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)
- 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. 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—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

- Code Size Summary
- Experimental Setup
- Global STREAM Triad Description
- EP STREAM Triad Description
- STREAM Triad Performance
- Random Access Description
- Random Access Performance
- FFT Discussion
- HPL Discussion
- Portability
- Summary
- EP STREAM Triad Code
- Global Random Access Code
- Global FFT Code
- Global HPL Code
- Shared Problem Size Module Code

*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.

2http://www.highproductivity.org/
3http://icl.cs.utk.edu/hpcc/
4http://www.hpccchallenge.org/
## 2 Code Size Summary

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>2</td>
<td>6</td>
<td>7 + 25 = 34</td>
<td>75</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>Declarations</td>
<td>12</td>
<td>8</td>
<td>20 + 12 = 32</td>
<td>32</td>
<td>63</td>
<td>34</td>
</tr>
<tr>
<td>Total kernel</td>
<td>14</td>
<td>14</td>
<td>27 + 37 = 66</td>
<td>107</td>
<td>113</td>
<td>34</td>
</tr>
<tr>
<td>Initialization</td>
<td>10</td>
<td>10</td>
<td>1 + 10 = 11</td>
<td>26</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Verification</td>
<td>8</td>
<td>10</td>
<td>9 + 0 = 9</td>
<td>11</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Results/output</td>
<td>32</td>
<td>43</td>
<td>21 + 0 = 21</td>
<td>21</td>
<td>39</td>
<td>21</td>
</tr>
<tr>
<td>Total benchmark</td>
<td>64</td>
<td>77</td>
<td>58 + 47 = 107</td>
<td>165</td>
<td>176</td>
<td>55</td>
</tr>
<tr>
<td>Debug/test</td>
<td>7</td>
<td>0</td>
<td>3 + 2 = 3</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Comments</td>
<td>72</td>
<td>131</td>
<td>94 + 31 = 125</td>
<td>140</td>
<td>170</td>
<td>39</td>
</tr>
<tr>
<td>Blank</td>
<td>28</td>
<td>29</td>
<td>23 + 8 = 31</td>
<td>47</td>
<td>61</td>
<td>8</td>
</tr>
<tr>
<td>Total program</td>
<td>171</td>
<td>237</td>
<td>178 + 88 = 266</td>
<td>357</td>
<td>408</td>
<td>102</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Machine Characteristics</th>
<th>XT4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of compute nodes</td>
<td>7,832</td>
</tr>
<tr>
<td>Compute node processor</td>
<td>Quad-core AMD Opteron</td>
</tr>
<tr>
<td>Processor speed</td>
<td>2.1 GHz</td>
</tr>
<tr>
<td>Memory per node</td>
<td>8 GB</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Problem Sizes Per Node</th>
<th>XT4</th>
</tr>
</thead>
<tbody>
<tr>
<td>STREAM Triad Problem Size</td>
<td>85983914 (1.92 GB)</td>
</tr>
<tr>
<td>Random Access Problem Size</td>
<td>$2^{28}$ (2.0 GB)</td>
</tr>
<tr>
<td>Random Access Number of Updates</td>
<td>$2^{18}$</td>
</tr>
</tbody>
</table>

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 for inter-process coordination and data transfer. The following table specifies software versions and settings used to obtain the results in this report:

<table>
<thead>
<tr>
<th>Software</th>
<th>Flags/Settings</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>chpl</td>
<td>--fast</td>
<td>1.0.2</td>
</tr>
<tr>
<td>gcc</td>
<td>--target=linux --O3 --std=c99 --fopenmp</td>
<td>4.3.2</td>
</tr>
<tr>
<td></td>
<td>--param max-inline-insns-single=35000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--param inline-unit-growth=10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--param large-function-growth=200000</td>
<td></td>
</tr>
<tr>
<td>GASNet</td>
<td>conduit=mpi, segment=fast</td>
<td>1.12.0</td>
</tr>
</tbody>
</table>

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`.

