Global HPC Challenge Benchmarks in Chapel[†]

Revision 2.1 — November 2008 Release

Bradford L. Chamberlain, Steven J. Deitz
Samuel A. Figueroa, David M. Iten
Chapel Team
Cray Inc.
chapel_info@cray.com

Andrew Stone
Department of Computer Science
Colorado State University
stonea@cs.colostate.edu

***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. This report describes Chapel implementations of the global HPC Challenge benchmarks for STREAM Triad, Random Access, FFT, and HPL. Chapel is a work in progress. As such, this report serves as a snapshot of our current status as we work toward implementations of the HPCC benchmarks that are elegant and efficient. The highlights of our submission this year include: (i) the first publicly-released performance results for Chapel including a 1.69 TFlop/s execution of STREAM Triad; (ii) distributed memory executions of STREAM and RA implemented using Chapel's user-defined distribution strategy; (iii) our first executions of FFT at full problem sizes; (iv) our first version of HPL with a focus on exploiting locality. All codes in this report compile and execute correctly with version 0.8 of the Chapel compiler. The full code listings are provided in appendices to this report.**

I. Introduction

Chapel [3] is a new parallel programming language being developed by Cray Inc. as part of its participation in DARPA's High Productivity Computing Systems (HPCS) program.[1,2] The Chapel team is working to design and implement a language that improves parallel programmability, portability, and code robustness as compared to current programming models while producing programs with performance comparable to or better than MPI. Chapel is very much a work in progress, and as such, this article should be viewed as a snapshot of Chapel's current status rather than the final word on its capabilities.

In this article, we present our current Chapel implementations of four of the global HPC Challenge (HPCC) benchmarks[3,4]—STREAM Triad (STREAM), Random Access (RA), 1D Fast Fourier Transform (FFT), and High Performance Linpack (HPL). We provide performance results for the STREAM and RA benchmarks on up to 512 nodes of a Cray XT4, running at the full problem size for STREAM but a reduced problem size (2^{19} table elements per node) for RA due to long execution times. Our FFT and HPL implementations do not yet run on multiple nodes, nor do they achieve competitive performance on a single node, so we provide an overview of the codes themselves and a status report on their implementations.

For the 2006 HPC Challenge competition, we submitted earlier (yet very similar) versions of the STREAM, RA, and FFT benchmarks as an introduction to the Chapel language and an indicator of where we were headed. Since then, the Chapel compiler has achieved a number of important milestones:

- **December 2006:** First limited release of Chapel (subsequent limited releases were made available in June 2007, December 2007, and March 2008)
- **July 2007:** First Chapel codes executing across nodes of a distributed memory platform
- **March 2008:** First complete support for Chapel's task parallel features on distributed memory platforms
- **September 2008:** First codes executing using Chapel's distributed domains and arrays
- **November 2008:** First public release of Chapel

The September 2008 milestone is particularly noteworthy for a few reasons. First, because our HPCC benchmark implementations rely on distributed domains and arrays in order to execute at scale on distributed memory platforms. Second, because Chapel's support for distributed domains and arrays has long been considered one of its most promising productivity contributions (not to mention one of our most daunting research challenges). Third, because our distributions are themselves implemented in Chapel using the same mechanisms that advanced programmers would use to write their own distributions. In particular, for the 1D Block distributions used in this report, the Chapel compiler has no specific semantic knowledge about what a Block distribution is. It only knows that, as with any Chapel distribution, *Block1D* provides a set of classes that support well-defined interfaces including methods to locate, access, and iterate over domain indices and array elements.

Because our support for distributed domains and arrays is scarcely a month old, most of our effort in preparing our entry this year has gone into ensuring that the codes

[†]This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

[1]http://www.darpa.mil/IPTO/programs/hpcs/hpcs.asp

[2]http://www.highproductivity.org/

[3]http://icl.cs.utk.edu/hpcc/

[4]http://www.hpcchallenge.org/

work correctly on distributed memory machines rather than optimizing the performance of Chapel distributions. To this end, our performance results include a discussion of the current scalability bottlenecks and our plans for addressing them in the coming year.

The rest of this report is organized as follows. The next section gives a summary of our code sizes as required by the competition while Section III describes the experimental platform used in the preparation of this report. Sections IV, V, VI, and VII each give a brief overview of one of the benchmarks, describing our approach in Chapel, our implementation status, and our next steps. Section VIII concludes with a brief summary. Our complete source listings are provided in Appendices A–E.

II. Code Size Summary

The following table categorizes and counts the number of lines of code utilized by our HPCC implementations:

	Benchmark Code				
line count type	*STREAM Triad*	*Random Access*	*FFT*	*HPL*	*Problem Size*
Kernel computation	2	3 + 25 = 28	57	50	0
Kernel declarations	11	20 + 13 = 33	22	63	34
Total kernel	*13*	*23 + 38 = 61*	*79*	*113*	*34*
Initialization	9	1 + 9 = 10	26	8	0
Verification	8	9 + 0 = 9	11	16	0
Results and Output	32	21 + 0 = 21	21	39	21
Total Benchmark	***62***	***54 + 47 = 101***	***137***	***176***	***55***
Debug and Test	7	3 + 0 = 3	3	1	0
Comments	72	94 + 31 = 125	109	170	39
Blank	27	23 + 8 = 31	40	61	8
Total Program	*168*	*174 + 86 = 260*	*289*	*408*	*102*

The line counts for each benchmark are represented using a column of the table. The final data column represents the shared *HPCCProblemSize* module that is used by the benchmarks to automatically compute the appropriate problem size for a machine and to print it. For the Random Access benchmark, each entry is expressed as a sum—the first value represents the benchmark module itself, the second represents a helper module used to define the stream of pseudo-random update values, and the final value is the sum of the two.

The rows of the table are used to group the lines of code into various categories and running totals. The first two rows 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 two lines of code, while its supporting variable and subroutine declarations require eleven 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—thirteen in the case of STREAM.

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 the 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 giving 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 *Comments* row indicates the number of comment lines and 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 serve as a checksum against the line number labels that appear in the appendices.

The next table compares the total SLOC for the standard HPCC reference implementations with that of our Chapel codes:

	Benchmark Code			
	STREAM Triad	*Random Access*	*FFT*	*HPL*
HPCC SLOC	433	1668	1406	11,674
Chapel SLOC	117	156	192	231
SLOC Ratio	*3.70*	*10.69*	*7.32*	*50.53*

The HPCC SLOC results are the sum of the *Framework* and *Parallel* numbers reported for the reference versions of the benchmarks in the table from the HPCC website's FAQ.[5] The Chapel result for each code is obtained by summing its *Total Benchmark* result from the previous table with that of the Problem Size module (55 lines) to compute the problem size.

This table shows that our Chapel codes are approximately 3.7–50× smaller than the reference implementations. Note that this isn't an apples-to-apples comparison since some of the HPCC codes implement several variations on an algorithm while our benchmarks implement a single algorithm. Moreover, it is commonly understood that shorter codes are not necessarily easier to understand. That said, having browsed both source bases, we believe that our Chapel implementations are not only succinct, but also clearer representations of the benchmarks than the reference implementations, and that they would serve as a better reference for future programmers tackling the HPC Challenge benchmarks.

III. Experimental Platform

This section describes the experimental platform that we used in preparing this report. Our performance results were obtained on Jaguar, a Cray XT4 located at Oak Ridge National Laboratory (ORNL). The following table provides a brief overview of Jaguar:

machine characteristic	*value*
# compute nodes	7,832
compute node processor	2.1 GHz AMD Opteron
cores per node	4
total usable RAM per node (as reported by `/proc/meminfo`)	7.68 GB

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

In terms of software, our experiments were conducted using our current version of the Chapel compiler which uses a source-to-C compilation approach for portability. On Jaguar, we used Cray's *PrgEnv-gnu* programming environment module which provides a Cray C compiler wrapper around gcc. We used this compiler to compile both Chapel's generated C code and the standard reference implementation of the HPCC benchmarks. Our runtime libraries use POSIX threads (*pthreads*) to implement tasks and Berkeley's GASNet communication library [2] for inter-process coordination and data transfer. The software versions and settings that we used are given in the following table:

software *flags/settings*	*version*
chpl −−fast	0.8
PrgEnv-gnu	2.0.49a
cc, gcc -target=linux -O3 -std=c99 −−param max-inline-insns-single=35000 −−param inline-unit-growth=10000 −−param large-function-growth=200000	4.2.0
pthreads	NPTL
GASNet conduit=portals, segment=fast GASNET_MAX_SEGSIZE 4294967296	1.12.0

The Chapel flag "−−fast" turns off a number of runtime checks that are enabled by default for safety, including guards against out-of-bounds array accesses, null pointer dereferences, and violations of locality assertions. The flags used for the C compilation were chosen by GASNet's auto-configuration process and were used both for the generated Chapel code and the HPCC reference implementations. The GASNet conduit and segment choices are the recommended settings for running on a Cray XT. The GASNET_MAX_SEGSIZE setting is required to support data set sizes larger than the default of 2GB per node.

IV. STREAM Triad

The STREAM Triad benchmark asks the programmer to generate two vectors of random 64-bit floating-point values, b and c, and to compute $a = b + \alpha \cdot c$ for a scalar value α. We express this computation in our entry this year using the following lines of Chapel code:

```
forall (a, b, c) in (A, B, C) do
  a = b + alpha * c;
```

This pair of statements says to iterate in parallel over the vectors A, B, and C in a *zippered* manner, referring to corresponding elements as a, b, and c for the purposes of the loop body. Within the loop, standard multiplication, addition, and assignment are applied to the component scalar values.

