# HPC Challenge Benchmarks in Chapel*

Bradford L. Chamberlain     Sung-Eun Choi     Steven J. Deitz     David Iten

Cray Inc.
chapel_info@cray.com

**Abstract**

This report presents our best to-date Chapel implementations of the global HPC Challenge benchmarks STREAM Triad, Random Access, FFT, and HPL. They improve upon our 2008 Chapel submission using the same hardware. The highlights of this year's submission include:

- Global STREAM Triad performance of 10.8 TB/s on 8192 cores of a Cray XT4 (up from 1.69 TB/s)
- EP STREAM Triad performance of 12.2 TB/s on 8192 cores of a Cray XT4.
- Random Access performance of 0.122 GUP/s on 8192 cores of a Cray XT4 (up from 0.0011 GUP/s)
- A first distributed implementation of FFT which makes use of two distinct distributions: Block and Cyclic.
- A demonstration of Chapel's portability on a Cray XT4, a Cray CX1, and an IBM pSeries 575.
- The Chapel compiler and these benchmarks are publicly available at http://sourceforge.net/projects/chapel.

All codes in this report compile and execute correctly with version 1.0.2 of the Chapel compiler. All reported Chapel performance results were obtained with this version of the Chapel compiler. The full code listings are provided in appendices to this report.

## 1 Overview and Contents

Chapel is a new parallel programming language under development at Cray Inc. as part of its participation in DARPA's High Productivity Computing Systems (HPCS) program.[1,2] The goal of the Chapel project is to improve 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 report should be viewed as a snapshot of Chapel's current status.

In this report, we present our best to-date Chapel implementations of four HPC Challenge (HPCC) benchmarks[3,4]—STREAM Triad, Random Access, FFT, and HPL. We summarize the Chapel implementations by categorizing and counting the source lines of code. For each benchmark, we provide a brief overview of our Chapel implementation. For STREAM Triad and Random Access, we present performance results on up to 2048 nodes of a Cray XT4. We provide discussions of our ongoing implementation work for the FFT and HPL benchmarks.

**Contents**

---

*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/

## 2 Code Size Summary

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

| **Code Category** | *Global STREAM Triad* | *EP STREAM Triad* | *Global Random Access* | *Global FFT* | *Global HPL* | *Problem Size* |
|---|---|---|---|---|---|---|
| Computation | 2 | 6 | 7 + 25 = 34 | 75 | 50 | 0 |
| Declarations | 12 | 8 | 20 + 12 = 32 | 32 | 63 | 34 |
| *Total kernel* | *14* | *14* | *27 + 37 = 66* | *107* | *113* | *34* |
| Initialization | 10 | 10 | 1 + 10 = 11 | 26 | 8 | 0 |
| Verification | 8 | 10 | 9 + 0 = 9 | 11 | 16 | 0 |
| Results/output | 32 | 43 | 21 + 0 = 21 | 21 | 39 | 21 |
| ***Total benchmark*** | ***64*** | ***77*** | ***58 + 47 = 107*** | ***165*** | ***176*** | ***55*** |
| Debug/test | 7 | 0 | 3 + 2 = 3 | 5 | 1 | 0 |
| Comments | 72 | 131 | 94 + 31 = 125 | 140 | 170 | 39 |
| Blank | 28 | 29 | 23 + 8 = 31 | 47 | 61 | 8 |
| *Total program* | *171* | *237* | *178 + 88 = 266* | *357* | *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 check sum against the line number labels that appear in the appendices.

# 3 Experimental Setup

Our main experimental platform was the Cray XT4 partition of Jaguar at Oak Ridge National Laboratory (ORNL). The following table provides a brief overview of these platforms:

| *Machine Characteristics* | *XT4* |
|---|---|
| Number of compute nodes | 7,832 |
| Compute node processor | Quad-core AMD Opteron |
| Processor speed | 2.1 GHz |
| Memory per node | 8 GB |

The problem sizes used for STREAM Triad and Random Access were computed by quartering the result returned by Chapel's built-in method `physicalMemory`, which returns the amount of physical memory on a locale. (For Random Access, this was then rounded up to the nearest power of two.) For both benchmarks, this is the smallest possible problem size that exceeds 25% of the total memory as listed in `/proc/meminfo`. These same problem sizes were used for our runs of the MPI and OpenMP reference versions of these benchmarks by brutally inserting these values into the elaborate framework surrounding these codes. The following table specifies the per-node problem sizes used to obtain the results in this report:

| *Problem Sizes Per Node* | *XT4* |
|---|---|
| STREAM Triad Problem Size | 85983914 (1.92 GB) |
| Random Access Problem Size | $2^{28}$ (2.0 GB) |
| Random Access Number of Updates | $2^{18}$ |

Note that we execute a reduced number of updates ($2^{n-10}$ instead of $2^{n+2}$ where $n$ is large enough to create a table that uses 25% of the total system memory) for Random Access due to long execution times. The GUP/s rate does not appear to be affected by this change.

The Chapel compiler is a source-to-C translator that invokes a C compiler to create an executable. We used the same C compiler, with the same flags, to finish the Chapel compilation and to compile the MPI and OpenMP reference implementations of the benchmarks. The Chapel runtime uses POSIX threads (*pthreads*) to implement tasks and Berkeley's GASNet communication library [5] for inter-process coordination and data transfer. The following table specifies software versions and settings used to obtain the results in this report:

| *Software* | *Flags/Settings* | *Version* |
|---|---|---|
| chpl | ––fast | 1.0.2 |
| gcc | –target=linux –O3 –std=c99 –fopenmp<br>––param max-inline-insns-single=35000<br>––param inline-unit-growth=10000<br>––param large-function-growth=200000 | 4.3.2 |
| GASNet | conduit=mpi, segment=fast | 1.12.0 |

On the Cray XT4, we used Cray's *PrgEnv-gnu* programming environment module which provides a Cray C compiler wrapper around gcc.

