User-Defined Parallel Zippered Iterators in Chapel *

Bradford L. Chamberlain
Sung-Eun Choi
Cray Inc.
{bradc|sungeun}@cray.com

Steven J. Deitz
Microsoft Corporation
stdeitz@microsoft.com

Angeles Navarro
Dept. of Computer Architecture
University of Málaga
angeles@ac.uma.es

Abstract

A fundamental concept in Chapel is zippered iteration, in which multiple iterators of compatible shape are evaluated in an interleaved manner, yielding tuples of corresponding elements. Defining language semantics in terms of zippered iteration has several advantages, including defining whole-array operations without requiring temporary arrays. That said, parallel zippered iteration also leads to challenges since the component iterators may wish to use different degrees of parallelism or distinct mappings of iterations to tasks. This paper describes our approach to addressing these challenges in Chapel. In particular, it defines Chapel's use of leader-follower iterators for defining the semantics of parallel zippered iteration. The leader iterator generates the parallelism and performs work scheduling, while the follower iterators are responsible for implementing the serial work generated by the leader. We motivate leader-follower iterators, show several examples of their use, and describe their implementation in the Chapel compiler. Our experimental results demonstrate that our implementation of zippered iteration results in reasonable performance compared to hand-coded C with OpenMP and MPI. We wrap up by pointing out shortcomings in our current support for parallel zippered iteration and describe our plans for addressing them in future work.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures

General Terms PGAS languages, parallel control structures

Keywords Chapel, zippered iteration

1. Introduction

Chapel is an emerging parallel language being developed by Cray Inc. with the goal of improving programmer productivity on large-scale systems as well as on the desktop. Among Chapel's features for expressing parallelism are *forall loops* which indicate a loop whose iterations can and should be executed in parallel. Forall loops are commonly used to express the parallel traversal of a data structure or iteration space, but they can also invoke user-defined parallel iterators. Chapel's forall loops support a *zippered* format in which multiple iterators or data structures are traversed simultaneously, such that corresponding iterations are processed together. More precisely, when looping over n iterators in a zippered manner, the resulting index values are n-tuples in which component d of the i^{th} index generated by the loop represents the i^{th} index generated by iterator d. For example, the following loop traverses arrays *A* and *B* in a zippered manner:

```
forall (a, b) in (A, B) do
  a = b
```

Here, the tuple (a, b) generated by iteration i represents the pair of variables corresponding to the i^{th} element from each array. Thus, the loop specifies that every element of *B* should be assigned to its corresponding element of *A*, and that the collection of assignments can and should be performed in parallel.

In addition to supporting explicitly zippered loops, Chapel defines many of its built-in operations, such as array assignment or promotion of scalar functions, in terms of parallel zippered loops. For example, a trivial whole-array assignment in Chapel, `A = B`, is defined in terms of the zippered forall loop shown above. Because such loops are prevalent when using Chapel's data parallel features, it is crucial to define and implement parallel zippered iteration appropriately.

One of the challenges of supporting parallel zippered iteration is that it can require arbitrating between multiple contradictory implementations of the parallel loop. For example, in the zippered loop above, the *A* and *B* arrays could have distinct data distributions and memory layouts with corresponding parallelization strategies and work decompositions. In more complex zipperings involving multiple data structures or iterators, the number of implementation options could be even greater. Given this choice between potentially contradictory parallelization strategies, how should Chapel decide which implementation to use, and how should users reason about the execution of these high-level parallel loops?

Chapel addresses this challenge by defining zippered iteration in terms of *leader-follower semantics*. In particular, the first item being iterated over is identified as the *leader*. As such, it is responsible for introducing the parallel tasks that will be used to implement the loop, and for assigning iterations to those tasks. All other items in the zippered iteration are classified as *followers*. As such, they are only responsible for serially executing the iterations specified by the leader and do not play a role in determining the parallel implementation of the loop.

This paper contributes the first published description of Chapel's leader-follower iteration strategy. It motivates the use of leader-

* This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. It was also funded in part by the Ministry of Education of Spain under grant PR2010-0247 and by Junta de Andalucía Project P08-TIC-3500. 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.

PGAS 2011 October 15–18, 2011, Galveston Island, TX.

follower iterators, describes their role in Chapel's definition, and provides a number of examples that illustrate the variety of parallel iteration styles that a user can write using this mechanism. We describe the implementation of leader-follower iterators in our open-source compiler and demonstrate the performance achieved by using them in a number of shared- and distributed-memory computational kernels. While several languages have previously supported high-level data parallelism and zippered iteration, to our knowledge, this is the first time that a language has supported parallel zippered iteration of general user-defined iterators, and thus the first published paper on the topic. We believe that such support is essential for productive parallel programming and expect it to become an important idiom and implementation strategy for parallel languages going forward.

The rest of this paper is organized as follows: The next section provides a brief overview of Chapel in order to motivate, and provide context for, the rest of the paper. Section 3 describes the benefits of defining data parallelism in terms of parallel zippered iteration. We describe our leader-follower iterator strategy in detail in Section 4 and provide some examples of leader-follower iterators written in Chapel in Section 5. Section 6 provides an overview of our implementation while Section 7 evaluates it using some kernel computations for shared and distributed memory parallelism. In Section 8, we contrast our work with previous languages' support for data parallelism and zippered iteration. We provide a discussion of some of the limitations of our current approach and our plans for addressing them in Section 9, concluding in Section 10.

2. Chapel Overview

Chapel is defined using what we refer to as a *multiresolution language design*. This term refers to our approach of providing a mix of higher- and lower-level features in order to support varying levels of abstraction and control within a unified set of concepts. In particular, the lower-level features are designed to be closer to the physical hardware and to give the user greater control over their program's implementation details. In contrast, the higher-level features support abstractions that are further removed from the hardware with the goal of improved programmability. Moreover, Chapel's high-level features are implemented in terms of the lower-level ones in order to simplify the implementation and ensure that all features are inter-compatible. The result is that programmers may mix features from different levels at arbitrary points in their programs.

A key example of multiresolution design for this paper is seen in Chapel's support for data- and task-parallel features. Chapel's task-parallel features support the ability to create and synchronize between tasks, supporting general concurrent programming abstractions. As such, task parallelism is considered a fairly low-level feature in Chapel. At the higher level, Chapel's data parallel features include a rich set of data types that support a variety of parallel operations. Yet, all parallelism on these high-level abstractions is ultimately implemented in terms of task parallel features. Thus, all parallel execution within a Chapel program is executed using tasks, though they may be hidden by higher-level abstractions.

At its lower levels, Chapel also has a serial base language and features for locality control. The rest of this section provides a brief overview of Chapel features from each of these layers as context for the rest of this paper. For a more complete treatment of Chapel, the reader is directed to previous descriptions [4, 7, 9].

