" indices
var aPoint: index(opaque) = Points.new();
// create and store a new anonymous index
```

As shown above, Chapel arrays are declared by specifying the domain(s) that define the array's size and shape in combination with an element type that is to be stored for each element within the array. In general, domains may be dynamically modified to add or subtract indices from the set that they describe. When this occurs, any arrays defined using the domain are reallocated to reflect the changes to the domain's index set. Thus, the declaration of an array over a domain establishes an invariant relationship between the two concepts that is maintained dynamically throughout the array's lifetime.

Index types and subdomains Chapel supports an *index type* that is parameterized by a domain value. A variable of a given index type may store any of the indices that its domain describes. The following code declares and initializes an index variable for each of the three domains declared above:

```
var centerD: index(D) = (m/2, n/2);
// the center of domain D
var name: index(People) = "John";
// a name from domain People
var anotherPoint: index(Points) = Points.new();
// a new point from the Points domain
```

A domain variable may be declared as a *subdomain* of another domain, constraining its indices to be a subset of those described by its parent. For example, the following code creates a subdomain for each of the three domains declared above:

```
var innerD: subdomain(D) = [2..m-1, 2..n-1];
// the inner indices of domain D
var Adults: subdomain(People) = ...;
// the adults in the People domain
var HeavyPoints: subdomain(Points) = ...;
// the heavy points from the Points domain
```

Index types and subdomains are beneficial because the compiler can often reason about them to eliminate bounds checks. For example, if an index value from *innerD* is used to access array *A*, no bounds check is required since *innerD* is a subdomain of *A*'s domain *D*. Index types and subdomains also provide readers of Chapel code with additional semantic information about a variable's value. In contrast, code can be difficult to understand when every variable used to access an arithmetic array is declared as an integer.

Iteration, slicing, and promotion Parallel iteration is specified in Chapel using *forall loops* in either statement or expression form. Forall loops iterate over domains, arrays, or other expressions that evaluate to a sequence of values. Forall loops may optionally define *loop variables* that take on the values described by the collection being traversed. For example, a forall loop over a domain will create loop variables of that domain's index type that will take on the index values described by the domain. Loop variables are only defined for the body of the forall loop. Figure 4 illustrates forall loops for our running domain examples.

Chapel users can index into arrays an element at a time using individual index values, such as those resulting from a forall loop over a domain. Alternatively, they may index into arrays in an aggregate manner by using a domain expression to specify an arbitrary subset of the array's values. This provides an array slicing mechanism similar to that of Fortran 90, yet with richer semantics since Chapel domains can describe arbitrary index sets. For example, the following expressions use our previous subdomain declarations to slice into arrays declared over their parent domains:

```
... A(innerD) ...
... Age(Adults) ...
... Weight(HeavyPoints) ...
```

Promotion, reductions, and scans Chapel supports the *promotion* of scalar operators and functions over data

```
forall ij in D {
  A(ij) = ...;
  ...
}

... [ij in D] A(ij) ...
```

(a)

```
forall i in People {
  Age(i) = ...;
  ...
}

... [i in People] Age(i) ...
```

(b)

```
forall pt in Points {
  Weight(pt) = ...;
  ...
}

... [pt in Points] Weight(pt) ...
```

(c)

Fig. 4 Forall loops in statement and expression form for the three domains declared at the beginning of Section 4.2.1. The loop variables are declared implicitly to be the domain's index type, and are scoped by the loop body. More specifically, *ij* is an *index*(*D*)—a 2-tuple of integers from domain *D*; *i* is an *index*(*People*)—a string from the *People* domain; and *p* is an *index*(*Point*)—an opaque index from the *Point* domain.

aggregates such as whole array references or array slices. Whole-array assignment is a trivial example involving the promotion of the scalar assignment operator. The semantics of such promotions are defined using sequence-based interpretations of the array expressions. Promoted operators take sequence expressions of compatible rank and shape. Chapel also supports a cross-product style of promotion. Promotions over aggregates are implicitly parallel.

Chapel also supports other specialized forms of data parallel computation such as *reductions* and parallel prefix computations (*scans*). These operators can be invoked using built-in operations like *sum* or *min*, or users can specify their own operations that define the component functions of a scan or reduction (Deitz et al. 2006).

As example uses of promotion and reductions, consider the following Chapel statements which compute approximate norms for a 3D $n \times n \times n$ array, A. These promote the standard exponentiation operator and absolute value function across A's elements, using sum and max reductions to collapse the resulting values to a scalar:

