HPCC STREAM and RA in Chapel: Performance and Potential[*]

Steven J. Deitz Bradford L. Chamberlain Samuel Figueroa David Iten

Cray Inc.
chapel_info@cray.com

Abstract

Chapel is a new parallel programming language under development at Cray Inc. as part of the DARPA High Productivity Computing Systems (HPCS) program. This paper reports on recent progress to implement Chapel for distributed-memory computers. It provides a brief overview of Chapel and then discusses the concept of *distributions* in Chapel. Perhaps the most promising abstraction in Chapel, the distribution is a mapping of the data in a program to the (potentially distributed) memory in a computer. The paper presents preliminary results for two simple benchmarks, HPCC STREAM Triad and HPCC RA, that make use of distributions and other features of the Chapel language. The highlights of this paper include a presentation of performance results (STREAM performance of 5.68 TB/s on 4096 cores of an XT4), a detailed discussion of the core components of the STREAM and RA benchmarks, a thorough analysis of the performance achieved by the current compiler, and a discussion of future work.

1 Introduction

Chapel is a new parallel programming language being developed by a small team at Cray Inc. as part of its participation in DARPA's High Productivity Computing Systems program (HPCS). Our goal is to produce a language and compiler that makes it easier to prototype and develop general high-performance computing applications on both single processor and distributed memory systems.

While Chapel has proved to be an interesting language for writing both serial and multi-threaded programs, it really begins to shine when used to develop programs that run on distributed-memory systems. The programming model for Chapel differs from most common approaches, including Unified Parallel C (UPC), Co-Array Fortran (CAF), and C or Fortran with MPI.

In these other models, program execution starts in `main()` in every process. In Chapel, program execution starts in `main()`, but only in a single process. This seemingly minor distinction in program start-up leads to fundamentally different views of parallel data and execution within the programming model. To understand these distinctions, let's look at the MPI, CAF, and UPC models briefly.

When using MPI [7], the programmer must partition the program's data across processes with few abstractions. In the MPI program, each processor must be directed to first determine how many processors are in play and its relationship to the other processors. Then, to allocate the data within the program, each processor needs to use this information to divide the problem space amongst all of the processess and read in or generate its portion of the input data. For the computation, each processor needs to determine what portion of the problem to compute. All communication between the processors is written by making calls into the MPI library to pass messages between them. The programmer must thus view every aspect of the problem in per-process fragments of data and control flow.

Co-Array Fortran [6] improves the situation significantly but still requires the programmer to fragment both the data and the computation. The *co-dimension* of an array, the key abstraction in CAF, is useful for setting up an array that is accessible from every processor, but the data still needs to be partitioned by the programmer. Thus, even though CAF supports a *global address space* that makes it easier to access remote data, the data must still be partitioned across the processors by the programmer. For the computation, the programmer is responsible for ensuring that each process computes its own portion of the problem, thus fragmenting the computation as well.

In addition to supporting a global address space like CAF, Unified Parallel C [5] adds two abstractions that handle partitioning of data and computation in some simple cases. For data partitioning, UPC supports distributed one-dimensional arrays in a block-cyclic round-robin layout. However, if a problem's data is not one-dimensional or otherwise amenable to this data distribution, which is narrow in its applicabil-

[*]This material is based upon work supported by the Defense Advanced Research Projects Agency under its Contract No. NBCH3039003. Performance on the Cray XT4 was measured by a grant of HPC time by Oak Ridge National Laboratory.

ity, then the programmer must revert to fragmenting the data within the global address space. For computations, UPC supports a parallel loop that can subdivide a one-dimensional iteration space. Even with these abstractions, however, the programmer is responsible for managing and synchronizing the work of each processor with every other.

In Chapel, the program begins with a single logical thread of control. This single thread of control is key to how a Chapel programmer can view the data and computation. Parallelism is orthogonal to the processors, so the creation of parallelism serves as a natural point of synchronization. This is a stronger model than just supporting a global address space. We say that Chapel supports a *global-view programming model*, meaning that it provides abstractions that manage the details of partitioning the data and computation across the processors. With fragmented models like MPI, CAF, and UPC, partitioning data and computation across processors is a big challenge. Abstractions for distributed multi-dimensional arrays with a layout that can be specified by the programmer (or supplied in a library) allow the distributed data to be regarded as a whole, rather than in parts. Partitioning of the computation is also handled via these same abstractions; thus programmers can separate their concerns for the implementation on multiple processors from the computation itself.

In the past year, we've started to implement Chapel on distributed memory systems, allowing us to begin work on issues of remote communication and scaling. This paper reports on this progress and captures where we are in the implementation at this specific point. We present performance results for our implementations of two of the HPCC benchmarks [4]: STREAM and RA.

This paper is organized as follows: In Section 2, we discuss Chapel's language support for multi-locale task parallelism, *i.e.*, task parallelism across processesors with (potentially) distributed memory. Data parallelism in Chapel is built on top of this more fundamental language support. In Section 3, we discuss Chapel's abstractions for multi-locale data parallelism: domains, arrays, and distributions. For a more thorough treatment of the syntax and semantics of Chapel, refer to the Chapel specification [2]. In Sections 4 and 5, we discuss Chapel versions of the HPCC STREAM Triad and HPCC RA benchmarks. We conclude in Section 6.