The Chapel flag "––fast" turns off a number of runtime checks that are enabled by default for safety, including checks for 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 primarily used for GASNet portability. We did not use the portals conduit due to a combination of GASNet and Chapel bugs. Since the portals conduit is chosen by the Chapel system by default, we explicitly set the environment variable `CHPL_COMM_SUBSTRATE` to `mpi` and `CHPL_GASNET_SEGMENT` to `fast`.

---

[5] For details on GASNet, refer to the GASNet specification.

# 4 Global STREAM Triad Description

The core of the global version of STREAM Triad in Chapel is unchanged from last year:

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

This pair of lines specifies parallel, element-wise iteration over the vectors $A$, $B$, and $C$, referring to corresponding elements as $a$, $b$, and $c$ in the loop body.

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

The first declaration[6]

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

creates a distribution $BlockDist$ that maps indices across the entire set of *locales*[7] according to the implementation of the *Block* class. This mapping is computed by partitioning the specified bounding box, $1 \dots m$, across the locales using blocks of approximately equal sizes. The $tasksPerLocale$ argument, which is passed the value of a *configuration constant* of the same name, specifies how many tasks should be used on each locale to implement parallel loops over the distribution's domains and arrays.

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. It is initialized to store the index set $1 \dots m$ which will be divided between the locales according to the mapping defined by $BlockDist$.

The third declaration

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

creates the 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). The elements of the vectors are mapped to the same locales as the corresponding indices in the $ProblemSpace$ domain.

Chapel distributions like *Block* 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 computation—down to the individual data structures and tasks that will implement the computation across the locales. In the case of a forall loop like this one, the compiler rewrites the loop using multiple iterators defined by the distribution which specify how parallel, element-wise 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. The compiler contains no semantic knowledge specific to the *Block* distribution, only that distributions implement a structural interface that it can target when lowering and optimizing high-level operations. This lack of distribution-specific knowledge is the backbone of our plan to support user-defined distributions and a large collection of provided distributions.

---

[6] The current distribution syntax is temporary, pending a better way to distinguish between the class that implements a distribution and the distribution value over which a domain is declared.

[7] A Chapel *locale* is an architectural unit of locality. 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 the Cray XT4, it is a single quadcore node.

# 5 EP STREAM Triad Description

New to this year's Chapel entry, we present both EP (embarrassingly parallel) and global versions of the STREAM Triad benchmark. The global version is far more elegant in Chapel due to its support for a global-view programming model. Nevertheless, Chapel's multi-level design allows us to fragment execution across the locales and implement an EP version of the STREAM Triad benchmark as well.

The 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 Chapel's support for multiple levels of design is of the utmost importance for high-level languages that seek to support both programmability and performance, if for no other reason than to enable programmers to work around cases where the compiler or high-level abstractions fail them.

The core of the EP version of STREAM Triad uses a *coforall* loop over all of the locales to create a new task per locale. When coupled with the *on* statement to specify where these tasks should execute, the following loop creates remote task on each locale:

```
coforall loc in Locales do on loc {
```

Within this loop, we can create non-distributed arrays to contain the locale's portion of the STREAM vectors as follows:

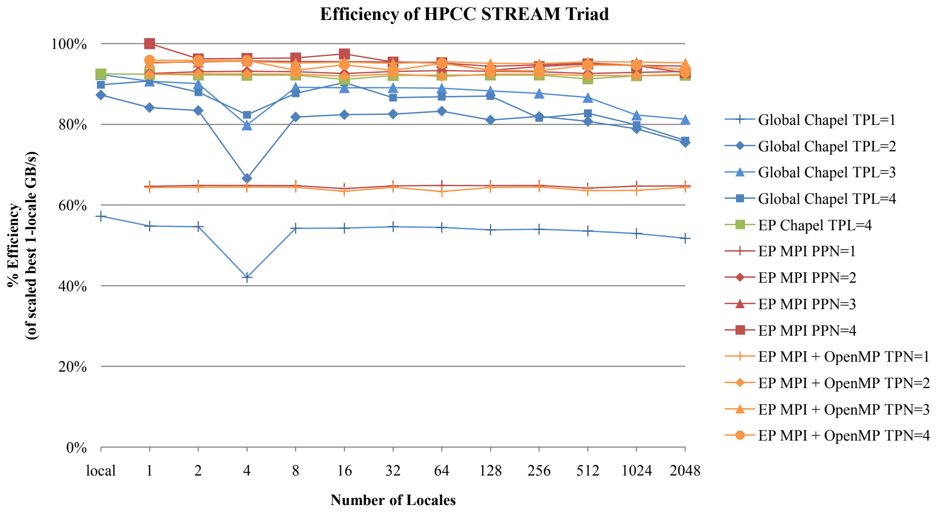
```
var A, B, C: [1..m] elemType;
```

For this version of the benchmark, like the reference version, $m$ refers to a per-locale problem size. The actual computation looks identical to the computation in the global version above:

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

In our implementation of this benchmark, we use a *local* block to guarantee that there will be no communication within this computation. This results in a faster implementation with our current version of the Chapel compiler.

