Iterators in Chapel*

Mackale Joyner[1], Bradford L. Chamberlain[2], and Steven J. Deitz[2]

[1]Rice University
Dept. of Computer Science
Houston, TX 77005 USA
mjoyner@cs.rice.edu

[2]Cray Inc.
Seattle, WA 98104 USA
{bradc, deitz}cray.com

Abstract

A long-held tenet of software engineering is that algorithms and data structures should be specified orthogonally in order to minimize the impact that changes to one will have on the other. Unfortunately, this principle is often not well-supported in scientific and parallel codes due to the lack of abstractions for factoring iteration away from computation in traditional scientific languages. The result is a fragile situation in which complex loop nests are used to express parallelism and maximize performance, yet must be maintained individually as the algorithm and data structures evolve. In this paper, we introduce the iterator concept in the Chapel parallel programming language, designed to address this problem and provide a means for factoring iteration away from computation. The paper illustrates iterators using several examples, compares our approach with those taken in other languages, and describes our implementation in the Chapel compiler.

1. Introduction

Novice programmers are taught that they should separate the specification of their algorithms from the data structures used to implement them, in order to create code that is more robust in the face of changes to either. Unfortunately, scientific computing has a history of mixing the specification of algorithms with their implementations, due in part to the need for performance and in part to the languages that are traditionally used for such applications.

Scientific programmers targeting uni-processors take great care to iterate over their data structures in a manner that will maximize performance by generating loops that will walk through memory in a beneficial order, take advantage of the cache, enable vectorization, and so forth. Since C and Fortran are the most prevalent languages used in this domain, iterations are typically expressed using carefully-architected scalar loop nests. As an example, programmers who wish to iterate over their array elements in a tiled manner will typically need to intersperse all the details associated with tiling (extra loops, bounds calculations, etc.) in with their computation, even though the algorithm probably does not care about these implementation details.

As a scientific code evolves or is ported to new machines, each of these loop nests may need to be rewritten to match the new parameters. One typical scenario involves porting a multidimensional array code from C to Fortran and changing all of its loops to deal with the conversion between arrays allocated in row-major and column-major order. Other porting efforts may require the loops to change due to new cache parameters or vectorization opportunities. In the worst case, every loop nest that contributes to the code's performance may need to be considered and rewritten during this porting process.

When coding for a parallel environment, the problem tends to be even more difficult due to the fact that data structures are potentially distributed between multiple processors. As a result, loops tend to be cluttered by additional details, such as the specification of each processor's local bounds, in addition to the traditional uni-processor concerns described above. By embedding such details within every loop that accesses a distributed data structure, a huge effort is typically required to change the distribution or implementation of the data structure, resulting in code that is brittle and difficult to experiment with. In short, our community has failed to separate algorithms from data structures as intended.

*This work was funded in part by the Defense Advanced Research Projects Agency under its Contract No. NBCH3039003.

This paper describes our attempts to address this fragility within scientific codes by introducing an iterator abstraction within the Chapel parallel programming language [3]. An *iterator* is a software unit that encapsulates general computation, defining the traversal of a possibly multidimensional iteration space. Iterators are used to control loops simply by invoking them within the loop header. Moreover, multiple iterators may be invoked within a single loop using either cross-product or *zippered* semantics. Just as functions allow repeated subcomputations to be factored out of a program and replaced with function calls, iterators support a similar ability to factor common looping structures away from the computations contained within the bodies of those loops. Changes to an iterator's definition will be reflected in all uses of the iterator, and loops can alter their iteration method either by modifying the arguments passed to the iterator or by invoking a different iterator. The result is that users (and in some cases the compiler) can switch between different iteration methods without cluttering the expression of the algorithm or requiring changes to every loop nest.

The contributions of this paper are as follows:

- It provides the first published description of iterators in Chapel and compares them to iteration techniques supported by other languages.
- It describes two implementations of iterators done within the prototype Chapel compiler.
- It provides examples of using iterators that suggest their utility within larger scientific codes.
- It describes different implementation strategies for zippered iteration.