```
var rnm2 = sqrt((sum reduce A**2) / n**3);
var rnmu = max reduce abs(A);
```

Domain distributions A domain's indices may be decomposed between multiple locales, resulting in a distributed allocation for each array defined using that domain. Domain distributions also provide a default work location for threads implementing parallel computation over a domain's indices or an array's values. Because of the constrained relationship between an array and its defining domain, two arrays declared using the same domain are guaranteed to have the same size, shape, and distribution throughout their lifetimes, allowing the compiler to reason about the distribution of aligned arrays. Subdomains also inherit their parent domain's distribution unless explicitly over-ridden, providing additional alignment information.

Chapel will support a number of traditional distributions (e.g. block, cyclic, recursive bisection, graph partitioning algorithms) as part of its standard library. However, a primary goal of the language is to allow users to implement their own distributions when they have application-specific knowledge for how an array should be distributed that is not captured by the standard distributions.

Support for these user-defined distributions is one of our most aggressive research areas in the Chapel project, and a rich enough topic to warrant a paper of its own (Diaconescu and Zima 2007). For this discussion, it suffices to say that a distribution specifies how a domain's indices are mapped to a set of target locales, as well as how domain indices and array elements should be stored within each locale's memory. Our current framework requires distribution authors to create five classes that represent: the distribution; a domain declared using the distribution; a single locale's portion of such a domain; an array declared using the domain; and a single locale's portion of such an array. These classes will be expected to support a standard interface of accessors, iterators, query functions, and data transfer methods which the compiler will target during its rewrites of the user's global-view code. Domain distributions are a key part of Chapel's separation of concerns, allowing users to define data structures independently of the computations that operate on them.

4.2.2 Task parallelism Chapel supports task parallelism using stylized forms of specifying parallel statements and synchronizing between them. The primary way to launch a series of parallel tasks is to use the *cobegin statement*, a compound statement which asserts that each of its component statements can be executed in parallel. A variation on the cobegin statement is the *begin statement* which identifies a statement that can execute in parallel with the statements that follow it. Parallel tasks can also be specified symmetrically using the forall loops described in the previous section.

Chapel's parallel tasks can coordinate with one another using *synchronization variables* that support *full/empty* *semantics* in addition to their normal data values. By default, reads to a synchronization variable only complete when the variable is "full" and leave it "empty", while writes do the opposite. The user may explicitly specify alternate semantics such as writing to the variable and leaving it full regardless of the initial state. Synchronization variables support a structured form of producer/consumer semantics on arbitrary variables. Begin statements can store their results to synchronization variables as a means of expressing *futures* in Chapel.

Chapel also supports *atomic sections* to indicate that a group of statements should be executed as though they occurred atomically from the point of view of any other thread. Atomic sections have been of recent interest to the software transactional memory community because of their intuitiveness and ability to replace explicit lock management in user code (Fraser 2003; Harris et al. 2005, 2006; Adl-Tabatabai et al. 2006). By specifying intent rather than mechanism, atomic sections result in a higher-level abstraction for users than locks. We also value them for their role in allowing code from different authors to compose cleanly.

4.2.3 Composition of parallelism Because of its multithreaded execution model, Chapel does not restrict parallel code from appearing within the body of a forall loop or a cobegin statement. As a result, each statement body can contain statements of either type, resulting in arbitrarily nested parallelism and general support for composition of parallel tasks. The key to achieving this in a scalable manner is to realize that specifying that work *may* execute in parallel does not imply that the runtime *must* spawn a new thread for each parallel computation. Instead, we anticipate using work sharing and stealing techniques combined with a certain degree of laziness in evaluation to throttle the creation of parallel threads and avoid swamping the system, as in Cilk and the Cray MTA (Blumofe and Leiserson 1994; Alverson et al. 1995).

Parallelism summary Many of the desirable language characteristics laid out in Section 2.2 are fulfilled in Chapel by the concepts described in this section. Chapel's multithreaded execution model and support for domains, arrays, forall loops, and cobegin statements support its global view of computation. Combining these concepts with synchronization variables and atomic sections fulfills our requirements for supporting general parallel computation. Domains, arrays, and index types help support our need for rich data abstractions, and the ability to change a domain's distribution and implementation at its declaration point without touching the rest of the code supports Chapel's goal of separating algorithms from their implementing data structures. Similarly, in many codes a domain may be changed from dense to sparse without altering the code that operates on it. As an example, the following loop can be used to perform a matrix–vector multiplication in Chapel without specifying whether the matrix A is sparse or dense, how it is distributed between locales, or how the indices and array elements are stored in each locale's memory.