# 6 STREAM Triad Performance



| | *Global Chapel* | | | | *EP Chapel* | *EP MPI* | | | | *EP MPI + OpenMP* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Locales* | *1 TPL* | *2 TPL* | *3 TPL* | *4 TPL* | *4 TPL* | *1 PPN* | *2 PPN* | *3 PPN* | *4 PPN* | *1 TPN* | *2 TPN* | *3 TPN* | *4 TPN* |
| local | 3.80 | 5.80 | 6.14 | 5.97 | 6.15 | | | | | | | | |
| 1 | 3.64 | 5.59 | 6.03 | 6.03 | 6.15 | 4.30 | 6.16 | 6.34 | 6.65 | 4.28 | 6.14 | 6.32 | 6.38 |
| 2 | 7.26 | 11.09 | 11.97 | 11.69 | 12.26 | 8.62 | 12.37 | 12.69 | 12.79 | 8.56 | 12.30 | 12.71 | 12.74 |
| 4 | 11.19 | 17.71 | 21.23 | 21.90 | 24.51 | 17.24 | 24.77 | 25.45 | 25.62 | 17.14 | 24.59 | 25.43 | 25.48 |
| 8 | 28.85 | 43.51 | 47.44 | 46.62 | 49.04 | 34.46 | 49.50 | 50.83 | 51.29 | 34.26 | 49.17 | 50.63 | 49.65 |
| 16 | 57.70 | 87.63 | 94.74 | 96.11 | 96.95 | 68.13 | 98.48 | 101.67 | 103.64 | 67.43 | 97.78 | 101.51 | 100.83 |
| 32 | 116.22 | 175.60 | 189.52 | 184.23 | 195.89 | 137.71 | 198.02 | 202.99 | 203.06 | 136.96 | 196.70 | 202.37 | 198.60 |
| 64 | 231.61 | 354.28 | 378.55 | 369.46 | 392.33 | 275.89 | 397.03 | 405.22 | 405.39 | 269.54 | 390.95 | 406.62 | 404.65 |
| 128 | 458.08 | 689.86 | 751.34 | 740.20 | 784.63 | 551.10 | 792.50 | 803.33 | 794.76 | 547.72 | 786.69 | 809.55 | 794.79 |
| 256 | 918.85 | 1394.52 | 1492.43 | 1388.25 | 1569.33 | 1103.24 | 1583.39 | 1611.91 | 1605.32 | 1097.32 | 1573.16 | 1617.36 | 1587.71 |
| 512 | 1823.46 | 2747.25 | 2949.44 | 2815.62 | 3105.15 | 2185.09 | 3152.08 | 3239.64 | 3232.55 | 2164.25 | 3133.62 | 3250.04 | 3221.81 |
| 1024 | 3604.22 | 5371.51 | 5602.11 | 5433.64 | 6265.50 | 4401.32 | 6323.29 | 6441.16 | 6436.95 | 4330.40 | 6266.72 | 6498.65 | 6443.33 |
| 2048 | 7046.64 | 10278.50 | 11058.00 | 10346.20 | 12543.12 | 8814.57 | 12677.93 | 12858.80 | 12628.04 | 8767.55 | 12587.24 | 12961.92 | 12725.32 |

This efficiency graph shows the percentage efficiency of the global and EP Chapel versions and the MPI and MPI+OpenMP reference versions of the STREAM Triad benchmark. For the global Chapel version, results are reported for between 1 and 4 tasks per locale (TPL). For the MPI version, results are reported for between 1 and 4 processes per node (PPN). For the MPI+OpenMP version, results are reported for between 1 and 4 threads per node (TPN). The current implementation of Chapel does not allow for varying the number of tasks per locale for non-distributed arrays. The efficiency is taken with respect to linear scaling of the best 1-locale performance (6.64 GB/s for the MPI version with 4 MPI processes per node). The *local* results refer to compiling the Chapel codes with the "––local" flag. This flag optimizes the program for running on a single locale.

We make the following observations:

- No global version can perform as well as an EP version because there is inherent synchronization overhead in the global version that does not exist in the EP version. The global Chapel version shows little overhead and scales very well to 512 locales and reasonably well to 2048 locales.
- The EP Chapel implementation is competitive with the MPI and MPI+OpenMP implementations less a small amount of scalar overhead.
- The scalar overhead is larger for the global Chapel version than the EP Chapel version and larger with a smaller degree of intranode parallelism (especially no intranode parallelism) because the implementation is memory-bound at higher levels.
- The *local* flag removes up to 5% of the scalar overhead in the global Chapel version by eliminating the *coforall* loop over the locales and all "potentially remote" references.
- There is an oddity in the results on 4 locales that is consistent across multiple trials. However, in timings for our original submission, we did not see this behaviour.

The global Chapel implementation scales much better than our last year's entry because we eliminated all communication after the initial remote task invocation. In last year's entry, the remote tasks in the global Chapel version made remote accesses back to locale 0 after being spawned. At high node counts, this bottleneck became a serious issue.

# 7 Random Access Description

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 two *Block* 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.

While the official benchmark 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.

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.

To improve performance, we write the *on*-statement as follows:

```
  on TableDist.ind2loc(r & indexMask) do
```

This expression says "Access $TableDist$, $T$'s distribution, and call its index-to-locale mapping function to determine which locale owns the index `r & indexMask`." With our current compiler, the on-statement with the array access will result in extra communication in order to determine where the array element exists in memory (not just on which locale). As the Chapel compiler and the Block distribution improves, we expect to be able to write the code with the array access and get similar performance.