The distributed implementation of these vectors and the parallel implementation of the loop are both controlled by the *distribution* of A, B, and C, specified using a series of three declarations. The first:

```
const BlockDist = new Block1D(bbox=[1..m],
                              tasksPerLocale=tasksPerLocale);
```

creates a distribution named $BlockDist$ and assigns it a new instance of the distribution class *Block1D* which maps 1D indices across the set of *locales*[6] executing the program. *Block1D* computes this mapping by partitioning the specified bounding box, $1 \ldots m$, across the locales using evenly-sized blocks (± 1). It also takes an argument $tasksPerLocale$ indicating how many tasks should be used on each locale to implement parallel loops over the distribution's domains and arrays. Here, we are passing it a *configuration constant* of the same name that can be used to vary this number from one execution of the program to the next.

The second declaration:

```
const ProblemSpace: domain(1, int(64))
                    distributed BlockDist = [1..m];
```

creates a *domain*—a first-class language concept representing an index set—to describe the set of indices that define the problem space. This domain, $ProblemSpace$, is declared to be a 1-dimensional domain of 64-bit integer indices, distributed using the $BlockDist$ distribution created previously. It is initialized to store the index set $1 \ldots m$ which will be divided between the locales according to the mapping defined by $BlockDist$.

The third declaration:

```
var A, B, C: [ProblemSpace] elemType;
```

creates our three vectors, A, B, and C, specifying that each index in $ProblemSpace$ should be mapped to a variable of type $elemType$ (defined previously to be a 64-bit real floating-point value). These vectors are implemented using $ProblemSpace$'s distribution and therefore have their elements mapped to the locales' memories in a blocked manner according to $BlockDist$.

Chapel distributions like *Block1D* not only map domain indices and array elements to locales, they also serve as recipes for mapping high-level operations—such as the forall loop used for the Triad computation—down to the individual data structures and tasks that will implement the computation across the locales. In the case of a zippered forall loop like this one, the compiler rewrites the loop using *leader/follower iterators* defined by the distribution which specify how zippered parallel iteration should be implemented for its domains and arrays. The distribution itself is written in Chapel using standard features such as *coforall loops* to create tasks and *on-clauses* to specify the locales on which the tasks should run.

As mentioned earlier, the Chapel compiler contains no semantic knowledge specific to *Block1D* distributions. It only knows that, as a distribution, *Block1D* will support a standard

[6]A *locale* in Chapel is an architectural unit of locality. Locales have the ability to execute computation and store data. Tasks running within a locale are considered to have uniform access to local data; they can also access data in other locales, but with greater overhead. On a commodity cluster, a multicore processor or SMP node would typically be considered a locale. On jaguar, it is a single quadcore node.

interface of methods and iterators that it can target when lowering and optimizing high-level operations on its domains and arrays. This philosophy forms the basis of our plan to support user-defined distributions in Chapel and to implement Chapel's Standard Distribution Library using this same mechanism. To our knowledge, this is the first time that such capabilities have been implemented in a global-view parallel language, and the first time that parallel zippered iteration has been implemented using a leader/follower iterator scheme. This report constitutes the first public mention of these concepts in print, and we intend to write technical papers describing our approach in more detail in the coming year.

We ran our Chapel STREAM Triad benchmark on Jaguar using up to 512 locales (nodes). The problem sizes that we used and their respective memory requirements are summarized in this table:

STREAM Characteristic	*Chapel*	*HPCC*
number of vectors	3	3
element size (in bytes)	8	8
per-locale problem size	85,985,408	87,469,200
per-locale memory required	1.92 GB	1.95 GB
percent of available memory	25.0%	25.3%

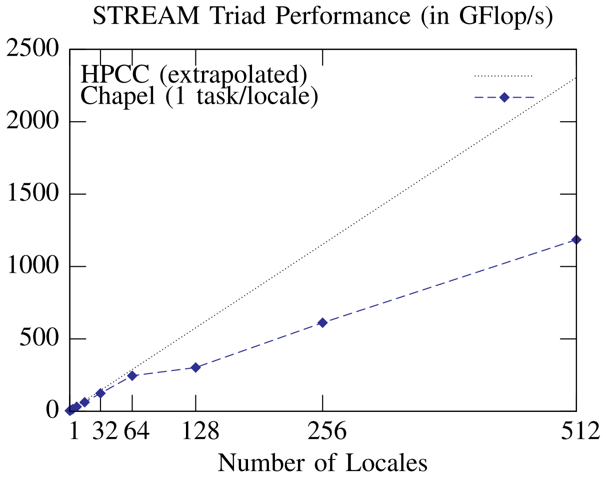
The Chapel problem size was automatically computed using the *HPCCProblemSize* module given in Appendix E. The reference version of HPCC does not support the direct specification of STREAM's problem size—only indirectly through the size of a 2D HPL matrix size—so the problem size for the HPCC version represents a size that we were able to coerce it into running which approximates the Chapel problem size.

The following table gives an indication of our single-locale, single-task execution times:

STREAM Version	*Single-Task Performance*
HPCC Single	4.506 GFlop/s
HPCC Star	4.505 GFlop/s
Chapel −−local	4.030 GFlop/s
Chapel	4.038 GFlop/s

HPCC Single and *HPCC Star* are the standard HPCC results for the reference implementation of STREAM Triad. The *Chapel −−local* entry refers to a run of the Chapel benchmark compiled with a flag that asserts to the compiler that it will only be run on one locale, removing parallel overheads related to distributed memory execution. *Chapel* is the multi-locale executable running on a single locale. As can be seen, the Chapel implementations lag the reference version by approximately 9–10%, due primarily to the parallel loops that are generated in the code which are degenerate for this single-locale, single-task run. Previously, we have demonstrated a sequential Chapel STREAM implementation with performance identical to hand-coded sequential C and Fortran on desktop workstations, so this gives us some hope of closing this scalar performance gap.

Our multi-locale performance results are shown in the following graph:

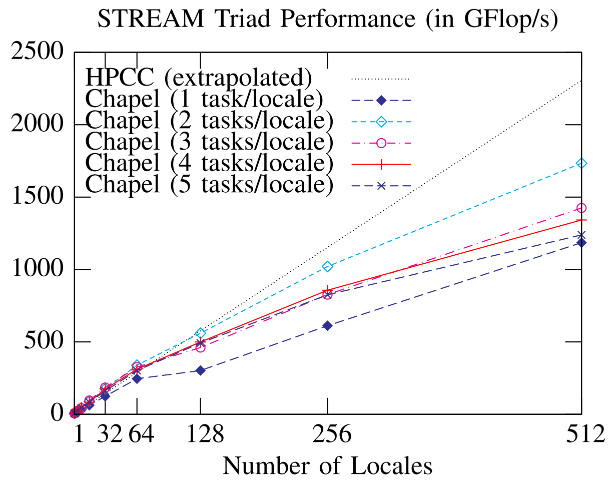


Since the reference implementation of HPCC Stream does not compute an aggregate GFlop/s performance when executing on multiple nodes, we extrapolated its performance by taking the 1-node HPCC Star timing and scaling it linearly with the number of locales. This is a reasonable assumption given that the multi-node reference implementation simply executes multiple copies of the single-node computation, each with its own local timing loop and no communication.

Although STREAM Triad is an embarrassingly parallel benchmark, our current Chapel compiler does not generate the perfect scaling that one should expect. The culprit is our current implementation of the leader iterator in the *Block1D* distribution. In particular, the leader spawns off a task on each of the remote locales one after the other, introducing $O(p)$ overhead to the forall loop when running on p locales. Similarly, the synchronization used to terminate the leader is performed by having each of the p locales indicate to the leader that they have completed their local work. As the number of locales increases, these linear bottlenecks start to cut into our scalability as should be expected. Apart from these startup and teardown overheads, the computation itself is completely local and ought to result in perfect speedup as we demonstrate below.

Note that due to long queuing times leading up to SC08, we only had time to run each experiment once. Thus, we believe that some characteristics of our results, such as the dip at 128 locales in this graph, are due to an insufficient number of experimental runs rather than something deeper.

As mentioned above, our implementation of STREAM supports the ability to run a user-specified number of tasks per locale to take advantage of intra-locale parallelism—in this case the 4 cores on each node. We ran our implementation varying the number of tasks from 1 to 5 and show those performance results here:



Interestingly, while the 3- and 4 task/locale numbers are quite competitive (and often the fastest at lower numbers of locales), from 64 locales onwards, the 2 task/locale case becomes the best, achieving a maximum of 1.69 TFlop/s on 512 locales. Even at its best, though, the Chapel implementation continues to lag behind the single task per locale MPI implementation by a significant margin due to the startup/teardown reasons described above.