2 Multi-Locale Task Parallelism

In Chapel, the definition of *locale* is specific to a machine, but it is typically defined to be a unit in a system that includes memory and processors such that all of the processors in that unit have roughly uniform access to any given memory location. For example, a locale may refer to a workstation in a cluster of workstations or to a node in a cluster of SMPs.

Chapel supports multi-locale programs with a `locale` type and an `on` statement, along with some intrinsics and operators. However, none of these constructs introduce concurrency. Task parallelism is independent of, but compatible with, the `locale` type and `on` statement.

These basic concepts offer the programmer great flexibility and form the basis of the more advanced abstractions in Chapel that support data parallelism.

2.1 The Locale Type

The locales available to a Chapel program are values of a type called `locale`. The locale type supports several standard methods including `id` which returns a unique integer between zero and one less than the number of locales on which the program is executing. (Other methods include `name`, which returns a machine-dependent locale name, and `physicalMemory()`, which returns the amount of local memory on the locale.)

2.1.1 Execution Context

When a multi-locale Chapel program is executed, the program has access to the following standard constants:

```
config const numLocales: int;
const LocaleSpace = [0..numLocales-1];
const Locales: [LocaleSpace] locale;
```

The configuration constant `numLocales` specifies the number of locales that were requested at program start-up. This is fixed throughout execution and is usually specified on a command line via the `-nl`, or `--numLocales`, flag. (A convenient feature of Chapel is that a program can contain many configuration constants that can be set on the command line.) The domain `LocaleSpace` contains the indices from `0` to `numLocales-1` and defines the array `Locales`. The elements of the `Locales` array are the locale values on which the program is executing.

2.1.2 Querying a Locale

In a Chapel program, the locale on which any variable (or, more generally, *lvalue*) exists is easy to query. The expression `expr.locale` evaluates to the locale on which `expr` is allocated. For example, to find out where the `i`th element of array `A` exists, a programmer can write `A(i).locale`.

2.2 The On Statement

To execute code on a remote locale, Chapel supports the on statement. The on statement evaluates an expression to determine a locale and then executes a statement block on that locale. For example, the simple code

```
on Locales(1) do
  compute();
```

results in the `compute` function being executed on locale 1.

The expression part of the on statement, say `expr`, is implicitly evaluated as `expr.locale` as discussed in Section 2.1.2. Thus it is easy to execute a task where the element of an array is stored. For example, to call a function `compute` on the `i`th element of array `A` on the locale on which it is stored, one could write

```
on A(i) {
  compute(A(i));
}
```

Chapel supports a partitioned global address space that permits variables to be accessed regardless of locale. Consider the simple example below:

```
on Locales(1) {
  var x: int = 1;
  on Locales(2) do
    x = x + 1;
}
```

This code increments the value of `x` on locale 2; `x` is stored on locale 1.

2.3 Task Parallelism

The abstractions discussed above do not introduce any parallelism even though they involve multiple processors. In Chapel, parallelizing a code is orthogonal to executing a code on multiple locales. (Chapel programs can thus be written for both multi-core and multi-processor systems.) Two abstractions for creating parallel tasks are `begin` and `coforall`.

2.3.1 The Begin Statement

The begin statement spawns a new task. There is no implicit join, and the execution of the spawning task continues with the next statement. Synchronization between tasks can be handled via synchronization variables or the sync statement as described in the language specification [2].

The begin statement can be coupled with the on statement to spawn a non-blocking remote task. For example, the code in Section 2.2 can be modified to increment `x` in a remote task as follows:

```
on Locales(1) {
  var x: int = 1;
  on Locales(2) do begin {
    x = x + 1;
  }
}
```

2.3.2 The Coforall Loop

The coforall loop is a typical loop except that each iteration of the loop is executed in a new task. Unlike the task that executes a begin statement, the task that initiates a coforall loop will wait for each of its spawned tasks to complete before continuing to the next statement.

One common idiom uses the coforall loop to spawn a task on each locale. Consider the following code:

```
def main() {
  coforall loc in Locales {
    on loc {
      fragmentedMain();
    }
  }
}
```

When this idiom is used in `main()`, as above, a Chapel program can support the fragmented model of MPI, CAF, and UPC. However, this style does not benefit as greatly from Chapel's global-view abstractions.

2.4 Implementation

The current implementation of multi-locale task parallelism in Chapel is built on top of other technologies. The Chapel compiler is source-to-source, translating Chapel to C. Multithreading is supported via POSIX Threads (pThreads). Remote memory operations and remote task invocation is supported via GASNet.

3 Multi-Locale Data Parallelism

Distributions are perhaps Chapel's most promising abstraction. By changing the arrays in a program to use one distribution over another, a programmer can impact how the data is partitioned across the locales and which computations end up on which locales. In this section, we will first discuss domains and arrays, Chapel's data-parallel abstractions. Next we will discuss distributions, first from the perspective of using them and then from the perspective of defining them.