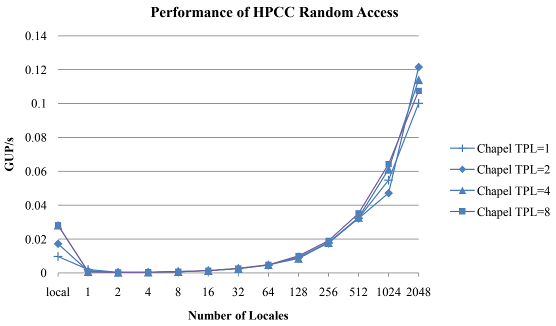
To further improve performance, we introduce a *local*-statement and write the core of the benchmark as follows:

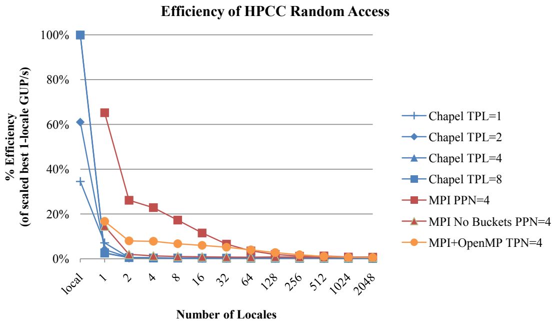
```
  forall (_, r) in (Updates, RAStream()) do
    on TableDist.ind2loc(r & indexMask) do {
      const myR = r;
      local {
        T(myR & indexMask) ^= myR;
      }
   }
```

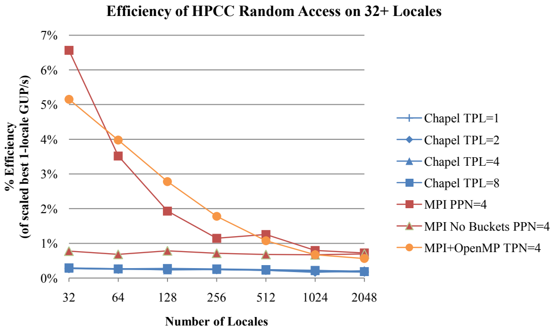
The *local*-block tells the compiler that there will not be any communication within this block. The array access function is cloned as a result. This enables an optimization that allows us to execute the body of the *on*-statement in the GASNet handler. Without this optimization, the compiler has to generate code that will create or reuse a thread on the remote locale. Interestingly, without this optimization, using the *local* keyword does not impact performance because there is a conditional in the array access function that quickly determines the access is local.

Another interesting observation pertains to the declaration of the constant $myR$. This constant is declared on the remote locale so it can be accessed in the *local*-block with impunity. The index $r$, on the other hand, exists on the locale that executes the forall loop. Another optimization ensures that we pass the value of $r$ to the remote locale that executes the *on*-statement, rather than a reference to $r$ which would then require a remote read. This optimization applies because $r$ is not changed on the remote locale and because there is no synchronization with another thread before the value is read. That said, the semantics of the language, and thus the *local*-block, assume that $r$ is not necessarily local and require the copy.

# 8 Random Access Performance







| | *Chapel* | | | | *MPI* | *MPI (No Buckets)* | *MPI + OpenMP* |
|---|---|---|---|---|---|---|---|
| *Locales* | *1 TPL* | *2 TPL* | *4 TPL* | *8 TPL* | *4 PPN* | *4 PPN* | *4 TPN* |
| local | 0.00972525 | 0.0172033 | 0.0281936 | 0.0281603 | | | |
| 1 | 0.00200099 | 0.00116716 | 0.000714489 | 0.000741223 | 0.018384637 | 0.00410575 | 0.004713876 |
| 2 | 0.000317063 | 0.000315868 | 0.000293979 | 0.00032569 | 0.014732936 | 0.001107038 | 0.004503648 |
| 4 | 0.000413006 | 0.000410412 | 0.000423378 | 0.000427791 | 0.025734469 | 0.001485908 | 0.008797149 |
| 8 | 0.000706298 | 0.000710561 | 0.000710417 | 0.000742912 | 0.038924038 | 0.002301621 | 0.015105049 |
| 16 | 0.00125442 | 0.00132351 | 0.00131812 | 0.00137499 | 0.051985597 | 0.003872435 | 0.02702586 |
| 32 | 0.00247275 | 0.00254758 | 0.00255019 | 0.00265785 | 0.059201854 | 0.007007821 | 0.046489749 |
| 64 | 0.00456788 | 0.00463443 | 0.00483206 | 0.00476748 | 0.063466622 | 0.012337323 | 0.07174715 |
| 128 | 0.00930662 | 0.00904224 | 0.00852665 | 0.00999743 | 0.069669644 | 0.028327548 | 0.100320905 |
| 256 | 0.0175101 | 0.0180818 | 0.017865 | 0.0189707 | 0.082709838 | 0.051622158 | 0.128392311 |
| 512 | 0.0323605 | 0.0321982 | 0.032745 | 0.0351792 | 0.181213405 | 0.098146035 | 0.155228153 |
| 1024 | 0.0547629 | 0.0471243 | 0.0611901 | 0.0642561 | 0.229965659 | 0.193492656 | 0.194191687 |
| 2048 | 0.100144 | 0.121527 | 0.114039 | 0.10749 | 0.41723 | 0.402235 | 0.325563 |

The performance graph on top shows the raw performance (GUP/s) of the Chapel version of the Random Access benchmark varying the number of tasks per locale between 1 and 8. The efficiency graphs below show the percentage efficiency of the Chapel version and MPI and MPI+OpenMP reference versions of the Random Access benchmark. For the MPI versions, 4 MPI processes were used per node. For the MPI+Open version, 4 tasks were used per node. The "No Bucket" variation on the MPI reference version shows performance without bucketing. In some ways, this is a purer implementation of random accesses and it also more closely matches the Chapel implementation. The efficiency is taken with respect to linear scaling of the best 1-locale performance (0.028 GUP/s for the Chapel version compiled with the "––local" flag and executed with 4 tasks per node).

We make the following observations:

- On high node counts, the Chapel implementation exhibits about a quarter of the performance of the reference version.
- The reference version with bucketing does not scale because the buckets decrease in size.
- The "––local" flag produces scalar performance that exceeds that of an MPI implementation.

The Chapel implementation performs far better than last year's entry because of eliminated communication (as described in Section 6) and because of the optimization of on-statements described in Section 7.