2.1 Serial Base Language Features

Chapel's base serial language was created from scratch, though its design was informed by concepts and syntax from previous languages such as C, Modula, Java, C++, C#, Fortran, CLU, and ML. One of the most crucial base language features for this paper is Chapel's support for CLU-style *iterator functions* (*iterators* for short). Unlike a normal function which returns only once for any given call, an iterator function may *yield* multiple values back to the callsite before eventually returning and completing. For example, the following iterator generates the first *n* elements of the Fibonacci sequence, as specified by its input argument:

```
1 iter fib(n) {
2   var current = 0,
3       next = 1;

5   for i in 1..n {
6     yield current;
7     current += next;
8     current <=> next;
9   }
10 }
```

In this iterator, the local variables *current* and *next* store the values of the next two Fibonacci numbers. The `yield` statement on line 6 returns the *current* value back to the callsite before incrementing it by the *next* value and swapping the two values (lines 7–8). The iterator continues until its loop to *n* finishes, at which point control falls out of the iterator, terminating it.

Chapel's iterators are typically invoked within a loop header. For example, to print out the first ten Fibonacci numbers in sequence, we could use a serial *for loop* as follows:

```
for f in fib(10) do
  writeln(f);
```

This loop introduces a new variable *f* to represent the values yielded by the iterator. Such *loop index variables* are lexically associated with the loop's body. Thus, each iteration can be thought of as having its own private value of *f*. As written, the `fib()` iterator returns constant values; therefore, in this loop, *f* may not be reassigned. More generally, iterators can be declared to yield variable references, permitting loops to modify data via their index variables.

Chapel's iterators can also be associated with data structures by creating a zero-argument iterator method with the special name `these()`. Such iterators are implicitly invoked when a loop traverses a variable of that type. For example, given a record definition as follows:

```
record R {
  ...
  iter these() { ... }
}
```

the following loop would invoke *R*'s `these()` iterator for the *myR* instance:

```
var myR: R;
for r in myR do ...
```

As an example of multiresolution language design, our compiler uses a user-level record to implement Chapel's built-in *range* type, which represents regularly-strided integer sequences. Like record *R* above, the range record supports a `these()` method to implement loops over ranges like the one over `1..n` in the `fib()` example above (line 5). An example of this approach is illustrated in Section 5.1.

Multiple iterators may be invoked in an interleaved manner using zippered iteration. For example, the following loop iterates over our `fib()` iterator, a range, and *myR* simultaneously. In order for this loop to be legal, all three iterators must generate the same number of values:

```
for ind in (fib(10), 1..10, myR) do ...
```

In a zippered iteration, the index variable is a tuple whose d^{th} component represents the values generated by the d^{th} iterator in the

zippering. Standard Chapel syntax for de-tupling can be applied to separate the variables:

```
for (f, i, r) in (fib(10), 1..10, myR) do ...
```

2.2 Task Parallel Features

In Chapel, a *task* is a computation to be executed concurrently with other tasks. Chapel has a handful of constructs for introducing new tasks into a program. When defining iterators, one of the most useful is the *coforall loop*. This is a variant of the for-loop that creates a distinct task to execute each iteration's body. Thus, the following loop:

```
coforall f in fib(10) do
  writeln(f);
writeln("Continuing...");
```

creates 10 tasks, each of which has its own private copy of f representing the Fibonacci number for its iteration. Because these tasks can execute concurrently, there is no guaranteed order in which the Fibonacci numbers are printed. Coforall loops have an implicit *join* that waits for the component tasks to complete before execution continues. Thus, we are guaranteed that all ten of the Fibonacci numbers will print before the string "*Continuing...*" is.

Unlike many task-parallel programming models, Chapel permits dependences between tasks. For this reason, coforall loops may have synchronization events between distinct iterations. Like for-loops, coforall loops also support a zippered form.

Chapel's task parallel layer has other features that support non-blocking tasks, heterogeneous tasks, and synchronization between tasks, but those features are beyond the scope of this paper.

2.3 Locality Features

To help programmers control and reason about locality, Chapel supports a built-in type called the *locale*. Locales represent the unit of the target architecture that is most valuable when reasoning about locality. Chapel's PGAS (Partitioned Global Address Space) memory model permits a task to access lexically visible variables whether they are stored on its locale or a remote one. Locales define what is local and cheap to access versus what is remote and therefore more expensive. On most conventional large-scale architectures, Chapel defines a locale as the processors and memory that make up a compute node. All Chapel programs are seeded with a built-in *Locales* array that permits programmers to query and refer to the machine resources on which their programs are running.

Each Chapel task executes on a given locale. The Chapel programmer can control where a task executes by using *on-clauses*. If the on-clause's argument is of locale type, the task logically migrates to the specified locale and continues executing there. For any other variable expression, it will migrate to the locale on which the variable is stored. As an example, the following loop will create one task for each locale on which the Chapel program is running, migrating it to its respective locale:

```
coforall (loc, id) in (Locales, 1..) do
  on loc do
    writeln("Task ", id, " executes on ", loc);
```

2.4 Data Parallel Features

Chapel's data parallel features are based on its third form of loop, the *forall loop*. Whereas for-loops are executed serially using just the original task, and coforall loops are implemented using one task per iteration, forall loops are typically implemented using some number of tasks between these two extremes. In practice, this is often based on the degree of hardware parallelism used to execute the loop. The precise number of tasks is controlled by the loop's leader iterator.

Forall loops also differ from coforall loops in that they must be *serializable*—that is, it must be legal to execute the whole loop serially with a single task. In particular, there may not be synchronization dependences between forall loop iterations, as there can be in a coforall loop.

Chapel's most prominent features for data parallel programming are its *domains* and *arrays*. The *domain* is a first-class language concept representing an index set, while an array is a mapping from a domain's indices to a set of variables. Domains are also used to operate on arrays in parallel, supporting slicing, reindexing, and iteration. In addition to traditional dense rectangular arrays as in Fortran, Chapel supports strided, sparse, associative, and unstructured domains and arrays. As described in our previous work [6], advanced Chapel programmers can provide their own implementations of domains and arrays by authoring *domain maps* specifying how their values should be distributed between locales, stored within each locale's memory, and operated upon.

All Chapel domains and arrays support loops that iterate over their values. Serial iteration over a domain or array is implemented by providing `these()` iterator methods for the domain map's descriptors. The implementation of a forall loop uses the leader-follower techniques introduced in Section 4.

In addition to supporting explicit parallel loops over domains and arrays, Chapel also supports implicit parallelism via the concept of *scalar function promotion* (*promotion* for short). Promotion occurs when an array or domain argument is passed to a function or operator that expects a scalar argument. For example, given the traditional scalar `sin()` function and an array A, `sin(A)` specifies the parallel application of `sin()` to every element of A, generating a logical array of results.