```
s = sum reduce(dim=2)
        [i,j in A.domain] A(i,j) * v(j);
```

4.3 Locality Control in Chapel

Chapel's goals include allowing the programmer to control where data is stored and where computation takes place on the machine, and to allow this information to be added to a program incrementally as additional performance tuning is required. Domain distribution is one example of locality control that has been described here, and it supports this principle: a domain can initially be declared without distributing it, and later, a distribution can be added without changing any of the code that refers to the domain or its arrays. Similarly, users may initially wish to use a distribution from the standard library, and later write their own custom distribution once performance concerns become more critical.

When iterating over a domain, each iteration will typically execute on the locale that owns that index, as specified by the domain's distribution. Users can override this behavior—or control the locality of any other executable statement—using an *on clause* which references a locale and specifies that the given computation should take place there. An extreme example of an on clause can be used to revert to a fragmented programming style:

```
forall loc in Locales do
  on loc do fragmentedFunction();
```

On clauses can also be used in a data-driven manner by naming a variable rather than a locale. This causes the computation to take place on the locale that owns the variable in question. For example, in the following forall loop, the loop body will be executed on the locale that owns element $A_{j/2,2\cdot i}$.

```
forall (i,j) in D do
  on A(j/2, 2*i) do A(i,j) = foo(A(j/2, 2*i));
```

On clauses can also be applied to variable declarations to cause variables to be allocated on a specific locale. Typically, variables that are not explicitly managed in this way are allocated in the locale on which the thread is executing.

The locality-oriented features of this section are designed to help Chapel programmers tune their program for performance, since so much parallel performance is determined by the relative placement of threads and the data that they access. These features also provide an execution model for Chapel programmers, albeit one that is fairly low-level.

4.4 Object-Oriented Programming in Chapel

Object-oriented programming (OOP) has proven to be a very useful concept in mainstream programming languages, providing a foundation for separating interfaces from implementations, enabling code reuse, and generally helping programmers reason about their code using data structures whose composition reflects the systems they are striving to model. Chapel supports object-oriented features in order to leverage this productivity boost in a parallel setting.

Chapel provides two basic flavors of classes: *traditional classes* and *value classes*. Traditional classes are assigned and passed by reference, mimicking the semantics of traditional object-oriented languages (Java being a notable recent example). Value classes are assigned and passed by value, making them similar to structures or records in traditional languages, yet ones that support method invocation (making them similar to C++ classes). Chapel also supports *garbage collection* due to its productivity benefit in eliminating common memory management errors in modern OOP languages.

We have decided against making Chapel a pure object-oriented language, both to reflect the necessity of operating on value types efficiently in the HEC market, and because of the fact that a large fraction of the HEC community has not been trained to think and program in the object-oriented paradigm since Fortran and C have been the dominant HEC languages for decades. As a result, though Chapel supports OOP features, it typically does not require users to adopt them unless they want to. In practice, Fortran and C programmers are free to ignore Chapel's classes and program in a traditional block-imperative style while Java and C++ programmers are likely to utilize a more object-based style in their code.

The one exception to this rule is in the specification of advanced Chapel concepts such as user-defined reductions, scans, and domain distributions. In these cases, the OOP framework provides a very natural means of specifying a required interface, as well as a mechanism for specializing existing implementations without redefining each component manually. For this reason, we have chosen to express such concepts using a class library. Note that even if Fortran and C programmers need to utilize these advanced concepts in their codes, the bulk of their programs can still be written in a non-OOP style.

The features described in this section are supported in Chapel to provide broad-market language concepts in a parallel programming language, and to leverage the productivity improvements that mainstream computing has enjoyed through their use. Support for classes also provides us with a strong foundation for building higher-level language features such as domains and user-defined distributions.

4.5 Generic Programming in Chapel

Chapel supports language features that enable *generic programming*, which we define as the ability to write a subcomputation that is flexible with respect to the variable types to which it is applied. We implement this flexibility in the source code by aggressive program analysis and cloning, potentially creating multiple statically-typed instances of the generic code to satisfy the cases that the user's program requires. In particular, these generic programming features are implemented to avoid introducing unnecessary runtime overhead.

Latent types In most cases, the declared type of a variable or argument can be elided, in which case the compiler will infer the type from the program's context. If a variable's type is elided, it is typically inferred by inspecting its initializing expression. If it is a class member, its initialization within the class constructor is analyzed to determine its type. Function argument types are inferred by examining the actual arguments at the callsites for that function. If the inferred type of a variable or argument is unclear, the compiler will declare an ambiguity and request additional information from the user rather than making an arbitrary choice that may be surprising. This is a performance-motivated policy, in order to avoid storing dynamically tagged values to perform semantic checks and dispatches at run-time.

As an example, consider the code in Figure 5a in which a function is written to return the result of the + operator applied to its two arguments. Note that the formal arguments, the function's return type, and the type of the local variable are all elided. In this case, the available callsites will be analyzed to determine the types of the arguments, while the result value of the + operator will determine the type of the local variable, and therefore the function's return value. Since the function is invoked using integers and strings, two versions of the function will be created by the compiler, one that adds integers and the second which concatenates strings.

Type variables For programmers who prefer a more structured approach to generic programming, function arguments and classes may contain *type variables* that can be used to explicitly parameterize variable types. For example, Figure 5b contains the same function as before, but with an initial type variable argument describing a type *t*. This type is then used to declare the types of argu-