# 9 FFT Discussion

Although last year's HPCC entry made use of a Block distribution for STREAM and RA, support for slicing was incomplete. Since our Chapel implementation of FFT makes extensive use of array slicing, last year's HPCC FFT entry only executed on a single locale. With array slicing now supported through some Chapel distributions, this year's HPCC FFT entry can run on multiple locales. The Chapel FFT implementation makes use of a localSlice() method that is useful when a slice's elements are known to be local to the current locale. This permits a user to store an alias to the local portion of a distributed array while only paying local array overheads for its operations. This can be thought of, in Fortran terms, as creating a local dope vector for the alias portion of the array.

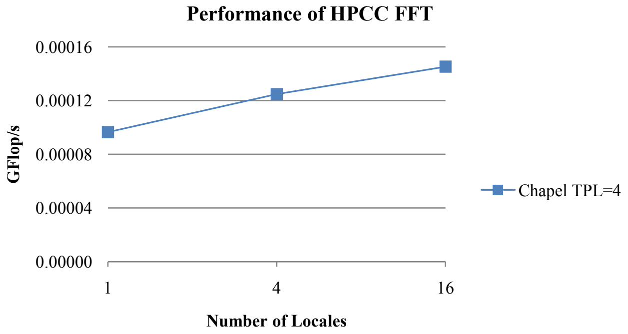
The Chapel FFT implementation makes use of the Cyclic distribution as well as the Block distribution, both standard to Chapel version 1.0.2. The Cyclic distribution maps indices to locales in a round-robin fashion starting from a user-specified index, or 0 by default. The Chapel implementation uses a Block-distributed vector for the first half of the computation and then a Cyclic-distributed vector for the second half. When the number of locales is a power of four, all of the butterflies within the radix-4 implementation only access local data. The only communication in the main computation is in the assignment between the two vectors; this requires all-to-all communication.

The following code excerpts the main structure of the Chapel version of FFT:

```
for (str, span) in genDFTStrideSpan(numElements, cyclicPhase) {
  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..#str do
      on ADom.dist.ind2loc(lo) do
        local butterfly(wk1, wk2, wk3, A.localSlice(lo..by str #radix));
```

The outer *for* loop iterates over the serial phases of the algorithm, each of which has a unique stride and distance between its butterflies. The outer *forall* loop is used to create batches of butterflies that share the same twiddle factors while the inner *forall* loop describes that set of butterflies. Nested parallelism is beneficial because the trip counts of the inner and outer loops vary dramatically between the earlier and later phases.

This implementation pushes on Chapel features that have not been optimized for performance. Most egregious, assignment between vectors of different distributions uses a naive and slow implementation. (This will eventually be optimized within the Chapel distributions.) The following graph shows the performance that we are currently getting on 1–16 locales of a Cray XT4:



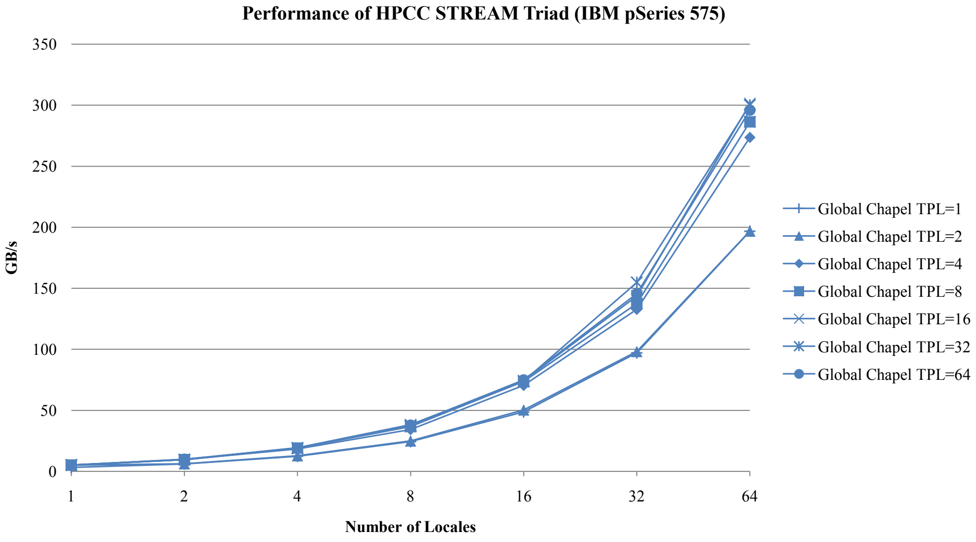
# 10 HPL Discussion

Although an initial Block-Cyclic distribution (which is a good match for HPL) is part of Chapel v1.0.2, its implementation is incomplete. In particular, the Chapel implementation of HPL relies on rank change and reindexing, but these operations are not yet implemented for any Chapel distribution. Thus there is not yet a distributed-memory implementation of HPL in Chapel.