Function promotion is defined in terms of forall loops in Chapel. For example, the promotion of the `sin()` function above is equivalent to the following loop expression:

```
forall a in A do sin(a)
```

When a scalar function has multiple promoted arguments or an expression contains multiple promotions, its definition is equivalent to a zippered forall loop. For example, given arrays A, B, and C, the expression: `exp(A, B + C)` is equivalent to:

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

This discussion of data parallelism returns us to the central challenges that this paper addresses: How does the programmer write a parallel iterator function that is suitable for use in a forall loop (zippered or otherwise)? Given a zippered parallel forall loop in Chapel, how do we define its implementation given that each of its component iterators may have its own notion of how many parallel tasks should be used to execute the loop, where those tasks should execute, and which iterations each task should own? And how do we avoid making the loop's implementation opaque to the user? These questions are addressed by our approach, described in Section 4.

3. Benefits of Zippered Iteration Semantics

Defining a language's data parallel semantics in terms of parallel zippered iteration has a few benefits that may not be immediately obvious. The first is that by defining implicit parallelism in terms of zippered forall loops, we can rewrite whole-array operations in terms of existing looping and scalar operations within the language. This permits Chapel to define higher-level abstractions in terms of lower-level concepts, resulting in a more unified language design. It also provides a means for users to implement their own customized parallel abstractions directly within the language.

The bigger benefit is that defining data parallel operations in terms of zippered iteration permits us to define Chapel in a man-

ner that does not rely on the introduction of temporary arrays. For example, a traditional data parallel language might define the final promotion example from the previous section in terms of the following whole-array operations:

```
const T1 = B + C;
const T2 = exp(A, T1);
```

Where *T1* and *T2* are temporary arrays of the same size and shape as *A*, *B*, and *C*.

Defining a language's semantics in this operation-centric manner has a few disadvantages: The first is that when users compute with very large arrays that occupy a majority of the machine's physical memory (as is common in HPC), the introduction of temporary arrays like this can surprise programmers, causing them to exceed their memory budget.

The second problem is that performing whole-array operations pair-wise like this will typically result in poor cache performance since arrays are traversed in their entirety before moving on to the next operation. This causes problems when the same array is used within several expressions in a single statement. For example, consider the expression `(A + B) * (A - B)`. Implemented using traditional whole-array semantics, this would correspond to:

```
const T1 = A + B;
const T2 = A - B;
const T3 = T1 * T2;
```

Here, for large arrays *A* and *B*, by the time *T1* has been computed, the initial values from *A* and *B* will have been flushed from the cache, requiring them to be re-loaded when computing *T2*. In contrast, adopting zippered semantics permits *A* and *B* to each be traversed just once in computing the expression:

```
forall (a, b) in (A, B) do
  (a + b) * (a - b)
```

For languages that adopt an operation-centric model rather than our zippering-based semantics, compiler optimizations can help improve the overall memory requirements and cache behavior for simple cases like these by fusing loops and contracting semantically unnecessary arrays [14]. However, for more complex idioms, temporary arrays are likely to be required, either for correctness or due to conservatism in the compiler's analysis. This reliance on the compiler has the downside of muddling the language's execution and portability models since users must conservatively assume that a compiler may need to introduce such temporary arrays to guarantee correct execution. In contrast, languages that adopt zippered semantics provide users with an unambiguous execution model while requiring less memory overall and using it in a more cache-friendly way.

Note that zippered semantics have a downside as well, related to aliasing in array assignments. For example, consider the following statement that seems to replace each element with the average of its neighbors.

```
A[1..n] = (A[0..n-1] + A[2..n+1]) / 2;
```

Using zippered semantics, this is equivalent to the following:

```
forall (a0, a1, a2)
    in (A[1..n], A[0..n-1], A[2..n+1]) do
      a0 = (a1 + a2) / 2;
```

This loop is problematic because it contains races between the reads and writes of *A*, thereby breaking the probable intention of the whole-array operation.

We believe the benefits of zippered iteration outweigh the potential confusion of cases like this, and so have chosen to adopt zippered execution semantics in Chapel. Compiler warnings can help prevent unintentional races like this one by detecting the aliasing and suggesting that the user may wish to use a second array to capture the resulting array expression. In practice, stencil operations like this one often use distinct result arrays anyway, in order to perform comparisons across time steps.

4. Leader-Follower Iterators

This section defines how leader-follower iterators are defined in our Chapel implementation today. For this discussion we gloss over a few coding details for the sake of clarity. In the following section, we will introduce these concrete details as we introduce some examples of actual leader-follower iterators.

As described in the introduction, Chapel converts zippered parallel forall loops into invocations of leader and follower iterators. As an example, given a loop of the following general form:

```
forall (a, b, c) in (A, B, C) do
  ...
```

the compiler converts it into the following conceptual loop structure:

```
...inlined A.lead() iterator creates tasks that yield work...
   for (a, b, c) in (A.follow(work, ...),
                     B.follow(work, ...),
                     C.follow(work, ...)) do
      ...
```

Here, `lead()` refers to a variable's leader iterator while `follow()` refers to its follower.

Recall that the responsibilities of a leader iterator are to create the tasks that will implement the forall loop's parallelism and to assign work to them. Leaders are typically written using Chapel's task parallel features to create the tasks. Each task then generates the work it owns back to the loop header via one or more `yield` statements. In the code above, the leader iterator is inlined and generates work items that are passed on to the follower iterators.

In contrast, a follower iterator is simply a traditional serial iterator that takes a leader's work representation as an input argument. It executes the iterations represented by that work unit, yielding its corresponding variables or values back to the callsite. Followers are executed in a zippered manner using Chapel's traditional serial zippered implementation.

As a simple example, consider a leader-follower pair that is defined to use a specified number of tasks, *numTasks*, to iterate over a range. A simple leader for such a type might appear as follows:

```
iter range.lead() {
  coforall tid in 0..#numTasks {
    const mywork = this.getChunk(tid, numTasks);
    yield mywork.translate(-this.low);
} }
```

This leader uses a coforall loop to create the tasks that will implement its parallelism. It then generates work for each task using a helper method `getChunk(i, n)` that divides the range into *n* approximately equal pieces and returns the i^{th} sub-range. For reasons that will be explained in a few paragraphs, the leader shifts the result by the original range's low bound before yielding it.

While this example contains only a single yield per task, more generally, a leader's tasks can use additional control flow and/or yield statements to create any number of work items per task. Section 5 contains some interesting examples of such leaders.

To illustrate the leader in practice, consider the following zippered forall loop over a set of range values:

```
forall (i,j,k) in (1..n, 0..n-1, 2..n+1) do
  ...
```

Given the leader iterator above, this loop would be conceptually rewritten as follows. Note that `1..n` is the leader since it is the first item in the zippering:

```
coforall tid in 0..#numTasks {
  const mywork = (1..n).getChunk(tid, numTasks);
  const work = mywork.translate(-1);
  for (i,j,k) in ((1..n).follow(work),
                  (0..n-1).follow(work),
                  (2..n+1).follow(work)) do
    ...
}
```

This translation demonstrates that in order for a set of iterators to be zippered together, their followers must accept the same work representation generated by the leader. In this example, since our leader yields a range, each follower must accept a range as input. More importantly however, a common encoding for the work items must be used such that each follower can transform the leader's representation into its own iteration space. For instance, Chapel's built-in leader-follower iterators use a zero-based coordinate system by convention so that each follower can convert the work from a neutral representation back into its respective iteration space. This is why our sample leader iterator shifted its results by its low bound before yielding them—to represent them using zero-based coordinates.

