Affine Loop Optimization Based on Modulo Unrolling in Chapel

Aroon Sharma, Darren Smith,
Joshua Koehler, Rajeev Barua
Dept. of Electrical and Computer Engineering
University of Maryland, College Park
{asharma4|darrenks|jskoeh9|barua}@umd.edu

Michael Ferguson
Laboratory for Telecommunication Sciences
College Park, MD
mferguson@ltsnet.net

ABSTRACT

This paper presents modulo unrolling without unrolling (modulo unrolling WU), a method for message aggregation for parallel loops in message passing programs that use affine array accesses in Chapel, a Partitioned Global Address Space (PGAS) parallel programming language. Messages incur a non-trivial run time overhead, a significant component of which is independent of the size of the message. Therefore, aggregating messages improves performance. Our optimization for message aggregation is based on a technique known as modulo unrolling, pioneered by Barua [3], whose purpose was to ensure a statically predictable single tile number for each memory reference for tiled architectures, such as the MIT Raw Machine [18]. Modulo unrolling WU applies to data that is distributed in a cyclic or block-cyclic manner. In this paper, we adapt the aforementioned modulo unrolling technique to the difficult problem of efficiently compiling PGAS languages to message passing architectures. When applied to loops and data distributed cyclically or block-cyclically, modulo unrolling WU can decide when to aggregate messages thereby reducing the overall message count and runtime for a particular loop. Compared to other methods, modulo unrolling WU 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, as opposed to creating a traditional compiler transformation. Results were collected that compare the performance of Chapel programs optimized with modulo unrolling WU and Chapel programs using the existing Chapel data distributions. Data collected on a ten-locale cluster show that on average, modulo unrolling WU used with Chapel's Cyclic distribution results in 64 percent fewer messages and a 36 percent decrease in runtime for our suite of benchmarks. Similarly, modulo unrolling WU used with Chapel's Block Cyclic distribution results in 72 percent fewer messages and a 53 percent decrease in runtime.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
PGAS '14 October 6-10, 2014, Eugene, OR, USA
Copyright 2014 ACM 978-1-4503-3247-7/14/10 ...$15.00
http://dx.doi.org/10.1145/2676870.2676877.

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 automates the production of message passing code from a shared memory programming model and 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.

Without aggregation, 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 reside on the same remote locale before the message is sent.

The vast majority of loops in scientific programs access data using *affine array accesses*. An affine array access is one whose indices are linear combinations of the loop's induction variables. For example, for a loop with induction variables i and j, accesses $A[i, j]$ and $A[2i-3, j+1]$ are affine, but $A[i^2]$

is not. Loops using affine array accesses are special because they exhibit regular and predictable access patterns within a data distribution. Compilers can use this information to decide when message aggregation can take place.

Existing methods for message passing code generation such as [10, 22] all have the following steps:

- **Loop distribution** The loop iteration space for each nested loop is divided into portions to be executed on each locale (message passing node), called iteration space tiles.
- **Data distribution** The data space for each array is distributed according to the directive of the programmer (usually as block, cyclic, or block-cyclic distributions.)
- **Footprint calculation** For each iteration space tile, the portion of data it accesses for each array reference is calculated as a formula on the symbolic iteration space bounds. This is called the data footprint of that array access.
- **Message aggregation calculation** For each array access, its data footprint is separately intersected with each possible locale's data tile to derive symbolic expressions for the portion of the data footprint on that locale's data tile. This portion of the data tile for locales other than the current locale needs to be communicated remotely from each remote data tile's locale to the current loop tile's locale. Since the entire remote portion is calculated exactly, sending it in a single aggregated message becomes possible.

Unfortunately, of the steps above, the message aggregation calculation is by far the most complex. Loop distribution and data distribution are straightforward. Footprint calculation is of moderate complexity using matrix formulations or the polyhedral model. However, it is the message aggregation calculation that defies easy mathematical characterization for the general case of affine accesses. Instead some very complex research methods [9, 22] have been devised that make many simplifying assumptions on the types of affine accesses supported, and yet remain so complex that they are rarely implemented in production compilers.

Although the steps above are primarily for traditional methods of parallel code generation, polyhedral methods don't fare much better. Polyhedral methods have powerful mathematical formulations for loop transformation discovery, automatic parallelization, and parallelism coarsening. However message aggregation calculation is still needed but not modeled well in polyhedral models, leading to less capable ad-hoc methods for it.

