Affine Loop Optimization Using Modulo Unrolling in Chapel

Aroon Sharma

June 3, 2014

Abstract

This paper presents a loop optimization for message passing programs that use affine array accesses in Chapel, a Partitioned Global Address Space (PGAS) parallel programming language. The optimization is based on a technique known as modulo unrolling, where the locality of any affine array access can be deduced at compile time if the data is distributed in a cyclic or block cyclic fashion. In this work, modulo unrolling is used to decide when to aggregate messages thereby reducing the overall message count and run time for a particular loop. Compared to other methods, modulo unrolling greatly simplifies the complex problem of automatic code generation of message passing code. It also results in substantial performance improvement compared to the non-optimized Chapel compiler.

To implement this optimization in Chapel, we modify the leader and follower iterators in the Cyclic and Block Cyclic data distribution modules. Results were collected that compare the performance of Chapel programs optimized with modulo unrolling with Chapel programs using the existing Chapel data distributions. Data collected for eleven parallel benchmarks on a ten-locale cluster show that on average, modulo unrolling used with Chapel's Cyclic distribution results in 76 percent fewer messages and a 45 percent decrease in runtime. Similarly, modulo unrolling used with Chapel's Block Cyclic distribution results in 72 percent fewer messages and a 52 percent decrease in runtime for data collected for two parallel benchmarks.

1 Introduction

Compilation of programs for distributed memory architectures using message passing is a vital task with potential for speedups over existing techniques. The partitioned global address space (PGAS) parallel programming model exposes locality of reference information to the programmer thereby improving programmability and allowing for compile-time performance optimizations. In particular, programs compiled to message passing hardware can improve in performance by aggregating messages and eliminating dynamic locality checks for affine array accesses in the PGAS model.

Message passing code generation is a difficult task for an optimizing compiler targeting a distributed memory architecture. These architectures are comprised of independent units of computation called locales. Each locale has its own set of processors, cores, memory, and address space. For programs executed on these architectures, data is distributed across various locales of the system, and the compiler needs to reason about locality in order to determine whether a program data access is remote (requiring a message to another locale to request a data element) or local (requiring no message and accessing the data element on the locale's own memory). Only a compiler with sufficient knowledge about locality can compile a program in this way with good communication performance.

Each remote data memory access results in a message with some non-trivial run time overhead, which can drastically slow down a program's execution time. This overhead is caused by latency on the interconnection network and locality checks for each data element. Accessing multiple remote data elements individually results in this run time overhead being incurred multiple times, whereas if they are transferred in bulk the overhead is only incurred once. Therefore, aggregating messages improves performance of message passing codes. In order to transfer remote data elements in bulk, the compiler must be sure that all elements in question are remote before the message is sent.

The vast majority of loops in scientific programs access data using affine array accesses. An affine array access is a linear expression of the loop's index variables. Loops using affine array accesses are special because they exhibit regular access patterns within a data distribution. Compilers can use this information to decide when message aggregation can take place.

Although a simple concept to understand, message aggregation is the most complicated part of message passing code generation. Existing methods [7, 18] split the loop iteration space into tiles (also known as footprints) of consecutive iterations, and each locale executes its tile's iterations in parallel. Communication using aggregation takes place between locales just before and after the computation within a single tile. These methods require the compiler to do complex footprint calculations, optimizing for tile size and shape, before message aggregation can take place. As [14] shows, determining tile shape quickly becomes difficult for more complicated loop structures, since the optimum tile shape for a computation is not always rectangular. It is our belief that message aggregation using tiling is not used in production quality compilers today because of the complexity of this footprint analysis. What is needed is a simple, robust, and widely applicable method for message aggregation that leads to improvements in performance.

This paper presents a loop optimization for message passing code generation based on a technique called modulo unrolling, where the locality of any affine array access can be deduced at compile time if the data is distributed in a cyclic or block cyclic fashion. The optimization can be performed by a compiler to aggregate messages and reduce a program's execution time and communication. Modulo unrolling in its original form, pioneered by [1], was meant to target tiled architectures such as the MIT Raw machine, not distributed memory architectures that use message passing. It has since been modified to apply to such machines in this work. In this work, modulo unrolling unrolls an affine loop by a factor equal to the number of locales of the machine being utilized by the program. If the arrays used in the loop are distributed cyclically or block cyclically, each array access is guaranteed reside on a single locale across all iterations of the loop. Using this information, the compiler can then aggregate all remote array accesses that reside on a remote locale into a single message before the loop. If remote array elements are written to during the loop, one message is required to store these elements back to each remote locale after the loop runs.

Chapel is an explicitly parallel programming language developed by Cray Inc. that falls under the Partitioned Global Address Space (PGAS) memory model. Here, a system's memory is abstracted to a single global address space regardless of the hardware architecture and is then logically divided per locale and thread of execution. By doing so, locality of reference can easily be exploited no matter how the system architecture is organized. The Chapel compiler is an open source project used by many in industry and academic settings. The language contains many high level features such as zippered iteration and array slicing that greatly simplify the implementation of modulo unrolling into the language.