Using this same convention, the corresponding follower for the *range* type would be written as follows:

```
iter range.follow(work: range) {
  const mywork = work.translate(this.low);
  for i in mywork do
    yield i;
}
```

This iterator takes a sub-range of work from the leader as its input argument, shifts it back into its natural coordinate system, and then iterates over the resulting range serially, yielding the corresponding values.

To wrap up this example, if *n* was 8 and *numTasks* was 2, the leader iterator would yield sub-range `0..3` for the first task and `4..7` for the second. The first task's follower iterators would convert the *work* sub-range into `1..4`, `0..3`, and `2..5` respectively. These iterators would then be invoked in a serial zippered manner yielding the tuples `(1,0,2)`, `(2,1,3)`, `(3,2,4)`, and `(4,3,5)`. The second task would do similarly for its *work* sub-range.

Note that this example is somewhat simplified by the fact that all of the items being zippered were of the same type and therefore used the same follower iterator. More generally, however, Chapel supports the zippering of any iterators whose followers use compatible work representations as the leader.

5. Sample Leader-Follower Iterators

In this section we provide a series of interesting leader-follower iterators in Chapel, from simple static blockings to dynamic and multi-locale iterators. For simplicity in presentation, we omit the 0-based normalization and de-normalization steps for these iterators.

A coding detail that was ignored for clarity in the previous section relates to the declaration of leader-follower iterators. In our implementation, a serial iterator and its leader-follower variants are correlated via overloading. The three iterators must have the same name, take the same user-supplied arguments (if any), and the follower overload must yield the same type as the serial iterator. The overloads are distinguished from one another through the use of an additional compile-time argument that is added by the compiler to indicate an invocation of the leader or follower.

```
1  // serial iterator
2  iter range.these() {
3    var i = first;
4    const end = if (low > high) then i
5                                else last + 1;
6    while (i != end) {
7      yield i;
8      i += 1;
9  } }

11 // leader iterator overload
12 iter range.these(param tag: iterKind)
13       where tag == iterKind.leader {
14   const numChunks = min(numTasks,
15                         length/minChunk);
16   coforall cid in 0..#numChunks do
17     yield this.getChunk(cid, numChunks);
18 }

20 // follower iterator overload
21 iter range.these(param tag: iterKind,
22                   followThis)
23       where tag == iterKind.follower {
24   for i in followThis do
25     yield i;
26 }
```

Listing 1. Chapel iterators for static fixed-chunk loops.

5.1 Fixed-Chunk Iterators

As an example of this overloading, consider the iterators for the Chapel range type in Listing 1. This set of iterators is designed to chunk the iteration space evenly between its tasks. In the parallel version, the number of tasks is determined by two variables, *numTasks*, specifying the preferred number of tasks, and *minChunk*, indicating the smallest chunk size a task should be assigned. While somewhat simplified, these iterators are illustrative of how we implement default iterators for Chapel's bounded, unstrided range type. This style of iterator is evaluated in Section 7.1.

Lines 2–9 define the serial iterator, which simply computes the start and end bounds and then uses a while loop to yield the integer values. The leader and follower overloads (lines 12–18 and 21–26) take the compile-time (`param`) argument *tag* as their first argument to distinguish themselves from the serial version. The follower also takes a *followThis* argument which represents the work yielded by the leader.

The bodies of the leader and follower are fairly straightforward and similar to those defined in the previous section. The leader computes the number of tasks to use, uses a coforall loop to create the tasks, and computes the work owned by each. The follower simply iterates serially over the indices yielded by the leader. The combination of these three iterators is sufficient to support serial and parallel iterations over simple ranges.

5.2 Dynamic OpenMP-style Leaders

The fixed-size chunking of the preceding example can be a reasonable default iteration style; yet in the presence of irregular workloads or asymmetric architectures, it will tend to result in sub-optimal performance due to load imbalance. In such situations, a dynamic scheduling heuristic that assigns work to tasks as they become idle can be more effective. OpenMP supports examples of such policies through its *dynamic* and *guided* schedule annotations [8]. In this section, we illustrate how such scheduling policies can be written using leader-follower iterators in Chapel.

In the dynamic scheduling case, the total iteration space is conceptually divided into chunks of a fixed, user-specified size. These chunks are then assigned to tasks as they become idle.

```
1  iter dynamic(param tag:iterKind, r:range,
2               chunkSize:int, numTasks:int)
3       where tag == iterKind.leader {
4    var workLeft = r;
5    var splitLock$:sync bool = true;

7    coforall tid in 0..#numTasks {
8      do {
9        splitLock$;          // grab lock
10       const myWork = splitChunk(workLeft,
11                                 chunkSize);
12       splitLock$ = true;  // release lock
13       const haveWork = (myWork.length > 0);
14       if haveWork then yield myWork;
15     } while (haveWork);
16   }
17 }

19 // Helper routine to compute the chunks
20 proc splitChunk(inout workLeft:range,
21                 in chunkSize:int) {
22   const totLen = workLeft.length;
23   chunkSize = min(chunkSize, totLen);
24   const chunk = workLeft#chunkSize;
25   workLeft = workLeft#(chunkSize-totLen);
26   return chunk;
27 }
```

Listing 2. Leader iterator for OpenMP-style dynamic scheduling.

Listing 2 shows an excerpt of a leader iterator named `dynamic()` that implements this strategy. The iterator takes a range (*r*) and chunk size (*chunkSize*) as arguments, as in OpenMP. The coforall loop (line 7) starts *numTasks* tasks. Then each task repeatedly takes chunks from the remaining iterations (*workLeft*) via calls to the helper routine `splitChunk()` (line 10). This operation must be performed in a critical section, so we use the synchronization variable *splitLock$* to ensure the mutual exclusion (lines 9–12). Once the task has its chunk, it yields it to the follower iterator (line 14), which will traverse the corresponding iterations serially.

Lines 20–27 show the details of the helper routine that assigns chunks. It first determines how much of the range to take based on the chunk size and number of elements remaining (line 23). It then uses Chapel's count operator (`#`) to compute the range representing the chunk (line 24) and update the range representing the remaining iterations (line 25).

Although dynamic scheduling may improve load balance compared to static scheduling, it has some drawbacks. The first is that it requires the programmer to select an appropriate chunk size—a poor choice can result in overdecomposition or a poor load balance. Moreover, the optimal choice may vary from one machine or workload to the next. In such cases, using a dynamically varying chunk size can be a better alternative.

OpenMP's guided scheduling option is one such example. It selects a new chunk size each time a task becomes idle [16]. The chunk size is computed by dividing the remaining iterations by a factor that considers the total number of cooperating tasks, in our case *numTasks*. As a result, the size of each new chunk decreases over time.