3.1 Domains and Arrays in Chapel

In Chapel, data parallelism is supported via arrays. Chapel extends the abstraction of the ZPL region [1], a first-class index set over which arrays are declared and computations are controlled, calling it a domain.

3.1.1 Arithmetic Domains and Arrays

An arithmetic Chapel domain is defined by a rank, an index type, and rank-number of ranges where each range defines a sequence of integers with a lower bound, an upper bound, and a stride. In addition, whether or not any of the ranges can be defined by a non-unit stride is part of its type. For example, consider the following three domains:

```
const TableSpace: domain(1,int(64)) = [0..m-1];
var World: domain(2) = [1..n,1..n];
var BigWorld: domain(2) = [0..n+1,0..n+1];
```

The constant `TableSpace` is a one-dimensional domain containing 64-bit integer indices from zero to `m-1`. The variables `World` and `BigWorld` are two-dimensional domains that each contain indices composed of two 32-bit integers. (The default integral type for index types for non-distributed arithmetic domains is 32 bits.) The `World` domain contains the indices (i, j) for all $1 \leq i, j \leq n$; the `BigWorld` domain contains the indices (i, j) for all $0 \leq i, j \leq n + 1$.

An array is defined by a domain and an element type. For example, consider the following array declarations:

```
var T: [TableSpace] uint(64);
var A,B,C: [BigWorld] real;
```

The variable `T` is a one-dimensional array of elements of 64-bit unsigned integer type. It is linked to the `TableSpace` domain which defines the indices for which `T` stores an element. The variables `A`, `B`, and `C` are two-dimensional arrays of elements of floating-point type linked to the domain `BigWorld`. Note that changes to `BigWorld` will ripple to these three arrays. For example, if the size of `BigWorld` quadruples (*i.e.*, changes to `[1..2*n,1..2*n]`), then each array will quadruple as new elements are allocated.

3.1.2 Other Domains and Arrays

In addition to arithmetic domains and arrays, Chapel supports sparse, associative, and opaque domains and arrays. Sparse domains and arrays support sparse computations by containing a subset of the indices in a bounding domain. Associative domains and arrays support dictionary-style data structures (perhaps with hash-table implementations) by containing a set of indices of any scalar type. Opaque domains and arrays support graph-based data structures.

These other kinds of domains and arrays are beyond the scope of this paper but it is worth mentioning that the distribution abstraction presented here also applies to them. They permit Chapel's data-parallel mechanisms to apply to a more varied set of data structures than standard arithmetic arrays and Cartesian index spaces.

3.1.3 The Forall Loop

The forall loop is the key abstraction to introducing data parallelism into a program. All domains and arrays support parallel iteration (as well as serial iteration via the for loop). For domains, this means iterating over their indices. For arrays, this means iterating over their elements. For example, we can specify a data-parallel computation in which we compute the element-wise sums of `A` and `B` and store the results in `C` in many ways, including the following:

1. We can iterate over the domain and index into the arrays to do the actual assignment:

   ```
   forall i in BigWorld do
     C(i) = A(i) + B(i);
   ```

2. We can iterate over the domain, destructure the index into its column and row components, and index into the arrays to do the actual assignment:

   ```
   forall (i,j) in BigWorld do
     C(i,j) = A(i,j) + B(i,j);
   ```

3. We can iterate over all three arrays simultaneously and assign the elements directly:

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

4. We can use an implicit forall loop via *operator promotion* and whole array assignment:

   ```
   C = A + B;
   ```

   This is implemented via parallel iteration over the arrays; it is similar to (3).

5. We can use slicing to limit the computation to the interior indices (those defined by `World`) and then use an implicit forall loop via operator promotion and whole array assignment:

   ```
   C(World) = A(World) + B(World);
   ```

In a sense, all domains and arrays in Chapel are distributed. If a distribution is not specified, the default distribution applies and this default distribution maps all of the indices to a single locale. The next section explains how to distribute domains and arrays to multiple locales using either standard or user-defined distributions.

3.2 Using Chapel Distributions

Distributions are classes in Chapel that can be instantiated with the `new` keyword to create an instance of the distribution. Domains can then be linked to instances of distributions in the same way that arrays are linked to domains. Then all domains and arrays linked to that distribution share it. Thus if arrays `A` and `B` are linked, via potentially different domains, to the same distribution, they share the same distribution, *i.e.*, `A(i)` will be stored on the same locale as `B(i)`.

3.2.1 Example: Using the Block Distribution

For example, reconsider the declarations of domain `BigWorld` and arrays `A`, `B`, and `C`:

```
var BigWorld: domain(2) = [0..n+1,0..n+1];
var A,B,C: [BigWorld] real;
```

We can use the `Block` distribution to partition this domain and its arrays across the locales by modifying the declarations as follows:

```
config const tpl = 4;
var Dist = new Block(2,int(32),[1..n,1..n],tpl);
var BigWorld: domain(2) distributed Dist
                        = [0..n+1,0..n+1];
var A,B,C: [BigWorld] real;
```

The `Block` distribution is defined by a rank and index type used to define its domains, *a bounding box*, a specified number of tasks to use on each locale, and an array of locales. The bounding box for the `Block` distribution is used to define the mapping from indices to locales such that the bounding box is partitioned into roughly equal-sized blocks across the locales. The configuration constant `tpl` (tasks per locale) is used to specify the number of tasks to use on each locale. (By making it a configuration constant, we can vary it at runtime.) The default array of locales is `Locales`, the global constant that specifies every locale accessible to the program. Since it is not specified in the above code, this is the assumed array of locales.