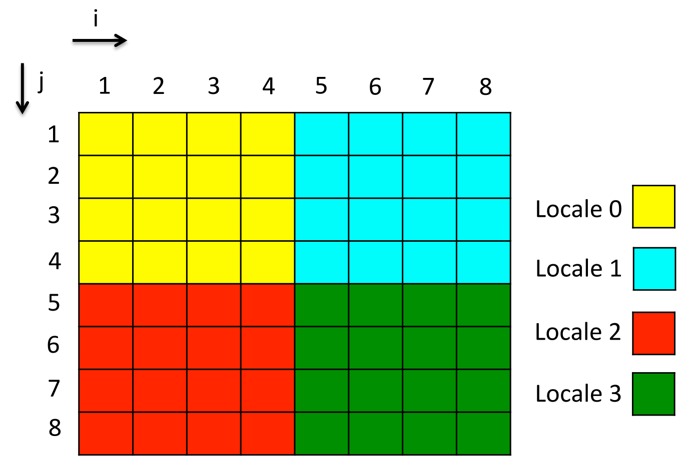

Figure 1: Chapel Block distribution.

1.1 Chapel's Data Distributions

In this work, we consider three types of data distributions: Block, Cyclic, and Block Cyclic. In a Block distribution, elements of an array are mapped to locales evenly in a dense manner. In a Cyclic distribution, elements of an array are mapped in a round-robin manner across locales. In a Block Cyclic distribution, a number of elements specified by a block size parameter is allocated to consecutive array indices in a round robin fashion. Figures 1 - 3 illustrate these three Chapel distributions for a two-dimensional array. In Figure 2, the array takes a 2 x 2 block size parameter.

The choice of data distribution to use for a program boils down to computation and communication efficiency. It has been shown that finding an optimal data distribution for parallel processing applications is an NP-complete problem, even for one or two dimensional arrays [11]. Certain program data access patterns will result in fewer communication calls if the data is distributed in a particular way. For example, many loops in stencil programs that contain nearest neighbor computation will have better communication performance if the data is distributed using a Block distribution. This occurs because on a given loop iteration, the elements accessed are near each other in the array and therefore more likely to reside on the same locale block. Accessing elements on the same block does not require a remote data access and can be done faster. However, programs that access array elements far away from each other will have better communication performance if data is distributed using a Cyclic distribution. Here, a Block distribution is almost guaranteed to have poor performance because the farther away accessed elements are the more likely they are to reside on different locales.

A programmer may choose a particular data distribution for reasons unknown to the compiler. These reasons may not even take communication behavior into account. For example, Cyclic and Block Cyclic distributions provide better load balancing of data across locales than a Block distribution because elements can be added or removed according to a regular predictable pattern. In many applications, data redistribution may be needed if elements of a data set are inserted or deleted. In particular, algorithms to redistribute data using a new block size exist for the Block Cyclic distribution [16, 12]. If an application uses a dynamic data set with elements being appended, a Cyclic or Block Cyclic distribution is superior to Block because new elements are added to the locale that follows the cyclic or block cyclic pattern. For Block, the entire

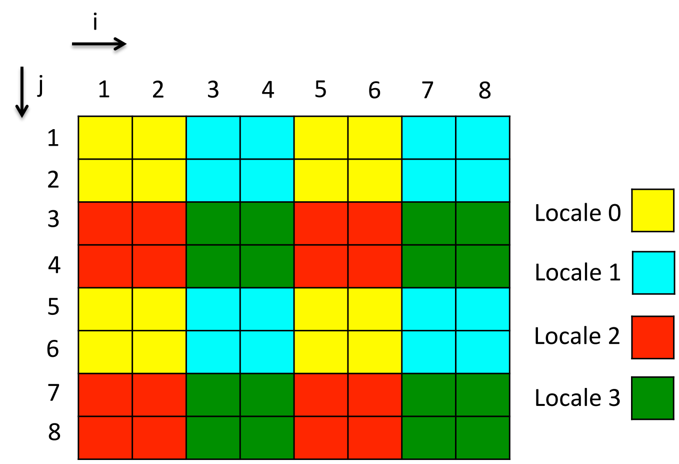

Figure 2: Chapel Block Cyclic distribution with a 2 x 2 blocksize parameter.

data set would need to be redistributed every time a new element is appended, which can be expensive.

Compatibility with other PGAS languages might be an important consideration for a programmer when selecting a data distribution. Data sets used by Chapel programs and other PGAS programs should use Cyclic or Block Cyclic distributions because other PGAS languages do not support the Block distribution. A programmer would benefit by distributing the same data set using only one scheme so the data would not have to be replicated for different programs.

Therefore, in this work, it is our view that the compiler should not change the programmer's choice of data distribution in order to achieve better runtime and communication performance. The compiler should attempt to perform optimizations based on the data distribution that the programmer specified.

2 Related Work

Compilation for distribution memory machines has two main steps: loop optimizations and message passing code generation. First, the compiler performs loop transformations and optimizations to

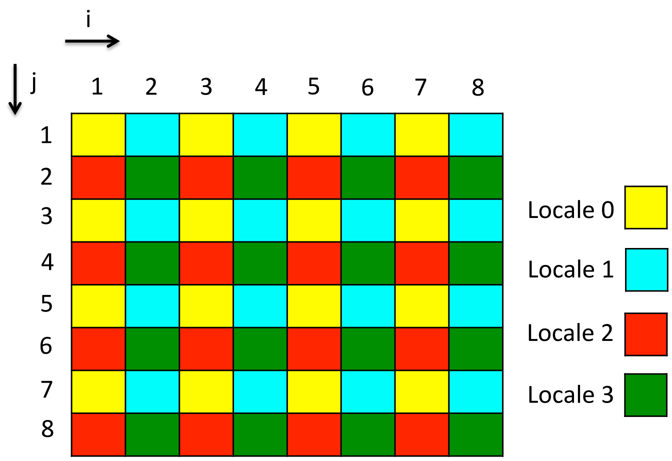

Figure 3: Chapel Cyclic distribution.

uncover parallelism, improve the granularity of parallelism, and improve cache performance. These transformations include loop peeling, loop reversal, and loop interchange. Chapel is an explicitly parallel language, so uncovering parallelism is not needed. Other loop optimizations to improve the granularity of parallelism and improve cache performance are orthogonal to this paper. The second step is message passing code generation, which includes message aggregation.