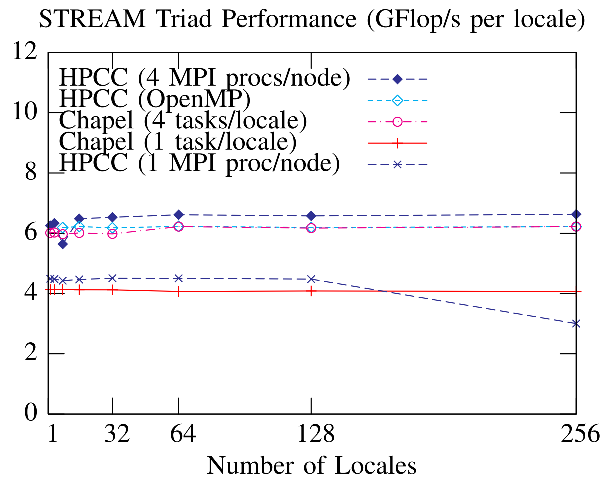
As our Chapel implementation matures, we expect that the performance of our submission will improve until it matches that of the SPMD reference version. In the short-term, we will be replacing the linear creation of tasks in the *Block1D* leader with a tree-based task spawning scheme in order to replace the $O(p)$ startup and teardown costs with an $O(\log p)$ version that ought to greatly reduce the overheads that we are currently seeing. This technique requires support for recursive leader iterators which we do not yet support in our implementation. In the longer-term, we plan to implement compiler optimizations for code segments like STREAM that can be implemented using a traditional SPMD execution. This supports Chapel's philosophy that programmers should not be constrained to SPMD programming models as they are in many current languages, but rather that SPMD should be an important common case of parallel execution to support and optimize for. Applying such an optimization to STREAM would move the creation and destruction of tasks into the program's initialization and teardown, removing the overheads from the user's code as in a traditional MPI execution.

Today, performance-minded Chapel programmers can manually remove these overheads from their code by programming Chapel in a more explicit SPMD style similar to MPI. This supports Chapel's *multiresolution design* philosophy which says that in providing high-level abstractions, a language should not prevent the programmer from dropping down to lower levels, closer to the machine. In particular, a version of STREAM can be written in which an explicit coforall loop and *on-clause* are used to create a task on each locale outside of the timing loop as in the reference version of the benchmark. The program would then manually fragment the problem space into per-locale chunks, performing the computation on the local chunks. In this version, separate timings could be taken on each locale and combined using reductions after the coforall loop as in the MPI version. A simplified version of this approach that omits details of initialization, multiple trials, and verification would appear as follows:

```
var localGBs: [LocaleSpace] real;

coforall loc in Locales do
  on loc {
    const myProblemSpace: domain(1, indexType)
            = BlockPartition(ProblemSpace,
                             here.id, numLocales);

    var myA, myB, myC: [myProblemSpace] elemType;

    const startTime = getCurrentTime();
    local {
      for (a, b, c) in (myA, myB, myC) do
        a = b + alpha * c;
    }
    const execTime = getCurrentTime() - startTime;

    localGBs(here.id) = timeToGFlops(execTime);
  }

const avgGBs = (+ reduce localGBs) / numLocales;
```

Note that our ability to abandon Chapel's global-view array abstractions and elegantly step into an explicit SPMD-style programming model is in stark contrast to most previous languages with support for global arrays. We believe that such multiresolution capabilities are of the utmost importance for languages like Chapel that want to support both programmability and performance, if for no other reason than to work around cases where the compiler or high-level abstractions fail them. Furthermore, we believe our SPMD implementation of STREAM is far more elegant than the equivalent MPI program due to Chapel's support for global-view task parallelism at the language level.

Comparing the SPMD-style Chapel results with the HPCC reference implementation in terms of average GFlop/s per locale, we see that the Chapel version does perform quite competitively once the task startup and teardown has been removed from the timing loop:



This graph shows the HPCC reference code running in three configurations: (a) 4 MPI processes per node, (b) 1 MPI process per node using 4 OpenMP threads per process, and (c) 1 MPI process per node. The Chapel is run with 4 tasks per locale and with 1 task per locale. As can be seen, running 4 tasks per locale results in approximately 6 GFlop/s per locale whether the tasks are implemented using Chapel, MPI, or OpenMP. Meanwhile, running 1 task per locale results in similar performance for Chapel or MPI of 4 GFlop/s. These results confirm our hypothesis that task startup and teardown overheads are the cause of our STREAM entry's current lack of scalability.

As in the previous graph, we did not have time to run multiple trials of these experiments, nor to run the 512 locale experiments. Due to this, we interpret the outlier values (1 MPI task on 256 nodes and 4 MPI tasks on 8 nodes) as being anomalous rather than an indication of a scalability problem.

As we have argued, in the case of our STREAM implementation we believe that our scalability overheads are due primarily to the immaturity of our distributed array implementation rather than a fundamental flaw in Chapel's design. For this reason we chose not to pursue an explicitly fragmented STREAM implementation like the one above as our official submission to the HPCC competition—it is not the approach we wish to promote for Chapel. That said, even when our compiler is mature, there will always be cases when a performance-driven programmer will want to dive below the high-level abstractions and program as close to the machine as possible. Chapel's support for multiresolution parallel programming enables this better than previous languages while still permitting the programmer to use higher-level abstractions in sections of the code where performance is not as critical.

As a closing note, readers who are familiar with Chapel may notice that our STREAM Triad entry this year differs from our traditional one-statement version, which appears as follows:

```
A = B + alpha * C;
```

This version uses *promotion* by applying the scalar operators + and * to the vectors A, B, and C, resulting in semantics that are identical to the zippered parallel iteration of our forall-loop-based entry. While the promotion-based version works correctly today, the use of promoted operators currently thwarts a crucial compiler analysis that optimizes our leader/-follower iterators for well-aligned cases like this one. This is again a symptom of the immaturity of our distributed array implementation, and we expect that our 2009 HPC Challenge entry will demonstrate the one-statement promoted version at scale.

V. RANDOM ACCESS (RA)

The Random Access benchmark computes pseudo-random updates to a large distributed table T of 64-bit unsigned integer values. As in STREAM, our distributed memory implementation uses *Block1D* distributions—one to distribute the set of N_U table updates represented using a domain named *Updates*, and a second to distribute the table T and its corresponding domain.

The core of the Chapel implementation can be summarized by the following three lines of code:

```
forall (_, r) in (Updates, RAStream()) do
  on T(r & indexMask) do
    T(r & indexMask) ^=r;
```

As in STREAM, we use a parallel zippered iteration to express the main computation but rather than traversing arrays, this forall loop iterates over *Updates* and *RAStream()*—an iterator defined elsewhere in the benchmark to generate the pseudo-random stream of values. Each random value is referred to as r for the purposes of the loop body while the values representing the update indices are neither named nor used (as indicated by the underscore).

Since the table location corresponding to r is increasingly likely to be owned by a remote locale as the number of locales grows, we use an on-clause to specify that the update should be computed on the locale that owns the target table element. This results in the creation of a remote task, passing it the value r, and having it perform the update, after which it signals to the main loop that it is done.

Though the above version of RA works in our current implementation, the version of RA that we used for our timings (and which appears in the appendices) uses a different on-clause than the one above. In particular, our compiler does not currently optimize the table access appearing within the on-clause by realizing that it does not need to access the array element in question, but only needs to determine the locale on which it lives. As a result, today, the version above results in an unnecessary remote communication in order to access that value of T, only to drop it on the floor. To manually optimize this away, we rewrite the on clause as follows in our entry:

```
  on T.domain.dist.ind2loc(r & indexMask) do
```

This expression says "Access T's distribution and call its index-to-locale mapping function to determine which locale owns the index `r & indexMask`." Once we implement the optimization described above, we will be able to replace this with the simpler and more elegant reference to `T(r & indexMask)`.

As permitted by the benchmark, our RA implementation contains races since two iterations of the loop could attempt to update the same table location simultaneously, in which case one could miss the other's write. In practice, we never saw this cause more than a handful of conflicts for any of our executions. Our verification loop uses Chapel's *atomic statement* to indicate that each update should be implemented safely, without conflicts. This feature is currently unimplemented, suggesting that our verification loop is likely to increase the number of errors due to races. We are currently working with researchers at the University of Notre Dame and ORNL to add software transactional memory (STM) mechanisms to Chapel's runtime libraries in support of its atomic blocks. This will build on previous work we conducted with UIUC that demonstrated the potential of supporting STM on distributed memory architectures [1].

While the official benchmark also permits updates to be batched to amortize the communication overheads, in this entry, we have opted to take a pure update-at-a-time approach for the sake of elegance and to see how far we can push the performance of this implementation.

Our RA problem sizes are given in the following table:

RA Characteristic	*Chapel*	*HPCC*
number of tables	1	1
element size (in bytes)	8	8
per-locale problem size	2^{19}	2^{27}
number of updates per locale	2^{21}	2^{29}
per-locale memory required	0.0039 GB	1.00 GB
percent of available memory	0.05%	13.0%

We did not run the official problem size and number of updates in Chapel due to the amount of time required to execute them. We chose the problem size here due to the amount that we estimated we would be able to run in time for this release's deadline. That said, we found that our results on a given number of locales scaled fairly linearly as the problem size and number of updates increased, which is not surprising since the work in our implementation is not influenced by the table size and should scale linearly with N_U. In retrospect, it also appears that we did not configure the reference HPCC version correctly since we did not meet the 25% threshold.[7]

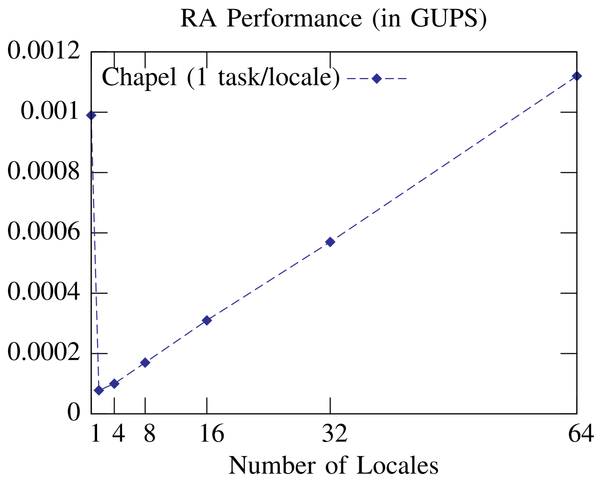
The following table gives our single-task, single-locale performance results for RA:

RA Version	*Performance*
HPCC Single	0.0105 GUPS
HPCC Star	0.0105 GUPS
HPCC MPI	0.0102 GUPS
Chapel −−local	0.0209 GUPS
Chapel	0.00099 GUPS

The *HPCC Single*, *Star*, and *MPI* entries are the standard HPCC timings using the optimized Sandia algorithm. As in our STREAM results, the *Chapel −−local* results represent a compilation of the benchmark in which the compiler can assume it will never be run on more than one locale. As can be seen, this results in a very nice GUPS figure due to the fact that the compiler can optimize away all of the on-clauses and inter-locale communication that the implementation typically assumes it will need. Running our multi-locale implementation on a single locale results in a major performance hit due to the overheads of spawning tasks, resulting in a GUPS figure that is an order of magnitude worse than the HPCC implementation. In future work, we will investigate the root causes of this performance gap to see how much of it is due to our element-by-element implementation versus our early support for multi-locale parallelism.