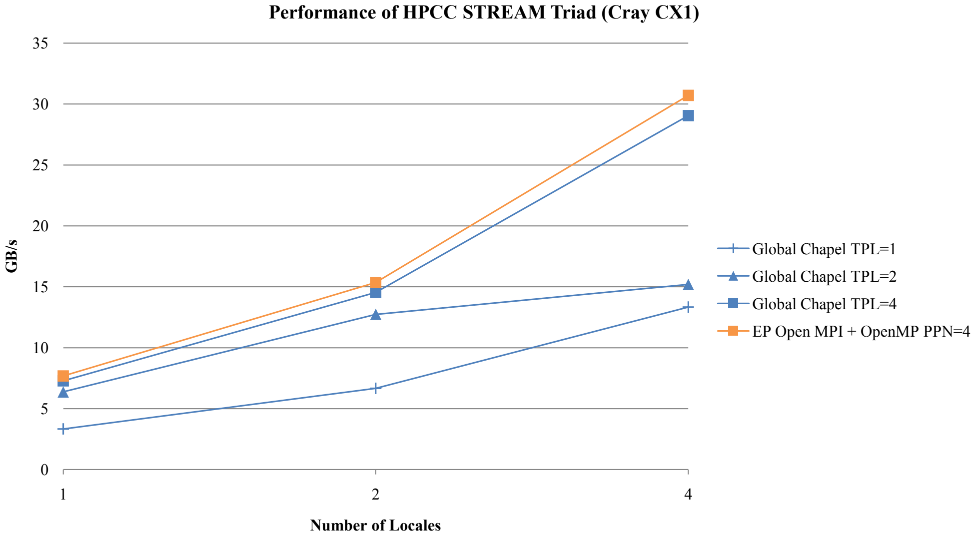
The single-locale implementation presented in this report continues to execute correctly. It is largely unchanged from last year's entry. However, this year it does take advantage of parallelism within a locale because, as of Chapel v1.0, data parallel constructs like *forall* loops now use multiple tasks within a locale.

# 11 Portability

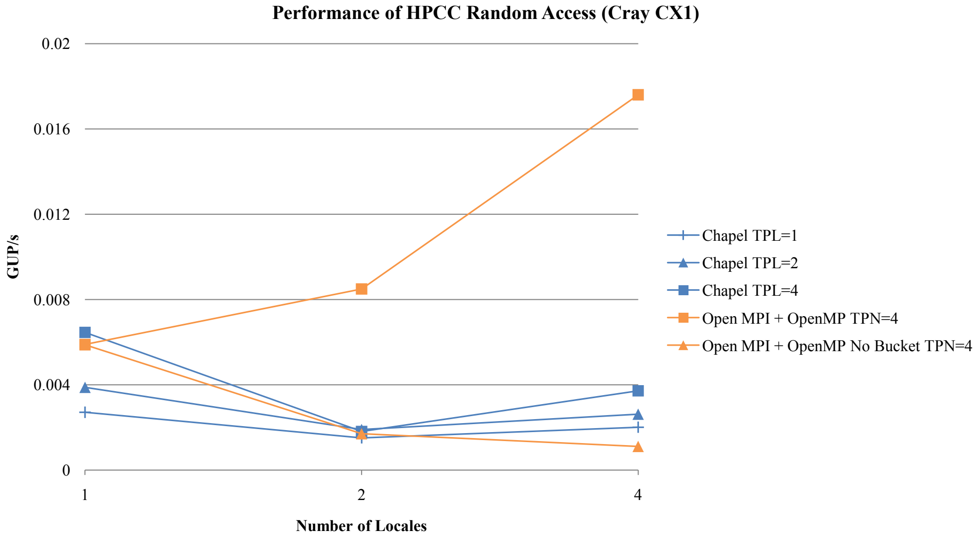
Chapel's source-to-source compilation and use of GASNet and pthreads makes our current implementation of Chapel highly portable. To demonstrate this, we have included some graphs showing performance numbers for the STREAM Triad and Random Access benchmarks on an IBM pSeries 575 and a Cray CX1. These graphs demonstrate portability of the Chapel implementation, but the results, gathered without serious exploration, are not representative of these systems. On the IBM platform, we met our quota of CPU usage before having a change to run the Random Access benchmark.



This performance graph shows raw GB/s performance of the Chapel implementation of STREAM Triad on an IBM pSeries 575 varying the tasks per locale between 1 and 64. The IBM pSeries 575 on which these results were obtained has 104 nodes. Each node has 16 dual core IBM Power6 processors running at 4.7 GHz and either 128 GB or 256 GB of memory. The benchmark was run with a problem size of 132146193 per locale (2.95 GB/locale) and the locales were mapped to nodes.



This performance graph shows raw GB/s performance of the Chapel implementation of STREAM Triad on a Cray CX1 varying the tasks per locale between 1 and 4. The Cray CX1 on which these results were obtained has 8 nodes and an Infiniband network. Each node contains 4 dual core Intel Xeon processors running at 3 GHz and 16 GB of memory. Chapel locales were mapped to nodes on this machine.



This performance graph shows raw GUP/s performance of the Chapel implementation of Random Access on a Cray CX1 (see characteristics above) varying the tasks per locale between 1 and 4.

## 12 Summary

We hope this report provides a glimpse into the Chapel implementation, our progress and the language design. The performance of the Chapel implementation is improving steadily. Both STREAM Triad and Random Access are much closer to optimal than a year ago. The FFT benchmark is now running on multiple locales on small problem sizes, and we've made significant progress on the multi-locale implementation of HPL.

## Acknowledgments

The authors would like to gratefully acknowledge our former team members and collaborators who assisted with our 2006 and 2008 entries in the HPCC competition: Samuel Figueroa, Mary Beth Hribar, John Lewis, Andrew Stone, Adrian Tate, and Wayne Wong. In addition, we thank all of Chapel's past contributors and early users for helping us reach this stage.

We would like to thank Oak Ridge National Laboratory and the National Center for Computational Sciences for the computing time on the Cray XT4 partition of Jaguar. We would also like to thank SARA for their generous donation of time and user support on the Huygens IBM pSeries 575 clustered SMP system, and by extension the Netherlands National Computing Facilities foundation (NCF) who funds Huygens.