We have also implemented a leader iterator for this strategy in Chapel, though space does not permit listing or discussing the code here (it is provided in the Appendix for reference). The approach is very similar to that taken in the dynamic scheduling leader, except that the chunk size is re-computed each time according to the heuristic described above. Our implementations of both of these OpenMP-style iterators are evaluated in Section 7.1.

5.3 Adaptive Work-stealing Leader

A drawback to the dynamic and guided leader iterators of the preceding section is their use of a centralized mechanism for calculating the next chunk of work, as represented by the critical section. For processors supporting large numbers of threads, this can result in contention that will impact the performance of the loop.

One solution to this problem is to implement an adaptive splitting algorithm with work-stealing. In an initial step, the iteration space is divided and assigned evenly across tasks, as in static scheduling. Next, each task starts to perform adaptive splitting of its assigned work to get a new chunk. Once a task exhausts its local iterations, it tries to split and steal work from another task that has iterations remaining. In our work, we use a binary splitting strategy in which each chunk is computed by dividing the remaining iterations in half. The leader terminates when no tasks have local work remaining. During the loop's execution, tasks will typically only contend for ranges pairwise, avoiding a centralized bottleneck.

We have implemented this adaptive work-stealing schedule in Chapel, making it the most elaborate leader iterator implemented so far. Space constraints do not permit us to include the code in the body of this paper, though we do include it in the Appendix for reference. Section 7.2 evaluates the scalability of this iterator with respect to a sequential C loop.

Advanced iterators like this one demonstrate a major advantage to permitting users to define their own scheduling policies within a language. Namely, rather than restricting programmers to a finite number of choices defined by a language and its compiler, advanced users are able to implement or modify scheduling policies to suit their needs, while still being able to invoke them using high-level looping constructs like Chapel's forall loop.

5.4 Random Stream Follower

Though most of the examples in this paper have iterated over non-strided integer ranges, more generally leaders and followers can be written to traverse arbitrary data structures and iteration spaces. In the Chapel implementation, we use such iterators to support parallel iteration over a wide variety of regular and irregular array types.

Here, we present another such example: a follower iterator for a stream of pseudo-random numbers. The concept is that the random stream is optimized for the case of generating a series of consecutive random numbers via a pair of helper functions: `getNthRandom()` and `getNextRandom()`.

```
iter RAStream(param tag: iterKind, followThis)
     where tag == iterKind.follower {
  var r = getNthRandom(followThis.low);
  for i in followThis {
    yield r;
    getNextRandom(r);
  }
}
```

We use this follower to implement the HPC Challenge RA benchmark by zippering it with a Block-distributed domain representing the set of updates we need to perform:

```
forall (_, r) in (Updates, RAStream()) do
  // perform random update
```

This benchmark is evaluated in Section 7.3.

5.5 Multi-locale Leader-Followers

Leader-follower iterators can also be used to traverse data structures that are distributed across multiple locales by writing them using Chapel's locality features. Such iterators are used to implement distributed domains and arrays in Chapel. As an illustration,

the following leader distributes its iteration space across locales as well as tasks executing on those locales:

```
iter domain.these(param tag: iterKind)
     where tag == iterKind.leader {
  coforall loc in Locales do
    on loc {
      const locChunk = this.getChunk(loc.id,
                                     numLocales);
      coforall tid in 0..#numTasks do
        yield locChunk.getChunk(tid, numTasks);
    }
}
```

This leader iterator is very similar to those used by default for distributed rectangular domains and arrays in Chapel. In particular, the zippered parallel loops in the HPC Challenge benchmarks evaluated in Section 7.3 are implemented using leader iterators of this form.

Follower iterators for multi-locale settings are also interesting because when asked to follow iterations that are not owned by the task's locale, the follower is responsible for localizing remote elements before yielding them. This gives it control over the communication and buffering granularity. At one extreme, the follower can prefetch everything before yielding anything; at the other, it can fetch and yield elements one at a time. Due to the PGAS model, such remote fetches are expressed simply as references to remote variables in the follower.

6. Implementation

This section provides a very brief description of our implementation of serial and parallel iterators in the Chapel compiler. In the future, we plan to write a follow-on paper to give a more detailed description of our implementation and several key optimizations.

Serial Iterators While new users often worry that high-level iterators like Chapel's will be prohibitively expensive, it's worth noting that many standalone iterators (including all the ones in this paper) can be implemented by replacing the loop header with the iterator's definition and inlining the loop's body in place of the yield statement. The Chapel compiler uses this implementation approach for sufficiently simple iterators: those that are non-recursive, have a small number of yield statements, and are not used in a zippered context.

When serial iterators are used in a zippered context, the compiler creates a class to implement the iterator, generating methods to initialize and advance it, and member variables to store live state across yield statements. The objects for each iterator are then advanced in a lockstep manner.

Parallel Iterators For parallel zippered iteration, the Chapel compiler inlines the leader iterator and then zippers together the follower iterators as in the serial case. In our current implementation, we also implement non-zippered parallel loops in the same manner even though leader-follower iterators are arguably overkill in such cases. As future work, we are considering adding support for a fourth overload that would be used to implement standalone forall loops like this, as a means of optimizing away overheads due to the leader-follower interface.

Status Everything described in this paper is implemented in our open-source Chapel compiler and can be used today under the terms of the BSD license. The release can be obtained from our SourceForge website[1] and contains a primer example demonstrating the authoring of leader-follower iterators[2]. In the release, leader-follower iterators are used to implement parallel iteration for all of our standard domain maps.

[1] http://sourceforge.net/projects/chapel/

[2] $CHPL_HOME/examples/primers/leaderfollower.chpl

workload	delay	# iters
fine-grain	1 microsecond	1e+6
coarse-grain	10 milliseconds	100
triangular	100*(1000-i) microseconds	1000
random	100*rand() milliseconds	1000

Table 1. Synthetic workload parameters to simulate loops with various characteristics.

7. Experimental Results

In this section, we present performance results for our parallel zippered iterators. We show that our Chapel implementations of the dynamic scheduling iterators described in Section 5 perform as well as the corresponding code written using OpenMP. Moreover, we show that the work-stealing leader iterator (an option not available when using OpenMP) can lead to a much better schedule for imbalanced workloads. Finally, we demonstrate the scalability of our iterators by recapping multi-locale performance results from our 2009 HPC Challenge entry [5] that utilized zippered leader-follower iterators.

7.1 Comparison with OpenMP scheduling algorithms

In this section, we compare the Chapel implementations of the dynamic and guided scheduling algorithms presented in Section 5.2 to the corresponding OpenMP versions. We ran our experiment on a 32-core HP ProLiant server with four Intel Xeon X7550 octo-core 2GHz processors, running a SUSE Linux Enterprise Server 11 OS. The Chapel codes were compiled using version 1.3.0 of the Chapel compiler (--fast). Both the Chapel-generated C code and OpenMP were compiled with gcc 4.3.4 (-O3).

