Global HPCC Benchmarks in Chapel: STREAM Triad, Random Access, and FFT

(Revision 1.5 — December 2007 Release)

Bradford L. Chamberlain, Steven J. Deitz, Mary Beth Hribar, Wayne A. Wong
Chapel Team, Cray Inc.
chapel_info@cray.com

***Abstract*—Chapel is a new parallel programming language being developed by Cray Inc. as part of its participation in DARPA's High Productivity Computing Systems program. In this article, we describe Chapel implementations of the global HPC Challenge (HPCC) benchmarks for the STREAM Triad, Random Access, and FFT computations. The Chapel implementations use 5–13× fewer lines of code than the reference implementations supplied by the HPCC team. All codes in this article compile and run with the current version of the Chapel compiler. We provide an introduction to Chapel, highlight key features used in our HPCC implementations, and discuss our plans and challenges for obtaining performance for these codes as our compiler development progresses. The full codes are listed in appendices to this article.**

I. Introduction

Chapel is a new parallel programming language being developed by Cray Inc. as part of its participation in DARPA's High Productivity Computing Systems program (HPCS). The Chapel team is working to design, implement, and demonstrate a language that improves parallel programmability, portability, and code robustness as compared to current practice while producing programs whose performance is comparable to or better than MPI.

In this article, we present Chapel implementations of three of the global HPC Challenge (HPCC) benchmarks—STREAM Triad, Random Access (RA), and Fast Fourier Transform (FFT). These benchmarks were developed as a means of measuring sustained parallel performance by stressing traditional system bottlenecks. We describe our Chapel implementations of the benchmarks, highlighting key language concepts used to implement them. We submitted an earlier version of these codes to the second annual 2006 HPC Challenge competition for "most productive" implementations, judged 50% on elegance and 50% on performance. We were selected as one of six finalists to present at the HPCC BOF at SC06.

The codes presented in this article compile and execute with our current Chapel compiler (version 0.5). To date, our compiler development effort has focused on providing a single-node prototype of Chapel's features in order to support experimentation with the language and generate feedback on its specification. While our language specification and compiler architecture have both been designed with distributed-memory execution and performance optimizations in mind, minimal development effort has been invested toward these activities to date. As a result, this article does not contain performance results since they would not reflect our team's emphasis. Beginning in 2007, we have focused our implementation effort on serial performance optimizations and distributed-memory execution, and we expect to have publishable performance results for our codes within the coming year.

Though this article does not report on performance, we did write our benchmark implementations to be portable, performance-minded parallel codes rather than simply optimizing for elegance, clarity, and size. Our goal was to write versions of the benchmarks that would serve as clear reference implementations for future HPCC competitors while also producing codes that should result in good performance as our compiler matures. In lieu of performance results, this article contains a discussion of the top performance-related issues for each benchmark and our strategies for achieving good performance in the future.

The rest of this paper is organized as follows: In Section II we give an overview of our results. In Section III we provide an introduction to Chapel's motivating themes. In Section IV we describe the coding conventions that we adopted in writing these benchmarks. Sections V, VI, and VII each describe one of the three benchmarks, providing a high-level overview of our approach in Chapel as well as performance-related issues that highlight Chapel's benefits and challenges. Finally in Section VIII, we summarize and provide a brief status report for the Chapel project. Our complete source listings are provided in the appendices.

II. Overview of Results

Table I categorizes and counts the number of lines of code utilized by our HPCC implementations. The line counts for each benchmark are represented in a column of the table. The fourth data column represents the common *HPCCProblemSize* module that is shared by the benchmarks to compute and print the problem size. For the Random Access benchmark, each entry is expressed as a sum—the first value represents the benchmark module itself, the second represents an additional module used to generate the random number stream, and the final value is the sum for both modules.

The rows of the table are used to group the lines of code into various categories and running totals. The first two rows

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-001.

TABLE I
LINE COUNTS AND CLASSIFICATIONS FOR HPCC CODES IN CHAPEL

	Benchmark Code			
line count type	*STREAM Triad*	*Random Access*	*FFT*	*Common*
Kernel computation	1	4 + 19 = 23	52	0
Kernel declarations	9	13 + 21 = 34	27	19
Total kernel	*10*	*17 + 40 = 57*	*79*	*19*
Initialization	8	1 + 0 = 1	20	0
Verification	6	15 + 0 = 15	13	0
Results and Output	28	18 + 0 = 18	15	12
Total Benchmark	***52***	***51 + 40 = 91***	***127***	***31***
Debug and Test	9	6 + 0 = 6	8	3
Blank	27	26 + 12 = 38	53	9
Total Program	*88*	*83 + 52 = 135*	*188*	*43*

indicate the number of lines required to express the kernel of the computation and its supporting declarations, respectively. For example, in the STREAM Triad benchmark, writing the computation takes a single line of code, while its supporting variable and subroutine declarations require 9 lines of code. The next row presents the sum of these values to indicate the total number of lines required to express the kernel computation.

The next three rows of the table count lines of code related to setup, verification, and tear-down for the benchmark. *Initialization* indicates the number of lines devoted to initializing the problem's data set, *Verification* counts lines used to check that the computed results are correct, and *Results and Output* gives the number of lines for computing and outputting results for timing and performance. These three rows are then combined with the previous subtotal in the next row to indicate the number of source lines used to implement the benchmark and output its results. This subtotal should be interpreted as the SLOC (*Source Lines of Code*) count for the benchmark as specified.

The *Debug and Test* row indicates the number of lines added to make the codes more useful in our nightly regression testing system, while the *Blank* row indicates the number of blank lines. These values are added to the previous subtotal to give the total number of lines in each program, and they are provided to serve as a checksum against the line number labels that appear in the appendices.

Table II compares the total SLOC for our Chapel codes with the standard HPCC reference implementations. The Chapel result for each code is obtained by summing its *Total Benchmark* result from Table I with that of the common module. The reference results are the sum of the *Framework* and *Parallel* numbers reported in the table from the HPCC website's FAQ.[1]

This table shows that our Chapel codes are approximately 5–13× smaller than the reference implementations. While shorter codes are not necessarily better or easier to understand, we believe that our Chapel implementations are not only succinct, but also clear representations of the benchmarks that will perform well as our compiler matures. The rest of this paper examines the codes qualitatively to complement the quantitative results in this section.