[7]This may indicate something about the productivity of having your user reverse engineer their desired 1D problem size based on the problem size of an all-but-unrelated 2D benchmark.

The following graph shows our performance as we increase the number of locales:



We chose not to plot against the reference HPCC version due to the different approaches taken and the difference in magnitude between our results. We should also note that these performance results are from our previous version of the benchmark that was hard-coded to only execute a single task per locale. In practice, the multiple task-per-locale implementation that we list in this report and include in our v0.8 release has resulted in improved GUPS performance over the single-task version, but we were not able to get timings for more than 16 locales in time for this release and so will report on that version in a future draft of this paper.

As one would expect given our implementation, our GUPS rate drops significantly as we move from one locale to two since an ever-increasing fraction of the on clauses that had been able to execute locally on a single locale must now create tasks on remote locales. However, once we have taken that performance hit, our execution scales linearly through the 64-locale timings that we were able to run. At the time of this writing, we have not had a chance to successfully get our 128- and 256-locale executions through the queues.

While the performance gaps between the HPCC implementation and our implementation and between 1 locale and 2 are large, we have had almost no time to investigate and optimize the causes and are confident that improvements can be made. Moreover, the fact that we are scaling linearly after taking those hits is encouraging since large parallel machines are not always used to get speedups compared to uniprocessor timings, but also to run increasingly large data set sizes.

One variation on this year's entry that we are exploring uses a "fire and forget" approach in which remote tasks are launched asynchronously using a *begin* statement, leaving the forall loop free to start working on the next iteration. This represents an interesting approach because it would be difficult to write elegantly in MPI and other conventional parallel programming models due to their reliance on a cooperating executables model of parallelism. In Chapel, we would write this as follows:

```
sync {
  forall (_, r) in (Updates, RAStream()) do
    on T(r & indexMask) do
      begin T(r & indexMask) ^=r;
}
```

The *sync* statement ensures that all tasks created within its dynamic scope using begin statements will have terminated before execution continues—in other words, that all the updates are complete. Our current implementation of this approach has a memory leak that prevents it from executing at large problem sizes, so this version is currently an avenue for future exploration.

As mentioned above, we are also exploring using multiple tasks per locale—optionally oversubscribing the processor cores—with the goal of keeping the processors busy and hiding the network latency associated with firing off remote tasks and waiting on them.

Due to the overheads of remote task spawning on conventional architectures, we will also be exploring versions of RA that batch their updates, as in the MPI reference implementation, to see how the elegance and performance of such versions in Chapel compares with MPI.

VI. Fast Fourier Transform (FFT)

The FFT benchmark asks the programmer to compute a 1D discrete Fourier transform on a vector of pseudo-random values. Our implementation uses a radix-4 algorithm in order to take advantage of its improved Flops-to-memory operations ratio. This affects the elegance of the code somewhat, but still results in an implementation that is clearer to read than most publicly-available C/Fortran implementations.

As described in our 2006 HPC Challenge entry, we believe that the strengths of our FFT implementation are its clean expression of the multiple levels of the parallelism in the algorithm; its use of a generic butterfly routine to support real or complex multipliers with a single source routine; and its use of domain striding and vector slicing to express the FFT's access patterns in a concise yet readable way. Our FFT implementation has changed in only minor ways since the 2006 competition:

- The main loop over the phases of the DFFT has been cleaned up by pushing the logic that enumerates the powers of four defining the stride and span of the butterflies into a user-defined iterator, $genDFTStrideSpan()$.
- The almost-universally reviled "open interval" syntax in which `[0..n)` served as sugar for the range $0 \dots n-1$ has been replaced with a more general and powerful range operator, #, that specifies the number of values in the range. As examples, "`lo..#num`" starts at lo and counts num elements while "`lo.. by str #num`" starts at lo and enumerates num elements stride by str. This operator allowed us to simplify a number of range arithmetic expressions in our original entry. For example, the strided range example above would have appeared as the less-clear "`[0..num) * str + lo`" in our 2006 entry.
- Rather than passing the $radix$-element vector slice into $butterfly()$ using an *inout intent* (intended to create a local copy of the vector on the locale implementing the butterfly), we now do an explicit copy to and from the slice within the butterfly routine. We made change in order to make the copy more explicit and avoid the semantic question of whether or not a copy of a distributed array slice should remain distributed or be localized.
- Identifier names have generally been improved in hopes of making the code more comprehensible.

The bulk of our work on FFT in recent months has been focused on plugging memory leaks which have prevented it from executing at the full problem size. Since implementation on the Chapel compiler began, we have been overly cavalier about failing to free compiler-allocated memory due to a combination of competing priorities and our long-term plan to address the issue via garbage collection. During the past year, memory leaks have become a growing concern for us, both due to their impact on the problem sizes we can run and their impact on performance. While all of our HPCC benchmarks suffered from memory leaks in 2006, FFT has required the most effort due to its heavy use of array slice descriptors and array copies within its inner loops, all of which were being leaked.

In the weeks leading up to this release, we have been able to get our compiler generating code that cleans up these temporaries, permitting us to compile and run FFT at the full problem size on a single locale for the first time. So far, we have only had the chance to perform initial performance comparisons against a C version of FFT on which our implementation was based. Anecdotally, the performance difference between the codes is around a factor of four. This is encouraging given that the Chapel compiler currently does nothing to optimize the general domains and arrays used to implement its vector butterfly slices, all of which should be amenable to a lighter-weight implementation given their invariant nature and short lifetimes. That said, we recognize that FFT is a challenging code to tune and anticipate that additional work will be required when comparing against more highly-optimized versions.

Our FFT implementation is currently unable to run on multiple locales due to the fact that our *Block1D* distribution is not yet mature enough to support the array slices used in the benchmark. This should not require significant effort, but as the HPC Challenge deadline approached, we decided to focus most of our efforts on the scalability of STREAM and RA rather than on the generalization of the *Block1D* distribution.

Our next steps with FFT are to tune the single-locale performance, to finish the multi-locale implementation, and to make our memory deallocation more robust. We also plan to explore the use of redistributing the vector's domain midway through the FFT's phases in order to guarantee that all of the butterflies are local to a single locale when running on 2^k locales.

VII. High Performance Linpack (HPL)

The HPL benchmark requires the user to solve a dense LU factorization problem using pivoting. Our 2006 HPC Challenge entry did not include an HPL implementation, and this year's entry marks our team's first implementation of LU written with an eye toward scalability and locality. The version of the code in this paper compiles and executes correctly with our current compiler, but has not yet been run on the full problem size due to memory leaks, nor evaluated for performance due to time constraints.

While we have approached this implementation with an eye toward locality, it does not yet execute using multiple locales due to its need for multidimensional block-cyclic and replicated distributions. We have started the implementation of both of these distributions in Chapel, but they are not yet mature enough to support codes of HPL's complexity. In spite of this, as we have worked through the algorithm, we have mentally anticipated the introduction of these distributions in order to remain aware of which accesses will be local vs. remote. Moreover, for key routines like *schurComplement*, we have used Chapel's local-statement to assert that communication should not be required for the component dgemm operations. By default, such assertions are checked at runtime, though the checks may be turned off for production runs using a compiler flag.

Our implementation benefits from Chapel's support for multidimensional domains/arrays, array views, and domain/array slicing. Our implementation particularly benefits from the use of unbounded ranges and the #-operator, both of which eliminate opportunities for introducing trivial errors in bounds arithmetic when slicing into the distributed array+vector *Ab*. While we anticipate a lot of work ahead of us to get HPL running at competitive speeds, we believe that Chapel's clean implementation of HPL will simplify future changes to the code and to the compiler's analysis and optimization. We also anticipate using HPL as a motivating case for our language interoperability features in order to demonstrate a version that will pass slices of our distributed block-cyclic array into BLAS routines to take advantage of their highly-tuned implementations on each platform. Over time, we plan to write a tutorial-like document that walks through our HPL code in detail as we have done previously for STREAM, RA, and FFT.

Our current implementation of HPL is very synchronous in that it performs the various stages of the algorithm sequentially, one after the other. Once we get this version executing competitively with an equivalent hand-coded version, we plan to explore a more asynchronous/dataflow-based implementation using Chapel's begin statements and *synchronization variables* to execute stages of the algorithm in parallel, pipelining the computation to avoid well-known bottlenecks in step-by-step implementations like ours. We realize that HPL is a well-studied benchmark and invite comments from experienced HPL programmers as to how we might improve our code for efficiency and clarity.