The rest of this paper is organized as follows: *Section 2* provides an overview of Chapel with emphasis on Chapel iterators. *Sections 3.1* and *3.2* describe implementation approaches that we have implemented in the Chapel compiler, using sequences and nested functions, respectively. *Section 4* explores implementation strategies for zippered iteration, an iteration method that combines the yielded values provided by multiple iterators. The final three sections contain related work, future work, and conclusions, respectively.

2. Overview of Chapel

Chapel is an object-oriented language that, along with Fortress [1] and X10 [5], is being developed as part of DARPA's High-Productivity Computing Systems (HPCS) program, challenging supercomputer vendors to increase *productivity* in high-performance computing. The design of Chapel is guided by four key areas of programming language technology: multithreading, locality-awareness, object-orientation, and generic programming. The Object-oriented programming area, which includes Chapel's iterators, helps in managing complexity by separating common function from specific implementation to facilitate reuse. The common function or specification in scientific loops is how to specify the traversal of the iteration space for the data structures referenced inside loops in a way that maximizes reuse and minimizes clutter within the algorithm. This specification can be reused if it is factored away from the implementation of the algorithm. The benefit comes from saving programmers from having to rewrite the specification alongside their computations each time the code traverses those data structures. The separation also allows the programmer to focus on the iteration and computation separately. Chapel iterators provide a framework to achieve this goal effectively.

2.1. Chapel Iterators

Chapel iterators are semantically similar to iterators in CLU [9]. Chapel implements iterators using a function-like syntax, although the semantic behavior of an iterator differs from that of a function in some important ways. Unlike functions, instead of returning a value, Chapel iterators typically return a sequence of values. The *yield* statement, legal only within iterator bodies, returns a value and temporarily suspends the execution of the code within the iterator. As an example, the following Chapel code defines a trivial iterator that yields the first n values from the Fibonacci sequence:

```
iterator fibonacci(n):integer {
  var i1 = 0, i2 = 1;
  var i = 0;
  while i <= n {
    yield i1;
    var i3 = i1 + i2;
    i1 = i2;
    i2 = i3;
    i += 1;
  }
}
```

Chapel invokes iterators using a syntax similar to function calls. Chapel iterator calls commonly appear in loop headers to model the idea of executing the loop body's computation once for each element in a data structure's iteration space. In Chapel, the ordering of a loop's iterations is specified by the iterator call located in the loop header. As a result, all the developer has to do to change the iteration space ordering is to modify the iterator invocation. As an example, the following loop invokes our Fibonacci iterator to generate

```
iterator rmo(d1,d2): 2*integer do
  for i in 1..d1 do
    for j in 1..d2 do
      yield (i,j);

iterator cmo(d1,d2): 2*integer do
  for j in 1..d2 do
    for i in 1..d1 do
      yield (i,j);

iterator tiledcmo(d1,d2): 2*integer {
  var (b1,b2) = computeTileSizes();
  for j in 1..d2 by b2 do
    for i in 1..d1 by b1 do
      for jj in j..min(d2,j+(b2-1)) do
        for ii in i..min(d1,i+(b1-1)) do
          yield (ii,jj);
}

function evolve(d1,d2) do
  for (i,j) in {rmo|cmo|tiledcmo}(d1,d2) {
    u0(i,j) = u0(i,j)*twiddle(i,j);
    u1(i,j) = u0(i,j);
  }
```

Figure 1. A basic iterator example showing how Chapel iterators separate the specification of an iteration from the actual computation.

```
iterator NWBorder(n: integer): 2*integer {
  forall i in 0..n do
    yield (i, 0);
  forall j in 0..n do
    yield (0, j);
}

iterator Diags(n: integer): 2*integer {
  for i in 1..n do
    forall j in 1..i do
      yield (i-j+1, j);
  for i in 2..n do
    forall j in i..n do
      yield (n-j+i, j);
}

var D: domain(2) = [0..n, 0..n],
    Table: [D] integer;

forall i,j in NWBorder(n) do
  Table(i,j) = initialize(i,j);

ordered forall i,j in Diags(n) do
  Table(i,j) = compute(Table(i-1,j),
                       Table(i-1,j-1),
                       Table(i,j-1));
```

Figure 2. A parallel excerpt from the Smith-Waterman algorithm written in Chapel utilizing iterators.