[1]http://www.hpcchallenge.org/faq/index.html

TABLE II
SLOC COMPARISON BETWEEN REFERENCE AND CHAPEL HPCC CODES

	Benchmark Code		
	STREAM Triad	*Random Access*	*FFT*
SLOC for Reference Implementation	433	1668	1406
SLOC for Chapel Implementation	83	122	158
SLOC Ratio	*5.21*	*13.67*	*8.89*

III. CHAPEL'S MOTIVATING THEMES

One of Chapel's primary themes is to support general parallel programming using high-level abstractions. Chapel does this through its support for a *global-view programming model* that raises the level of abstraction for both data structures and control flow as compared to parallel programming models currently used in production.

Global-view data structures are arrays and other data aggregates whose size and indices are expressed globally in spite of the fact that their implementations may be distributed across the memories of multiple nodes or *locales*.[2] This contrasts with most parallel languages used in practice, which require users to partition distributed data aggregates into per-processor chunks, either manually or using language abstractions. HPF and ZPL are two other recent parallel languages that support global-view data structures [15], [6], though in a more restricted form than Chapel.

A global view of control means that a user's program commences execution with a single logical thread of control and that additional parallelism is introduced through the use of specific language concepts. All parallelism in Chapel is implemented via multithreading, though these threads are created via high-level language concepts and managed by the compiler and runtime, rather than through explicit fork/join-style programming. An impact of this approach is that Chapel can express parallelism that is more general than the Single Program, Multiple Data (SPMD) model that today's most common parallel programming approaches use as the basis for their programming and execution models. Examples include Co-Array Fortran, Unified Parallel C (UPC), Titanium, HPF, ZPL, SHMEM, and typical uses of MPI [18], [11], [20], [15], [6], [2], [19], [12]. Our multithreaded execution model is perhaps most similar to that which is supported by the Cilk language or the Cray MTA's runtime libraries [13], [1]. Moreover, Chapel's general support for parallelism does not preclude the user from coding in an SPMD style if they wish.

Supporting general parallel programming also means targeting a broad range of parallel architectures. Chapel is designed to target a wide spectrum of HPC hardware including clusters of commodity processors and SMPs; vector, multithreading, and multicore processors; distributed memory, shared address space, and shared memory architectures; networks of any topology; and custom vendor architectures. Our portability

[2]A *locale* in Chapel is a unit of the target architecture that supports computation and data storage. Locales are defined for each architecture such that a locale's threads will all have similar access times to any specific memory address. For commodity clusters, each of their (single-core) processors, multicore processors, or SMP nodes would be considered a locale.

goal is to have any legal Chapel program run correctly on all of these architectures, and for Chapel programs that express parallelism in an architecturally-neutral way to perform reasonably on all of them. Naturally, Chapel programmers can tune their codes to more closely match a particular machine's characteristics, though doing so may cause the program to be a poor match for other architectures. In this article we present codes written in an architecturally-neutral manner and note places where they could be tuned to better match specific architectural characteristics, including memory models and network capabilities.

A second theme in Chapel is to allow the user to optionally and incrementally specify where data and computation should be placed on the physical machine. We consider this control over program locality to be essential for achieving scalable performance on large machine sizes given current architectural trends. Such control contrasts with shared-memory programming models like OpenMP [7] which present the user with a flat memory model. It also contrasts with SPMD-based programming models in which such details are explicitly specified by the programmer on a process-by-process basis via the multiple cooperating program instances.

A third theme in Chapel is support for object-oriented programming (OOP), which has been instrumental in raising productivity in the mainstream programming community. Chapel supports traditional reference-based classes as well as value classes. Programmers are not required to use an object-oriented style in their code, so that traditional Fortran and C programmers need not adopt a new programming paradigm in order to use Chapel effectively.[3]

Chapel's fourth theme is support for generic programming and polymorphism, allowing code to be written in a style that is generic across types and thereby applicable to variables of various types, sizes, and precisions. The goal of these features is to support exploratory programming as in popular interpreted and scripting languages, and to support code reuse by allowing algorithms to be expressed without explicitly replicating them for each possible type. This flexibility at the source level is implemented by having the compiler create versions of the code for each required type signature rather than by relying on dynamic typing which would incur unacceptable runtime overheads.

Chapel's first two themes are designed to provide support for general, performance-oriented parallel programming through high-level abstractions. The second two themes are supported to help narrow the gulf that exists between parallel programming languages and popular mainstream programming or scripting languages. The benchmarks in this paper illustrate Chapel's support for global-view programming, for locality control, and for generic programming. Due to the relatively straightforward style of parallel computation required for these benchmarks, this article does not demonstrate many of Chapel's features for more general parallelism and locality control. We also chose not to utilize OOP concepts in our benchmark codes since we do not believe that the introduction of objects would greatly improve the benchmarks' clarity, generality, organization, or performance.

For a more complete introduction to Chapel, the reader is referred to our project's website[4], overview publications, and draft language specification [8], [5], [3].

IV. Coding Conventions

In writing these codes, we used the HPC Challenge Class 2 Official Specification as our primary guide for defining the computations [10]. We studied and benefited from the HPCC reference implementations as well as the 2005 finalist codes, but typically chose to express the benchmarks in our own style rather than trying to mimic pre-existing implementations.

In particular, we chose names for our variables and subroutines that we found descriptive and appealing rather than trying to adhere to the naming conventions of previous implementations. The primary exception to this rule is for variables named in the written specification, such as m, n, and N_U. For these variables, we adopted the specification's names in order to clarify the ties between our code and the official description.

Several concerns directed our coding design (in roughly this order):

- faithfulness to the written specification
- ability to generate code with good performance
- clarity and elegance of the solution, emphasizing readability over minimization of code size
- appropriate and demonstrative uses of Chapel features
- implementations that are generic with respect to types and problem parameters
- support for execution-time control over key program parameters like problem sizes
- ability to be tested in our nightly regression suite

Some of these motivations, particularly the last three, cause our programs to be slightly more general than required by the written specification. However, we believe that they also result in more interesting and realistic application codes.

Structurally, we tried to keep the timed kernel of the computation in the program's `main()` procedure, moving other functionality such as initialization, verification, and I/O into separate routines. In the Random Access and FFT benchmarks, we also abstracted kernels of the computation into helper iterators and routines to improve abstraction and reuse.