---

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:

\[
\text{forall } (a, b, c) \text{ in } (A, B, C) \text{ do }
\]
\[
a = b + \alpha * c;
\]

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]

\[
\text{const BlockDist = distributionValue(new Block(rank=1,bbox=[1..m],tasksPerLocale=tasksPerLocale));}
\]

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, \(1 \ldots m\), 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

\[
\text{const ProblemSpace: domain(1, int(64)) distributed BlockDist = [1..m];}
\]

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 \(1 \ldots m\) which will be divided between the locales according to the mapping defined by \(BlockDist\).

The third declaration

\[
\text{var A, B, C: [ProblemSpace] elemType;}
\]

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:

```chapel
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:

```chapel
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:

```chapel
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

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:

```chapel
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:

```chapel
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:

```chapel
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.
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:

```chapel
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:

![Performance of HPCC FFT](image)

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.

![Performance of HPCC STREAM Triad (IBM pSeries 575)](image1)

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.

![Performance of HPCC STREAM Triad (Cray CX1)](image2)

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

```chapel
// The main loop: Iterate over the vectors A, B, and C in 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...
printStats(validAnswer, execTime); // ...and print the results
```

```chapel
// ProblemSpace describes the index set for the three vectors. It
// is a 1D domain storing 64-bit ints and is distributed across
// BlockDist, included in stream.chpl.
const ProblemSpace: domain(1, int64(4)); def blockDist = distributionValue(1..m, tasksPerLocale); def ProblemSpace = domain(1, int64(4)); blockDist = [1..m];
```

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

```chapel
// A, B, and C are the three distributed vectors, declared to store
// a variable of type eleType for each index in ProblemSpace.
forall (a, b, c) in (A, B, C) do
  var A, B, C: [ProblemSpace] eleType;
  var initVectors(B, C); // Initialize the input vectors, B and C
forall (i in 1..numTrials) { // loop over the trials
  const startTime = getCurrentTime(); // capture the start time
  const time = [numVectors] real; // an array of timings
  for trial in 1..numTrials | // use Time, Types, Random
```

### B EP STREAM Triad Code

```chapel
// The main loop: Iterate over the vectors A, B, and C in 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...
printStats(validAnswer, execTime); // ...and print the results
```

```chapel
// Use standard modules for Block distributions, Timing routines, Type utility functions, and Random numbers
use BlockDist, included in stream.chpl.
```
24 //
25 // Use shared user module for computing HPCC problem sizes
26 //
27 use HPCCProblemSize;
28 //
29 // The number of vectors and element type of those vectors
30 //
31 const numVectors = 3;
32 type elemType = real(64);
33 //
34 // To ensure a local problem size, we spread the number of vectors
35 // passed to the computeProblemSize function to be the number of
36 // vectors times the number of locales.
37 //
38 // Configuration constants to set the problem size (m) and the scalar
39 // multipliers, alpha
40 //
41 const alpha = 3.0;
42 //
43 // Arrays to hold the input vectors, B and C
44 //
45 const initVectors(B, C);
46 //
47 // The number of vectors and element type of those vectors
48 //
51 // The program entry point
54 main() {
55 // print the problem size, number of trials, etc.
56 printProblemSize(elemType, numVectors, m * numLocales);
57 // verify...}
58 //
59 // Verify that the computation is correct
60 if (printResults)
61 {
62 def initVectors(B, C) {
63 var randlist = new RandomStream
64 (seed = -1, fillRandom(B);
65 randlist.fillRandom(C);
66 delete randlist;
67 }
68 //
69 // Compute the infinity-norm by computing the maximum reduction of the
70 // absolute value of A's elements minus the new result computed in B.
71 // "[i in I]" represents an expression-level loop: "forall i in I"
72 // recompute the computation, destructively storing into B to save space
73 //
74 // reconstruction of input vectors using the random seed
75 //
76 //
77 fortrial in 1..numTrials {
78 // loop over the trials
79 const startTime = getCurrentTime();
80 //
81 // The program entry point
82 //
83 def main() {
84 printConfiguration(); // print the problem size, number of trials, etc.
85 // *** Aggregate for collecting per-locale results for the minimum
86 // *** execution time per trial, and whether verification passed
87 var minTimes: [LocaleIndex] real;
88 var validAnswers: [LocaleIndex] bool;
89 //
90 // *** Fragment control so that we have a single task running on
91 // *** every locale.
92 //
93forall loc in Locales do on loc |
94 //
95 // *** We declare these variables outside of the local block since
96 // *** we'll need to access them when we write back to the global
97 // *** aggregates declared above.
98 //
99 var validAnswer; bool;
100 var execTime: [1..numTrials] real;
101 //
102 // *** Indicates that all of the code in this block is local to
103 // *** this locale. There is no communication. A violation will
104 // *** result in an error, though error checking is disabled with
105 // *** --fast or --no-checks.
106 //
107 // local {
108 //
109 // *** A, B, and C are the three local vectors
110 //
111 var A, B, C: [1..M] elemType;
112 //
113 // Initialize input vectors, B and C
114 //
115 fortrial in 1..numTrials {
116 // loop over the trials
117 const startTime = getCurrentTime();
118 //
119 // The main loop looks identical to stream.chpl. However,
120 // *** in this version we are iterating over arrays that are
121 // *** not distributed.
122 //
123forall (a, b, c) in (A, B, C) do
C Global Random Access Code

C.1 Benchmark Code