```
function addem(x, y): {
  var result = x + y;
  return result;
}

...addem(1, 2)...;
...addem("hi", "ya")...;
```

(a)

```
function addem(type t, x: t,
               y: t): t {

  var result: t = x + y;
  return result;
}

...addem(int, 1, 2)...;
...addem(string, "hi", "ya")...;
```

(b)

```
function addem(x: ?t, y: t)
               : t {
  var result: t = x + y;
  return result;
}

...addem(1, 2)...;
...addem("hi", "ya")...;
```

(c)

Fig. 5 Chapel functions showing a generic addition function in which (a) the types are elided, (b) a type variable *t* is used, (c) a type *t* is set using the query syntax (?*t*) and then used.

ments *x* and *y*, as well as the function's return type and local variable. Note that this code differs slightly from the previous version since it constrains *x* and *y* to have the same type, whereas the previous version permitted them to have different types (so long as the + operator supported them). Apart from this distinction, the two versions would generate identical code.

Type variables may also be defined using a stylized declaration syntax, using the "?" operator to mnemonically query the type of an argument and bind it to a type variable. Using this syntax, our generic function would appear as in Figure 5c and is semantically identical to the previous case.

Users have the ability to express additional constraints on the relationships between type variables via a *where clause* which can follow a type variable's declaration. We refer the reader to the language specification for more information on this feature (Cray Inc. 2005).

The features in this section are designed to help with Chapel's separation of algorithms from data structures. They are also designed to narrow the gap between parallel languages and mainstream languages. In particular, Chapel's type variables support generic programming in a manner similar to C++, Java, and C#, yet using a more intuitive syntax. Chapel's latent types are designed to emulate mainstream scripting languages that tend to support untyped variables. Our approach differs from most scripting languages in that Chapel's latently-typed variables must be declared and will only have a single static type during their lifetime. These choices were made to prevent common errors and to ensure that performance is not compromised by dynamic changes to a variable's type.

4.6 Other Chapel Language Features

Chapel has several other features that are not covered in this article because of lack of space. Primary among these are its support for iterators, similar to those in CLU (Liskov et al. 1977); curried function calls; and modules for namespace management.

4.7 Chapel's Execution Model Transparency

In any language design, there exists a tension between programmability and execution model transparency. A simple example of this in parallel language design can be seen in specifying assignments between distributed arrays.

In a language geared toward programmability, two arrays would likely be assigned using a single mechanism regardless of whether or not they shared the same distribution. In such a model, the simplicity of performing array assignment is favored over the desire to make the anticipated differences in performance visible in the source text. The programmer would therefore need to rely on feedback from the compiler or performance analysis tools (or their own comprehension of the execution model) to understand the difference in performance between the two cases.

In a language geared toward execution model transparency, the assignment between two aligned arrays would likely take a different—and simpler— form than the assignment between two arrays with different distributions. Here, the programmer must do more work, and potentially clone code manually to handle multiple cases. The advantage is that the program's syntax keeps the execution model clear to the programmer and compiler.

In Chapel's design, we have chosen to emphasize programmability over execution model transparency, in part to maximize genericity of the user's code and in part because of a belief that the 90/10 rule suggests that most of the time users want to do what's easiest and not worry about performance.

As the project has evolved, we have considered whether it might be useful and possible to statically switch between two semantic models in order to support different

Table 1
The ten design principles from Section 2.2 and how Chapel addresses them as described in Section 4

Desiderata	Chapel Response
Global view of parallelism	supported via: distributed domains and arrays; cobegin/begin statements; forall loops; iterators
General parallelism	data and task parallelism supported; cobegin statements and forall loops support composability; synchronization variables and atomic sections for task cooperation
Separation of algorithm and implementation	supported via: domains; user-defined distributions; multidimensional iterators; OOP features