It is our belief that message aggregation using tiling is not used in production quality compilers today because of the complexity of message aggregation calculations, described above. What is needed is a simple, robust, and widely applicable method for message aggregation that leads to improvements in performance.

This paper presents modulo unrolling without unrolling (WU), a loop optimization for message passing code generation based on a technique called modulo unrolling, whose advantage is that it makes the message aggregation calculation above far simpler. Using modulo unrolling WU, the locality of any affine array access can be deduced if the data is distributed in a cyclic or block-cyclic fashion. It is possible for the optimization to 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 [3], was meant to target tiled architectures such as the MIT Raw machine. Its purpose for tiled architectures was to allow the use of the compiler-routed static network for accessing array data in unrolled loops. It was not meant for message passing architectures, nor was it used to perform message aggregation. It has since been modified to apply to message passing machines in this work.

We build on the modulo unrolling method to solve the very difficult problem of message aggregation for message passing machines inside PGAS languages. In the PGAS model, 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.

Modulo unrolling WU takes as input a parallel loop containing affine accesses from arrays distributed cyclically or block-cyclically. It has three steps. First, in the *block cyclic transformation for static disambiguation*, the original loop header is transformed using a strip mining technique to ensure that affine accesses are statically disambiguated for both block-cyclic and cyclic data throughout the loop. Next, in the *owning expression calculation*, loop iterations are assigned to locales according to the owning expression of the loop in order to ensure that the fewest number of remote data accesses occur during each loop iteration. Finally, in the *message aggregation step*, for each remote affine access that is non-owned in the loop, the footprint of data that it accesses during the entire loop is calculated statically and communicated to the owning locale before the loop executes in an aggregate message. The remote affine accesses inside the loop can now be replaced with accesses to local storage, thereby eliminating locality checks for each loop iteration. If any data elements that were communicated to the owning locale are written to during the loop, another aggregate message sends these elements back to the remote locale after the loop finishes.

Our evaluation is for Chapel, an explicitly parallel programming language developed by Cray Inc. that falls under the PGAS memory model. 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, leader and follower iterator semantics, and array slicing that greatly simplify the implementation of modulo unrolling WU into the language. In particular, we implement modulo unrolling WU in Chapel not as a traditional compiler pass or loop transformation, but as a portion of the Cyclic and Block Cyclic data distribution modules. This allows us to express the optimization directly using the Chapel language. It also gives us the ability to reason about the number of locales being used to run the program. The number of locales is generally unknown at compile time, but the Chapel language exposes this information to the programmer via built-in constructs such as the `Locales` array and `numLocales` constant.

Although our method is implemented in Chapel, we describe it using pseudocode in Section 6, showing how it can be adapted to any PGAS language. However, for other lan-

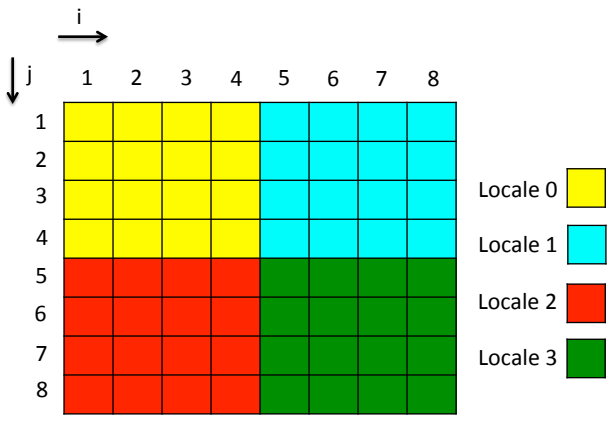

Figure 1: Chapel Block distribution.

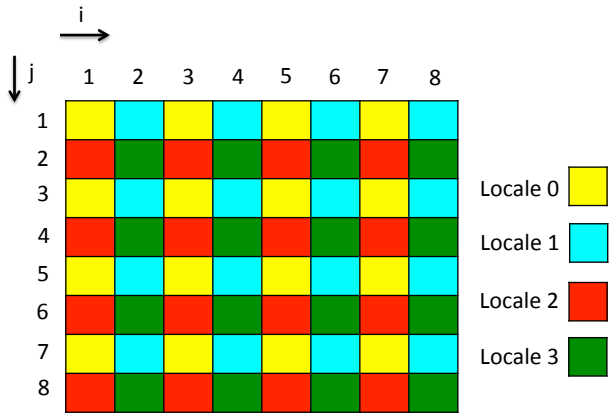

Figure 2: Chapel Cyclic distribution.

guages the implementation may differ. For example, if the language does not use leader and follower iterator semantics to implement parallel for loops, the changes to those Chapel modules that we present here will have to be implemented elsewhere in the other PGAS language where **forall** loop functionality is implemented.