Message passing code generation in the traditional model is exceedingly complex, and practical robust implementations are hard to find. These methods [7, 18, 2, 13] require footprint calculations for each tile. A footprint is the span of data elements accessed by all iterations of a single tile. It is common for a tile's footprint to span across multiple locales, requiring communication between locales. Footprint calculations are modeled by matrices and need to be intersected with the data distribution in order to determine the locality of data elements. Message aggregation can only be done once the compiler determines which data elements of a tile's footprint are remote. These footprint calculations quickly become more complex as the number of locales scales. Our method does not require any footprint calculations, thereby simplifying code generation.

The polyhedral method is another branch of compiler optimization that seeks to speed up parallel programs on distributed memory architectures [4, 6, 8, 9, 10, 17]. However, the method at its core does not compute information for message passing code generation.

Similar work to take advantage of communication aggregation on distributed arrays has already been done in Chapel. Like distributed parallel loops in Chapel, whole array assignment suffered from locality checks for every array element, even when the locality of certain elements is known in advance. In [15], aggregation was applied to improve the communication performance of whole array assignments for Chapel's Block and Cyclic distributions. Our work goes beyond array assignments and applies aggregation to affine array accesses within parallel loops for Chapel's Cyclic and Block Cyclic distributions. One of the contribution's of [15] included two new bulk communication primitives for Chapel developers as library calls, `chpl_comm_gets` and `chpl_comm_puts`. They both rely on the GASNet networking layer, a portion of the Chapel runtime. Our optimization uses these new communication primitives in our implementation directly to perform bulk remote data transfer between locales.

Extensive work has been done with the UPC compiler (another PGAS language) by [5] to improve on its communication performance. This method targets fine-grained communication and uses techniques such as redundancy elimination, split-phase communication, and communication coalescing (similar to message aggregation) to reduce overall communication. However, it is not clear whether this method can be used to improve communication performance across distributed parallel loops. There is no locality analysis that statically determines whether an array access is shared or remote. Our method, modulo unrolling, can determine which accesses are local purely on the affine array access and data distribution. Another major limitation to this work's aggregation scheme is that only contiguous data can be sent in bulk. To aggregate data across an entire loop in a single message, it must be possible to aggregate data elements that are far apart in memory, separated by a fixed stride. Our method handles this by using the strided get and put calls (`chpl_comm_gets` and `chpl_comm_puts`) from [15], described earlier.

3 Modulo Unrolling

Modulo unrolling [1] is a bank disambiguation method used in tiled architectures that is applicable to loops with affine array accesses. An affine function of a set of variables is defined as a linear combination of those variables. An affine array access is any array access where each dimension of the array is accessed by an affine function of the loop induction variables. For example, for loop index variables i and j and array A, $A[i+2j+3][2j]$ is an affine access, but $A[ij+4][j^2]$ and $A[2i^2+1][ij]$ are not. Modulo unrolling works by unrolling the loop by a factor equal to the number of memory banks on the architecture. If the arrays accessed within the loop are distributed using low-order interleaving (a Cyclic distribution), then after unrolling, each array access will be guaranteed to reside on a single bank for all iterations of the loop. This is achieved with a modest increase of the code size.

To understand modulo unrolling, refer to Figure 4. In Figure 4a there is a code fragment consisting of a sequential **for** loop with a single array access $A[i]$. The array A is distributed over four memory banks using a Cyclic distribution. As is, the array A is not bank disambiguated because accesses of $A[i]$ go to different memory banks on different iterations of the loop. The array access $A[i]$ has bank access patterns 0, 1, 2, 3, 0, 1, 2, 3, ... in successive loop iterations.

A naive approach to achieving bank disambiguation is to fully unroll the loop, as shown in Figure 4b. Here, the original loop is unrolled by a factor of 100. Because each array access is independent of the loop induction variable i, bank disambiguation is achieved trivially. Each array access resides on a single memory bank. However, fully unrolling the loop is not an ideal solution to achieving bank disambiguation because of the large increase in code size. This increase in code size is bounded by the unroll factor, which may be extremely large for loops iterating over large arrays. Fully unrolling the loop may not even be possible for a loop bound that is unknown at compile time.

A more practical approach to achieving bank disambiguation without a dramatic increase in code size is to unroll the loop by a factor equal to the number of banks on the architecture. This is shown in Figure 4c and is known as modulo unrolling. Since we