Stylistically, we tend to use mixed-case names to express multi-word identifiers, rather than underscores. We typically use an initial lower-case letter when naming procedures and non-distributed variables, while domains, distributed arrays, and class instances start with an upper-case letter.

In our code listings, **boldface** text is used to indicate Chapel's reserved words and standard types, while "..." represents code that is elided in an excerpt for brevity.

As mentioned previously, we approached these codes as we would for a large-scale parallel machine—thus they contain distributed data structures, parallel loops, and array-based parallelism. Since our current compiler only supports execution

[3]Note that many of Chapel's standard library capabilities are implemented using objects, so these may require Fortran and C programmers to utilize a method-invocation style of syntax when using them. However, such use does not necessitate broader adoption of OOP methodologies.

[4]http://chapel.cs.washington.edu

```
const ProblemSpace: domain(1, int(64)) distributed(Block) = [1..m];
var A, B, C: [ProblemSpace] elemType;

A = B + alpha * C;
```

Fig. 1. STREAM Triad Kernel in Chapel

on a single locale, these constructs will necessarily fall into the degenerate cases of allocating data structures on a single locale and, for the single-core processors we use, executing parallel loops and array statements using a single thread.

V. STREAM TRIAD

The STREAM Triad benchmark asks the programmer to take two vectors of random 64-bit floating-point values, b and c, and to use them to compute $a = b + \alpha \cdot c$ for a given scalar value α. As with all of the HPCC benchmarks, the problem size for the vectors must be chosen such that they consume $1/4 - 1/2$ of the system memory. STREAM Triad is designed to stress local memory bandwidth since the vectors may be allocated in an aligned manner such that no communication is required to perform the computation.

A. Overview of STREAM Triad in Chapel

Our approach to the STREAM Triad benchmark is summarized by the lines of code in Figure 1. This code excerpt presupposes the definition of two named values, *m* defining the problem size and *alpha* defining the scalar multiplication value for the Triad computation (α). It also refers to a named type, *elemType*, that represents the element type to be stored in the vectors.

The first line declares a constant named *ProblemSpace* that is defined to be a *domain*—a first-class representation of an index set, potentially distributed across the memories of multiple locales. In this instance, the index set is declared to be 1-dimensional and to describe the indices $\{1, 2, \ldots, m\}$ using 64-bit integers. *ProblemSpace*'s definition also specifies that it should be distributed across the user's locale set using the "Block" distribution. Such distributions specify how a domain's indices should be mapped to a set of locales and how they should be represented within each locale's memory. In this case, the *Block* distribution is a standard Chapel distribution that assigns contiguous blocks of indices to each locale.

The next line uses the *ProblemSpace* domain to declare three arrays—*A*, *B*, and *C*—used to represent the vectors, a, b, and c from the written specification. The domain's index set defines the size and shape of these arrays, and its distribution specifies the arrays' distributed implementation across the locales and within their local memories. The identifier *elemType* specifies the type of each array element (defined to be a 64-bit floating-point value in our implementation).

The final line expresses the computation itself, using whole-array syntax to specify the elementwise multiplications, additions, and assignments needed to perform the Triad computation. Whole-array operations like this one are implicitly parallel and each locale will perform the operations for the array elements that it owns, as defined by *ProblemSpace*'s distribution (since that was the domain used to define all three arrays).

B. STREAM Triad Performance Notes

Once we have a distributed-memory implementation of Chapel and some additional scalar optimizations, we expect our STREAM Triad implementation to perform as well as the target architecture allows—constrained only by the rate of a locale's local memory accesses. The timed Triad computation is embarrassingly parallel, and its expression in Chapel makes this obvious through the use of whole-array operations on arrays declared using a single domain. This demonstrates one of the many advantages of domains, which is that they often provide the compiler with valuable information about the relative alignment of arrays in distributed memory [4].

Our main implementation challenge in obtaining good performance relates to the definition of the *Block* distribution itself. While our team has extensive experience implementing block distributions whose performance competes with or outperforms hand-coded benchmarks [4], [9], a major goal for Chapel is to write all of its standard distributions using the same mechanism that advanced Chapel users would use to author their own distributions. This contrasts with prior language work in which compilers typically have special knowledge of standard distributions. Our challenge is to design a distribution interface that allows the compiler to implement computations using standard distributions without compromising performance. If we fail to meet this challenge, we can always fall back on the approach of embedding knowledge of standard distributions into the compiler; however, this would be unfortunate because it suggests that users may face a performance penalty if they need to use a distribution that Chapel does not provide.

Our final performance-related note for STREAM Triad relates to the fact that Chapel uses a multithreaded execution model to implement its program semantics. Users who are accustomed to an SPMD execution model may worry that for embarrassingly parallel applications like STREAM Triad, the overhead of supporting a general multithreaded execution environment may be overkill since simpler SPMD execution is sufficient. While we believe that multithreaded execution is required in the next generation of parallel languages to support general parallel programming, we also recognize the efficiency advantages of the SPMD model due to its simplicity. For this reason, we expect to optimize our implementation for phases of computation in which statically scheduled threads suffice, either in the compiler-generated code or in its runtime support system.

Using STREAM Triad as a specific example, when the compiler analyzes the source code, it can trivially see that

```
  const TableSpace: domain(1, indexType) distributed(Block) = [0..m);
  var T: [TableSpace] elemType;

  const UpdateSpace: domain(1, indexType) distributed(Block) = [0..N_U);

  [i in TableSpace] T(i) = i;

  forall block in UpdateSpace.subBlocks do
    for r in RAStream(block) do
      T(r & indexMask) ^= r;

...

iterator RAStream(block) {
  var val = getNthRandom(block.low);
  for i in block {
    getNextRandom(val);
    yield val;
  }
}
```

Fig. 2. Random Access Kernel and Stream Iterator in Chapel

there is neither task parallelism nor explicit parallelism specified in the code. All parallelism comes from performing implicitly parallel operations on distributed arrays—whole-array operations, pseudo-random fills, and reductions. Given such information, the compiler may choose to generate code that schedules a single thread per core across the distributed machine, using them in an SPMD fashion. Recall that Chapel's global-view model of control specifies that the entry point is executed by a single *logical* thread, allowing an SPMD execution strategy to be used as long as the illusion of a single startup thread is maintained and the program's semantics allow it. Similarly, the compiler may use an SPMD execution strategy for a single phase of the user's program and more general techniques for other phases when it can prove such an approach is legal.