Broad-market language features	support for: OOP; function and operator overloading; garbage collection; latent types; modules; generic programming
Data abstractions	support for: sequences; arithmetic (dense, strided, sparse) arrays; indefinite and opaque arrays; OOP
Performance	achieved by locality tuning via: domain distributions; "on" clause
Execution model transparency	currently only supported in the program text via "on" clauses; may add support for switching to stricter semantic models in the future
Portability	assumes abstract machine model supporting multithreading and single-sided communication; if not supported by hardware, will have to simulate in software at some cost
Interoperability	anticipate supporting via Babel and domain distributions
Bells and Whistles	implementation-dependent/TBD

degrees of execution transparency. For example, one could imagine that for the 10% of a code where programmers really care about performance, they might use a special scoping mechanism or keyword to switch into a stricter semantic model (such as ZPL's WYSIWYG model), allowing them to keep a better grasp on the execution model. Such a mechanism would also support Chapel's goal of allowing programmers to write code quickly and then tune it incrementally. This idea is still under study.

4.8 Chapel Portability

Chapel was designed with an idealized, productive parallel architecture in mind—in particular, one with a global shared address space, hardware support for multithreading, a high-bandwidth, low-latency network, and latency tolerant processors. This choice was made believing that to best improve productivity in a parallel language, one should not assume a virtual architecture that represents an unproductive least-common denominator for existing machines.

One obvious result of this decision is that when compiling Chapel for a less-ideal architecture, certain features are likely not to work as well. For example, on an architecture that lacks hardware support for a shared address space or multithreading, one would expect there to be greater overheads in software to make up for these deficiencies. Similarly, on an architecture with a lesser network or with no latency tolerance, one would expect the impact of communication on a program's performance to be greater.

Our guiding principle here has been to design Chapel features such that they can be implemented on various architectures, though possibly at some cost. We are working on implementing Chapel so that when programmers write code that matches their current programming model on such architectures, the performance is comparable to what they would achieve today with that model. If, on the other hand, they use more advanced features on such architectures for the sake of programmability, they should expect to take a performance hit due to the paradigm mismatch.

4.9 Chapel Interoperability with Other Languages

Chapel's plan for interoperability with other languages is to implement support for calling between Chapel and C. We then intend to use the Babel tool for scientific language interoperation (Dahlgren 2006) in order to interface with Fortran, C++, Python, Java, and any other languages that it supports at that time. In this way we should be able to interoperate with most standard languages that are used by the HEC community.

With respect to the second aspect of interoperability—that of operating on another parallel language's data in-place—our plan is to use Chapel's domain distribution capabilities to address this challenge. As an example, to interoperate with an existing MPI code, the user would need to use a Chapel distribution to describe the layout of their MPI program's data in memory in order to access that data in-place. This could either be a custom distribution written by the user, or for simple decompositions (e.g. a simple blocked array in MPI) it could be a distribution from Chapel's standard library. As with Chapel distributions in general, this remains an area of active research.

5 Summary

Table 1 lists our ten design principles for productive parallel languages from Section 2.2 and summarizes the discussion from Section 4 to describe how Chapel addresses each area. Since we created both the wishlist of features and the Chapel language itself, it should come as no surprise that Chapel addresses most of our areas of concern fairly well. The weakest area is in the realm of execution model transparency, where we have chosen to emphasize programmability over performance transparency. We also have some work to do in constructing a proof-of-concept portable implementation of Chapel, and to flesh out our language interoperability story, particularly in the area of domain distributions. This is work that is currently in progress.