10 values, printing them out as they are yielded:

```
for val in fib(10) do
  write(val);
```

Conceptually, control of execution switches between the iterator and the loop body. Semantically, the loop body executes each time a *yield* statement inside the iterator executes. Upon completion, the loop body transfers control back to the statement following the *yield*. However, control of execution does not switch to the loop body when a *return* statement inside the iterator executes. *Figure 1* provides a more detailed view of how iterators in Chapel may be utilized, using an example based on the NAS parallel benchmark FT [2], where we use the simplicity of our iterators to experiment with tiling. This example shows three iterators that might be used to traverse a 2D index space, and shows that the *evolve* client code can switch between them simply by invoking a different iterator.

Chapel's iterators may be invoked using either sequential *for* loops, as shown above, or parallel *forall* loops. The iterator's body may also be written to utilize parallelism, potentially yielding values using multiple threads of execution. In such cases, the *ordered* keyword may be used when invoking the iterator in order to respect any sequential constraints within the iterator's body. *Figure 2* illustrates this utilizing two Chapel iterators for the Smith-Waterman algorithm, a well-known dynamic programming algorithm in scientific computing that performs DNA sequence comparisons. For more details, the reader is referred to the Chapel language specification [4]. This paper focuses primarily on the implementation of sequential iterators, which represent a crucial building block for efficiently supporting parallel iterators and iteration.

2.2. Invoking Multiple Iterators

Chapel supports two types of simultaneous iteration by adding additional iterator invocations in the loop header. Developers can express cross-product iteration in Chapel by using the following notation:

```
for (i,j) in [iter1(),iter2()] do ...
```

which is equivalent to the nested *for* loop:

```
for i in iter1() do
  for j in iter2() do
    ...
```

Zipper-product iteration is the second type of simultaneous iteration supported by Chapel, and is specified using the following notation:

```
for (i,j) in (iter1(),iter2()) do ...
```

which, assuming that both iterators yield k values, is equivalent to the following pseudocode:

```
for p in 1..k {
  var i = iter1().getNextValue();
  var j = iter2().getNextValue();
  ...
}
```

In this case, the body of the loop will execute each time both iterators yield a value. However, recall that the semantics of the Chapel iterators, differing from normal functions, require that once program execution reaches the last statement in the loop body, control resumes inside the iterator body on the statement immediately following the *yield* statement for each iterator. Zippered iteration would be implemented naturally using coroutines [8], which allow for execution to begin anywhere inside of a function, unlike functions in most current languages. However, without coroutines, zipper-product iteration may still be implemented using techniques we describe in *Section 4*.

2.3. Arrays and Domains

A domain [4] describes a collection of indices for data. The type of a domain's indices may be one of several types including primitive types and class references. Domains are used to represent multidimensional iteration spaces and arrays. Domains are important in Chapel because they enable a high-level distribution of data collections at the collection level rather than at the object level. Conceptually, arrays [4] in Chapel are functions that map the domain indices to variables. Domains and arrays are both first class entities in Chapel. Users may control the distribution of an array by defining the array over a particular domain.

3. Implementation Techniques

Chapel has two iterator implementation techniques, an iterator approach using sequences and an alternate approach using nested functions. We implemented the sequence-based approach first. The motivation for implementing the second technique was to overcome the disadvantages of the first. These two iterator implementation techniques will be described in detail in the next sections.