For these experiments, we use four synthetic workloads: fine-grain, coarse-grain, triangular and random. Each loop body computes a delay to simulate the particular workload. The rationale for using delays for the loop bodies is to avoid array accesses or other features whose overheads could obscure or mask the comparison of the scheduling algorithms. In this way, we can compare the Chapel scheduling algorithms directly with OpenMP C code. Table 1 lists the delay and number of iterations for each workload. In the triangular workload, i represents the index of the parallel loop. In these experiments, we compare the execution time of Chapel's default fixed-chunk iterators (Section 5.1) to the dynamic schedules as implemented in Chapel and OpenMP. The goals of this comparison are to measure the improvement that can be gained by using a dynamic scheduling strategy on different types of workloads and to compare the Chapel iterator implementations with OpenMP.

Figure 1 shows the speedups vs. the sequential C code for the dynamic scheduling algorithms in Chapel and OpenMP, for 16 and 32 tasks. The speedup of the base static Chapel iterator is shown for reference. The chunk size was set to 10,000 in the fine-grain workload, 2 in the coarse, and 20 in the other two. The figure shows that for all workloads, Chapel's dynamic version can match the performance of the OpenMP version.

The fine-grain workload represents a loop with many independent iterations, each with a very light load. In this workload, the slight variation in the load balance introduced by the partition method of the dynamic strategy when compared to the equal partition of the statically scheduled base iterator is unlikely to improve the performance. In this case, the slowdown over the base version illustrates the overhead of using dynamic scheduling; about 9% for Chapel and OpenMP using 16 tasks, and 20%-24% respectively for 32 tasks. The coarse-grain workload represents a loop with a small

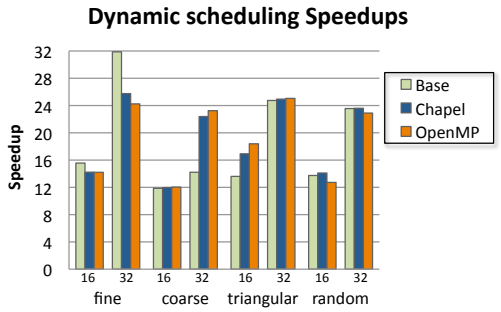

Figure 1. Speedups of the dynamic Chapel and OpenMP scheduling strategies for 16 and 32 tasks. The speedup of the base Chapel iterator (Base) is shown for reference.

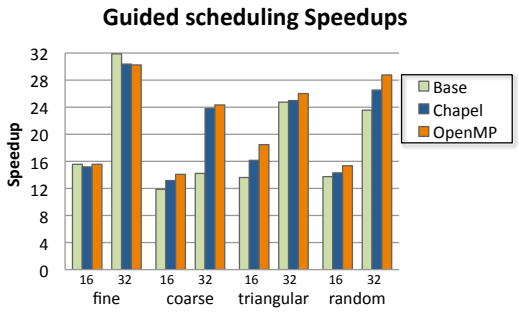

Figure 2. Speedups of the guided Chapel and OpenMP scheduling strategies for 16 and 32 tasks. The speedup of the base Chapel iterator (Base) is shown for reference.

number of independent iterations, each requiring a long execution time. In this workload, the performance will be limited by the equal distribution of work at runtime, not by the overhead. Since the number of iterations in this workload is not a multiple of the number of tasks, we see that the dynamic partitioning strategy outperforms the static one: the improvement is about 57% for Chapel and 63% for OpenMP using 32 tasks.

The other two workloads, triangular and random, are representative of irregular loops in which the parallel iterations are not balanced. The triangular workload represents irregular workloads with a pattern while the random workload represents ones with no pattern. In the triangular workload, the dynamic version outperforms the basic static one for small task counts (seen in the 16-task results). However, when the number of tasks increases, the problem of contention arises in this centralized splitting algorithm, becoming the factor that limits improvement in the 32-task results. In the random workload we observe that the selection of the chunk size is suboptimally balancing the load. As a result, the dynamic scheduling strategy does not improve upon the results of the basic static scheduling. In this case an adaptive splitting strategy would be more appropriate.

Figure 2 shows the performance of the guided scheduling strategy for OpenMP and Chapel using 16 and 32 tasks as compared to Chapel's default static iterator. Again, we see that for all workloads, Chapel's guided version is competitive with the performance of the OpenMP version. Here, the guided strategy achieves better results than the dynamic one, especially for the coarse-grain and random workloads. For instance, an improvement of 12% (Chapel) and 22% (OpenMP) for the irregular random workload is seen for 32 tasks, thanks to the adaptive chunk splitting used by the approach.

7.2 Performance of adaptive work-stealing iterator

In this section, we measure the performance of the Chapel adaptive work-stealing iterator described in Section 5.3. We show the speedups versus the sequential C code for the triangular and random workloads in Figure 3 since we are now interested in a scalability analysis of this more advanced scheduling approach. For reference, the OpenMP guided version is also included in the figure. We see that as we add more tasks, the work-stealing algorithm maintains nearly linear speedup, while the guided strategy drops off. The distributed splitting strategy of our work-stealing iterator reduces the problem of contention when the number of tasks increases, explaining the improved scalability.

Another observation is that our binary adaptive splitting strategy reduces the cost of splitting as compared to the OpenMP guided strategy. The binary splitting strategy performs less splitting which

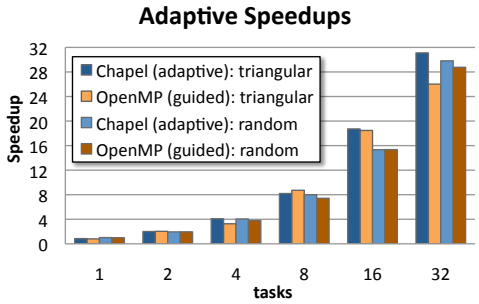

Figure 3. Speedups of the adaptive work-stealing algorithm in Chapel and the guided dynamic scheduling algorithm in OpenMP for the triangular and random workloads.

explains the better results for lower task counts (e.g., speedups on 4–8 tasks).

7.3 Scalability Results

In this section, we evaluate the scalabilty of our multi-locale leader iterator from Section 5.5 as well as our random follower from Section 5.4. These performance results are taken from our 2009 HPC Challenge entry [5] which won the *Most Elegant* prize for that year and produced the fastest STREAM Triad performance in the Class 2 competition.

7.3.1 STREAM Triad

STREAM Triad is a benchmark designed to measure sustainable local memory bandwidth using an embarrassingly parallel vector computation. The Chapel version of the code is implemented using zippered iteration as follows:

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

Here, the *A*, *B*, and *C* variables are 1D arrays.

We have implemented two versions of the benchmark: *stream* and *stream-ep*. Stream-ep is a completely local computation, as in the HPCC reference version. In it, each locale allocates its own local copy of the arrays. As a result, the leader iterator used in the forall loop is similar to the statically blocked leader in Section 5.1. The only overhead required to execute the forall loop is the time required for a locale to create its local tasks.