Alternatively, Chapel's runtime support for multithreading may be tuned so that when a single thread is running per core, the overheads required by more general multithreading capabilities are minimized. In the limit, this may allow multithreaded Chapel codes executing in an SPMD style to be competitive with explicit SPMD execution models. Even in this scenario it should be expected that compiler-generated information about the threading and communication requirements of a computation phase would help Chapel's runtime libraries shut down unnecessary sources of runtime overhead, such as services for creating remote threads or servicing one-sided communication requests on a commodity cluster.

VI. Random Access (RA)

The Random Access benchmark computes pseudo-random updates to a large distributed table of 64-bit integer values, T. Each table location T_i is initialized to store its index i in order to support verification of the computation afterwards. The pseudo-random values that are generated are used both to specify the table location to be updated and the value to use for the update. The updates themselves are performed by xor-ing the random value into the table location. Because multiple updates may attempt to modify the same table location simultaneously, the benchmark permits a certain fraction of such conflicting updates to be lost. Random Access is designed to test a system's ability to randomly access memory at a fine granularity. The benchmark permits updates to be performed in batches of a certain size as a concession to architectures and programming models that do not support fine-grain communication.

A. Overview of Random Access in Chapel

Figure 2 shows an excerpt from the Chapel implementation of the Random Access benchmark that summarizes the computation. It assumes the definition of three named values: *m* which defines the problem size, *N_U* which represents the number of updates (N_U), and *indexMask* which stores the bitmask used to create legal indices into the table. It also refers to two named types, *indexType* and *elemType*, used to represent the types that should be used to store the indices and elements of table T, respectively.

The first two lines define a domain and array used to represent the distributed table, T, that is randomly accessed by the benchmark. The next line defines a second domain, *UpdateSpace* that represents the distribution of the table update work across the locale set. Note that *UpdateSpace* is not used to allocate any arrays, only to distribute the computation space.

Both of the domain declarations in this excerpt specify their index type in terms of a named type, *indexType*. In this benchmark, we use a 64-bit unsigned integer as the index type to simplify the indexing operations using the unsigned 64-bit pseudo-random values. Note furthermore that the domain initializers use a slightly different domain specifier than in the STREAM Triad benchmark: [*lo*..*hi*). This domain syntax is intended to suggest open intervals in math, and defines the index set $\{lo, lo+1, \ldots, hi-1\}$.

The computation starts on the fourth line of code which initializes the table using a *forall expression*, representing the parallel loop: "for all indices i in $TableSpace$, assign T_i the value i."

The main loop in the code computes the random accesses using a nested loop which is defined by invoking two *iterators*.

In Chapel, an iterator is like a function, but rather than creating a single result via a `return`, it generates a stream of results via a `yield` keyword. Chapel's yield statement is like a traditional return except that the iterator's execution conceptually continues onwards from that point. Iterators are typically used to specify loops, and can be viewed as a concept for abstracting complex control flow away from loop headers just as functions are used to abstract code away from an invocation point. As an example, the complicated loop nest used to implement a multidimensional tiled iteration can be specified once using an iterator and then invoked multiple times to tile many loops in a clean, modular manner.

The outer loop used to compute the random updates invokes a standard iterator defined by the *Block* distribution to generate sub-blocks of work that can be performed in parallel. The number of sub-blocks created will be determined by the degree of parallelism supported by the target architecture. For example, on a machine with 16 locales of 4 cores each, 64 sub-blocks would be generated by the iterator, one for each core.

The inner loop uses a user-defined iterator `RAStream()` that is shown at the bottom of Figure 2. This iterator is written to generate a stream of values corresponding to the indices specified by its argument *block*. It is written using two helper functions not shown here, `getNthRandom()` and `getNextRandom()`. Upon computing each value, the iterator *yields* it and then goes on to compute the next one. After exiting the for loop, the end of the iterator is reached, and it terminates.

The body of the loop uses the values yielded by the iterator to update the table using Chapel's xor operator (`^`) used in its read-modify-write form (`^=`). Since multiple tasks created by the outer loop may attempt to update a given table position simultaneously, this code contains a race condition as permitted by the benchmark specification. In order to avoid the race condition (as we must during the benchmark's verification stage) the loop body can be prefixed by Chapel's `atomic` keyword to specify that the update must be computed as though it was instantaneous from the point-of-view of other threads. In such a case, the loop body would be written:

```
atomic T(r & indexMask) ^= r;
```

By implementing the random stream using an iterator, our code allows different pseudo-random number generation techniques to be swapped in and out without modifying the computation itself. Moreover, the use of an iterator allows additional parallelism to be introduced by rewriting the `RAStream()` iterator to return multiple values rather than singletons. Such an implementation would cause the table update operation to be promoted (as in the STREAM Triad benchmark) resulting in implicit parallelism and the potential for batching communication.

B. Random Access Performance Notes

For most platforms, the limiting factor for Random Access performance is the hardware rather than the software. We anticipate this to be the case for our parallel Chapel implementation as well. Without significant optimization, the default implementation that we provide will utilize the full parallelism of a machine's processors, but will make updates to the global table one element at a time, requiring a lot of fine-grain communication. For architectures that are tuned for message-passing workloads this is a worst-case scenario, while for others, having many small messages in flight is not an issue. As we noted in the previous section, the iterators used to describe the computation can be modified to create chunks of work of varying sizes to amortize these communication overheads or to use vector scatter technologies (if available on the architecture). In addition, a programmer who is committed to tuning the computation for a specific architecture could restructure the loops, add new levels of parallelism, or batch updates manually in order to make the computation match the architecture's characteristics. For example, on an architecture with single-threaded processors and poor fine-grain communication, the programmer might choose to rewrite the outer loop to oversubscribe each locale since the communication overheads are likely to dominate the costs of software multithreading.

Our second performance-related note for Random Access relates to iterators. Iterators are one of several language concepts in Chapel that define semantics without specifying a specific implementation approach. This provides the user with a nice abstraction while allowing the compiler to consider a palette of implementation options and select one that is well-suited for the specific case. In particular, an iterator's definition and uses can be analyzed to select an implementation technique that is appropriate for each invocation. While iterator bodies can contain complicated control structures and multiple yield statements in the general case, and while they can be invoked in complicated ways such as *zippered iteration*, many common cases simply use iterators as a clean and modular way of expressing a loop. The `RAStream()` iterator in our Random Access code is one such example, defining a straightforward loop structure and invoking it in a straightforward way. In such cases, the compiler can simply inline the iterator's definition in place of the loop statement and then inline the loop's body into the iterator's yield statement. This results in no runtime overhead compared to writing the loop out explicitly, while supporting loop-level modularity and the ability for the iterator to be invoked in more complicated ways in other loops. For an overview of some of our early and more general implementation approaches for iterators, please refer to our previous work [14].