The rest of this paper is organized as follows. Section 2 describes three Chapel data distributions: Block, Cyclic, and Block Cyclic. Section 3 discusses related work. A brief background on modulo unrolling for tiled architectures [3] is presented in Section 4. Section 5 illustrates how message aggregation is applied to parallel affine loops using modulo unrolling WU with an example. Section 6 formally describes the mathematical transformations of modulo unrolling without unrolling (WU), our communication optimization. Section 7 explains how we adapted modulo unrolling WU into the Chapel programming language. Section 8 presents our results. Finally, Section **??** describes our future work.

2. CHAPEL'S DATA DISTRIBUTIONS

Figures 1 - 3 illustrate the Chapel data distributions that we explored in this work: Block, Cyclic, and Block Cyclic. Each figure shows how a two-dimensional 8 x 8 array can be distributed in Chapel using each distribution. Figure 1

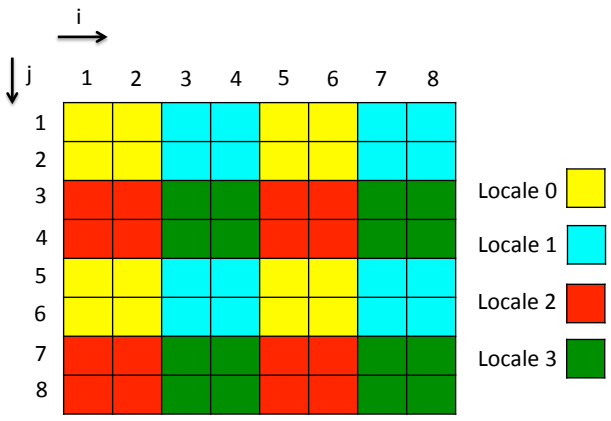

Figure 3: Chapel Block Cyclic distribution with a 2 x 2 block size parameter.

illustrates the Block distribution. Elements of the array are mapped to locales evenly in a dense manner. In Figure 2, the Cyclic distribution, elements of the array are mapped in a round-robin manner across locales. Finally, in Figure 3 the Block Cyclic distribution is shown. Here, a number of elements specified by a block size parameter is allocated to consecutive array indices in a round-robin fashion. In Figure 3, the distribution takes in a 2 x 2 block size parameter. Further details about Block, Cyclic, and Block Cyclic distributions in Chapel are described in [17].

The choice of data distribution to use for a program boils down to computation and communication efficiency. Different programs and architectures may require different data distributions. 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 [13]. 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 are 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 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 when array sizes may be changed dynamically because in Cyclic and Block Cyclic distributions, the locales of existing array elements do not change when new array elements are added at the end of the array. In many applications, data redistribution may be needed if elements of a data set

are inserted or deleted at the end of the array. In particular, algorithms to redistribute data using a new block size exist for the Block Cyclic distribution [14, 19]. If an application uses a dynamic data set with elements that are 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 data set would need to be redistributed every time a new element is appended, which can be expensive.

The compiler should attempt to perform optimizations based on the data distribution that the programmer specified. Our optimization is meant to be applied whenever the programmer specifies a Cyclic or Block Cyclic distribution. It is not applied when the programmer specifies a Block distribution.

3. 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 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 [22, 9, 4, 15] require not only footprint calculations for each tile but also the intersection of footprints with data tiles. As described in detail in Section 1, calculating such intersections is very complex, which explains the complexity and simplifying limitations of many existing methods. Such methods are rarely if ever implemented in production compilers.

The polyhedral method is another branch of compiler optimization that seeks to speed up parallel programs on distributed memory architectures [10, 6, 8, 11, 12, 20]. Its strength is that it can find sequences of transformations in one step, without searching the entire space of transformations. However, the method at its core does not compute information for message passing code generation. Message passing code generation does not fit the polyhedral model, so ad-hoc methods for code generation have been devised to work on the output of the polyhedral model. However they are no better than corresponding methods in the traditional model, and suffer from many of the same difficulties.