# A Global STREAM Triad Code

```
//
// 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 = here.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 = distributionValue(
    new Block(rank=1,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");
  }

  delete randlist;
}

//
// 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);
  }
}
```

# B EP STREAM Triad Code

```
//
// Embarassingly Parallel Implementation of STREAM Triad
//
// This version of the stream benchmark is not as elegant as
// stream.chpl.  It is a per-locale code with no communication in the
// actual computation.  It highlights the ability of a Chapel
// programmer to escape the global-view programming model and write
// codes with a fragmented, per-locale model.
//
// Comments marked with '***' point out differences with the global
// version of this benchmark in stream.chpl.
//

//
// *** Since this benchmark is written on a per-locale basis, there is no
// *** need to use the Block distribution.  The following use omits
// *** BlockDist, included in stream.chpl.
//
// Use standard modules for Timing routines, Type utility functions,
// and Random numbers
//
use 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);

//
// *** To ensure a local problem size, we spoof the number of vectors
// *** passed to the computeProblemSize function to be the number of
// *** vectors times the number of locales.
//
// Configuration constants to set the problem size (m) and the scalar
// multiplier, alpha
//
config const m = computeProblemSize(numVectors*numLocales, 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;

//
// *** There isn't (yet) a way to set the number of tasks to use for
// *** implementing a forall loop over a default array.  When there is
// *** such a way, we will want to set it via this configuration
// *** constant to get functionality like we have with stream.chpl.
//
// config const tasksPerLocale = here.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;

//
// *** To ensure determinism of output, there is no more printing of the
// *** arrays in initVectors and verifyResults.
//
//
// Configuration constants to control what's printed -- benchmark
// parameters and/or statistics
//
config const printParams = true,
             printStats = true;

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

  //
  // *** Aggregates for collecting per-locale results for the minimum
  // *** execution time per trial, and whether verification passed
  //
  var minTimes: [LocaleSpace] real;
  var validAnswers: [LocaleSpace] bool;

  //
  // *** Fragment control so that we have a single task running on
  // *** every locale.
  //
  coforall loc in Locales do on loc {

    //
    // *** We declare these variables outside of the local block since
    // *** we'll need to access them when we write back to the global
    // *** aggregates declared above.
    //
    var validAnswer: bool;
    var execTime: [1..numTrials] real;

    //
    // *** Indicates that all of the code in this block is local to
    // *** this locale.  There is no communication.  A violation will
    // *** result in an error, though error checking is disabled with
    // *** --fast or --no-checks.
    //
    local {

      //
      // *** A, B, and C are the three local vectors
      //
      var A, B, C: [1..m] elemType;

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

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

        //
        // *** The main loop looks identical to stream.chpl.  However,
        // *** in this version we are iterating over arrays that are
        // *** not distributed.
        //
        forall (a, b, c) in (A, B, C) do
          a = b + alpha * c;

        execTime(trial) = getCurrentTime() - startTime;
      }

      validAnswer = verifyResults(A, B, C);            // verify...
    }

    //
    // *** Write times and verification result into aggregates
    // *** declared above.  These are declared over LocaleSpace so we
    // *** can write to them in parallel.
    //
    minTimes[here.id] = min reduce execTime;
    validAnswers[here.id] = validAnswer;
  }

  //
  // *** Pass minimum, average, and maximum times to printResults
  //
  printResults(&& reduce validAnswers, minTimes);
}

//
// Print the problem size and number of trials
//
def printConfiguration() {
  if (printParams) {
    //
    // *** Here we multiply m by the number of locales so that we can
    // *** print out the global problem size.
    //
    printProblemSize(elemType, numVectors, m * numLocales);
    writeln("Number of trials = ", numTrials, "\n");
  }
}

//
// *** Both initVectors and verifyResults are almost identical to
// *** stream.chpl even though they are called with arrays that are
// *** not distributed.  For initialization, the same random stream is
// *** used on each locale.  In the global version, a single logical
// *** stream of random numbers is used across all of the locales.
// ***
// *** In this version, we've omitted a way to print the arrays.  This
// *** ensures determinism of output.  Printing the arrays also
// *** violates the locality constraint imposed by the local block
// *** from which these functions are called.
//
// Initialize vectors B and C using a random stream of values
//
def initVectors(B, C) {
  var randlist = new RandomStream(seed);

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

  delete randlist;
}

//
// Verify that the computation is correct
//
def verifyResults(A, B, C) {

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

  //
  // 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.
//
// *** Here we report maximum, average, and minimum times instead of
// *** total, average, and minimum.
//
def printResults(successful, minTimes) {
  writeln("Validation: ", if successful then "SUCCESS" else "FAILURE");
  if (printStats) {
    const maxTime = max reduce minTimes,
          avgTime = + reduce minTimes / numLocales,
          minTime = min reduce minTimes;
    writeln("Execution time:");
    writeln("  max = ", maxTime);
    writeln("  avg = ", avgTime);
    writeln("  min = ", minTime);

    const maxGBPerSec = numVectors * numBytes(elemType) * (m / minTime) * 1e-9,
          avgGBPerSec = numVectors * numBytes(elemType) * (m / avgTime) * 1e-9,
          minGBPerSec = numVectors * numBytes(elemType) * (m / maxTime) * 1e-9;
    writeln("Performance (GB/s):");
    writeln("  max = ", maxGBPerSec);
    writeln("  avg = ", avgGBPerSec);
    writeln("  min = ", minGBPerSec);
  }
}
```

# C Global Random Access Code

## C.1 Benchmark Code

```
//
// 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 = here.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 = distributionValue(new Block(1, indexType, bbox=[0..m-1],
                                              tasksPerLocale=tasksPerLocale)),
      UpdateDist = distributionValue(new Block(1, 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 TableDist.ind2loc(r & indexMask) do {
      const myR = r;
      local {
        T(myR & indexMask) ^= myR;
      }
   }

  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 TableDist.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);
  }
}
```

## C.2 Supporting Module Code

```
//
// 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 m2: randWidth*randType = computeM2Vals();

  //
  // 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 {
    if follower.size != 1 then
      halt("RAStream cannot use multi-dimensional iterator");
    var val = getNthRandom(follower(1).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 0..#randWidth do
        if ((ran >> j) & 1) then val ^= m2(j+1);
      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 tuple, m2
  //
  def computeM2Vals() {
    var m2tmp: randWidth*randType;
    var nextVal = 0x1: randType;
    for param i in 1..randWidth {
      m2tmp(i) = nextVal;
      getNextRandom(nextVal);
      getNextRandom(nextVal);
    }
    return m2tmp;
  }
}
```