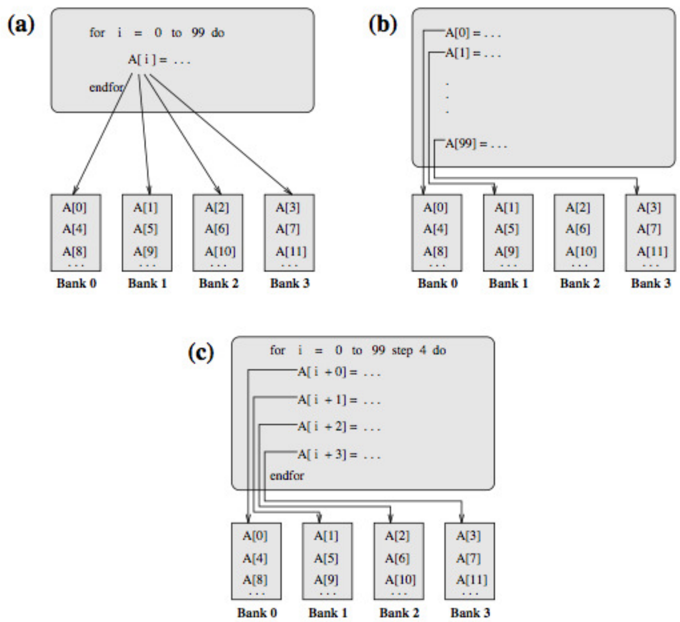

Figure 4: Modulo unrolling example. (a) Original sequential for loop. Array A is distributed using a Cyclic distribution. Each array access maps to a different memory bank on successive loop iterations. (b) Fully unrolled loop. Trivially, each array access maps to a single memory bank because each access only occurs once. This loop dramatically increases the code size for loops traversing through large data sets. (c) Loop transformed using modulo unrolling. The loop is unrolled by a factor equal to the number of memory banks on the architecture. Now each array access is guaranteed to map to a single memory bank for all loop iterations and code size increases only by the loop unroll factor.

have 4 memory banks in this example, we unroll the loop by a factor of 4. Now every array reference in the loop maps to a single memory bank on all iterations of the loop. Specifically, $A[i]$ refers to bank 0, $A[i+1]$ refers to bank 1, $A[i+2]$ refers to bank 2, and $A[i+3]$ refers to bank 3.

Modulo unrolling, as used in [1] provides bank disambiguation and memory parallelism for tiled architectures. That is, after unrolling, each array access can be done in parallel because array accesses map to a different memory banks. However, as we will show, modulo unrolling can also be used to aggregate messages and reduce communication costs in message passing machines.

4 Motivation for Message Aggregation

In Chapel, a program's data access patterns and the programmer's choice of data distribution greatly influence the program's runtime and communication behavior. There are some situations where programs exhibit predictable patterns of communication that the compiler can detect. In doing so, the compiler can aggregate remote data elements coming from one locale into one local buffer via a single message and then access this local buffer on subsequent iterations of the loop.

For example, consider Chapel code for the Jacobi computation shown in Figure 5, a common stencil operation that computes elements of a two dimensional array as an average of that element's four adjacent neighbors. On each iteration of the loop, five array elements are accessed in an affine manner: the current array element $A_{new}[i,j]$ and its four adjacent neighbors $A[i+1,j]$, $A[i-1,j]$, $A[i,j+1]$, and $A[i,j-1]$. Naturally, the computation will take place on the locale of $A_{new}[i,j]$, the element being written to. If arrays A and A_{new} are distributed with a Cyclic distribution as shown in Figure 3, then it is guaranteed that $A[i+1,j]$, $A[i-1,j]$, $A[i,j+1]$, and $A[i,j-1]$ will not reside on the same locale as $A_{new}[i,j]$ **for all iterations of the loop**. These remote elements are transferred over to $A_{new}[i,j]$'s locale in four individual messages during every loop iteration. For large data sets, transferring four elements individually per loop iteration drastically slows down the program because the message overhead is incurred more than once.

Because the data is distributed using a Cyclic distribution, we notice that the data is accessed in the same way every cycle. Consider two iterations that are on different cycles, $(i,j) = (2,2)$ and $(i,j) = (4,2)$. Both iterations of the loop take place on locale 3, and both access 2 remote data elements from locale 1 and 2 remote data elements from locale 2. The remote data elements being accessed each cycle are a known fixed distance away from each other within the array A. We can therefore bring in all remote data elements accessed by iterations where $A_{new}[i,j]$ resides on locale 3 to locale 3 before the loop executes, access them from this local storage, and write them back to locales 1 and 2 after the loop finishes. Figure 6 illustrates this process in detail.

If we focus on locale 3, there will be four buffers containing remote data elements after aggregation has occurred, one for each affine array access in the loop in Figure 5. Now that a copy of all remote data elements reside on the locale that they are used from, the affine array accesses other than $A_{new}[i,j]$ can be replaced with accesses to the local buffers. After the loop has finished, any buffer elements that have been written to are communicated back to their respective remote locales in their own aggregate messages. This optimization can also be applied to the Block Cyclic distribution, as the data access pattern is the same for elements in the same position within a block.

If arrays A and A_{new} are instead distributed using Chapel's Block or Block Cyclic distributions as shown in Figure 1 and Figure 2 respectively, the program will only perform remote data accesses on iterations of the loop where element $A_{new}[i,j]$ is on the boundary of a block. As the block size increases, the number of remote data accesses for the Jacobi computation decreases. For the Jacobi computation, it is clear that distributing the data using Chapel's Block distribution is the best choice in terms of communication. Executing the program using a Block distribution will result in fewer remote data accesses than when using a Block Cyclic distribution. Similarly, executing the program using a Block Cyclic distribution will result in fewer remote data accesses than when using a Cyclic distribution.

It is important to note that the Block distribution is not the best choice for all programs using affine array accesses. Programs with strided access patterns that use a Block distribution will have poor communication performance because accessed array elements are more likely to reside outside of a block boundary. For these types of programs, a Cyclic or Block Cyclic distribution will perform better.

```
var n: int = 8;
var LoopSpace = {2..n-1, 2..n-1};

//Jacobi relaxation pass
forall (i,j) in LoopSpace {
  A_new[i,j] = (A[i+1, j] + A[i-1, j] + A[i, j+1] + A[i, j-1])/4.0;
}

//update state of the system after the first relaxation pass
A[LoopSpace] = A_new[LoopSpace];
```

Figure 5: Chapel code for Jacobi computation over an 8 x 8 two dimensional array. Arrays A and A_{new} are distributed with a Cyclic distribution and their declarations are not shown. During each iteration of the loop, the current array element $A_{new}[i, j]$ gets the average of the four adjacent array elements of $A[i, j]$.