For the `Block` distribution, the rank of the array of locales must be one or must match the rank of the distribution. If the rank does not match the distribution, then the locales are copied into an array of appropriate rank such that each dimension is assigned some number of locales. The `Block` distribution currently uses a heuristic to try to make the allocation of locales to dimensions even. For example, if the above code is run on 6 locales, the locales will be reorganized into a 3×2 array; on 16 locales, into a 4×4 array.

3.2.2 Future Distributions

One of our implementation goals is to support several common distributions as part of the standard Chapel programming environment. Other distributions can be written by and shared amongst Chapel programmers. The following list includes a small selection of the standard distributions, other than `Block`, that we expect will become part of the Chapel environment:

1. *Cyclic*: In each dimension, the indices are distributed round-robin to the locales.

2. *BlockCyclic*: In each dimension, uniformly sized blocks of indices are distributed round-robin to the locales.

3. *CutBlocks*: In each dimension, $l_i - 1$ positions are specified that divide that dimension, i, across its l_i locales.

4. *Scatter*: Each index is distributed to a locale based on the values in an array.

5. *RecursiveBisection*: The index space is divided in the first dimension, then both parts are divided in the second dimension, then all four parts are divided in the third dimension, etc., but the dimensions are iterated over in a round-robin fashion. This distribution is particularly useful for sparse arrays where load-balancing concerns make the divisions non-equal.

Note that there will be a different set of distributions for supporting associative and opaque arrays.

3.2.3 Distribution Advantages

Distributions allow for a valuable separation of concerns. The programmer can focus on the correctness of an array computation and then later plug in different distributions. In the fragmented programming model, changing a code from a block distribution to a cyclic distribution requires a complete rewrite. With the global-view programming model, it can be as easy as changing the type of distribution. The complexity is thus hidden in the implementation of the distribution.

3.3 Defining Chapel Distributions

The implementation of a distribution is not meant to be an easy undertaking. That said, we expect there to be a (relatively small) core interface that needs to be implemented and a (very large) extended interface that may be implemented to improve performance. Moreover, the artifact of a distribution can be reused in many applications and cleanly separates the concerns that we expect applications programmers to encounter.

A distribution defines the implementation of domains and arrays (what data they store and where). It is defined in a class. The domain and array data structures associated with this distribution are also defined in a class. These domain and array classes are used indirectly; the programmer writes codes with domains and arrays as described in Section 3.1, and the compiler transforms that code to use the classes.

3.3.1 Example: Defining the Block Distribution

The block distribution is a class that stores a target array of locales and a bounding box as described in Section 3.2. It is also specialized to the rank and index type that define it. Along with the block distribution class, classes are defined for implementing arithmetic domains and arrays over this distribution.

All of these classes inherit from classes in a standard hierarchy. This hierarchy—with leaf classes associated with the Block distribution's classes, `Block`, `BlockDom`, and `BlockArr`—is listed as follows:

```
BaseDist
  Block

BaseDom
  BaseArithmeticDom
    BlockDom
  BaseSparseDom
  BaseAssociativeDom
  BaseOpaqueDom

BaseArr
  BlockArr
```

Indentation indicates a "derived from" relationship.

3.3.2 Block Distribution Core Interface

The Block distribution must define a set of methods that compose a core interface. The compiler can then insert calls to these methods when processing domains and arrays declared over the Block distribution.

For example, the distribution class includes, but is not limited to, the following methods:

- `newArithmeticDomain` supports arithmetic domains. The compiler generates code to call this function when domains are linked to this distribution in their declaration.

- `ind2loc` takes an index and returns the locale where an element of one of its distributed arrays would exist.

The domain class includes, but is not limited to, the following methods:

- `buildArray` supports arrays declared over this domain. The compiler generates code to call this function when arrays are linked to this domain in their declaration.
- `these` is an overloaded iterator that supports both serial and parallel iteration over the domain.
- `dim` takes an integer and returns the range in this dimension.

The array class includes, but is not limited to, the following methods:

- `this` takes a domain index and returns the corresponding element in the array.
- `these` is an overloaded iterator that supports both serial and parallel iteration over the array.
- `reallocate` takes a domain of the same type and distribution as its domain and reallocates itself over this domain.

Distributions are defined using the constructs we have already discussed. They can be written entirely in Chapel using its task-parallel and locality-aware constructs. The compiler thus can transform high-level data-parallel array code into low-level task-parallel fragmented code.