Another Chapel concept that defines semantics without specifying an implementation mechanism is the *atomic section*. Atomic sections that contain arbitrary code can be implemented using a number of sophisticated techniques, including the body of research known as *Software* and *Hardware Transactional Memory* in which groups of memory operations appear to execute atomically from the point of view of other threads. Implementing Software Transactional Memory effectively on a distributed memory machine remains an open problem, and one that we are actively investigating. For simple atomic statements like the one contained in this code, however, simpler implementation techniques can be utilized. For example, by guarding the verification loop containing the atomic updates with the proper synchronization, the compiler can use

```
  for (str, span) in genDFTPhases(numElements, radix) {
    forall (bankStart, twidIndex) in (ADom by 2*span, 0..) {
      var wk2 = W(twidIndex),
          wk1 = W(2*twidIndex),
          wk3 = (wk1.re - 2 * wk2.im * wk1.im,
                 2 * wk2.im * wk1.re - wk1.im):elemType;

      forall lo in bankStart + [0..str) do
        butterfly(wk1, wk2, wk3, A[[0..radix)*str + lo]);

      wk1 = W(2*twidIndex+1);
      wk3 = (wk1.re - 2 * wk2.re * wk1.im,
             2 * wk2.re * wk1.re - wk1.im):elemType;
      wk2 *= 1.0i;

      forall lo in bankStart + span + [0..str) do
        butterfly(wk1, wk2, wk3, A[[0..radix)*str + lo]);
    }
  }

...

def butterfly(wk1, wk2, wk3, inout A:[1..radix]) { ... }
```

Fig. 3. 1D Radix-4 FFT in Chapel

locks to ensure that no two threads try to access the same table element simultaneously. In the common case where there are no conflicts, the overhead of these locks should be negligible compared to the overhead of the fine-grain remote accesses. Moreover, on architectures with support for *atomic memory operations (AMOs)*, *extended memory semantics (EMS)*, or *compare-and-swap (CAS)* instructions, the Chapel compiler can utilize such capabilities to further reduce overheads.

VII. Fast Fourier Transform (FFT)

The FFT benchmark requires the user to compute a 1D discrete fourier transform on a vector of pseudo-random values. The user is given a fair amount of latitude in choosing an FFT algorithm and approach as long as the outcome is correct. Depending on how it is written, the FFT benchmark could stress a number of system parameters including floating point operations, local memory bandwidth, cache size, and network.

A. Overview of FFT in Chapel

We chose to implement a radix-4 FFT in order to take advantage of its improved Flops-to-MemOps ratio. Our main loop for FFT is shown in Figure 3. The outer loop iterates over the FFT phases using an iterator that we wrote to generate the stride and span of the butterflies for that phase. The next loop uses *zippered iteration* to traverse two iteration spaces simultaneously, the second of which is defined using the *indefinite range* "`0..`" to indicate that it should start counting at zero and iterate as many times as the iterator with which it is zippered. The iterator indices, *bankStart* and *twidIndex* are defined using a tuple-style variable declaration.

The inner loops describe the parallelizable calls to the `butterfly()` routine, which is written to take three twiddle values of unspecified type that it uses to update values within a 4-element array, *A*. The argument types are unspecified and are typically represented using Chapel's 128-bit complex type. However, by omitting these argument types, programmers can optimize calls that they know to use twiddle factors with zero components by passing in values of Chapel's real or pure imaginary types. Such calls cause the compiler to instantiate the butterfly routine for these types, thereby eliminating the extraneous floating point operations against zero values that would be incurred by a routine written specifically for complex values.

The calls to `butterfly()` are also of interest because they use range arithmetic to pass a strided slice of *A* to the formal vector argument that was defined using the anonymous domain "[1..*radix*]". This results in the creation of an *array view*, allowing the original array elements to be accessed within the routine using the local indices $1, 2, \ldots radix$.

B. FFT Performance Notes

It is generally known to be very challenging to express a global 1D FFT that is clear while also being well-tuned for general parallel architectures—there are simply too many degrees of freedom and too many architectural characteristics that can impact performance. In our approach, we have implemented the computation cleanly and expressed both outer- and inner-loop concurrency in order to maximize the parallelism available to the compiler. An aggressive compiler might recognize that the trip counts of these loops vary with the FFT's phases and create specialized versions of `dfft()`'s loops to optimize for outer- and inner-loop parallelism, or a blend of the two. However, even in such a case, the target architecture's memory model and network can still play a key role in how the performance-minded user might express additional optimizations to maximize efficiency.

At one end of the spectrum, even on a parallel machine with a flat shared memory like the Cray MTA, the clean expression of the computation as given here is unlikely to perform optimally. In our 2006 HPCC entry, we provided an alternate Chapel implementation of FFT that is based on the 2005 HPCC submission for the MTA2 [16]. In that

implementation, several specialized versions of the inner loops were created in order to maximize outer-loop parallelism, take advantage of twiddle values with zero components and tune for the MTA's characteristics. While we are optimistic that our simpler implementation of the benchmark would come close to achieving the performance of this specialized version, it will always be the case that programmers willing to expend additional effort to tune for a given platform are likely to produce better performance. Optimizations such as those expressed in this implementation would be likely to improve performance for flat shared-memory contexts such as an SMP node, multicore processor, or the MTA.

On the other end of the spectrum is the distributed-memory case. Here, our implementation faces the problem that in later phases of the computation, the array slices become increasingly spread out, accessing values stored in the memories of one or more remote locales. As in the Random Access benchmark, with no optimization this is likely to result in fine-grain messages that exercise the machine's ability to handle such communications efficiently. Recent work by John Mellor-Crummey's team on the Rice D-HPF compiler strives to statically analyze loops similar to the ones in our solution so that the programmer can express the computation naturally while having the compiler generate an efficient FFT implementation [17]. In their work, they analyze loop bounds and array access patterns in order to create specialized versions of the loops for butterflies that are completely local or distributed. In the latter case, communication is overlapped with computation in order to bring remote values into a locale's memory in a way that hides the communication latency. We hope to duplicate their success in the Chapel compiler and believe that Chapel's support for domains and index types should support such analysis cleanly.