```c
// Use standard modules for Block distributions and Timing routines
use BlockList, Time;
// Use the 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 n Tables = 1;
type elemType = randType;
indexType = randType;
// Constants defining the problem size (m) and a bit mask for table indexing
// of values
param m = 2**n,
indexMask = m-1;
// Configuration constants defining the number of errors to allow (as a fraction of the number of updates, N_U)
config const errorTolerance = 1e-2;
// Configuration constants to control what's printed -- benchmark parameters, input and output arrays, and/or statistics
printConfiguration()
printParams = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (printStats)
printStats = (prints...```
The program entry point

// The number of tasks to use per Chapel locale in parallel loops

14
15 const m2: randWidth= randType = computeNVals();
16
17 // A serial iterator for the random stream that resets the stream
18 // to its 0th element and yields values endlessly.
19
20 def &RandStream() {
21 var val = getRandomNumber(0);
22 while (1) |
23 yield val;
24 | yield val;
25 | return val;
26 }

// A "follower" iterator for the random stream that takes a range of
0-based indices (followers) and yields the pseudo-random values
// corresponding to those indices. Follower iterators like these
// are required for parallel pipelined iteration.

29 def &RandStream\(param \tag iterator, follower\) where tag == iterator.follower {
30 if follower.size != 1 then
31 yield \(\"RandStream cannot use multi-dimensional iterator\";\)
32 var val = getRandomNumber(follower().low);
33 for follower |
34 yield val;
35 | return val;
36 }

// A helper function for advancing a value from the random stream,\n// \&x, to the next value

39 def getNextRandom\(\{Param x\} \{\}
40 param POLY = 0xT;
41 param hRandBit = 0x1; randType == randWidth-1;
42 \(x = (x << 1) | (x \& hRandBit) \& POLY else 0;\}
43 }

// A helper function for computing the values of the helper tuple, \x22m2\x22
46 def computeM2Vals\(\) {
47 var m2tmp: randType = computeM2Vals();
48 for param i \in 1..randWidth \{|\}
49 m2tmp(i) = nextVal;
50 return m2tmp;\}
51 }