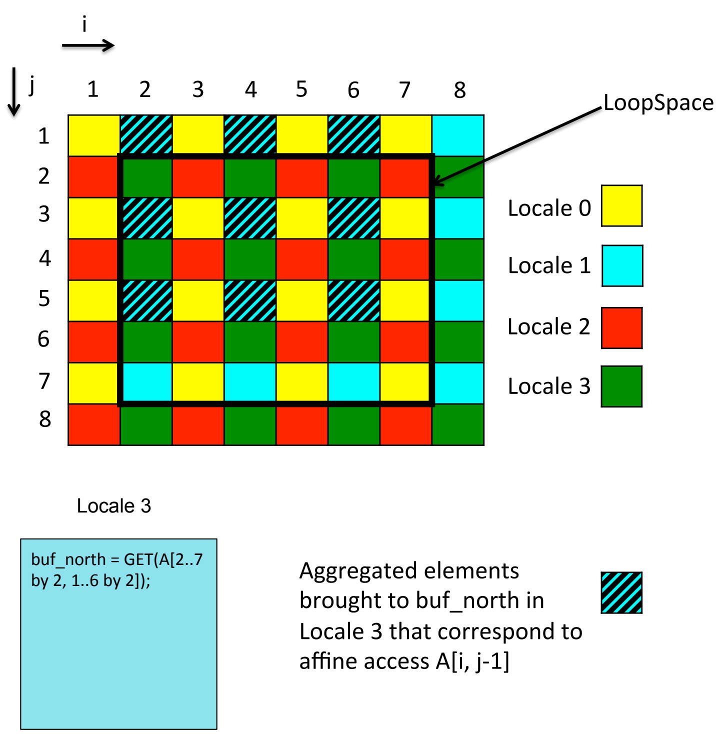

Figure 6: Illustration of message aggregation for the $A[i, j-1]$ affine array access of the Jacobi relaxation computation. The region *LoopSpace* follows from Figure 5. When $(i, j) = (2, 2)$, $A_{new}[2, 2]$ resides on locale 3. $A[2, 1]$ corresponds to the $A[i, j-1]$ access during this iteration, and it resides remotely on locale 1. If we now look at the next iteration where $A_{new}[i, j]$ resides on locale 3 (the next cycle, which is $(i, j) = (4, 2)$), we see that $A[4, 1]$ also resides on locale 1. We notice a pattern that all remote data accesses with respect to locale 3 corresponding to the $A[i, j-1]$ access in the loop **form an array slice** $A[2..7\ by\ 2, 1..6\ by\ 2]$, which we can aggregate with a single GET call and bring into buf_north on locale 3 before the loop begins. The array slice contains strided accesses of 2 in both the i and j dimensions, denoted using the Chapel keyword *by* within the array slice. The striped elements form the elements that have been aggregated. This same procedure occurs on each locale for each affine array access that is deemed to be remote for all iterations of the loop.

5 Chapel Language Features Necessary for Modulo Unrolling

The goal of this section is to provide a basic understanding of Chapel's zippered iteration, array slicing, and how the modulo unrolling optimization described in Sections 3 and 4 fits in naturally with these language constructs. Any parallel loop with affine array accesses can be written using zippered iteration. Therefore, zippered iteration serves as a clear spot within the language to implement the optimization.

5.1 Zippered Iteration

Iteration is a widely used language feature in the Chapel programming language. Chapel iterators are blocks of code that are similar to functions and methods except that iterators can return multiple values back to the call site with the use of the *yield* keyword instead of *return*. Iterators are commonly used in loops to traverse data structures in a particular fashion. For example, an iterator $fibonacci(n : int)$ might be responsible for yielding the first n Fibonacci numbers. This iterator could then be called in a loop's header to execute iterations 0, 1, 1, 2, 3, and so on. Arrays themselves are iterable in Chapel by default.

Chapel allows multiple iterators of the same size and shape to be iterated through simultaneously. This is known as *zippered iteration* [3]. When zippered iteration is used, correpsonding iterations are processed together. On each loop iteration, an n-tuple is generated, where n is the number of items in the zippering. The d^{th} component of the tuple generated on loop iteration j is the j^{th} item that would be yielded by iterator d in the zippering. Figure 7 shows an example of zippered iteration used in a Chapel **for** loop.

Zippered iteration can be used with either sequential **for** loops or parallel **forall** loops in Chapel. Parallel zippered iteration is implemented in Chapel using leader-follower semantics. That is, a leader iterator is responsible for creating tasks and dividing up the work to carry out the parallelism. A follower iterator performs the work specified by the leader iterator for each task and generally resembles a serial iterator.

5.2 Array Slicing

Chapel supports another useful language feature known as *array slicing*. This feature allows portions of an array to be accessed and modified in a succinct fashion. For example, consider two arrays A and B containing indices from 1..10. Suppose we wanted to assign elements $A[6]$, $A[7]$, and $A[8]$ to elements $B[1]$, $B[2]$, and $B[3]$ respectively. We could achieve this in one statement by writing $B[1..3] = A[6..8]$. Here, $A[6..8]$ is a slice of the original array A, and $B[1..3]$ is a slice of the original array B. An array slice can support a range of elements with a stride in some cases. For example, in the previous example, we could have made the assignment $B[1..3] = A[1..6\ by\ 2]$. This would have assigned elements $A[1]$, $A[3]$, and $A[5]$ to elements $B[1]$, $B[2]$, and $B[3]$ respectively. Since all array slices in Chapel are arrays themselves, array slices are also iterable.