3.1. Sequence Implementation

Our Chapel compiler's first implementation approach for iterators uses sequences to store the iteration space of the data structures traversed by the loop. Subsequently, the loop body is executed once for each element in the sequence.

Sequences in Chapel are homogeneous lists which support iteration via a built-in iterator. Chapel supports declarations of sequence variables and iterations over them using the following syntax:

```
var aseq: seq(integer) = (/ 1, 2, 4 /);
for myInt in aseq do ...
```

```
// Illustration of compiler transform
function tiledcmo(d1,d2): seq(2*integer) {
  var resultSeq: seq(2*integer);
  var (b1,b2) = computeTileSizes();
  for j in 1..d2 by b2 do
    for i in 1..d1 by b1 do
      for jj in j..min(d2,j+(b2-1)) do
        for ii in i..min(d1,i+(b1-1)) do
          resultSeq.append(ii,jj);
  return resultSeq;
}

function evolve(d1,d2) {
  var resultSeq = tiledcmo(d1,d2);
  for (i,j) in resultSeq {
     u0(i,j) = u0(i,j)*twiddle(i,j);
     u1(i,j) = u0(i,j);
  }
}
```

Figure 3. An implementation of tiled iteration using the sequence-based approach.

where *integer* in this example can be replaced by any type.

In our sequence-based implementation, Chapel first evaluates the iterator call and builds up the sequence of yielded values before executing the loop body. Each time the iterator yields a value, instead of executing the loop body, Chapel appends the value to a sequence. When execution reaches either the end of the iterator or a *return* statement, the iterator returns the constructed sequence of yielded values. Once the iterator returns its sequence of values, Chapel begins executing the loop body once for each element in the sequence returned from the iterator. *Figure 3* illustrates the compiler rewrite that would take place using the sequence-based iteration approach for the tiled iterator of *Figure 1*.

The advantage to using this implementation approach is its simplicity—the Chapel compiler can use the language's built-in support for sequences to capture the iteration space and to control how many times the loop body executes. Another advantage is that the iterator function only needs to be called once. As a result, this approach saves the cost of transferring control back and forth between the iterator and the loop body.

The chief disadvantage to this approach is that it is not general—it can only be applied when the compiler can ensure that no side effects exist between the iterator and loop body. Chapel must impose the side effect restriction because the sequence gathers the iteration space before loop body execution begins. If there was a side effect inside the loop body, such as changing the bounds of the iteration space, incorrect code could be produced. A second disadvantage to this approach

```
// Illustration of compiler transform
function evolve(d1,d2) {
  function tiledcmo(d1,d2) {
    function loopbody(i,j) {
      u0(i,j) = u0(i,j)*twiddle(i,j);
      u1(i,j) = u0(i,j);
    }
    var (b1,b2) = computeTileSizes();
    for j in 1..d2 by b2 do
      for i in 1..d1 by b1 do
        for jj in j..min(d2,j+(b2-1)) do
          for ii in i..min(d1,i+(b1-1)) do
            loopbody(ii,jj);
  }
  tiledcmo(d1,d2);
}
```

Figure 4. An implementation of tiled iteration using the nested function-based approach.

is the space overhead required to store the sequence. The next section details our second implementation approach, which addresses these limitations.

3.2. Nested Function Implementation

Our Chapel compiler's second iterator implementation approach uses nested functions. Currently, this approach works well on a *for* loop containing one iterator call in its loop header. We provide insight on extending this approach to handle zipper-product iteration in *Section 4*.

There are two steps to implementing Chapel iterators with nested functions. The first step involves creating a nested function within the iterator's scope that implements the *for* loop's body and takes the loop indices as its arguments. The second step creates a copy of the iterator, converting it to a function and replacing each *yield* statement in the body with a call to the nested function created during the first step. The transformation passes the value of each yield statement as arguments to the nested function. Once the transformation completes this process, it replaces the original *for* loop with the cloned iterator call, previously located in its loop header. *Figure 4* demonstrates how the Chapel compiler implements iterators using nested functions for the tiling example.