Alternatively, the Chapel programmer can explicitly manage the array's distribution in order to minimize communication. For example, by *redistributing* the domain representing the problem space from a *Block* to a *Cyclic* distribution at the appropriate stage in the computation, accesses to values that would have been remote in the later phases can be localized (given an appropriate number of processors). This approach effectively implements the common technique of computing the 1D FFT using a 2D representation that is blocked in one dimension, and then transposing it partway through the computation to localize accesses.

Of course, the 2D approach to 1D FFTs can also be coded directly in Chapel by declaring 2D domains and arrays that are distributed in one dimension and local in the other. After the local butterflies have been exhausted, the array can be transposed or the domain can be redistributed in order to localize the butterflies in the second dimension. We have coded several of these variations in Chapel, but omit them from this article due to space limitations—such an exploration merits a paper of its own.

We conclude this discussion of FFT performance notes by echoing our opening statement: The FFT is a rich computation with many possible approaches that are sensitive to architectural details. As we target large-scale machines, our hope is that as we can achieve acceptable portable performance from our baseline implementation without requiring the algorithm to be cluttered beyond recognition. We also expect to demonstrate the ability for a user to further optimize performance for a particular architecture by performing the various rewrites described in this section.

VIII. Summary and Future Work

In summary, we have created implementations of the HPCC STREAM Triad, Random Access, and FFT benchmarks that we believe represent the computations cleanly, succinctly, and in a manner that will support good performance as our compiler development effort proceeds. As stated in the introduction, all codes listed in the appendices compile and run with our current compiler. The Chapel compiler is written in C++ and utilizes a Chapel-to-C compilation strategy for portability, allowing us to target any UNIX-like environment that supports a standard C compiler. Our compiler currently builds, compiles, and generates executables for Linux, 64-bit Linux, Cygwin, SunOS, and Mac OS X platforms. We run a nightly regression test suite of over 1,685 tests to ensure forward progress.

A limited initial release of the Chapel compiler was made available to members of the government High Productivity Language Systems (HPLS) team in December 2006. This release was designed to demonstrate many of Chapel's features in a shared-memory context and will enable experimentation with the Chapel language by Cray developers and external reviewers. Our implementation priorities for 2007 are to support multi-locale execution for distributed memory machines and to improve the performance of single-thread execution. We plan to minimize hardware and software assumptions during this phase in order to support portability of the implementation (*e.g.*, commodity Linux clusters are a primary target). We plan to release updated versions of the compiler in 2007.

In concluding this article, it is worth noting that while the HPCC benchmarks have demonstrated many of Chapel's productivity features for global-view programming and software engineering, they remain rather restricted in terms of the parallel concepts that they exercise. In particular, none of these benchmarks required any significant task parallelism, thread synchronization, or nested parallelism. Because the computations were typically driven by a single distributed problem vector, there was no need for Chapel's features for explicit locality control. And even within the data parallel space, all of these benchmarks used only dense 1D vectors of data, leaving Chapel's support for multidimensional, strided, and sparse arrays; associative and opaque arrays; and array composition on the table. In future work, we intend to undertake similar studies for computations that exercise a broader range of Chapel's features.

Acknowledgments

The authors would like to thank the 2005 HPC Challenge finalists who shared their implementations with us and fielded questions about them—Petr Konecny, Nathan Wichmann, Calin Cascaval, and particularly John Feo who helped us learn a lot about FFT in a short amount of time. We would also

like to recognize the following people for their efforts and impact on the Chapel language and its implementation—David Callahan, Hans Zima, John Plevyak, David Iten, Shannon Hoffswell, Roxana Diaconescu, Mark James, Mackale Joyner, and Robert Bocchino.

REFERENCES

[1] Gail Alverson, Simon Kahan, Richard Korry, Cathy McCann, and Burton Smith. Scheduling on the Tera MTA. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *LNCS*, pages 19–44. Springer Verlag, April 1995.

[2] Ray Barriuso and Allan Knies. SHMEM user's guide for C. Technical report, Cray Research Inc., June 1994.

[3] David Callahan, Bradford L. Chamberlain, and Hans Zima. The Cascade High Productivity Language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pages 52–60. April 2004.

[4] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.

[5] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *to appear in an upcoming special issue of the International Journal of High Performance Computing Applications on High Productivity Programming Languages and Models*, 2007.

[6] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[7] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, October 2000.

[8] Cray Inc., Seattle, WA. *Chapel Specification*. (http://chapel.cs.washington.edu).

[9] Steven J. Deitz. *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*. PhD thesis, University of Washington, 2005.

[10] Jack Dongarra and Piotr Luszczek. HPC challenge awards: Class 2 specification. Available at: http://www.hpcchallenge.org, June 2005.

[11] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, June 2005.

[12] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference, volume 2*. Scientific and Engineering Computation. MIT Press, September 1998.

[13] Supercomputing Technologies Group. *Cilk 5.3.2 Reference Manual*. MIT Laboratory for Computer Science, November 2001.

[14] Mackale Joyner, Steven J. Deitz, and Bradford L. Chamberlain. Iterators in Chapel. In *Eleventh International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'06)*. April 2006.

[15] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, September 1996.

[16] Petr Konecny, Simon Kahan, and John Feo. SC05 HPCChallenge class 2 award—Cray MTA2. November 2005.

[17] John Mellor-Crummey. Personal communication.

[18] Robert W. Numerich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[19] Marc Snir, Steve Otto, Steve Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference, volume 1*. Scientific and Engineering Computation. MIT Press, 2nd edition, September 1998.

[20] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamato, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, September 1998.