D Global FFT Code

1 // This implementation of the FFT benchmark uses radix-4 butterflies
2 and is divided into two main phases: one which uses a Block
3 distribution and the second which uses a Cyclic distribution. When
4 run on 4^n locales, this guarantees that each butterfly will only
5 access local data. In an optimized implementation, this should
6 cause most of the communication to occur when copying the vector
7 between Block and Cyclic storage formats.
8 */
9
10 // Use standard modules for Bit operations, Random numbers, Timing, and
11 // Block and Cyclic distributions
12 using BitOps, Random, Time, BlockDist, CyclicDist;
13
14 // Use shared user module for computing HPC problem sizes
15 using HPCProblemSize;
16
17 // This FFT benchmark: the indices 0..m-1
18 const ProblemSpace = [0..m-1];
19
20 using HPCCProblemSize;
21 const radix = 4; // the radix of this FFT implementation
22 const numVectors = 2; // the number of vectors to be stored
23 type elemType = complex(128); // the element type of the vectors
24 type idxType = int(64); // the index type of the vectors
25
26 // A configuration constant defining log2(problem size) --- n --- and a
27 // constant defining the problem size itself --- m
28 config const n = computeProblemSize(numVectors, elemType, returnLog2 = true);
29 config const m = \(2^n ;\)
30
31 // The number of tasks to use per Chapel locale in parallel loops
32 config const TasksPerLocals = here.numCores;
33
34 // Configuration constants defining the epsilon and threshold values
35 // used to verify the result
36 config const epsilon = 2.0, \(-1.0,\)
37 threshold = 16.0;
38
39 // Configuration constants to indicate whether or not to use a
40 // pseudo random seed (based on the clock) or a fixed seed, and to
41 // specify the fixed seed explicitly.
42 config const useRandomSeed = true,
43 seed = 1 \useRandomSeed then SeedGenerator.clockMS else 314159265;
44
45 // Configuration constants to control what's printed to benchmark
46 // parameters, input and output arrays, and/or statistics
47 config const printParams = true,
48 printArrays = false,
49 printStats = true,
50
51 n \& period;
52 if \(n == 0\) then return val;
53 var ran: randType = 0xT;
54 for i \in 0..log2(m-1) by -1 |
55 var val = randType = 0;
56 for i \in 0..randWidth do
57 if \(i \& m\) then val = m2[i-1];
58 ran = val;
59 if \(n >> i\ & 1\) then nextRandom\(ran\);
60 return ran;
61 }
62 }
63
64 // A helper function for advancing a value from the random stream,
65 // \&x, to the next value
66 def getNextRandom\(\{Param x\} \{\}
67 param POLY = 0xT;
68 param hRandBit = 0x1; randType == randWidth-1;
69 \(x = (x << 1) | (x \& hRandBit) \& POLY else 0;\}
70 }
71
72 // A helper function for computing the values of the helper tuple, \x22m2\x22
73 def computeM2Vals\(\) {
74 var m2tmp: randType = computeM2Vals();
75 for param i \in 1..randWidth \{|\}
76 m2tmp(i) = nextVal;
77 getNthRandom\(nextVal\);
78 getNthRandom\(nextVal\);
79 return m2tmp;\}
80 }
81
82 // print the problem size
83 printConfiguration();
84
85 // This implementation assumes 4\^k locales due to its assertion that
86 // all butterflies are local to a given locale
87 assert(4**log4(numLocales) == numLocales,
88 "numLocales must be a power of 4 for this fft implementation");
89
90 // TwiddleDom describes the index set used to define the vector of
91 // twiddle values and is a 1D domain indexed by 64-bit ints from 0
92 // to m/4-1 stored using the block distribution TwiddleDist.
93 // Twiddles is the vector of twiddle values.
94 const TwiddleList = distributionValue\(new Block(1, idxType, bbox=\[0..m/4-1\]);\)
95 const TwiddleDom: domain\(1, idxType\) distributed TwiddleList = \[0..m/4-1\];
96 var Twiddles: [TwiddleList] elemType;
97
98 // ProblemSpace describes the abstract problem space used for the
99 // FFT benchmark: the indices 0..m-1
100 //
101 // BlockDist describes the problem space as distributed in a Block
102 // manner between the Locales where ProblemSpace defines the
103 // bounding box used to compute the blocking. BlockDist defines the
104 // block-distributed problem space and is used to define the vectors
105 // \&x used to store the input vectors and Zblk (used for the first
106 // half of the FFT phases).
107 const BlockList = distributionValue\(new Block(1, idxType, bbox=ProblemSpace, tasksPerLocale=tasksPerLocals);\)
108 const BlockDom: domain\(1, idxType\) distributed BlockList = ProblemSpace;
109 var Zblk: [BlockList] elemType;
110
111 // CyclicDist describes the problem space as distributed in a Cyclic
112 // manner between the Locales where ProblemSpace defines the
113 // bounding box used to compute the blocking. CyclicDist defines the
114 // cyclic-distributed problem space and is used to define the Zcyc vector, used for the second half of the FFT
115 // phases.
116 const CyclicList = distributionValue\(new Cyclic(1, idxType, tasksPerLocale=tasksPerLocals);\)
117 const CyclicDom: domain\(1, idxType\) distributed CyclicList = ProblemSpace;
118 var Zcyc: [CyclicList] elemType;
119
120 // initialize twiddles and input vector x
121 const startTime = getCurrentTime();
122 \(\{\}
123 \{6, 8\} in [Zblk, x] = conjg(b);
124 \(\}
125 // compute the DFT, block phases
126 forall (b, x) in [Zblk, x] cycloPhase=false do
127 // copy vector to Cyclic storage
128 dfft\(\{Zcyc, Twiddles, cycloPhase=false\};\)
129 // compute the DFT, cyclic phases
forall (b, c) in ADoM by 2 span, 0.. } { 
    // the first set of multipliers for the low bank
    var wk1 = B[N2[wthIndex]],
    wk2 = wk1.wthIndex(),
    wk3 = wk1.re - 2 * wk2.im + wk1.im,
    wk4 = 2 * wk2.re + wk1.re - wk1.im):elemType;
    // loop in parallel over each of the banks of butterflies with
    // shared twiddle factors, avoiding the unbounded range
    // 0.. to get the base twiddle indices
    // forall lo in bankStart, #str do
    on ADoM.dist.ind2loc(lo) do
        local butterfly(wk1, wk2, wk3, A.localSlice(lo by str #radius));
    //
    // Update the multipliers for the high bank
    // forall lo in 0.., str do
    on ADoM.dist.ind2loc(lo) do
        local butterfly(wk1, wk2, wk3, A.localSlice(lo by str #radius));
### E Global HPL Code

```python
if (printStats) {
    writeln("Execution time = ", execTime);
}

writeln("Performance (Gflop/s) = ", 5 * (m * n / execTime) * 1e-9);
```