Together, array slicing and parallel zippered iteration can express any parallel affine loop in Chapel that uses affine array accesses. Each affine array access is replaced with a corresponding array slice, which produces the same elements as the original loop.

(a)

```
for (i, f) in zip(1..5, fibonacci(5)) {
  writeln("Fibonacci ", i, " = ", f);
}
```

(b)

```
Fibonacci 1 = 0
Fibonacci 2 = 1
Fibonacci 3 = 1
Fibonacci 4 = 2
Fibonacci 5 = 3
```

Figure 7: (a) Chapel code fragment showing a loop using zippered iteration. A tuple of loop index variables equal to the number of items in the zippering is declared in the loop header. If j is the current loop iteration, variable i is equal to the j^{th} element in the range $1..5$, and f corresponds to the j^{th} element in the iterator $fibonacci(5)$. The *zip* keyword tells the loop header which items to iterate over using zippered iteration. (b) Program output of the code fragment in Figure 7a.

(a)

```
forall i in 1..10 {
  A[i] = B[i+2];
}
```

(b)

```
forall (a,b) in zip(A[1..10], B[3..12]) {
  a = b;
}
```

Figure 8: (a) Original loop written using a single loop induction variable i ranging from 1 to 10. (b) The same loop written using zippered iteration. Instead of a loop induction variable and a range of values to denote the loop bounds, two array slices containing 10 elements each are specified.

Consider the code fragment in Figure 8a. There are two affine array accesses $A[i]$ and $B[i + 2]$ in Figure 8a. The loop is written in a standard way where the loop induction variable i takes on values from 1 to 10. Because the loop is a **forall** loop, loop iterations are not guaranteed to complete in a specific order. This loop assigns elements of array B to A such that the i^{th} element of A is equal to the $(i + 2)^{th}$ element of B after the loop finishes. In Figure 8b, the same loop is written using zippered iteration. The loop induction variable i no longer needs to be specified, and each affine array access has been replaced with an array slice in the zippering of the loop header. It is possible to transform an affine loop in this fashion even when an affine array access has a constant factor multiplied by the loop induction variable. The resulting array slice will contain a stride equal to the constant factor.

Because any parallel affine loop can be transformed into an equivalent parallel loop that uses zippered iteration, we observe a natural place in the Chapel programming language in which to implement modulo unrolling: the leader and follower iterators of the Cyclic and Block Cyclic distribution. The leader iterator divides up the loop's iterations according to the locales they are executed on and passes this work to each follower iterator in the zippering. The follower iterator can then perform the aggregation of remote data elements according to the work that has been passed to it.

6 Cyclic Distribution with Modulo Unrolling

Modulo unrolling is implemented in the follower iterator of the Cyclic distribution. Based on the semantics of parallel zippered iteration, the leader iterator will divide up the iterations of the loop across the locales of the machine according to the first item in the zippering. This could mean that some portions of work will not be local to where the computation is taking place. The follower iterator in the Cyclic distribution recognizes whether or not its chunk of work is local or remote. If remote, all of the remote array elements are brought to the present locale in a local buffer using one `chpl_comm_gets` call. Finally, elements of the local buffer are now yielded back to the loop header. A loop body may modify the elements that are yielded to it via zippered iteration. To account for this, the follower iterator compares the element before it was yielded to the element after it was yielded. If any of the elements in the follower's chunk of work were modified, the entire local buffer is stored back to the remote local via one `chpl_comm_puts` call.

7 Block Cyclic Distribution with Modulo Unrolling

For the Chapel Block Cyclic implementation, both the leader and follower iterators have been modified to support the modulo unrolling optimization. Modulo unrolling, unaltered, is not compatible with the Block Cyclic distribution because consecutive array elements can reside on the same locale, which destroys the static locality information that we were able to use in the Cyclic distribution. The Block Cyclic leader iterator is now modified to choose slices of work such that the new "stride" is equal to the product of the block size and the cycle size. This way, when the work is passed to the follower iterator, elements that are in the same position within each block are guaranteed to be on the same locale. The follower iterator of the Block Cyclic distribution can now perform modulo unrolling in the same way as the Cyclic distribution.

8 Results

To demonstrate the effectiveness of modulo unrolling in the Chapel Cyclic and Block Cyclic distributions, we present our results. We have compiled a suite of seventeen parallel benchmarks shown in Figure 9. Each benchmark is written in Chapel and contains loops with affine array accesses that use zippered iteration, as discussed in 5.2. Our suite of benchmarks contains programs with single, double, and triple nested affine loops. Additionally, our benchmark suite contains programs operating on one, two, and three-dimensional distributed arrays. Fourteen of the seventeen benchmarks are taken from the Polybench suite of benchmarks and translated from C to Chapel by hand. The *stencil9* benchmark was taken from the Chapel source trunk directory. The remaining two benchmarks, *pascal* and *folding*, were written by our group. *pascal* is an additional benchmark other than *jacobi1D* that is able to test Block Cyclic with modulo unrolling. *folding* is the only benchmark in our suite that has strided affine array accesses.

To evaluate improvements due to modulo unrolling, we run our benchmarks using Cyclic and Block Cyclic distributions from the 1.8.0 release of the Chapel compiler as well as Cyclic and Block Cyclic distributions that have been modified to perform modulo unrolling, as described in Sections 6 and 7. We measure both runtime and message count for each benchmark.