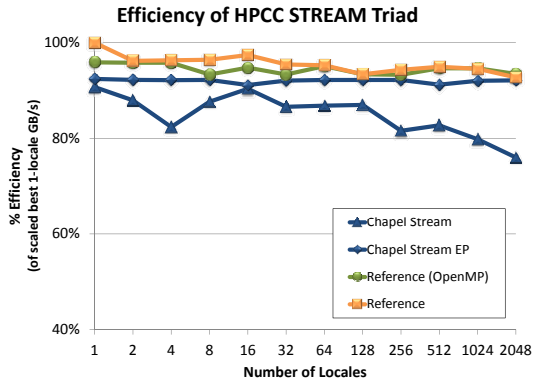

Figure 4. Efficiency of HPCC STREAM Triad on the ORNL XT4 Jaguar system.

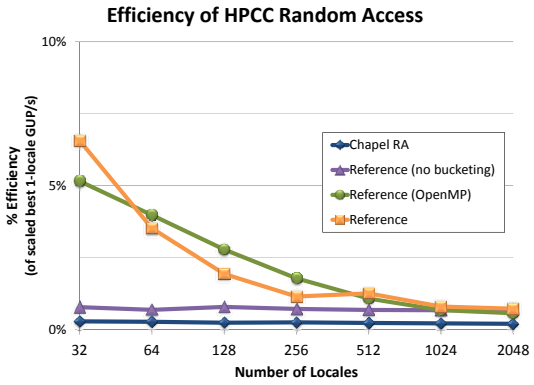

Figure 5. Efficiency of HPCC Random Access on the ORNL XT4 Jaguar system.

Stream is identical to stream-ep, except that the arrays are distributed across multiple locales using Chapel's standard *Block* distribution. The default leader iterator for a Block-distributed array uses a static blocking of the iterations across locales and tasks within each locale as illustrated in Section 5.5. For this reason, the stream benchmark has greater overhead as the number of locales grows due to the time required to launch tasks across all the nodes.

Figure 4 shows the percentage efficiency of stream, stream-ep, and the HPCC reference version with and without OpenMP on the ORNL XT4 Jaguar system (since retired) for 1–2048 locales. The efficiency is computed with respect to linear scaling of the best single locale performance.

We see that stream-ep closely matches the performance of the reference version. This demonstrates that our single-locale iterators have negligible overhead at this scale, similar to OpenMP.

Our global version of stream starts out scaling well, but drops off 20% from the reference version at 2048 locales. This is due to the overhead for loop startup and teardown across thousands of nodes as compared to a completely local parallel loop whose cross-node startup is performed prior to starting the benchmark's timer. Our current Chapel implementation also has some inefficiencies for this style of task launching that we are working on optimizing to reduce this gap. Naturally, such overheads could be amortized by parallel loops that perform more complex computations than a simple vector scale-add. We are also considering changes to the leader-follower interface that would support amortizing these overheads across multiple loops, described in Section 9.

7.3.2 Random Access

The Random Access (RA) benchmark measures the rate of random updates to a distributed table of integers. For the Chapel implementation, we used the random stream follower shown in Section 5.4. The leader iterator for this loop is the same as for the global stream benchmark—that of a Block-distributed domain.

Figure 5 shows the percentage efficiency of RA in Chapel on the ORNL XT4 Jaguar system as compared to three versions of the HPCC reference code: MPI only, MPI and OpenMP, and MPI without a bucketing optimization that coalesces several updates into a single message. This optimization is effective for small numbers of locales, yet as the number of locales approaches the maximum permitted lookahead, the number of updates per locale approaches 1, and the optimization becomes useless. This can be seen as the efficiency of the bucketing-based versions plummet to approach that of the Chapel version, which does no bucketing.

At high locale counts, the performance of the Chapel version is about 25–30% of the best reference version. This gap is due to inefficiencies in the body of our RA loop, such as remote task invocations and array access idioms. rather than overhead from the leader-follower iterators. In fact, the efficiency of the Chapel version is very consistent across machine scale, indicating that the multi-locale leader-follower iterators are scaling well, and that their overheads are being amortized away by RA's expensive loop body, in contrast to the STREAM Triad results in the previous section.

8. Related Work

The most closely related work to ours comes from the NESL language [1, 2] which served as the initial inspiration for supporting parallel zippered iteration in Chapel. NESL supports an *apply-to-each* construct that permits a function to be applied to the elements of one or more sequences, much like the zippered iterator rewrite of function promotion in Chapel. NESL's implementation approach is simpler than Chapel's because all sequences are one-dimensional and implemented by the compiler. In contrast, our work supports zippering of arbitrary iterators, including user-defined iterator functions and data structures by exposing the execution model and implementing mechanisms to the end-user.

Many scripting languages like Python [15] and Ruby [18] support features like *lockstep iteration* or `zip()` methods that provide a similar capability for iterating over multiple objects simultaneously. However, these features are distinct from ours due to their sequential, interpreted context. In contrast, our zippered iterators are designed to support scalable parallel execution on distributed memory systems using compiled code with performance competitive to hand-coded equivalents.

Another class of related work comes from data parallel languages and libraries supporting global-view arrays, such as ZPL [3] and HPF [11, 13]. Such languages have typically supported whole-array operations similar to Chapel's promotion of functions and operators. However, rather than defining their semantics in terms of zippered iteration, they are defined in terms of the pairwise application of operators to the arrays, as discussed in Section 3. Like earlier languages in this section, these data parallel languages also tend to support a small number of compiler-provided array imple-

mentations and therefore do not result in general user-level mechanisms for zippering distinct data structures together as in our work.

A final area of related work is seen in parallel loop scheduling pragmas supported by programming models like OpenMP [8, 10] or that of the Cray XMT (TM) [17]. These pragmas permit the programmer to specify at a high-level how a loop should be parallelized by the underlying compiler. As we demonstrate in Section 5, Chapel's leader-follower iterators support a similar capability for its forall loops. Yet unlike such pragma-based approaches, Chapel's multiresolution design permits these loop schedules to be written by users within the language rather than baking them into the implementation of the compiler and/or runtime. The result is a system that subsumes the capabilities of these techniques while providing greater flexibility and user control.

Within Chapel, we have previously published a description of our initial implementation of serial iterators [12]. This paper greatly expands on that early work by extending it to include zippered parallel iteration.

9. Future Work

While we believe our work on leader-follower iterators constitutes a crucial step forward in terms of providing language-level support for specifying efficient and composable parallel iterators, our current approach has a number of limitations that should be improved over time. One such limitation is related to zippered iteration over multidimensional data structures. Because zippered serial iteration is implemented via methods on an object, the multidimensional control flow is pushed into the object's single *advance* method, which is then zippered with all of the other objects' methods in a 1D loop. This results in a suboptimal control structure compared to the loop nest a programmer would manually write for a multidimensional zippered iteration. To that end, we anticipate extending the leader-follower interface to support distinct iterators for each dimension. Multidimensional loop nests would then be generated by zippering each dimension's iterators separately.