Since the body of the nested function inside the iterator is small, it is often beneficial to inline it. Chapel inlines the nested function calls appearing inside the iterator to eliminate the costs of invoking the nested function every time the iterator yields a value.

The advantage of using the nested function approach for iterators is generality: side effects between the iterator and the *for* loop's body do not have to be identified in fear of producing incorrect code. The execution behavior of this approach is closer to that of CLU [10] and Sather [11] iterators. Another advantage over the sequence-based approach is that Chapel does not need to use storage for the iteration space. The chief disadvantage is that this approach doesn't apply to zipper iteration in the general case.

```
iterator fibonacci(n):integer {
  var i1 = 0, i2 = 1;
  var i = 0;
  while i <= n {
    yield i1;
    var i3 = i1 + i2;
    i1 = i2;
    i2 = i3;
    i += 1;
  }
}

iterator squares(n):integer {
  var i = 0;
  while i <= n {
    yield i * i;
    i += 1;
  }
}

for i, j in fibonacci(12), squares(12) do
  writeln(i, ", ", j);
```

Figure 5. An example of zippered iteration in Chapel.

4. Zippered Iteration

Zipper-product iteration is the process of traversing through multiple iterators simultaneously where each iterator must yield a value once before execution of the loop body can begin. *Figure 5* shows an example of zippered iteration in Chapel. This section describes possible zipper-product implementation approaches that we are exploring as we go forward. Chapel's semantics define that zippered iteration is performed by requiring the iterators involved in the loop to each yield values before the loop body is executed. Recall that semantically, when an iterator yields a value, execution suspends from inside the iterator until the loop body has completed once. When execution resumes inside the iterator, Chapel will execute the statement immediately following the *yield* statement.

In modern languages, the only point of entry for functions is at the top. Coroutines are functions that can have multiple entry points and properly simulate the producer/consumer relationship that simultaneous iteration between two iterators introduces. However, because most modern languages do not support coroutines, programmers must utilize other methods to properly simulate the producer/consumer relationship. Here we consider two techniques, one that uses state variables and one that uses multiple threads via synchronization variables.

Figure 6 shows one technique for implementing zipper-product iteration. The example implements the zippered iteration using state variables. Both iterators use Chapel's *select* statement with *goto* statements to enable simulation of a coroutine, similar to checkpointing in the porch compiler [14]. The state is preserved via the class that is passed into the function. The semantic execution behavior of the iterators is preserved by ensuring that the statement immediately following the *yield* is executed when the iterators are invoked on subsequent calls. Once the last yield is executed, the iterator will not be called again. The advantage of using this approach is that it eliminates the synchronization costs that are associated with our second approach. Also, by having the compiler simulate the coroutine, dead variables do not need to have their state saved. For example, an optimization could be performed to eliminate *i3* from the state class for the Fibonacci iterator. The disadvantage of this approach is the overhead associated with entering and exiting the routine. This could be especially significant in recursive iterators where the stack would result in a large saved state class.

Our second implementation approach for zippered iteration uses multiple threads and synchronization (sync) variables. A *sync variable*[4] transitions to an undefined state when read. When a sync variable is undefined and a computation tries to read from it, the computation will stall until the sync variable is defined. As a result, sync variables allow us to model the producer/consumer relationship of coroutines that is needed to support zippered iteration. Note that the multi-threaded solution requires analysis which determines whether the iterators are parallel-safe or semantics which imply that iterators in a zippered context are executed in parallel.