When evaluating modulo unrolling used with the Block Cyclic distribution, we could only run two benchmarks out of our suite of seventeen because of limitations within the distribution. Many of our benchmarks operate on two or three-dimensional arrays and all require array slicing for the modulo unrolling optimization to apply. Both array slicing of multi-dimensional arrays and array slicing containing strides for one-dimensional arrays are not yet supported in the Block Cyclic distribution. Implementing such features remained outside the scope of this work. There was no limitation when evaluating modulo unrolling with the Cyclic distribution, and all seventeen benchmarks were tested.

Figures 10 and 11 compare the normalized runtimes and message counts respectively for the Cyclic distribution and Cyclic distribution with modulo unrolling. For 8 of the 11 benchmarks, we see reductions in runtime when the modulo unrolling optimization is applied. On average, modulo unrolling results in a 45 percent decrease in runtime. For 9 of the 11 benchmarks, we see reductions in message counts when the modulo unrolling optimization is applied. On average, modulo unrolling results in 76 percent fewer messages. Two of the benchmarks, *cholesky* and *fw*, showed slight improvements in message count when using modulo unrolling but did not show improvements in runtime. For the *2mm* benchmark, both runtime and message count did not improve when using modulo unrolling. For these benchmarks, the ratio of the problem size to number of locales may not have been high enough, leading to an insufficient amount of aggregation possible for the computation to see performance improvements. An increase in the number of locales on a system leads to fewer data elements per locale, which naturally means fewer data elements can be aggregated. When this occurs, the cost of performing bulk transfers of a few data elements is more expensive than transferring elements individually.

Figures 12 and 13 compare the normalized runtimes and message counts respectively for the Block Cyclic distribution and Block Cyclic distribution with modulo unrolling. For both benchmarks, we see reductions in runtime when the modulo unrolling optimization is applied. On average, modulo unrolling results in a 52 percent decrease in runtime. For both benchmarks, we see reductions in message counts when the modulo unrolling optimization is applied. On average, modulo unrolling results in 72 percent fewer messages.

9 Future Work

As presented, the modulo unrolling optimization can be improved upon in a few ways to achieve even better performance in practice. First, there is currently no limit on the number of array elements that an aggregate message may contain. For applications with extremely large data sets, buffers containing remote data elements may become too large and exceed the memory budget of a particular locale. This may slow down other programs running on the system. To solve this, the modulo unrolling optimization should perform strip mining where the aggregate message is broken down into smaller sections if it contains too many elements in order to conserve memory.

The modulo unrolling optimization currently performs both aggregate reading of remote data elements before the loop and aggregate writing of remote data elements after the loop no matter what the loop body consists of. It is conceivable that some of the yielded elements during zippered iteration will not be read or written to at all during the loop. An improvement to the optimization would be to avoid prefetching elements that are not read in the loop body

Name	Lines of Code	Input Size	Description
2mm	221	128 x 128	2 matrix multiplications (D=A*B; E=C*D)
fw	153	64 x 64	Floyd-Warshall all-pairs shortest path algorithm
trmm	133	256 x 256	Triangular matrix multiply
correlation	235	512 x 512	Correlation computation
covariance	201	512 x 512	Covariance computation
cholesky	182	256 x 256	Cholesky decomposition
lu	143	256 x 256	LU decomposition
mvt	185	4000	Matrix vector product and transpose
syrk	154	128 x 128	Symmetric rank-k operations
syr2k	160	256 x 256	Symmetric rank-2k operations
fdtd-2d	201	1000 x 1000	2D Finite Different Time Domain Kernel
fdtd-apml	333	128 x 128 x 128	FDTD using Anisotropic Perfectly Matched Layer
jacobi1D	138	10000	1D Jacobi stencil computation
jacobi2D	152	400 x 400	2D Jacobi stencil computation
stencil9†	142	400 x 400	9-point stencil computation
pascal‡	126	100000, 100003	Computation of pascal triangle rows
folding‡	139	50400	Strided sum of consecutive array elements

Figure 9: Benchmark suite. Benchmarks with no symbol after their name were taken from the Polybench suite of benchmarks and translated to Chapel. Benchmarks with † are taken from the Chapel Trunk test directory. Benchmarks with ‡ were developed on our own in order to test specific data access patterns.

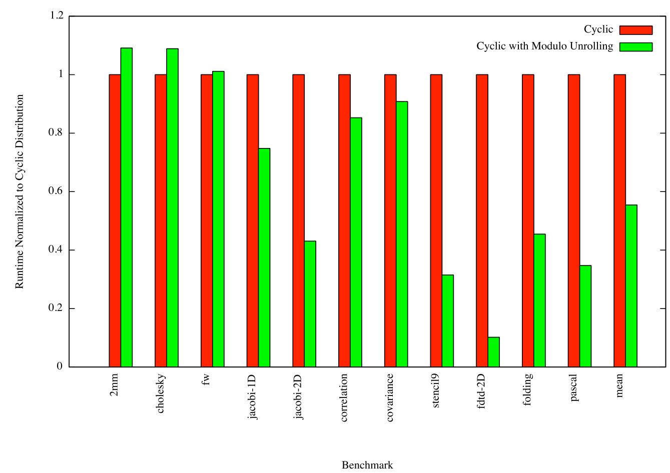

Figure 10: Cyclic runtime.