In the coming years, Chapel will be evaluated by the HPCS funding agencies, academics, other HPCS vendors, and members of the HEC user community. The goal is to work toward constructing a common set of features that the HEC vendors and community would be interested in supporting as a consortium language effort. During this period of evaluation and experimentation, it will be interesting to see how our design goals and preferences compare to those of the broader community once users have the chance to investigate Chapel more closely. We encourage anyone who is interested in parallel language design to explore chapel at our website (http://chapel.cs.washington.edu) and to send us feedback on where you believe we have made better and worse decisions (chapel_info@cray.com). Language design clearly involves making tradeoffs according to priorities and taste, and we are interested in hearing how your tastes compare with ours.

Acknowledgments

The authors would like to acknowledge John Plevyak, Steven Deitz, Roxana Diaconescu, Shannon Hoffswell, Mark James, Mackale Joyner, and Wayne Wong for their many contributions to the Chapel language and its ongoing implementation. Thanks also to Robert Bocchino, Paul Cassella, and Greg Titus for reading and providing valuable feedback on drafts of this article. This material is based upon work supported by the Defense Advanced Research Projects Agency under its Contract No. NBCH3039003. The research described in this paper was partially carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

Author Biographies

Bradford Chamberlain is a principal engineer at Cray Inc., where he works on parallel programming models, particularly on the design and implementation of the Chapel parallel language. Before coming to Cray in 2002, he spent time at a start-up working at the opposite end of the hardware spectrum by designing SilverC, a parallel language for reconfigurable embedded hardware. Brad received his Ph.D. in computer science from the University of Washington in 2001 where his work focused on the design and implementation of the region concept for the ZPL parallel array language. While there, he also dabbled in algorithms for accelerating the rendering of complex 3D scenes. Brad remains associated with the University of Washington as an affiliate faculty member. He received his Bachelor's degree in computer science from Stanford University in 1992.

David Callahan is a distinguished engineer at Microsoft where he works on concurrency related issues as part of the Visual Studio suite of developer tools. Prior to Microsoft, he worked for many years at Cray Inc. and Tera Computer Company developing language extensions, compilers, runtime support and tools for developing scalable parallel programs for high-performance computing. He is a graduate of Rice University and his interests include optimizations compilers, parallel languages, parallel algorithms, and hardware architecture. He is a member of the ACM.

Hans P. Zima is a Professor of Applied Computer Science at the University of Vienna, Austria, and a Principal

Scientist at the Jet Propulsion Laboratory in Pasadena. His major research interests have been in the fields of high-level programming languages, compilers, and advanced software tools. In the early 1970s he designed and implemented one of the first high-level real-time languages for the German Air Traffic Control Agency. During his tenure as a Professor of Computer Science at the University of Bonn, Germany, he contributed to the German supercomputer project "SUPRENUM", leading the design of the first Fortran-based compilation system for distributed-memory architectures (1989). After his move to the University of Vienna, he became the chief designer of the Vienna Fortran language (1992) that provided a major input for the High Performance Fortran de-facto standard. Since 1997, Dr. Zima has been heading the Priority Research Program "Aurora", a ten-year program funded by the Austrian Science Foundation. His research over the past four years focused on the design of the Chapel programming language in the framework of the DARPA-sponsored HPCS project "Cascade".

Notes

1 Note that we use the term "parallel language" loosely in this paper, since approaches like MPI and SHMEM are actually library-based approaches. In defense of our laxness, consider the impact of such libraries on the execution models of the base languages with which they are used—particularly the assumption that multiple program instances will be executing simultaneously. Compared with the relative unobtrusiveness of traditional libraries on a language's programming model, our casual use of the term "language" is not completely unwarranted. If it helps, think of "language" in terms of "a means of expressing something" rather than strictly as a "programming language."

2 Note that a language or library may support a global view of computation and yet be implemented using SPMD or fragmented techniques. Here, we use the terms to characterize the programmer's model of writing computation, not the program's execution model.