3.3.3 Extended Interface Discussion

The extended interface defines methods that are optional. If they exist, the compiler may generate calls to them in order to optimize the Chapel program. For example, the extended interface includes, but is not limited to, methods that support the following functionality:

- Replicating the classes across the locales in order to optimize accesses from any given locale.
- Accessing or iterating over an array in a way that the compiler can prove is aligned.
- Managing halos (*i.e.*, ghost cells or fluff) so that array accesses to elements that border local elements are optimized.
- Optimizing array assignment from arrays with specific distributions.

The first item, called privatization, is discussed further in Section 4.2.1. The second item, alignment detection, is discussed further in Section 4.2.2.

4 HPCC STREAM in Chapel

The HPCC STREAM Triad benchmark implements the simple vector computation $a = b + \alpha \cdot c$ where a, b, and c are vectors and α is a scalar. The problem size is large, roughly $1/4$ – $1/2$ of the system memory, requiring the arrays to be partitioned across the locales. Distributed arrays allow for an elegant implementation.

4.1 STREAM Computation Core

The core of the STREAM benchmark in Chapel consists of one or two lines of computation and a handful of declarations.

4.1.1 Declarations

Our version of STREAM defines a distribution, a domain, and three arrays as follows: (These declarations are similar to those in Section 3.2.1.)

```
config const m: int(64) = ...;
config const tpl = ...;
const BlockDist = new Block(1, int(64), [1..m], tpl);
const ProblemSpace: domain(1, int(64))
  distributed BlockDist = [1..m];
var A, B, C: [ProblemSpace] real(64);
```

The configuration constant `m` specifies the problem size. The Chapel benchmark uses the `locale.physicalMemory` method to set `m` if it is not specified at program start-up. The configuration constant `tpl` is passed to the `Block` distribution so that we can vary the number of tasks per locale that are used for parallel array and domain iteration.

4.1.2 Main Loop

Our version of STREAM computes the main kernel as follows:

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

While we could have written any of the alternatives enumerated in Section 3.1.3 including the one-liner `A = B + alpha * C`, we use the above statement because it has the best performance due to the current state of the implementation. All of the alternatives disable the alignment detection optimization discussed in Section 4.2.2. However, in the future, this optimization will be made more robust so that it applies to all of the alternatives.

This is a high-level implementation of STREAM. Using the interface of the Block distribution, the compiler is able to transform this high-level code into a low-level task-parallel code, partitioning the data and computation.

4.2 STREAM Optimization Notes

The STREAM benchmark is straightforward but a simple translation of the Chapel would not result in acceptable performance. There are two optimizations that we've implemented, along with tuning of the `Block` distribution, that are absolutely crucial to STREAM and probably most other programs. We call these optimizations privatization and alignment detection.

4.2.1 Privatization

In a simple translation of Chapel, distributions, domains, and arrays are all entities that are logically stored on the locale on which they are declared, even though the domain and array link to data that is stored on other locales. Access to any of these entities must thus go through the single locale on which they were declared. This is a tremendous bottleneck. Privatization is an optimization in which these entities are replicated across the locale. They can then be accessed from any locale without communication.

Privatization is implemented via the extended interface of the distribution as discussed in Section 3.3.3. The Chapel runtime keeps track of all privatized objects. Each privatized domain or array is given a unique integer that the runtime can then map to the appropriate replicated class on any given locale. When a domain or array that supports privatization is declared, this unique integer is generated and the compiler inserts a call to the `privatize()` method on every locale (other than the one on which the domain or array is declared) to create the replicated instances. This makes accesses quick from any locale.

4.2.2 Alignment Detection

Alignment detection optimizes the case in which a Chapel program iterates over multiple domains or arrays that share the same distribution instance. In the STREAM kernel, the arrays `A`, `B`, and `C` all share the same distribution. This means that the blocks of each array that we iterate over will be aligned. So the compiler can generate calls to specialized iterators that rely on the blocks of array elements being local. If the distributions were different, this may not be the case, because iterating over one array may result in blocks of elements that are not local to another array. The net result is that the compiler can emit faster code because it can assume there is no communication required to access the array elements in a block.

4.3 STREAM Performance

To evaluate the performance of the Chapel code, we examine the speedup curve to see if we can obtain linear speedup. Figure 1 shows the performance of our version of STREAM executed on a Cray XT4 installed at Oak Ridge National Laboratory. This particular machine has 7,832 compute nodes, each of which contains a quad-core 2.1 GHz AMD Opteron processor and 8 GB of memory. We use up to 1,024 compute nodes.

The STREAM benchmark should be expected to achieve linear speedup since the only overhead is that needed to synchronize between iterations of the computation kernel. The performance of the Chapel code is not quite linear because of the overhead involved in spawning the remote tasks that do the actual computation. We discuss future work to address the overhead in the next section. That said, it is worthy to note that near-linear scaling is achieved despite the high-level nature of the Chapel code and the indirect implementation via the distribution class.

The performance achieved, when varying the number of tasks per locale, peaks at 4 tasks per locale on fewer than eight locales; above eight locales, the performance peaks at 3 tasks per locale. Since the system has four cores, we don't see performance improve on more than 4 tasks per locale. That it peaks with 3 tasks per locale is due to a detail of the GASNet implementation; we use a polling thread to handle remote communication on every locale.