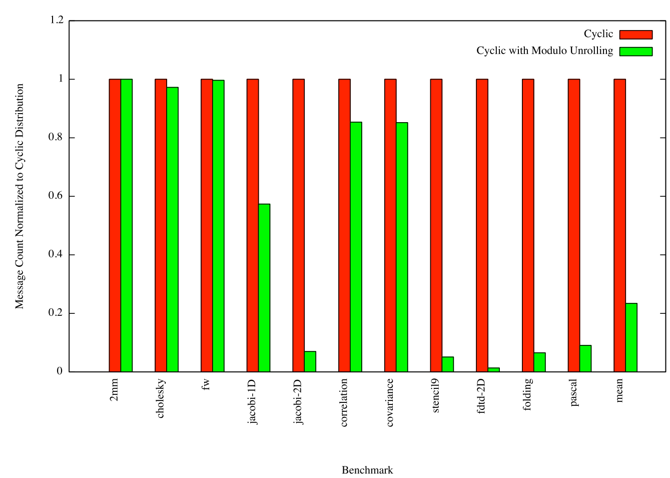

Figure 11: Cyclic message count.

and to avoid writing back elements that are not written to in the loop body.

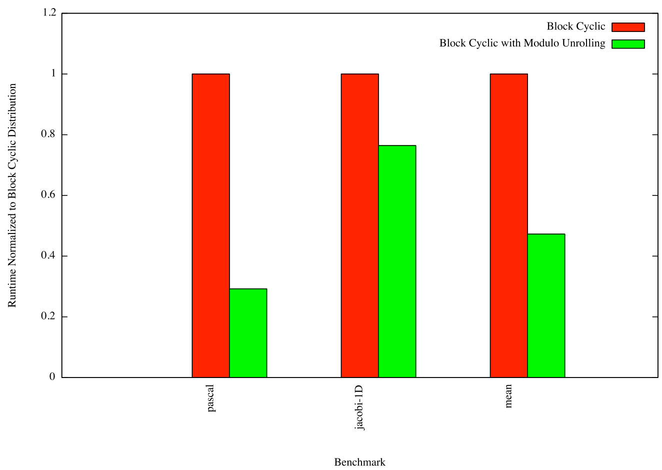

Figure 12: Block Cyclic runtime.

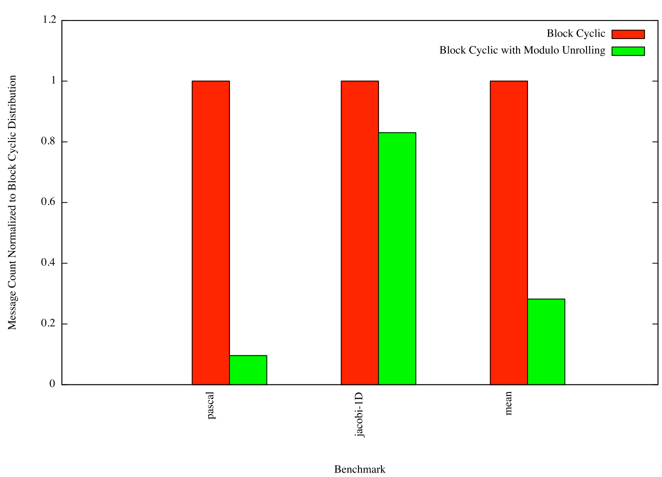

Figure 13: Block Cyclic message count.

References

[1] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: a compiler-managed memory system for raw machines. In *ACM SIGARCH Computer Architecture News*, volume 27, pages 4–15. IEEE Computer Society, 1999.

[2] D. Callahan and K. Kennedy. Compiling programs for distributed–memory multiprocessors. *The Journal of Supercomputing*, 2(2):151–169, 1988.

[3] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and A. Navarro. User-defined parallel zippered iterators in chapel. 2011.

[4] D. Chavarría-Miranda and J. Mellor-Crummey. Effective communication coalescing for data-parallel applications. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 14–25. ACM, 2005.

[5] W.-Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained upc applications. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 267–278. IEEE, 2005.

[6] C. Germain and F. Delaplace. Automatic vectorization of communications for data-parallel programs. In *EURO-PAR'95 Parallel Processing*, pages 429–440. Springer, 1995.

[7] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Message-passing code generation for non-rectangular tiling transformations. *Parallel Computing*, 32(10):711–732, 2006.

[8] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. Technical report, 1991.

[9] S. K. S. Gupta, S. Kaushik, C.-H. Huang, and P. Sadayappan. Compiling array expressions for efficient execution on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 32(2):155–172, 1996.

[10] C. Iancu, W. Chen, and K. Yelick. Performance portable optimizations for loops containing communication operations. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 266–276. ACM, 2008.

[11] M. E. Mace. *Memory storage patterns in parallel processing*. Kluwer Academic Publishers, 1987.

[12] L. Prylli and B. Tourancheau. Fast runtime block cyclic data redistribution on multiprocessors. *Journal of Parallel and Distributed Computing*, 45(1):63–72, 1997.

[13] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *Parallel and Distributed Systems, IEEE Transactions on*, 2(4):472–482, 1991.

[14] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.

[15] A. Sanz, R. Asenjo, J. López, R. Larrosa, A. Navarro, V. Litvinov, S.-E. Choi, and B. L. Chamberlain. Global data re-allocation via communication aggregation in chapel. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 235–242. IEEE, 2012.

[16] D. W. Walker and S. W. Otto. Redistribution of block-cyclic data distributions using mpi. *Concurrency Practice and Experience*, 8(9):707–728, 1996.

[17] W.-H. Wei, K.-P. Shih, J.-P. Sheu, et al. Compiling array references with affine functions for data-parallel programs. *J. Inf. Sci. Eng.*, 14(4):695–723, 1998.

[18] J. Xue. Communication-minimal tiling of uniform dependence loops. In *Languages and Compilers for Parallel Computing*, pages 330–349. Springer, 1997.