Each iteration of the loop increments a variable blk by blkSize. The point (blk, blk) is the upper-left location of the currently unfactored matrix (the dotted region represents the area factored in prior iterations). The unfactored matrix is partitioned into four subdomains: tl, tr, bl, and br, and an additional domain (not shown), i, that is the union of tl and bl.

```
for blk in 1..n by blkSize {
    const tl = Ab[blk..blk+blkSize, blk..blkSize],
    tr = Ab[blk..blk+blkSize, blk+blkSize..],
    bl = Ab[blk+blkSize.., blk..blkSize],
    br = Ab[blk+blkSize.., blk+blkSize..],
    l = Ab[blk.., blk..blkSize];
    // Now that we’ve sliced and diced Ab properly, do the blocked-LU
    // computation:
    panelSolve(Ab, 1, piv);
    if (Bl.numIndices > 0) then
        updateBlockRow(Ab, tl, tr);
    // update trailing submatrix (if any)
    if (Br.numIndices > 0) then
        schurComplement(Ab, bl, br);
} 
```

// Distributed matrix-multiply for HPL. The idea behind this algorithm is that some point the matrix will be partitioned as shown in the following diagram:

```
[1]-------------
| b |  
[2]-------------| a |
      |   
      |  
      |   
      |  
      | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a |a
```

The following diagram illustrates how we partition the matrix.
for i in trCols do
  for j in trRows do
  
  for i in tlRows do
  for j in tlCols do

  for i in trRows do
  for j in trCols do
// 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 {

        // 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 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");
        }

        // 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");

        // Compute the problem size as requested by the callee
        if returnLog2 then
            return lgProblemSize: retType;
        else
            if rank == 1 then
                return numIndices: retType;
            else
                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 then
            write(" (2**", lgProbSize);
            for i in 2..rank do write(" x 2**", lgProbSize);
            write(")");
        else
            write("(");
        write(n);
        writeln("Bytes per array = ", bytesPerArray);
        writeln("Total memory required (GB) = ", totalMemInGB);
    }

}