APPENDIX A
STREAM TRIAD CHAPEL CODE

```
use Time, Types, Random;

use HPCCProblemSize;


param numVectors = 3;
type elemType = real(64);

config const m = computeProblemSize(elemType, numVectors),
             alpha = 3.0;

config const numTrials = 10,
             epsilon = 0.0;

config const useRandomSeed = true,
             seed = if useRandomSeed then SeedGenerator.clockMS else 314159265;

config const printParams = true,
             printArrays = false,
             printStats = true;


def main() {
  printConfiguration();

  const ProblemSpace: domain(1, int(64)) distributed(Block) = [1..m];
  var A, B, C: [ProblemSpace] elemType;

  initVectors(B, C);

  var execTime: [1..numTrials] real;

  for trial in 1..numTrials {
    const startTime = getCurrentTime();
    A = B + alpha * C;
    execTime(trial) = getCurrentTime() - startTime;
  }

  const validAnswer = verifyResults(A, B, C);
  printResults(validAnswer, execTime);
}


def printConfiguration() {
  if (printParams) {
    printProblemSize(elemType, numVectors, m);
    writeln("Number of trials = ", numTrials, "\n");
  }
}


def initVectors(B, C) {
  var randlist = RandomStream(seed);

  randlist.fillRandom(B);
  randlist.fillRandom(C);

  if (printArrays) {
    writeln("B is: ", B, "\n");
    writeln("C is: ", C, "\n");
  }
}


def verifyResults(A, B, C) {
  if (printArrays) then writeln("A is: ", A, "\n");

  const infNorm = max reduce [i in A.domain] abs(A(i) - (B(i) + alpha * C(i)));

  return (infNorm <= epsilon);
}
```

```
74 def printResults(successful, execTimes) {
75   writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
76   if (printStats) {
77     const totalTime = + reduce execTimes,
78           avgTime = totalTime / numTrials,
79           minTime = min reduce execTimes;
80     writeln("Execution time:");
81     writeln("  tot = ", totalTime);
82     writeln("  avg = ", avgTime);
83     writeln("  min = ", minTime);

85     const GBPerSec = numVectors * numBytes(elemType) * (m / minTime) * 1e-9;
86     writeln("Performance (GB/s) = ", GBPerSec);
87   }
88 }
```

APPENDIX B
RANDOM ACCESS CHAPEL CODE

A. Random Access Computation

```
use Time;

use HPCCProblemSize, RARandomStream;


param numTables = 1;
type elemType = randType,
     indexType = randType;

config const n = computeProblemSize(elemType, numTables,
                                    returnLog2=true): indexType,
             N_U = 2**(n+2);

const m = 2**n,
      indexMask = m-1;

config const sequentialVerify = (numLocales < log2(m)),
             errorTolerance = 1e-2;

config const printParams = true,
             printArrays = false,
             printStats = true;


def main() {
  printConfiguration();

  const TableSpace: domain(1, indexType) distributed(Block) = [0..m);
  var T: [TableSpace] elemType;

  const UpdateSpace: domain(1, indexType) distributed(Block) = [0..N_U);

  const startTime = getCurrentTime();

  [i in TableSpace] T(i) = i;

  coforall block in UpdateSpace.subBlocks do
    for r in RAStream(block) do
      T(r & indexMask) ^= r;

  const execTime = getCurrentTime() - startTime;

  const validAnswer = verifyResults(T, UpdateSpace);
  printResults(validAnswer, execTime);
}


def printConfiguration() {
  if (printParams) {
    printProblemSize(elemType, numTables, m);
    writeln("Number of updates = ", N_U, "\n");
  }
}


def verifyResults(T: [?TDom], UpdateSpace) {
  if (printArrays) then writeln("After updates, T is: ", T, "\n");

  if (sequentialVerify) then
    for r in RAStream([0..N_U)) do
      T(r & indexMask) ^= r;
  else
    forall i in UpdateSpace {
      const r = getNthRandom(i+1);
      atomic T(r & indexMask) ^= r;
    }

  if (printArrays) then writeln("After verification, T is: ", T, "\n");

  const numErrors = + reduce [i in TDom] (T(i) != i);
  if (printStats) then writeln("Number of errors is: ", numErrors, "\n");
```

```
  return numErrors <= (errorTolerance * N_U);
}


def printResults(successful, execTime) {
  writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
  if (printStats) {
    writeln("Execution time = ", execTime);
    writeln("Performance (GUPS) = ", (N_U / execTime) * 1e-9);
  }
}
```

B. Random Access: Random Value Generation Module

```
module RARandomStream {
  param randWidth = 64;
  type randType = uint(randWidth);

  const bitDom = [0..randWidth),
        m2: [bitDom] randType = computeM2Vals(randWidth);


  def RAStream(block) {
    var val = getNthRandom(block.low);
    for i in block {
      getNextRandom(val);
      yield val;
    }
  }


  def getNthRandom(in n) {
    param period = 0x7fffffffffffffff/7;

    n %= period;
    if (n == 0) then return 0x1;

    var ran: randType = 0x2;
    for i in [0..log2(n)) by -1 {
      var val: randType = 0;
      for j in bitDom do
        if ((ran >> j) & 1) then val ^= m2(j);
      ran = val;
      if ((n >> i) & 1) then getNextRandom(ran);
    }
    return ran;
  }


  def getNextRandom(inout x) {
    param POLY = 0x7;
    param hiRandBit = 0x1:randType << (randWidth-1);

    x = (x << 1) ^ (if (x & hiRandBit) then POLY else 0);
  }


  def computeM2Vals(numVals) {
    var nextVal = 0x1: randType;
    for i in 1..numVals {
      yield nextVal;
      getNextRandom(nextVal);
      getNextRandom(nextVal);
    }
  }
}
```