# D Global FFT Code

```
/* This implementation of the FFT benchmark uses radix-4 butterflies
   and is divided into two main phases: one which uses a Block
   distribution and the second which uses a Cyclic distribution.  When
   run on 4**k locales, this guarantees that each butterfly will only
   access local data.  In an optimized implementation, this should
   cause most of the communication to occur when copying the vector
   between Block and Cyclic storage formats.
*/

//
// Use standard modules for Bit operations, Random numbers, Timing, and
// Block and Cyclic distributions
//
use BitOps, Random, Time, BlockDist, CyclicDist;

//
// 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
type idxType = int(64);        // the index 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;

//
// The number of tasks to use per Chapel locale in parallel loops
//
config const tasksPerLocale = here.numCores;

//
// 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

  //
  // This implementation assumes 4**k locales due to its assertion that
  // all butterflies are local to a given locale
  //
  assert(4**log4(numLocales) == numLocales,
           "numLocales must be a power of 4 for this fft implementation");

  //
  // 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 stored using the block distribution TwiddleDist.
  // Twiddles is the vector of twiddle values.
  //
  const TwiddleDist = distributionValue(new Block(1, idxType, bbox=[0..m/4-1]));
  const TwiddleDom: domain(1, idxType) distributed TwiddleDist = [0..m/4-1];
  var Twiddles: [TwiddleDom] elemType;

  //
  // ProblemSpace describes the abstract problem space used for the
  // FFT benchmark: the indices 0..m-1
  //
  const ProblemSpace = [0..m-1];

  //
  // BlkDist describes the problem space as distributed in a Block
  // manner between the Locales where ProblemSpace defines the
  // bounding box used to compute the blocking.  BlkDom defines the
  // Block-distributed problem space and is used to define the vectors
  // z (used to store the input vector) and ZBlk (used for the first
  // half of the FFT phases).
  //
  const BlkDist = distributionValue(new Block(1, idxType, bbox=ProblemSpace,
                                              tasksPerLocale=tasksPerLocale));
  const BlkDom: domain(1, idxType) distributed BlkDist = ProblemSpace;
  var Zblk, z: [BlkDom] elemType;

  //
  // CycDist describes the problem space as distributed in a Cyclic
  // manner between the Locales where locale #0 stores element 0.
  // CycDom defines the Cyclic-distributed problem space and is used
  // to define the Zcyc vector, used for the second half of the FFT
  // phases.
  //
  const CycDist = distributionValue(new Cyclic(1, idxType,
                                               tasksPerLocale=tasksPerLocale));
  const CycDom: domain(1, idxType) distributed CycDist = ProblemSpace;
  var Zcyc: [CycDom] elemType;

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

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

  [(a,b) in (Zblk, z)] a = conjg(b);      // store the conjugate of z in Zblk
  bitReverseShuffle(Zblk);                // permute Zblk

  dfft(Zblk, Twiddles, cyclicPhase=false); // compute the DFFT, block phases

  forall (b, c) in (Zblk, Zcyc) do       // copy vector to Cyclic storage
    c = b;

  dfft(Zcyc, Twiddles, cyclicPhase=true); // compute the DFFT, cyclic phases
```

```
  forall (b, c) in (Zblk, Zcyc) do        // copy vector back to Block storage
    b = c;

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

  const validAnswer = verifyResults(z, Zblk, Zcyc, Twiddles); // validate 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, cyclicPhase) {
  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, cyclicPhase) {
    //
    // 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
        on ADom.dist.ind2loc(lo) do
          local butterfly(wk1, wk2, wk3, A.localSlice(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
        on ADom.dist.ind2loc(lo) do
          local butterfly(wk1, wk2, wk3, A.localSlice(lo.. by str #radix));
    }
  }

  if cyclicPhase {
    //
    // 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) {
      forall lo in 0..#str do
        on ADom.dist.ind2loc(lo) do
          local butterfly(1.0, 1.0, 1.0, A.localSlice(lo.. by str # radix));
    }
    //
    // ...otherwise using a simple radix-2 butterfly scheme
    //
    else
      forall lo in 0..#str do
        on ADom.dist.ind2loc(lo) do
          local {
            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, X:[0..3]) {
  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;
}

//
// 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, cyclicPhase) {
  const (start, end) = if !cyclicPhase then (1, numLocales:idxType)
                                       else (numLocales, numElements-1);
  var stride = start;
  for i in log4(start)+1..log4(end):int {
    const span = stride * radix;
    yield (stride, span);
    stride = span;
  }
}

//
// Print the problem size
//
def printConfiguration() {
  if (printParams) {
    if (printStats) then printLocalesTasks(tasksPerLocale=tasksPerLocale);
    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, Zblk, Zcyc, Twiddles) {
  if (printArrays) then writeln("After FFT, Z is: ", Zblk, "\n");

  [z in Zblk] z = conjg(z) / m;
  bitReverseShuffle(Zblk);
  dfft(Zblk, Twiddles, cyclicPhase=false);
  forall (b, c) in (Zblk, Zcyc) do
    c = b;
  dfft(Zcyc, Twiddles, true);
  forall (b, c) in (Zblk, Zcyc) do
    b = c;

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

  var maxerr = max reduce sqrt((z.re - Zblk.re)**2 + (z.im - Zblk.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);
  }
}
```

# E Global HPL Code

```
//
// 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;
}
```

# F Shared Problem Size Module Code

```
//
// 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);
  }
}
```