VIII. Summary and Future Work

As stated at the outset, we have written this report to serve as an opportunity for the parallel programming community to peek over our shoulders by providing a snapshot of our current status with Chapel. This report makes it clear that Chapel is not yet ready for prime-time, yet we never intended for that to be our thesis this year. Rather, we are encouraged by the milestones that we have achieved since submitting our single-threaded, single-locale, memory-hogging implementations of STREAM, RA, and FFT in 2006. Moreover, we wish to point out how similar this year's entries are to their 2006 counterparts which were written with (eventual) large-scale parallel execution in mind.

We think it's worth repeating that the experimental results in this paper are based on a distributed array capability that is scarcely a month old, using a distribution written in Chapel, and without embedding any knowledge of block distributions into our compiler or runtime libraries. In the weeks leading up to this release, we came up with more ideas for new optimizations to our initial implementation than we had the time to pursue, and our inability to implement more than a handful of them in time for this deadline was a frequent but pleasant source of frustration.

We clearly have our work cut out for us before next year's competition. At this point, we anticipate focusing our efforts on performance optimizations, particularly in the area of automated locality inference and optimization. While we have made great strides in terms of memory leaks, HPL makes it clear that we still have more work ahead of us before the compiler can be considered a good steward of system memory. We also plan to build on our distribution story by fleshing out the missing capabilities for the *Block1D* distribution and by adding support for other standard distributions such as multi-dimensional Block, Block-Cyclic, and Replicated distributions using the same interface and mechanisms that we have for *Block1D*.

Though some amount of the coming year will be focused on improving the performance of the HPCC benchmarks, we also expect to spend a fair amount of time looking at more advanced computations, such as those that make greater use of dynamic and task-based parallelism; and those that use hierarchical, sparse, and unstructured data structures. We invite members of the community with favorite parallel coding challenges to contact us and explore how they might be expressed, implemented, and optimized in Chapel as our compiler continues to mature.

Acknowledgments

The authors would like to gratefully acknowledge our former team members, Mary Beth Hribar and Wayne A. Wong, who were instrumental in developing the original versions of these benchmarks for the 2006 HPC Challenge competition. We'd also like to thank John Lewis and Adrian Tate for their linear algebra expertise which enabled us to create our first reasonable version of HPL in Chapel. Thanks also to Paul Casella, Eric Jones, Mike Karo, and Steve Whalen for their role in helping us get our experimental results up and running in short order. Finally, we want to thank all of Chapel's past contributors and early users for helping us reach this stage.

References

[1] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 247–258, New York, NY, USA, 2008. ACM.

[2] Dan Bonachea. GASNet specification, v1.1. Technical Report CSD-02-1207, University of California – Berkeley, Berkeley, CA, October 2002.

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

APPENDIX A
STREAM TRIAD IN CHAPEL

```
//
// Use standard modules for Block distributions, Timing routines, Type
// utility functions, and Random numbers
//
use BlockDist, Time, Types, Random;

//
// Use shared user module for computing HPCC problem sizes
//
use HPCCProblemSize;

//
// The number of vectors and element type of those vectors
//
const numVectors = 3;
type elemType = real(64);

//
// Configuration constants to set the problem size (m) and the scalar
// multiplier, alpha
//
config const m = computeProblemSize(numVectors, elemType),
             alpha = 3.0;

//
// Configuration constants to set the number of trials to run and the
// amount of error to permit in the verification
//
config const numTrials = 10,
             epsilon = 0.0;

//
// The number of tasks to use per Chapel locale
//
config const tasksPerLocale = min reduce Locales.numCores;

//
// Configuration constants to indicate whether or not to use a
// pseudo-random seed (based on the clock) or a fixed seed; and to
// specify the fixed seed explicitly
//
config const useRandomSeed = true,
             seed = if useRandomSeed then SeedGenerator.clockMS else 314159265;

//
// Configuration constants to control what's printed -- benchmark
// parameters, input and output arrays, and/or statistics
//
config const printParams = true,
             printArrays = false,
             printStats = true;

//
// The program entry point
//
def main() {
  printConfiguration();   // print the problem size, number of trials, etc.

  //
  // BlockDist is a 1D block distribution that is computed by blocking
  // the bounding box 1..m across the set of locales
  //
  const BlockDist = new Block1D(bbox=[1..m], tasksPerLocale=tasksPerLocale);

  //
  // ProblemSpace describes the index set for the three vectors.  It
  // is a 1D domain storing 64-bit ints and is distributed according
  // to BlockDist.  It contains the indices 1..m.
  //
  const ProblemSpace: domain(1, int(64)) distributed BlockDist = [1..m];

  //
  // A, B, and C are the three distributed vectors, declared to store
  // a variable of type elemType for each index in ProblemSpace.
  //
  var A, B, C: [ProblemSpace] elemType;

  initVectors(B, C);    // Initialize the input vectors, B and C

  var execTime: [1..numTrials] real;                 // an array of timings

  for trial in 1..numTrials {                        // loop over the trials
    const startTime = getCurrentTime();              // capture the start time

    //
    // The main loop: Iterate over the vectors A, B, and C in a
    // parallel, zippered manner storing the elements as a, b, and c.
    // Compute the multiply-add on b and c, storing the result to a.
    //
    forall (a, b, c) in (A, B, C) do
      a = b + alpha * c;

    execTime(trial) = getCurrentTime() - startTime;  // store the elapsed time
  }

  const validAnswer = verifyResults(A, B, C);        // verify...
  printResults(validAnswer, execTime);               // ...and print the results
}

//
// Print the problem size and number of trials
//
def printConfiguration() {
  if (printParams) {
    if (printStats) then printLocalesTasks(tasksPerLocale);
    printProblemSize(elemType, numVectors, m);
    writeln("Number of trials = ", numTrials, "\n");
  }
}

//
// Initialize vectors B and C using a random stream of values and
// optionally print them to the console
//
def initVectors(B, C) {
  var randlist = new RandomStream(seed);

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

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

//
// Verify that the computation is correct
//
def verifyResults(A, B, C) {
  if (printArrays) then writeln("A is:     ", A, "\n");  // optionally print A

  //
  // recompute the computation, destructively storing into B to save space
  //
  forall (b, c) in (B, C) do
    b += alpha *c;

  if (printArrays) then writeln("A-hat is: ", B, "\n");  // and A-hat too

  //
  // Compute the infinity-norm by computing the maximum reduction of the
  // absolute value of A's elements minus the new result computed in B.
  // "[i in I]" represents an expression-level loop: "forall i in I"
  //
  const infNorm = max reduce [(a,b) in (A,B)] abs(a - b);

  return (infNorm <= epsilon);    // return whether the error is acceptable
}

//
// Print out success/failure, the timings, and the GB/s value
//
def printResults(successful, execTimes) {
  writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
  if (printStats) {
    const totalTime = + reduce execTimes,
          avgTime = totalTime / numTrials,
          minTime = min reduce execTimes;
    writeln("Execution time:");
    writeln("  tot = ", totalTime);
    writeln("  avg = ", avgTime);
    writeln("  min = ", minTime);

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

Appendix B
Random Access in Chapel

A. Random Access: Benchmark Module

```
//
// Use standard modules for Block distributions and Timing routines
//
use BlockDist, Time;

//
// Use the user modules for computing HPCC problem sizes and for
// defining RA's random stream of values
//
use HPCCProblemSize, RARandomStream;

//
// The number of tables as well as the element and index types of
// that table
//
const numTables = 1;
type elemType = randType,
     indexType = randType;

//
// Configuration constants defining log2(problem size) -- n -- and
// the number of updates -- N_U
//
config const n = computeProblemSize(numTables, elemType,
                                    returnLog2=true, retType=indexType),
             N_U = 2**(n+2);

//
// Constants defining the problem size (m) and a bit mask for table
// indexing
//
const m = 2**n,
      indexMask = m-1;

//
// Configuration constant defining the number of errors to allow (as a
// fraction of the number of updates, N_U)
//
config const errorTolerance = 1e-2;

//
// The number of tasks to use per Chapel locale
//
config const tasksPerLocale = min reduce Locales.numCores;

//
// Configuration constants to control what's printed -- benchmark
// parameters, input and output arrays, and/or statistics
//
config const printParams = true,
             printArrays = false,
             printStats = true;

//
// TableDist is a 1D block distribution for domains storing indices
// of type "indexType", and it is computed by blocking the bounding
// box 0..m-1 across the set of locales.  UpdateDist is a similar
// distribution that is computed by blocking the indices 0..N_U-1
// across the locales.
//
const TableDist = new Block1D(indexType, bbox=[0..m-1],
                              tasksPerLocale=tasksPerLocale),
      UpdateDist = new Block1D(indexType, bbox=[0..N_U-1],
                               tasksPerLocale=tasksPerLocale);

//
// TableSpace describes the index set for the table.  It is a 1D
// domain storing indices of type indexType, it is distributed
// according to TableDist, and it contains the indices 0..m-1.
// Updates is an index set describing the set of updates to be made.
// It is distributed according to UpdateDist and contains the
// indices 0..N_U-1.
//
const TableSpace: domain(1, indexType) distributed TableDist = [0..m-1],
      Updates: domain(1, indexType) distributed UpdateDist = [0..N_U-1];

//
// T is the distributed table itself, storing a variable of type
// elemType for each index in TableSpace.
//
var T: [TableSpace] elemType;

//
// The program entry point
//
def main() {
  printConfiguration();   // print the problem size, number of trials, etc.