APPENDIX C
FFT CHAPEL CODE

```
use BitOps, Random, Time;

use HPCCProblemSize;


param radix = 4;

param numVectors = 2;
type elemType = complex(128);


config const n = computeProblemSize(elemType, numVectors, returnLog2 = true);
const m = 2**n;

config const epsilon = 2.0 ** -51.0,
             threshold = 16.0;

config const useRandomSeed = true,
             seed = if useRandomSeed then SeedGenerator.clockMS else 314159265;

config const printParams = true,
             printArrays = false,
             printStats = true;


def main() {
  printConfiguration();

  const TwiddleDom: domain(1, int(64)) distributed(Block) = [0..m/4);
  var Twiddles: [TwiddleDom] elemType;

  const ProblemDom: domain(1, int(64)) distributed(Block) = [0..m);
  var Z, z: [ProblemDom] elemType;

  initVectors(Twiddles, z);

  const startTime = getCurrentTime();

  Z = conjg(z);
  bitReverseShuffle(Z);
  dfft(Z, Twiddles);

  const execTime = getCurrentTime() - startTime;

  const validAnswer = verifyResults(z, Z, Twiddles);
  printResults(validAnswer, execTime);
}


def printConfiguration() {
  if (printParams) then printProblemSize(elemType, numVectors, m);
}


def initVectors(Twiddles, z) {
  computeTwiddles(Twiddles);
  bitReverseShuffle(Twiddles);

  fillRandom(z, seed);

  if (printArrays) {
    writeln("After initialization, Twiddles is: ", Twiddles, "\n");
    writeln("z is: ", z, "\n");
  }
}


def computeTwiddles(Twiddles) {
  const numTwdls = Twiddles.numElements,
        delta = 2.0 * atan(1.0) / numTwdls;

  Twiddles(0) = 1.0;
  Twiddles(numTwdls/2) = let x = cos(delta * numTwdls/2)
```

```
                          in (x, x):elemType;
  forall i in [1..numTwdls/2) {
    const x = cos(delta*i),
          y = sin(delta*i);
    Twiddles(i)            = (x, y):elemType;
    Twiddles(numTwdls - i) = (y, x):elemType;
  }
}


def bitReverseShuffle(Vect: [?Dom]) {
  const numBits = log2(Vect.numElements),
        Perm: [i in Dom] Vect.eltType = Vect(bitReverse(i, revBits=numBits));
  Vect = Perm;
}


def bitReverse(val: ?valType, revBits = 64) {
  param mask = 0x0102040810204080;
  const valReverse64 = bitMatMultOr(mask, bitMatMultOr(val:uint(64), mask)),
        valReverse = bitRotLeft(valReverse64, revBits);
  return valReverse: valType;
}


def dfft(A: [?ADom], W) {
  const numElements = A.numElements;

  for (str, span) in genDFTPhases(numElements, radix) {
    forall (bankStart, twidIndex) in (ADom by 2*span, 0..) {
      var wk2 = W(twidIndex),
          wk1 = W(2*twidIndex),
          wk3 = (wk1.re - 2 * wk2.im * wk1.im,
                 2 * wk2.im * wk1.re - wk1.im):elemType;

      forall lo in bankStart + [0..str) do
        butterfly(wk1, wk2, wk3, A[[0..radix)*str + lo]);

      wk1 = W(2*twidIndex+1);
      wk3 = (wk1.re - 2 * wk2.re * wk1.im,
             2 * wk2.re * wk1.re - wk1.im):elemType;
      wk2 *= 1.0i;

      forall lo in bankStart + span + [0..str) do
        butterfly(wk1, wk2, wk3, A[[0..radix)*str + lo]);
    }
  }

  const str = radix**log4(numElements-1);
  if (str*radix == numElements) then
    forall lo in [0..str) do
      butterfly(1.0, 1.0, 1.0, A[[0..radix)*str + lo]);
  else
    forall lo in [0..str) {
      const a = A(lo),
            b = A(lo+str);
      A(lo)     = a + b;
      A(lo+str) = a - b;
    }
}


def genDFTPhases(numElements, radix) {
  var stride = 1;
  for 1..log4(numElements-1) {
    const span = stride * radix;
    yield (stride, span);
    stride = span;
  }
}


def butterfly(wk1, wk2, wk3, inout A:[1..radix]) {
  var x0 = A(1) + A(2),
      x1 = A(1) - A(2),
      x2 = A(3) + A(4),
      x3rot = (A(3) - A(4))*1.0i;
```

```
152   A(1) = x0 + x2;
153   x0 -= x2;
154   A(3) = wk2 * x0;
155   x0 = x1 + x3rot;
156   A(2) = wk1 * x0;
157   x0 = x1 - x3rot;
158   A(4) = wk3 * x0;
159 }


162 def log4(x) return logBasePow2(x, 2);


165 def verifyResults(z, Z, Twiddles) {
166   if (printArrays) then writeln("After FFT, Z is: ", Z, "\n");

168   Z = conjg(Z) / m;
169   bitReverseShuffle(Z);
170   dfft(Z, Twiddles);

172   if (printArrays) then writeln("After inverse FFT, Z is: ", Z, "\n");

174   var maxerr = max reduce sqrt((z.re - Z.re)**2 + (z.im - Z.im)**2);
175   maxerr /= (epsilon * n);
176   if (printStats) then writeln("error = ", maxerr);

178   return (maxerr < threshold);
179 }


182 def printResults(successful, execTime) {
183   writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
184   if (printStats) {
185     writeln("Execution time = ", execTime);
186     writeln("Performance (Gflop/s) = ", 5 * (m * n / execTime) * 1e-9);
187   }
188 }
```

APPENDIX D
HPCC PROBLEM SIZE COMPUTATION CODE

```
1  module HPCCProblemSize {
2    use Memory, Types;
3
4    config const memRatio = 4;
5
6    def computeProblemSize(type elemType, numArrays, returnLog2 = false) {
7      const totalMem = + reduce Locales.physicalMemory(unit = Bytes),
8            memoryTarget = totalMem / memRatio,
9            bytesPerIndex = numArrays * numBytes(elemType);
10
11     var numIndices = memoryTarget / bytesPerIndex;
12
13     var lgProblemSize = log2(numIndices);
14     if (returnLog2) {
15       numIndices = 2**lgProblemSize;
16       if (numIndices * bytesPerIndex <= memoryTarget) {
17         numIndices *= 2;
18         lgProblemSize += 1;
19       }
20     }
21
22     const smallestMem = min reduce Locales.physicalMemory(unit = Bytes);
23     if ((numIndices * bytesPerIndex)/numLocales > smallestMem) then
24       halt("System is too heterogeneous: blocked data won't fit into memory");
25
26     return if returnLog2 then lgProblemSize else numIndices;
27   }
28
29
30   def printProblemSize(type elemType, numArrays, problemSize: ?psType) {
31     const bytesPerArray = problemSize * numBytes(elemType),
32           totalMemInGB = (numArrays * bytesPerArray:real) / (1024**3),
33           lgProbSize = log2(problemSize):psType;
34
35     write("Problem size = ", problemSize);
36     if (2**lgProbSize == problemSize) {
37       write(" (2**", lgProbSize, ")");
38     }
39     writeln();
40     writeln("Bytes per array = ", bytesPerArray);
41     writeln("Total memory required (GB) = ", totalMemInGB);
42   }
43 }
```