3 Note that this is a completely distinct concept from Titanium's region, mentioned earlier.

References

Adl-Tabatabai, A.-R., Lewis, B. T., Menon, V., Murhpy, B. R., Saha, B., and Shpeisman, T. (2006). Compiler and runtime support for efficient software transactional memory, in *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 26–37, ACM Press, New York, NY.

Albert, E., Knobe, K., Lukas, J. D., and Steele, Jr., G. L. (1988). Compiling Fortran 8x array features for the Connection Machine computer system, in *PPEALS '88: Proceedings of the ACM/SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages, and Systems*, pp. 42–56, ACM Press, New York, NY.

Alverson, G., Kahan, S., Korry, R., McCann, C., and Smith, B. (1995). Scheduling on the Tera MTA, in D. G. Feitelson, and L. Rudolph, editors, *Job scheduling strategies for parallel processing*, vol. 949 of Lecture Notes in Computer Science, pp. 19–44, Berlin: Springer Verlag.

Barriuso, R. and Knies, A. (1994). SHMEM user's guide for C, Technical Report, Cray Research Inc.

Blumofe, R. D. and Leiserson, C. E. (1994). Scheduling multithreaded computations by work stealing, in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS `94)*, pp. 356–368, New Mexico.

Bonachea, D. (2002). GASNet specification v1.1, Technical Report UCB/CSD-02-1207, U.C. Berkeley. (newer versions also available at http://gasnet.cs.berkeley.edu)

Carlson, B., El-Ghazawi, T., Numerich, R., and Yelick, K. (2003). Programming in the partitioned global address space model, Tutorial at Supercomputing 2003. (notes available at http://upc.gwu.edu)

Carlson, W. W., Draper, J. M., Culler, D. E., Yelick, K., Brooks, E., and Warren, K. (1999). Introduction to UPC and language specification, Technical Report CCS-TR-99-157, Center for Computing Sciences, Bowie, MD.

Chamberlain, B. L. (2001). *The design and implementation of a region-based parallel language*, Ph.D. thesis, University of Washington.

Chamberlain, B. L., Choi, S.-E., Lewis, E. C., Lin, C., Snyder, L., and Weathersby, W. D. (1998). ZPL's WYSIWYG performance model, in *Proceedings of the Third International Workshop on High-Level Programming Models and Supportive Environments*, pp. 50–61, IEEE Computer Society Press.

Chamberlain, B. L., Lewis, E. C., Lin, C., and Snyder, L. (1999). Regions: An abstraction for expressing array computation, in *ACM/SIGAPL International Conference on Array Programming Languages*, pp. 41–49.

Chamberlain, B. L. and Snyder, L. (2001). Array language support for parallel sparse computation, in *Proceedings of the 2001 International Conference on Supercomputing*, pp. 133–145, ACM SIGARCH.

Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J. (2000). *Parallel programming in OpenMP*, Morgan Kaufmann, San Francisco, CA.

Chapman, B. M., Mehrotra, P., and Zima, H. P. (1992). Programming in Vienna Fortran, *Scientific Programming*, **1**(1): 31–50.

Chen, W.-Y., Bonachea, D., Duell, J., Husbands, P., Iancu, C., and Yelick, K. (2003). A performance analysis of the Berkeley UPC compiler, in *Proceedings of the International Conference of Supercomputing (ICS)*, San Francisco, CA.

Chen, W.-Y., Iancu, C., and Yelick, K. (2005). Communication optimizations for fine-grained UPC applications, in *14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, St. Louis, MO.

Cray Inc. (2005). *Chapel specification*, Cray Inc., Seattle, WA, 0.4 edition. (http://chapel.cs.washington.edu).

Dagum, L. and Menon, R. (1998). OpenMP: An industry-standard API for shared-memory programming, *IEEE Computational Science and Engineering*, **5**(1): 46–55.

Dahlgren, T., Epperly, T., Kumfert, G., and Leek, J. (2006). *Babel users' guide*, Lawrence Livermore National Laboratory, 0.11.0 edition.

Deitz, S. J. (2005). *High-Level programming language abstractions for advanced and dynamic parallel computations*, Ph.D. thesis, University of Washington.

Deitz, S. J., Callahan, D., Chamberlain, B. L., and Snyder, L. (2006). Global-view abstractions for user-defined reductions and scans, in *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 40–47, ACM Press, New York, NY.

Diaconescu, R. and Zima, H. P. (2007). An approach to data distributions in Chapel, *International Journal of High Performance Computing Applications*, **21**(3): 313–334.

Dotsenko, Y., Coarfa, C., and Mellor-Crummey, J. (2004). A multi-platform Co-Array Fortran compiler, in *Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques (PACT 2004)*, Antibes Juan-les-Pins, France.

El-Ghazawi, T., Carlson, W., Sterling, T., and Yelick, K. (2005). *UPC: Distributed shared-memory programming*, Wiley-Interscience, Hobohen, NJ.

Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C.-W., and Wu, M.-Y. (1990). Fortran D language specification, Technical Report CRPC-TR 90079, Rice University, Center for Research on Parallel Computation.