  //
  // In parallel, initialize the table such that each position
  // contains its index.  "[i in TableSpace]" is shorthand for "forall
  // i in TableSpace"
  //
  [i in TableSpace] T(i) = i;

  const startTime = getCurrentTime();             // capture the start time

  //
  // The main computation: Iterate over the set of updates and the
  // stream of random values in a parallel, zippered manner, dropping
  // the update index on the ground ("_") and storing the random value
  // in r.  Use an on-clause to force the table update to be executed on
  // the locale which owns the table element in question to minimize
  // communications.  Compute the update using r both to compute the
  // index and as the update value.
  //
  forall (_, r) in (Updates, RAStream()) do
    on T.domain.dist.ind2loc(r & indexMask) do
      T(r & indexMask) ^= r;

  const execTime = getCurrentTime() - startTime;   // capture the elapsed time

  const validAnswer = verifyResults();             // verify the updates
  printResults(validAnswer, execTime);             // print the results
}

//
// Print the problem size and number of updates
//
def printConfiguration() {
  if (printParams) {
    if (printStats) then printLocalesTasks(tasksPerLocale);
    printProblemSize(elemType, numTables, m);
    writeln("Number of updates = ", N_U, "\n");
  }
}

//
// Verify that the computation is correct
//
def verifyResults() {
  //
  // Print the table, if requested
  //
  if (printArrays) then writeln("After updates, T is: ", T, "\n");

  //
  // Reverse the updates by recomputing them, this time using an
  // atomic statement to ensure no conflicting updates
  //
  forall (_, r) in (Updates, RAStream()) do
    on T.domain.dist.ind2loc(r & indexMask) do
      atomic T(r & indexMask) ^= r;

  //
  // Print the table again after the updates have been reversed
  //
  if (printArrays) then writeln("After verification, T is: ", T, "\n");

  //
  // Compute the number of table positions that weren't reverted
  // correctly.  This is an indication of the number of conflicting
  // updates.
  //
  const numErrors = + reduce [i in TableSpace] (T(i) != i);
  if (printStats) then writeln("Number of errors is: ", numErrors, "\n");

  //
  // Return whether or not the number of errors was within the benchmark's
  // tolerance.
  //
  return numErrors <= (errorTolerance * N_U);
}

//
// Print out success/failure, the execution time, and the GUPS value
//
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

```
//
// A helper module for the RA benchmark that defines the random stream
// of values
//
module RARandomStream {
  param randWidth = 64;              // the bit-width of the random numbers
  type randType = uint(randWidth);   // the type of the random numbers

  //
  // bitDom is a non-distributed domain whose indices correspond to
  // the bit positions in the random values.  m2 is a table of helper
  // values used to fast-forward through the random stream.
  //
  const bitDom = [0..#randWidth],
        m2: [bitDom] randType = computeM2Vals(randWidth);

  //
  // A serial iterator for the random stream that resets the stream
  // to its 0th element and yields values endlessly.
  //
  def RAStream() {
    var val = getNthRandom(0);
    while (1) {
      getNextRandom(val);
      yield val;
    }
  }

  //
  // A "follower" iterator for the random stream that takes a range of
  // 0-based indices (follower) and yields the pseudo-random values
  // corresponding to those indices.  Follower iterators like these
  // are required for parallel zippered iteration.
  //
  def RAStream(param tag: iterator, follower) where tag == iterator.follower {
    var val = getNthRandom(follower.low);
    for follower {
      getNextRandom(val);
      yield val;
    }
  }

  //
  // A helper function for "fast-forwarding" the random stream to
  // position n in O(log2(n)) time
  //
  def getNthRandom(in n: uint(64)) {
    param period = 0x7fffffffffffffff/7;

    n %= period;
    if (n == 0) then return 0x1;
    var ran: randType = 0x2;
    for i in 0..log2(n)-1 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;
  }

  //
  // A helper function for advancing a value from the random stream,
  // x, to the next value
  //
  def getNextRandom(inout x) {
    param POLY = 0x7;
    param hiRandBit = 0x1:randType << (randWidth-1);

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

  //
  // A helper function for computing the values of the helper array,
  // m2
  //
  def computeM2Vals(numVals) {
    var nextVal = 0x1: randType;
    for i in 1..numVals {
      yield nextVal;
      getNextRandom(nextVal);
      getNextRandom(nextVal);
    }
  }
}
```

Appendix C
FFT in Chapel

```
//
// Use standard modules for Bit operations, Random numbers, and Timing
//
use BitOps, Random, Time;

//
// Use shared user module for computing HPCC problem sizes
//
use HPCCProblemSize;

const radix = 4;               // the radix of this FFT implementation

const numVectors = 2;          // the number of vectors to be stored
type elemType = complex(128);  // the element type of the vectors

//
// A configuration constant defining log2(problem size) -- n -- and a
// constant defining the problem size itself -- m
//
config const n = computeProblemSize(numVectors, elemType, returnLog2 = true);
const m = 2**n;

//
// Configuration constants defining the epsilon and threshold values
// used to verify the result
//
config const epsilon = 2.0 ** -51.0,
             threshold = 16.0;

//
// Configuration constants to indicate whether or not to use a
// pseudo-random seed (based on the clock) or a fixed seed; and to
// specify the fixed seed explicitly
//
config const useRandomSeed = true,
             seed = if useRandomSeed then SeedGenerator.clockMS else 314159265;

//
// Configuration constants to control what's printed -- benchmark
// parameters, input and output arrays, and/or statistics
//
config const printParams = true,
             printArrays = false,
             printStats = true;

//
// The program entry point
//
def main() {
  printConfiguration();          // print the problem size

  //
  // TwiddleDom describes the index set used to define the vector of
  // twiddle values and is a 1D domain indexed by 64-bit ints from 0
  // to m/4-1.  Twiddles is the vector of twiddle values.
  //
  const TwiddleDom: domain(1, int(64)) = [0..m/4-1];
  var Twiddles: [TwiddleDom] elemType;

  //
  // ProblemDom describes the index set used to define the input and
  // output vectors and is also a 1D domain indexed by 64-bit ints
  // from 0 to m-1.  Z and z are the vectors themselves
  //
  const ProblemDom: domain(1, int(64)) = [0..m-1];
  var Z, z: [ProblemDom] elemType;

  initVectors(Twiddles, z);            // initialize twiddles and input vector z

  const startTime = getCurrentTime();  // capture the start time

  Z = conjg(z);                        // store the conjugate of z in Z
  bitReverseShuffle(Z);                // permute Z
  dfft(Z, Twiddles);                   // compute the discrete Fourier transform

  const execTime = getCurrentTime() - startTime;     // store the elapsed time

  const validAnswer = verifyResults(z, Z, Twiddles); // validate the answer
  printResults(validAnswer, execTime);               // print the results
}

//
// compute the discrete fast Fourier transform of a vector A declared
// over domain ADom using twiddle vector W
//
def dfft(A: [?ADom], W) {
  const numElements = A.numElements;

  //
  // loop over the phases of the DFT sequentially using custom
  // iterator genDFTStrideSpan that yields the stride and span for
  // each bank of butterfly calculations
  //
  for (str, span) in genDFTStrideSpan(numElements) {
    //
    // loop in parallel over each of the banks of butterflies with
    // shared twiddle factors, zippering with the unbounded range
    // 0.. to get the base twiddle indices
    //
    forall (bankStart, twidIndex) in (ADom by 2*span, 0..) {
      //
      // compute the first set of multipliers for the low bank
      //
      var wk2 = W(twidIndex),
          wk1 = W(2*twidIndex),
          wk3 = (wk1.re - 2 * wk2.im * wk1.im,
                 2 * wk2.im * wk1.re - wk1.im):elemType;

      //
      // loop in parallel over the low bank, computing butterflies
      // Note: lo..#num         == lo, lo+1, lo+2, ..., lo+num-1
      //       lo.. by str #num == lo, lo+str, lo+2*str, ... lo+(num-1)*str
      //
      forall lo in bankStart..#str do
        butterfly(wk1, wk2, wk3, A[lo.. by str #radix]);

      //
      // update the multipliers for the high bank
      //
      wk1 = W(2*twidIndex+1);
      wk3 = (wk1.re - 2 * wk2.re * wk1.im,
             2 * wk2.re * wk1.re - wk1.im):elemType;
      wk2 *= 1.0i;

      //
      // loop in parallel over the high bank, computing butterflies
      //
      forall lo in bankStart+span..#str do
        butterfly(wk1, wk2, wk3, A[lo.. by str #radix]);
    }
  }

  //
  // Do the last set of butterflies...
  //
  const str = radix**log4(numElements-1);
  //
  // ...using the radix-4 butterflies with 1.0 multipliers if the
  // problem size is a power of 4
  //
  if (str*radix == numElements) then
    forall lo in 0..#str do
      butterfly(1.0, 1.0, 1.0, A[lo.. by str #radix]);
  //
  // ...otherwise using a simple radix-2 butterfly scheme
  //
  else
    forall lo in 0..#str {
      const a = A(lo),
            b = A(lo+str);
      A(lo)     = a + b;
      A(lo+str) = a - b;
    }
}

//
// this is the radix-4 butterfly routine that takes multipliers wk1,
// wk2, and wk3 and a 4-element array (slice) A.
//
def butterfly(wk1, wk2, wk3, A) {
  var X: [0..#radix] elemType = A;  // make a local copy of A on this locale
  var x0 = X(0) + X(1),
      x1 = X(0) - X(1),
      x2 = X(2) + X(3),
      x3rot = (X(2) - X(3))*1.0i;

  X(0) = x0 + x2;                   // compute the butterfly in-place on X
  x0 -= x2;
  X(2) = wk2 * x0;
  x0 = x1 + x3rot;
  X(1) = wk1 * x0;
  x0 = x1 - x3rot;
  X(3) = wk3 * x0;

  A = X;                            // copy the result back into A
}

//
// this iterator generates the stride and span values for the phases
// of the DFFT simply by yielding tuples: (radix**i, radix**(i+1))
//
def genDFTStrideSpan(numElements) {
  var stride = 1;
  for 1..log4(numElements-1) {
    const span = stride * radix;
    yield (stride, span);
    stride = span;
  }
}

//
// Print the problem size
//
def printConfiguration() {
  if (printParams) {
    if (printStats) then printLocalesTasks(tasksPerLocale=1);
    printProblemSize(elemType, numVectors, m);
  }
}

//
// Initialize the twiddle vector and random input vector and
// optionally print them to the console
//
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");
  }
}

//
```

```
// Compute the twiddle vector values
//
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/2-1 {
    const x = cos(delta*i),
          y = sin(delta*i);
    Twiddles(i)            = (x, y): elemType;
    Twiddles(numTwdls - i) = (y, x): elemType;
  }
}

//
// Perform a permutation of the argument vector by reversing the bits
// of the indices
//
def bitReverseShuffle(Vect: [?Dom]) {
  const numBits = log2(Vect.numElements),
        Perm: [i in Dom] Vect.eltType = Vect(bitReverse(i, revBits=numBits));
  Vect = Perm;
}

//
// Reverse the low revBits bits of val
//
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;
}

//
// Compute the log base 4 of x
//
def log4(x) return logBasePow2(x, 2);

//
// verify that the results are correct by reapplying the dfft and then
// calculating the maximum error, comparing against epsilon
//
def verifyResults(z, Z, Twiddles) {
  if (printArrays) then writeln("After FFT, Z is: ", Z, "\n");

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

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

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

  return (maxerr < threshold);
}

//
// print out sucess/failure, the timing, and the Gflop/s value
//
def printResults(successful, execTime) {
  writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
  if (printStats) {
    writeln("Execution time = ", execTime);
    writeln("Performance (Gflop/s) = ", 5 * (m * n / execTime) * 1e-9);
  }
}
```