In *figure 7*, the sync variables are initially undefined. Each sync variable can transition to the defined state inside an iterator. Chapel utilizes the *cobegin* statement to indicate that both iterators should be executed in parallel. The *while* loop inside the *cobegin* statement will stall until each iterator defines its sync variables. A sync variable is created for each iterator and a sync variable assignment replaces each *yield* statement inside the iterator. The chief disadvantage to using this approach lies in the synchronization costs associated with the sync variables. Both approaches enable the support of zippered iteration in Chapel.

```
// Illustration of compiler transform
class ss_fibonacci_state {
  var i1, i2, i3, i:integer;
  var jump = 1;
}
function ss_fibonacci(n, ss):integer {
  select ss.jump {
    when 1 do goto label1;
    when 2 do goto label2;
  }
  label label1 ss.i1 = 0;
  ss.i2 = 1;
  ss.i = 0;
  while ss.i <= n {
    ss.jump = 2;
    return ss.i1;
    label label2 ss.i3 = ss.i1 + ss.i2;
    ss.i1 = ss.i2;
    ss.i2 = ss.i3;
    ss.i += 1;
  }
  ss.jump = 0;
  return 0;
}

class ss_squares_state {
  var i:integer;
  var jump = 1;
}
function ss_squares(n, ss):integer {
  select ss.jump {
    when 1 do goto label1;
    when 2 do goto label2;
  }
  label label1 ss.i = 0;
  while ss.i <= n {
    ss.jump = 2;
    return ss.i * ss.i;
    label label2 ss.i += 1;
  }
  ss.jump = 0;
  return 0;
}

var ss1 = ss_fibonacci_state();
var ss2 = ss_squares_state();
while ss1.jump and ss2.jump do {
  var i = ss_fibonacci(12, ss1);
  var j = ss_squares(12, ss2);
  writeln(i, ", ", j);
}
if ss1.jump or ss2.jump then
  halt("non-equal zippering");
```

Figure 6. An implementation of zippered iteration using state variables.

```
// Illustration of compiler transform
class mt_fibonacci_state {
  sync var flag : boolean;
  sync var result : integer;
}
function mt_fibonacci(n, mt) {
  var i1 = 0, i2 = 1;
  var i = 0;
  while i <= n {
    mt.flag = false;
    mt.result = i1;
    var i3 = i1 + i2;
    i1 = i2;
    i2 = i3;
    i += 1;
  }
  mt.flag = true;
}

class mt_squares_state {
  sync var flag : boolean;
  sync var result : integer;
}
function mt_squares(n, mt) {
  var i = 0;
  while i <= n {
    mt.flag = false;
    mt.result = i * i;
    i += 1;
  }
  mt.flag = true;
}

var mt1 = mt_fibonacci_state();
var mt2 = mt_squares_state();
cobegin {
  mt_fibonacci(12);
  mt_squares(12);
  while not mt1.flag and not mt2.flag do
    writeln(mt1.result, ", ", mt2.result);
}
```

Figure 7. A multi-threaded implementation of zippered iteration using sync variables.

5. Related Work

CLU iterators are semantically similar to those in Chapel. Unlike Chapel, CLU iterators [9, 10] can only be invoked inside a loop header. Both Chapel and CLU support nested iteration. In CLU, only one iterator can be called in the loop header. As a result, CLU does not provide support for zippered iteration.

In contrast to CLU iterators, Sather iterators [11] can be invoked from anywhere inside the loop body. As a result, Sather iterators can support zippered iteration by invoking multiple iterator calls inside the loop body. Since Sather iterators may appear inside the loop body, iterator arguments may be reevaluated for each loop iteration. The semantics of Sather iterators are similar to both Chapel and CLU iterators. Sather iterators support zippered iteration as well as nested iterator calls. However, Chapel's focus on effective iteration in a parallel environment separates itself from Sather.

A coroutine [8] is a routine that yields or produces values for another routine to consume. Unlike functions in most modern languages, coroutines have multiple points of entry. When encountering the *yield* in a coroutine, execution of the routine is suspended. The routine saves the return value, program counter, and local static variables in some place other than a stack. When the routine invocation occurs again, the execution resumes after the yield.