4.4 STREAM Optimization Potential

The STREAM benchmark performs well but there is nothing inherent in the Chapel code that should keep it from achieving linear speedup. There are two optimizations, task tree and SPMD detection, that we expect will address the remaining overhead.

4.4.1 Task Tree

The code that implements the forall loop of the main kernel is encapsulated in iterators in the `Block` distribution, but it's basic structure is essentially equivalent to the following code:

```
coforall loc in Locales do
  on loc do
    coforall t in 1..tpl do
      for i in myPartOfTheWholeDomain do
        // local computation
```

Note that the idiom of using an on statement as the only statement in a coforall loop is optimized such that the system does not create all of the tasks locally and then do the remote spawns. Instead, these steps are combined to produce properly synchronized code from a serial loop.

It is this serial loop, however, that is the main issue. The loop becomes a performance bottleneck on many locales. The idea of the task tree is to parallelize this loop by creating a tree in which remote tasks spawn other remote tasks. If the locale that starts the loop just spawns a few tasks, then the rest of the spawning can be done concurrently.

4.4.2 SPMD Detection

Even with a task tree, starting up new tasks will introduce logarithmic overhead into the program. The idea of SPMD (Single Program, Multiple Data) detection is to identify data-parallel regions of code and implement them using a model more akin to MPI, UPC, and CAF in which we start remote tasks on each locale and use synchronization and communication primitives to coordinate them.

5 HPCC RA in Chapel

The HPCC RA benchmark tests the performance of making global pseudo-random updates to a large array of data. The benchmark tests the performance of many small communications.

5.1 RA Computation Core

The core of the RA benchmark in Chapel consists of a single loop that implements the random updates and a handful of declarations. The code relies on the same Block distribution used in our version of STREAM.

5.1.1 Declarations

Our version of RA defines two distributions, two domains, and a single array. The first distribution and domain are used for the array of data; the second distribution and domain are used to specify the update space. By default, it is four times the size of the table. These declarations are as follows:

```
config const m: uint(64) = ...;
config const tpl: int = ...;
const
  TableDist =
    new Block(1,uint(64),[0..m-1],tpl),
  UpdateDist =
    new Block(1,uint(64),[0..N_U-1],tpl),
  TableSpace: domain(1,uint(64))
    distributed TableDist = [0..m-1],
  Updates: domain(1,uint(64))
    distributed UpdateDist = [0..N_U-1];

var T: [TableSpace] uint(64);
```

The constant m specifies the size of the table, T. The configuration constant N_U specifies the number of random updates done in the benchmark. Note that we use the same configuration constant tpl, as in STREAM, to vary the number of tasks per locale that the distribution should use when iterating over domains and arrays.

5.1.2 Main Loop

The main loop of RA uses simultaneous iteration over the Updates domain and a parallel random number iterator, RAStream, that efficiently yields sequences of random numbers. Note that the first iterator controls the distribution of work. Thus each locale handles a roughly even number of updates because the domain Updates is evenly distributed across the locales using the Block distribution. We use an on statement to do the update on the locale that owns this portion of the array. Our version of RA computes the main loop as follows:

```
forall (i, r) in (Updates, RAStream()) do
  on T.domain.dist.ind2loc(r & (m-1)) do
    T(r & (m-1)) ^= r;
```

There are many other ways to write RA in Chapel. Some of these alternatives are discussed in Section 5.4.

5.2 RA Optimization Notes

Our implementation of RA in Chapel benefits immensely from privatization since the table needs to be accessed on all locales. In addition, the RA benchmark benefits from remote value forwarding.

5.2.1 Remote Value Forwarding

An unoptimized version of the main loop in RA requires communication that can be easily optimized away. Consider the accesses to r within the body of the on statement. Since r is stored on the locale from which the on statement was invoked, the access to r requires a remote read. Moreover, there are two such accesses.

Remote value forwarding is an optimization in which the value associated with a variable is forwarded to a spawned task rather than the address of the value. Our current implementation of remote value forwarding requires that the variable is only read within the on statement and that there is no synchronization within the on statement that would change the program semantics if the value is read early. Future work involves making this optimization flow sensitive so that it can apply in more cases.

As an aside, note that STREAM benefits from remote value forwarding as well; the benefits, however, apply to code embedded in the domain and array class iterator implementations.

5.3 RA Performance

We have more optimization work to do to make RA competitive with the reference version of the benchmark. To evaluate our current status, we have executed the Chapel code on the Cray XT4 at ORNL already mentioned in Section 4.3.

Figure 2 shows the results. The performance graph on the top clearly shows that the performance of RA on one locale, especially when optimized by the compiler under the assumption that the code will only be executed on one locale, is better than that achieved on between two and 64 locales. That is, we only see better performance on at least 128 nodes. This indicates first and foremost that there is significant overhead in the Chapel implementation. There are some relatively easy optimizations that we can make that should be able to improve performance greatly (decreasing message sizes, etc.). Other ideas are discussed in the next two sections.

All that said, one should expect to see this dip because when the data is distributed across two or more locales, there are remote accesses that will take more time. The efficiency graph on the left shows that the Chapel implementation fails to scale when compared to the best one-locale performance. On the other hand, the graph on the right shows that the Chapel code is scaling better when compared from eight locales onward.