Appendix D
HPL in Chapel

```
//
// Use standard modules for vector and matrix Norms, Random numbers
// and Timing routines
//
use Norm, Random, Time;

//
// Use the user module for computing HPCC problem sizes
//
use HPCCProblemSize;

//
// The number of matrices and the element type of those matrices
//
const numMatrices = 1;
type indexType = int,
     elemType = real;

//
// Configuration constants indicating the problem size (n) and the
// block size (blkSize)
//
config const n = computeProblemSize(numMatrices, elemType, rank=2,
                                    memFraction=2, retType=indexType),
             blkSize = 5;

//
// Configuration constant used for verification thresholds
//
config const epsilon = 2.0e-15;

//
// Configuration constants to indicate whether or not to use a
// pseudo-random seed (based on the clock) or a fixed seed; and to
// specify the fixed seed explicitly
//
config const useRandomSeed = true,
             seed = if useRandomSeed then SeedGenerator.clockMS else 31415;

//
// Configuration constants to control what's printed -- benchmark
// parameters, input and output arrays, and/or statistics
//
config const printParams = true,
             printArrays = false,
             printStats = true;

//
// The program entry point
//
def main() {
  printConfiguration();

  //
  // MatVectSpace is a 2D domain of type indexType that represents the
  // n x n matrix adjacent to the column vector b.  MatrixSpace is a
  // subdomain that is created by slicing into MatVectSpace,
  // inheriting all of its rows and its low column bound.  As our
  // standard distribution library is filled out, MatVectSpace will be
  // distributed using a BlockCyclic(blkSize) distribution.
  //
  const MatVectSpace: domain(2, indexType) = [1..n, 1..n+1],
        MatrixSpace = MatVectSpace[.., ..n];

  var Ab : [MatVectSpace] elemType,  // the matrix A and vector b
      piv: [1..n] indexType,         // a vector of pivot values
      x  : [1..n] elemType;          // the solution vector, x

  var A => Ab[MatrixSpace],          // an alias for the Matrix part of Ab
      b => Ab[.., n+1];              // an alias for the last column of Ab

  initAB(Ab);

  const startTime = getCurrentTime();     // capture the start time

  LUFactorize(n, Ab, piv);                 // compute the LU factorization

  x = backwardSub(n, A, b);  // perform the back substitution

  const execTime = getCurrentTime() - startTime;  // store the elapsed time

  //
  // Validate the answer and print the results
  const validAnswer = verifyResults(Ab, MatrixSpace, x);
  printResults(validAnswer, execTime);
}

//
// blocked LU factorization with pivoting for matrix augmented with
// vector of RHS values.
//
def LUFactorize(n: indexType, Ab: [1..n, 1..n+1] elemType,
                piv: [1..n] indexType) {
  const AbD = Ab.domain;    // alias Ab.domain to save typing

  // Initialize the pivot vector to represent the initially unpivoted matrix.
  piv = 1..n;

  /* The following diagram illustrates how we partition the matrix.
     Each iteration of the loop increments a variable blk by blkSize;
     point (blk, blk) is the upper-left location of the currently
     unfactored matrix (the dotted region represents the areas
     factored in prior iterations).  The unfactored matrix is
     partioned into four subdomains: tl, tr, bl, and br, and an
     additional domain (not shown), l, that is the union of tl and bl.

                (point blk, blk)
       +-------//------------------+
       |......//...................|
       |.....//....................|
       |....+-----+----------------|
       |....|     |                |
       |....| tl  |      tr        |
       |....|     |                |
       |....+-----+----------------|
       |....|     |                |
       |....|     |                |
       |....| bl  |      br        |
       |....|     |                |
       |....|     |                |
       +----+-----+----------------+
  */
  for blk in 1..n by blkSize {
    const tl = AbD[blk..#blkSize, blk..#blkSize],
          tr = AbD[blk..#blkSize, blk+blkSize..],
          bl = AbD[blk+blkSize.., blk..#blkSize],
          br = AbD[blk+blkSize.., blk+blkSize..],
          l  = AbD[blk.., blk..#blkSize];

    //
    // Now that we've sliced and diced Ab properly, do the blocked-LU
    // computation:
    //
    panelSolve(Ab, l, piv);
    if (tr.numIndices > 0) then
      updateBlockRow(Ab, tl, tr);

    //
    // update trailing submatrix (if any)
    //
    if (br.numIndices > 0) then
      schurComplement(Ab, blk);
  }
}

//
// Distributed matrix-multiply for HPL. The idea behind this algorithm is that
// some point the matrix will be partioned as shown in the following diagram:
//
//    [1]----+-----+-----+-----+
//     |     |bbbbb|bbbbb|bbbbb|  Solve for the dotted region by
//     |     |bbbbb|bbbbb|bbbbb|  multiplying the 'a' and 'b' region.
//     |     |bbbbb|bbbbb|bbbbb|  The 'a' region is a block column, the
//     +----[2]----+-----+-----+  'b' region is a block row.
//     |aaaaa|.....|.....|.....|
//     |aaaaa|.....|.....|.....|  The vertex labeled [1] is location
//     |aaaaa|.....|.....|.....|  (ptOp, ptOp) in the code below.
//     +-----+-----+-----+-----+
//     |aaaaa|.....|.....|.....|  The vertex labeled [2] is location
//     |aaaaa|.....|.....|.....|  (ptSol, ptSol)
//     |aaaaa|.....|.....|.....|
//     +-----+-----+-----+-----+
//
// Every locale with a block of data in the dotted region updates
// itself by multiplying the neighboring a-region block to its left
// with the neighboring b-region block above it and subtracting its
// current data from the result of this multiplication. To ensure that
// all locales have local copies of the data needed to perform this
// multiplication we copy the data A and B data into the replA and
// replB arrays, which will use a dimensional (block-cyclic,
// replicated-block) distribution (or vice-versa) to ensure that every
// locale only stores one copy of each block it requires for all of
// its rows/columns.
//
def schurComplement(Ab: [1..n, 1..n+1] elemType, ptOp: indexType) {
  const AbD = Ab.domain;

  //
  // Calculate location of ptSol (see diagram above)
  //
  const ptSol = ptOp+blkSize;

  //
  // Copy data into replicated array so every processor has a local copy
  // of the data it will need to perform a local matrix-multiply.  These
  // replicated distributions aren't implemented yet, but imagine that
  // they look something like the following:
  //
  //var replAbD: domain(2)
  //            distributed new Dimensional(BlkCyc(blkSize), Replicated))
  //          = AbD[ptSol.., 1..#blkSize];
  //
  const replAD: domain(2) = AbD[ptSol.., ptOp..#blkSize],
        replBD: domain(2) = AbD[ptOp..#blkSize, ptSol..];

  const replA : [replAD] elemType = Ab[ptSol.., ptOp..#blkSize],
        replB : [replBD] elemType = Ab[ptOp..#blkSize, ptSol..];

  // do local matrix-multiply on a block-by-block basis
  forall (row,col) in AbD[ptSol.., ptSol..] by (blkSize, blkSize) {
    //
    // At this point, the dgemms should all be local, so assert that
    // fact
    //
    local {
      const aBlkD = replAD[row..#blkSize, ptOp..#blkSize],
            bBlkD = replBD[ptOp..#blkSize, col..#blkSize],
            cBlkD = AbD[row..#blkSize, col..#blkSize];

      dgemm(aBlkD.dim(1).length,
            aBlkD.dim(2).length,
            bBlkD.dim(2).length,
            replA(aBlkD),
            replB(bBlkD),
            Ab(cBlkD));
    }
  }
```

```
}

//
// calculate C = C - A * B.
//
def dgemm(p: indexType,       // number of rows in A
          q: indexType,       // number of cols in A, number of rows in B
          r: indexType,       // number of cols in B
          A: [1..p, 1..q] ?t,
          B: [1..q, 1..r] t,
          C: [1..p, 1..r] t) {
  // Calculate (i,j) using a dot product of a row of A and a column of B.
  for i in 1..p do
    for j in 1..r do
      for k in 1..q do
        C[i,j] -= A[i, k] * B[k, j];
}

//
// do unblocked-LU decomposition within the specified panel, update the
// pivot vector accordingly
//
def panelSolve(Ab: [] ?t,
               panel: domain(2, indexType),
               piv: [] indexType) {
  const pnlRows = panel.dim(1),
        pnlCols = panel.dim(2);

  //
  // Ideally some type of assertion to ensure panel is embedded in Ab's
  // domain
  //
  assert(piv.domain.dim(1) == Ab.domain.dim(1));

  if (pnlCols.length == 0) then return;

  for k in pnlCols {             // iterate through the columns
    var col = panel[k.., k..k];

    // If there are no rows below the current column return
    if col.dim(1).length == 0 then return;

    // Find the pivot, the element with the largest absolute value.
    const (_, (pivotRow, _)) = maxloc reduce(abs(Ab(col)), col),
          pivot = Ab[pivotRow, k];

    // Swap the current row with the pivot row
    piv[k] <=> piv[pivotRow];

    Ab[k, ..] <=> Ab[pivotRow, ..];

    if (pivot == 0) then
      halt("Matrix can not be factorized");

    // divide all values below and in the same col as the pivot by
    // the pivot
    if k+1 <= pnlRows.high then
      Ab(col)[k+1.., k..k] /= pivot;

    // update all other values below the pivot
    if k+1 <= pnlRows.high && k+1 <= pnlCols.high then
      forall (i,j) in panel[k+1.., k+1..] do
        Ab[i,j] -= Ab[i,k] * Ab[k,j];
  }
}

//
// Update the block row (tr for top-right) portion of the matrix in a
// blocked LU decomposition.  Each step of the LU decomposition will
// solve a block (tl for top-left) portion of a matrix. This function
// solves the rows to the right of the block.
//
def updateBlockRow(Ab: [] ?t, tl: domain(2), tr: domain(2)) {
  const tlRows = tl.dim(1),
        tlCols = tl.dim(2),
        trRows = tr.dim(1),
        trCols = tr.dim(2);

  assert(tlCols == trRows);

  //
  // Ultimately, we will probably want to do some replication of the
  // tl block in order to make this operation completely localized as
  // in the dgemm.  We have not yet undertaken that optimization.
  //
  for i in trRows do
    forall j in trCols do
      for k in tlRows.low..i-1 do
        Ab[i, j] -= Ab[i, k] * Ab[k,j];
}

//
// compute the backwards substitution
//
def backwardSub(n: int,
                A: [1..n, 1..n] elemType,
                b: [1..n] elemType) {
  var x: [b.domain] elemType;

  for i in [b.domain by -1] {
    x[i] = b[i];

    for j in [i+1..b.domain.high] do
      x[i] -= A[i,j] * x[j];

    x[i] /= A[i,i];
  }

  return x;
}

//
// print out the problem size and block size if requested
//
def printConfiguration() {
  if (printParams) {
    if (printStats) then printLocalesTasks(tasksPerLocale=1);
    printProblemSize(elemType, numMatrices, n, rank=2);
    writeln("block size = ", blkSize, "\n");
  }
}

//
// construct an n by n+1 matrix filled with random values and scale
// it to be in the range -1.0..1.0
//
def initAB(Ab: [] elemType) {
  fillRandom(Ab, seed);
  Ab = Ab * 2.0 - 1.0;
}

//
// calculate norms and residuals to verify the results
//
def verifyResults(Ab, MatrixSpace, x) {
  var A => Ab[MatrixSpace],
      b => Ab[.., n+1];

  initAB(Ab);

  const axmbNorm = norm(gaxpyMinus(n, n, A, x, b), normType.normInf);

  const a1norm   = norm(A, normType.norm1),
        aInfNorm = norm(A, normType.normInf),
        x1Norm   = norm(x, normType.norm1),
        xInfNorm = norm(x, normType.normInf);

  const resid1 = axmbNorm / (epsilon * a1norm * n),
        resid2 = axmbNorm / (epsilon * a1norm * x1Norm),
        resid3 = axmbNorm / (epsilon * aInfNorm * xInfNorm);

  if (printStats) {
    writeln("resid1: ", resid1);
    writeln("resid2: ", resid2);
    writeln("resid3: ", resid3);
  }

  return max(resid1, resid2, resid3) < 16.0;
}

//
// print success/failure, the execution time and the Gflop/s value
//
def printResults(successful, execTime) {
  writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
  if (printStats) {
    writeln("Execution time = ", execTime);
    const GflopPerSec = ((2.0/3.0) * n**3 + (3.0/2.0) * n**2) / execTime * 10e-9;
    writeln("Performance (Gflop/s) = ", GflopPerSec);
  }
}

//
// simple matrix-vector multiplication, solve equation A*x-y
//
def gaxpyMinus(n: indexType,
               m: indexType,
               A: [1..n, 1..m],
               x: [1..m],
               y: [1..n]) {
  var res: [1..n] elemType;

  for i in 1..n do
    for j in 1..m do
      res[i] += A[i,j]*x[j];

  for i in 1..n do
    res[i] -= y[i];

  return res;
}
```