Java [7], Python [12], and C++ [13] STL provide iterators that are not tightly coupled to loops like Chapel, CLU, and Sather iterators. These iterators are normally associated with a container class. These languages support simultaneous iteration on containers. Within these languages, only Python provides built-in support to perform iteration using special *for* loops that implicitly grab each element in the container, thereby separating the specification of the algorithm from its implementation. However, Python's special *for* loops do not support zippered iteration since they may call only one iterator in the loop header.

Sisal [6] and Titanium [15] also provide some support for iterators using loops. Titanium has a *foreach* loop that performs iteration over arrays when given their domains. Sisal supports 3 basic types of iterators using a *for* loop. The first type iterates over a range specified by an lower and upper bound. The second type of iterator returns the elements of an array or stream (a stream is a data structure that is similar to an array). The third type of iterator returns tuples of a dot- or cross-product constructed from two range iterators. Sisal and Titanium iterators are limited when compared to Chapel iterators.

6. Future Work

Currently, our two Chapel iterator implementation approaches support sequential applications well. We will extend and combine these iterators with distributed domains to support parallel iteration over distributed arrays and index sets. We will also add Chapel iterator support for parallel zippered iteration to our implementation using the strategies outlined in this paper.

7. Conclusion

We showed that Chapel iterators can effectively separate the specification of an algorithm from its implementation, thereby enabling programmers to easily switch between different specifications while also allowing them to focus on the algorithm's implementation. Using iterators to handle specifications such as iteration space ordering allows programmers to reuse specifications instead of having to write a specification for an algorithm each time the programmer implements an algorithm. This paper describes two different strategies that we have implemented in the Chapel compiler to support Chapel iterators. The first approach was to implement Chapel iterators with sequences. This approach was satisfactory under imposed restrictions. The second approach was to implement Chapel iterators with nested functions. This strategy eliminates some of its imposed restrictions and spatial overhead.

We also gave two different techniques that Chapel could employ to support zippered iteration. The first technique used static variables. The advantage of using this technique was the savings in synchronization costs over alternative methods. The second technique took advantage of Chapel sync variables and may be more feasible to implement than the first approach, but relies on support for multithreading.

References

[1] E. Allen, D. Chase, V. Luchangco, J. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. *The Fortress Language Specification (version 0.785)*. Sun Microsystems Inc., Nov. 2005.

[2] D. Bailey, T. Harris, W. Saphir, R. F. Van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report RNR-95-020, NASA Ames Research Center, Moffett Field, CA, Dec. 1995.

[3] D. Callahan, B. Chamberlain, and H. Zima. The Cascade High Productivity Language. *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60, Apr. 2004.

[4] Cray Inc., Seattle, WA. *Chapel Specification (version 0.4)*, Feb. 2005. http://chapel.cs.washington.edu.

[5] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access. *3rd International Workshop on Language Runtimes*, Oct. 2004.

[6] J. Feo, D. Cann, and R. Oldehoeft. A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (3rd Edition)*. Addison-Wesley, 2005.

[8] D. Grune. A View of Coroutines. *ACM SIGPLAN*, 12(7):75–81, July 1977.

[9] B. Liskov. A History of CLU. *ACM SIGPLAN Conference on History of Programming Languages*, 28(3):133–147, 1993.

[10] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[11] S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski. Iteration Abstraction in Sather. *ACM TOPLAS*, 18(1):1–15, Jan. 1996.

[12] G. Rossum. Python Reference Manual. Technical Report CS-R9525, CWI, 1995. http://www.python.org/doc/ref/ref.html.

[13] B. Stroustrup. *The C++ Programming Language (Special Edition)*. Addison-Wesley, 2000.

[14] V. Strumpen. Compiler Technology for Portable Checkpoints. Technical report, Massachusetts Institute of Technology, 1998. Submitted for publication.

[15] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. *Concurrency: Practice and Experience*, 10(11), Sept. 1998.