5.4 Alternatives of RA in Chapel

There is much experimentation left to do with the RA benchmark. We plan to compare our current version with alternatives that use non-blocking updates with a begin statement, that aggregate updates, and that use remote updates without an on statement.

5.4.1 Non-Blocking Updates

A non-blocking version of RA changes the main loop as follows:

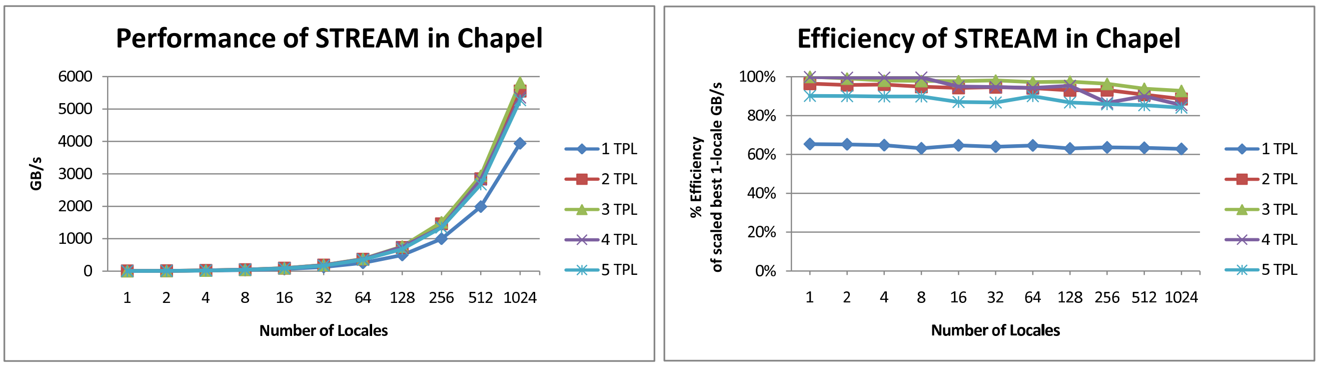

Figure 1: Graphs showing the performance and efficiency of the HPCC STREAM benchmark in Chapel. These graphs depict the same data. The graph on the left is included for completeness because we found it useful to use such a graph when discussing the HPCC RA results. The graph on the left shows the raw performance (GB/s) achieved by the Chapel version of the STREAM benchmark while varying the number of locales and the number of tasks per locale (TPL). The graph on the right shows the efficiency of the performance when compared to linear scaling of the best 1-locale performance. The results show near-linear scaling of the Chapel program.