Fraser, K. (2003). *Practical lock-freedom*, Ph.D. thesis, King's College, University of Cambridge.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994a). PVM 3 user's guide and reference manual, Technical Report ORNL/TM-12187, Oak Ridge National Laboratory.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994b). *PVM: Parallel virtual machine, a user's guide and tutorial for networked parallel computing*, Scientific and Engineering Computation, Cambridge, MA: MIT Press.

Gilbert, J. R., Moler, C., and Schreiber, R. (1992). Sparse matrices in MATLAB: Design and implementation, *SIMAX*, **13**(1): 333–356.

Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., and Snir, M. (1998). *MPI: The complete reference*, volume 2, Scientific and Engineering Computation, Cambridge, MA: MIT Press.

Harris, T., Marlow, S., Jones, S. P., and Herlihy, M. (2005). Composable memory transactions, in *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Chicago, IC.

Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. (2006). Optimizing memory transactions, in *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 14–25, ACM Press, New York, NY.

High Performance Fortran Forum (1993). High Performance Fortran language specification, *Scientific Programming*, **2**(1–2): 1–170.

High Performance Fortran Forum (1997). *High Performance Fortran Language Specification Version 2.0.*

Hilfinger, P. N., Bonachea, D. O., Datta, K., Gay, D., Graham, S. L., Liblit, B. R., Pike, G., Su, J. Z., and Yelick, K. A. (2005). Titanium language reference manual, Technical Report UCB/EECS-2005-15, Electrical Engineering and Computer Sciences, University of California at Berkeley.

Joerg, C. F. (1996). *The Cilk system for parallel multithreaded computing*, Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science.

Koelbel, C. H., Loveman, D. B., Schreiber, R. S., Guy L. Steele, Jr., and Zosel, M. E. (1996). *The High Performance Fortran handbook*, Scientific and Engineering Computation, Cambridge, MA: MIT Press.

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. (1977). Abstraction mechanisms in CLU, *ACM SIGPLAN*, **12**(7): 75–81.

Message Passing Interface Forum (1994). MPI: A message passing interface standard, *International Journal of Supercomputing Applications*, **8**(3/4): 169–416.

Message Passing Interface Forum (1997). *MPI-2: Extensions to the message-passing interface*.

Ngo, T. A., Snyder, L., and Chamberlain, B. L. (1997). Portable performance of data parallel languages, in *Proceedings of SC97: High Performance Networking and Computing*, San Jose, CA.

Nieplocha, J. and Carpenter, B. (1999). ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems, in J. Rolim et al., editors, *Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming*, Lecture Notes in Computer Science vol. 1586, pp. 533–546, San Juan, Puerto Rico, Berlin: Springer Verlag.

Numerich, R. W. and Reid, J. (1998). Co-array Fortran for parallel programming, *SIGPLAN Fortran Forum*, **17**(2): 1–31.

Numerich, R. W. and Reid, J. (2005). Co-arrays in the next Fortran standard, *SIGPLAN Fortran Forum*, **24**(2): 4–17.

Parzyszek, K., Nieplocha, J., and Kendall, R. A. (2000). A generalized portable SHMEM library for high performance computing, in M. Guizani, and Z. Shen, editors, *Twelfth IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 401–406, Las Vegas, Nevada.

Randall, K. H. (1998). *Cilk: Efficient multithreaded computing*, Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. (1998). *MPI: The complete reference*, volume 1, Scientific and Engineering Computation, 2nd edition, Cambridge, MA: MIT Press.

Snyder, L. (1999). *The ZPL programmer's guide*, Scientific and Engineering Computation, Cambridge, MA: MIT Press.

Su, J. and Yelick, K. (2005). Automatic support for irregular computations in a high-level language, in *19th International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO.

Supercomputing Technologies Group (2001). *Cilk 5.3.2 Reference manual*, MIT Laboratory for Computer Science.

Ujaldon, M., Zapata, E. L., Chapman, B. M., and Zima, H. P. (1997). Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation, *IEEE Transactions on Parallel and Distributed Systems*, **8**(10).

UPC Consortium (2005). *UPC Language specification (v 1.2)*. (available at http://upc.gwu.edu)

von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. (1992). Active Messages: A mechanism for integrated communication and computation, in *19th International Symposium on Computer Architecture*, Gold Coast, Australia, pp. 256–266.

Yelick, K., Semenzato, L., Pike, G., Miyamato, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. (1998). Titanium: A high-performance Java dialect, *Concurrency: Practice and Experience*, **10**(11–13): 825–836.

Zima, H., Brezany, P., Chapman, B., Mehrotra, P., and Schwald, A. (1992). Vienna Fortran—a language specification version 1.1, Technical Report NASA-CR-189629/ICASE-IR-21, Institute for Computer Applications in Science and Engineering.