A second concern relates to parallel loop startup overhead. Since each forall loop leader creates parallelism, tasks are created and destroyed for each forall loop. By breaking the leader iterator into two calls—one to create tasks, and the other to assign work to them—the compiler could fuse multiple forall loops, inserting appropriate synchronization to maintain correct semantics while reusing the same tasks.

In both of these scenarios, there will be a need to share state between the collection of iterators that define the leader and followers. To that end, we anticipate moving to more of an object-based framework for leader-follower iterators in which the various iterators would be defined as methods fulfilling an iterator interface. This would also solve some of the awkwardness of grouping iterators via overloading and the compiler-provided *tag* argument.

10. Summary

This paper constitutes the first published description of leader-follower iterators in Chapel and their use in implementing parallel zippered iteration. We have demonstrated that leader-follower iterators are a powerful means of supporting various parallel iteration strategies within a language while supporting invocation via high-level zippered forall loops. As demonstrated in our experiments, these leader-follower iterators can result in efficient and scalable performance. Though work remains to make our leader-follower strategy more efficient, we believe that our current approach serves as a crucial foundation for implementing high-level parallel loops using a multiresolution language philosophy.

Acknowledgments

We would like to thank David Callahan for originally proposing that Chapel should support parallel zippered iteration for the sake of productivity. We'd also like to thank our many colleagues and collaborators on the Chapel project, past and present, for helping us reach this stage of Chapel's development.

References

[1] G. E. Blelloch. NESL: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon, Pittsburgh, PA, September 1995.

[2] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):102–111, April 1994.

[3] B. L. Chamberlain, E. C. Lewis, C. Lin, and L. Snyder. Regions: An abstraction for expressing array computation. In *Proceedings of the ACM International Conference on Array Programming Languages*, 1999.

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

[5] B. L. Chamberlain, S.-E. Choi, S. J. Deitz and D. Iten. HPC Challenge benchmarks in Chapel. (available from http://chapel.cray.com), November 2009.

[6] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, D. Iten, and V. Litvinov. Authoring user-defined domain maps in Chapel. In *Cray Users Group (CUG) 2011*, Fairbanks, AK, May 2011.

[7] Chapel Team. Parallel programming in Chapel: The Cascade High-Productivity Language. http://chapel.cray.com/tutorials.html, November 2010.

[8] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. ISBN 0262533022, 9780262533027.

[9] *Chapel language specification (version 0.8)*. Cray Inc., Seattle, WA, April 2011. (Available at http://chapel.cray.com/).

[10] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January–March 1998.

[11] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1–2):1–170, Spring–Summer 1993.

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

[13] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, September 1996.

[14] E. C. Lewis, C. Lin, and L. Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1998.

[15] M. Lutz. *Learning Python*. Learning Series. O'Reilly Media, Inc., 2009. ISBN 9780596513986.

[16] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, December 1987.

[17] M. Ringenburg and S.-E. Choi. Optimizing loop-level parallelism in Cray XMT (TM) applications. In *Cray Users Group (CUG) 2009)*, Atlanta, GA, May 2009.

[18] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby (2nd edition): the pragmatic programmer's guide*. Pragmatic Bookshelf, October 2004.

Appendix: Additional Chapel Leader Iterators

For reference, this appendix contains the code for the Chapel iterators described in Section 5 and evaluated in Section 7 that did not fit into the paper due to space constraints. Listing 3 shows the leader iterator for our OpenMP-style guided scheduling iterator from Section 5.2. Listing 4 contains the helper function `adaptSplit()` that is used by both the guided and adaptive leader iterators to carve off a chunk of iterations using an adaptive size. Listing 5 contains our adaptive work-stealing leader iterator from Section 5.3.

```
// Guided leader iterator
iter guided(param tag:iterKind, r:range, numTasks:int)
      where tag == iterKind.leader {
  var workLeft = r;
  var moreWork = true;
  var splitLock$:sync bool = true;
  const factor = numTasks;

  coforall tid in 0..#numTasks {
   while moreWork {
      splitLock$;  // grab lock
      const myWork = adaptSplit(workLeft, factor,
                                moreWork);
      splitLock$ = true;  // release lock
      if myWork.length != 0 then yield myWork;
    }
  }
}
```

Listing 3. Leader iterator for OpenMP-style guided scheduling.

```
// Routine to perform adaptive chunk splitting
proc adaptSplit(inout rToSplit:range,
                factor:int, inout itLeft:bool) {
  var totLen, size: int;
  const profThreshold = 1;

  totLen = rToSplit.length;
  if totLen > profThreshold then
    size = max(totLen/factor, profThreshold);
  else {
    size = totLen;
    itLeft = false;
   }
  const firstRange = rToSplit#size;
  rToSplit = rToSplit#(size-totLen);
  return firstRange;
}
```

Listing 4. The `adaptSplit()` helper function.

```
// Leader iterator
iter adaptive(param tag:iterKind, r:range,
              numTasks:int)
     where tag == iterKind.leader {
  const SpaceThrs:domain(1) = 0..#numTasks;
  var localWork:[SpaceThrs] range;
  var moreLocalWork:[SpaceThrs] bool = true;
  var splitLock$:[SpaceThrs] sync bool = true;
  const factor:int = 2;

 var moreWork:bool = true;
  // Variables to put a barrier to
  // compute the initial range
  var barrierCount$:sync int = 0;
  var wait$:single bool;

  coforall tid in 0..#numTasks {

    // Step 1: Initial range per Task
    localWork[tid] = splitRange(r, tid);
    barrier();

    // Step 2: While tid has work, split locally
    while moreLocalWork[tid] {
      const myWork = splitWork(tid);
      if myWork.length != 0 then yield myWork;
   }

    // Step 3: Task tid finished its work, so
    //       will try to steal from a neighbor
    var nVisitedVictims: int = 0;
    var victim = (tid+1) % numTasks;

    while moreWork {
      while moreLocalWork[victim] {
        const myWork = splitWork(victim);
        if myWork.length != 0 then yield myWork;
     }
      nVisitedVictims += 1;
      // Check if there is no more work
      if nVisistedVictims >= numTasks-1
        then moreWork = false;
      else
        victim = (victim+1) % numTasks;
    }
  }

  // Nested helper functions:

  // Routine to distribute the initial range
  proc splitRange(r:range, tid:int) {
    const size = r.length/numTasks;
    var rLocal = (r+tid*size)#size;
    if tid==numTasks-1 then
      rLocal = r#(size*(numTasks-1)-r.length);
    return rLocal;
  }

  // Routine to perform a barrier
  proc barrier() {
    var tmp = barrierCount$;
    barrierCount$ = tmp+1;
    if tmp+1==numTasks then wait$ = true;
    wait$;
  }

  // Routine to split work in a critical section
  proc splitWork(taskid:int) {
    splitLock$[taskid];
    const myWork = adaptSplit(localWork[taskid],
                              factor, moreLocalWork[taskid]);
    splitLock$[taskid] = true;
    return myWork;
  }
}
```

Listing 5. Leader iterator for the adaptive work-stealing iterator.