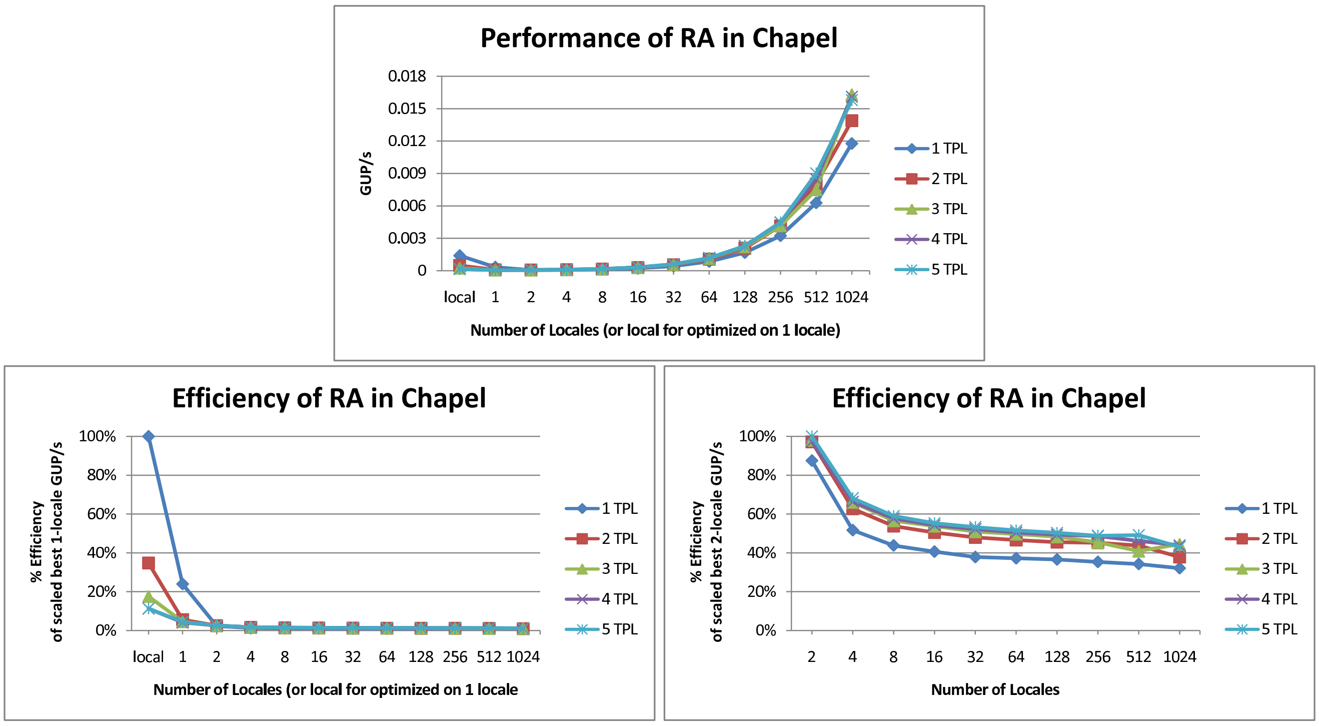

Figure 2: Graphs showing the performance and efficiency of the HPCC RA benchmark in Chapel. These graphs depict the same data but we found that examining all three makes it easier to understand the results. The graph on the top shows the raw performance (GUP/s) achieved by the Chapel version of the RA benchmark while varying the number of locales and the number of tasks per locale (TPL). The graphs on the bottom show the efficiency of the performance when compared to linear scaling of the best 1-locale performance and when compared to linear scaling of the best 2-locale performance. The results show a steep drop in performance when increasing the number of locales to eight, but near-linear scaling beyond eight locales.

```
  sync {
    forall (i, r) in (Updates, RAStream()) do
      on T.domain.dist.ind2loc(r & (m-1)) do begin
        T(r & (m-1)) ^= r;
  }
```

Note the introduction of the begin statement. This has the effect of spawning a non-blocking task for each update. The sync statement is used to wait until all of the spawned tasks complete.

The potential advantage to this variation is that it overlaps computation and communication. In order to make this scale, we need to improve our implementation of the sync statement. One possible implementation is to use the Dijkstra-Scholten algorithm. [3]

5.4.2 Aggregating Updates

An important optimization that has been implemented in the reference version of the RA benchmark is aggregation. In this approach, the loop is written to examine multiple updates, group them according to locale, and then do them all at once.

5.4.3 Remote Updates

Another way to write this in Chapel is to simply remove the on-statement. In this case, the update itself is done remotely. In order to optimize this alternative, we need to look at caching as described in Section 5.5.1.

5.5 RA Optimization Potential

The RA benchmark will allow us to explore a variety of optimizations depending on how it is written in Chapel. In this section, we'll discuss two such optimizations. First, we'll discuss remote descriptor caching which will amortize array accesses of remote data. Second, we'll discuss an optimization that will allow a simplification of our version of the RA benchmark.

5.5.1 Remote Descriptor Caching

In our current implementation of the `Block` distribution, accessing a remote element of an array involves more than just a single remote access. This is because we need to access the base address where the data exists on the remote locale and additional information stored in the remote local array class to compute the offset from this address used to locate the actual array element.

One optimization is to cache these values such that they can be reused. In that case, additional accesses of elements that exist on the same remote locale will require exactly one access.

5.5.2 Locale Query Optimization

In the main loop of the RA benchmark, we write the following line of code:

```
  on T.domain.dist.ind2loc(r & (m-1)) do
```

This is meant to compute the locale on which `T(r&(m-1))` exists. It would be cleaner to write simply:

```
  on T(r & (m-1)) do
```

However, doing so with our Block distribution requires making a number of additional remote accesses. This is a result of running the access function and remotely reading information stored in a remote local array class in order to compute the exact address of the remote data. However, we don't need the exact address. We only need to determine the locale. This is completely local as written above.

To implement this locale query optimization, we plan to use a compiler rewrite to convert locale accesses related to domains and arrays into more optimal code. For example, in the case above, we expect the compiler to effectively change the latter code into the former.

6 Summary

In this paper, we presented an overview of Chapel's task-parallel and data-parallel abstractions for programs that can execute on multiple nodes of a distributed-memory system. We've described a snapshot of the current implementation of Chapel and presented Chapel versions of two benchmarks, HPCC STREAM and RA, offering discussions of optimizations and performance.

The task-parallel multi-locale abstractions of Chapel can be used on their own to write general high-performance computing applications. When coupled with the data-parallel multi-locale abstractions, Chapel provides a high-productivity parallel programming environment for use on distributed memory systems. The library of distributions that we envision should make it easier for less sophisticated users to write applications on large-scale supercomputers.

As mentioned throughout this paper, we expect many details to change in the near future. That said, the code segments and interfaces listed in this paper are compatible with version 0.9.44 of Chapel. Chapel is available on SourceForge.net.

Acknowledgments

We would like to recognize the following people for their prior work and impact on the Chapel language and its implementation—Robert Bocchino, David Callahan, Roxana Diaconescu, James Dinan, Shannon Hoffswell, Mary Beth Hribar, Mark James, Mackale Joyner, John Plevyak, Andy Stone, Wayne Wong, and Hans Zima. In addition, we would like to thank Sung-Eun Choi for reading and commenting on an earlier draft of this paper.

References

[1] Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An abstraction

for expressing array computation. In *Proceedings of the ACM International Conference on Array Programming Languages*, 1999.

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

[3] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

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

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

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

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