APPENDIX E
HPCC PROBLEM SIZE COMPUTATION IN CHAPEL

```
//
// A shared module for computing the appropriate problem size for the
// HPCC benchmarks
//
module HPCCProblemSize {
  //
  // Use the standard modules for reasoning about Memory and Types
  //
  use Memory, Types;

  //
  // The main routine for computing the problem size
  //
  def computeProblemSize(numArrays: int,    // #arrays in the benchmark
                         type elemType,     // the element type of those arrays
                         rank=1,            // rank of the arrays
                         returnLog2=false,  // whether to return log2(probSize)
                         memFraction=4,     // fraction of mem to use (eg, 1/4)
                         type retType = int(64)): retType { // type to return
    //
    // Compute the total memory available to the benchmark using a sum
    // reduction over the amount of physical memory (in bytes) owned
    // by the set of locales on which we're running.  Then compute the
    // number of bytes we want to use as defined by memFraction and the
    // number that will be required by each index in the problem size.
    //
    const totalMem = + reduce Locales.physicalMemory(unit = MemUnits.Bytes),
          memoryTarget = totalMem / memFraction,
          bytesPerIndex = numArrays * numBytes(elemType);

    //
    // Use these values to compute a base number of indices
    //
    var numIndices = memoryTarget / bytesPerIndex;

    //
    // If the user requested a 2**n problem size, compute appropriate
    // values for numIndices and lgProblemSize
    //
    var lgProblemSize = log2(numIndices);
    if (returnLog2) {
      if rank != 1 then
        halt("computeProblemSize() can't compute 2D 2**n problem sizes yet");
      numIndices = 2**lgProblemSize;
      if (numIndices * bytesPerIndex <= memoryTarget) {
        numIndices *= 2;
        lgProblemSize += 1;
      }
    }

    //
    // Compute the smallest amount of memory that any locale owns
    // using a min reduction and ensure that it is sufficient to hold
    // an even portion of the problem size.
    //
    const smallestMem = min reduce Locales.physicalMemory(unit = MemUnits.Bytes);
    if ((numIndices * bytesPerIndex)/numLocales > smallestMem) then
      halt("System is too heterogeneous: blocked data won't fit into memory");

    //
    // return the problem size as requested by the callee
    //
    if returnLog2 then
      return lgProblemSize: retType;
    else
      select rank {
        when 1 do return numIndices: retType;
        when 2 do return ceil(sqrt(numIndices)): retType;
        otherwise halt("Unexpected rank in computeProblemSize");
      }
  }

  //
  // Print out the machine configuration used to run the job
  //
  def printLocalesTasks(tasksPerLocale=1) {
    writeln("Number of Locales = ", numLocales);
    writeln("Tasks per locale = ", tasksPerLocale);
  }

  //
  // Print out the problem size, #bytes per array, and total memory
  // required by the arrays
  //
  def printProblemSize(type elemType, numArrays, problemSize: ?psType,
                       param rank=1) {
    const bytesPerArray = problemSize**rank * numBytes(elemType),
          totalMemInGB = (numArrays * bytesPerArray:real) / (1024**3),
          lgProbSize = log2(problemSize):psType;

    write("Problem size = ", problemSize);
    for i in 2..rank do write(" x ", problemSize);
    if (2**lgProbSize == problemSize) {
      write(" (2**", lgProbSize);
      for i in 2..rank do write(" x 2**", lgProbSize);
      write(")");
    }
    writeln();
    writeln("Bytes per array = ", bytesPerArray);
    writeln("Total memory required (GB) = ", totalMemInGB);
  }
}
```