Similar work to take advantage of communication aggregation on distributed arrays has already been done in Chapel. Whole array assignment is the process of assigning an entire distributed array to another in one statement, where both arrays are not guaranteed to be distributed in the same way. Like distributed parallel loops in Chapel, whole array assignment suffers from locality checks for every array element, even when the locality of certain elements is known in advance. In [16], aggregation is applied to improve the communication performance of whole array assignments for Chapel's Block and Cyclic distributions. However, [16] does not address communication aggregation that is possible across general affine loops. Whole array assignment and affine loops in Chapel are fundamentally related because every whole array assignment can be written in terms of an equivalent affine **forall** loop. Yet, the contrapositive statement is not true: most affine loops can't be modeled as whole array assignments. Our method for communication aggregation in parallel loops encompasses more complex affine array accesses than those that are found in whole array assignments and addressed in [16]. Finally, our work applies to Chapel's Block Cyclic data distribution in addition to Cyclic, whereas the work in [16] does not.

One of the contribution's of [16] included two new strided 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. The methods in [16] are already in the current release of the Chapel compiler.

Work has been done with the UPC compiler (another PGAS language) by [7] to improve on its communication performance. Unlike our work, which takes as its input a distributed parallel affine loop, the work in [7] expects to aggregate communication across an entire program. 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. In communication coalescing, small puts and gets throughout the program are combined into larger messages by the compiler to reduce the number of times the per-message startup overhead is incurred. This work's aggregation scheme is only applicable to programs with many small, individual, and independent remote array accesses. This method can't be used to improve communication performance across more coarse-grained structures, such as distributed parallel loops. 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 when data is distributed cyclically, which is done in our work, it must be possible to aggregate data elements that are far apart in memory, separated by a fixed stride. In contrast, our method can aggregate data distributed cyclically and block-cyclically.

Another communication optimization targeting the X10 language [2] achieves message aggregation in distributed loops by using a technique called *scalar replacement with loop invariant code motion*. Here, the compiler copies *all* remote portions of a block-distributed array to each locale once before the loop. Then, each locale can access its own local copy of the array during each loop iteration. While this method does improve communication performance, it can potentially communicate extraneous remote array portions that the loop body never accesses. For large data sets, this could overwhelm a locale's memory. Modulo unrolling WU communicates only the remote portions of the distributed array that are used during the loop body.

4. BACKGROUND ON MODULO UNROLLING

Modulo unrolling [3] is a static 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

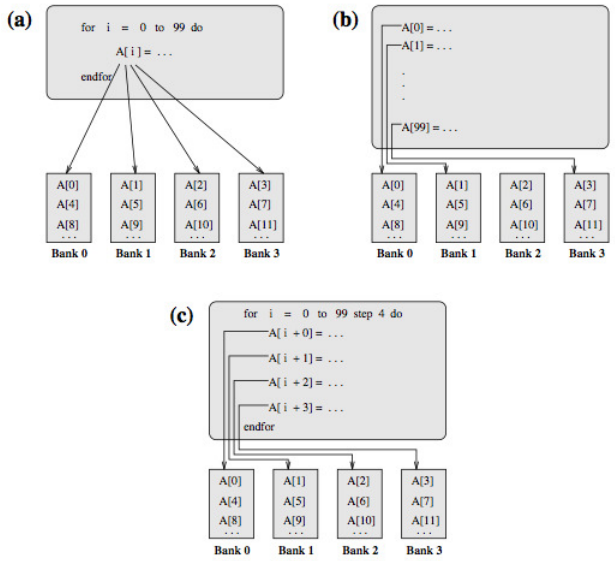

Figure 4: Modulo unrolling example. (a) Original sequential for loop. Array A is distributed using a Cyclic distribution. Each array access maps to a different memory bank on successive loop iterations. (b) Fully unrolled loop. Trivially, each array access maps to a single memory bank because each access 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.

```
1   var n: int = 8;
2   var LoopSpace = {2..n-1, 2..n-1};
3
4   //Jacobi relaxation pass
5   forall (i,j) in LoopSpace {
6         A_new[i,j] = (A[i+1, j] + A[i-1, j] +
7                       A[i, j+1] + A[i, j-1])/4.0;
8   }
9
10  //update state of the system after the first
11  //relaxation pass
12  A[LoopSpace] =  A_new[LoopSpace];
```

Figure 5: Chapel code for the Jacobi-2D 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 $A[i, j]$.

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 *statically disambiguated*, or 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 statically 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 static 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, static 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 static 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 static 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 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. The work in [3] shows that an unroll factor providing this property always exists not only for the code in Figure 4, but for the general case of any affine function in a loop. The unroll factor may not always equal the number of banks, but a suitable unroll factor can always be computed.

Modulo unrolling, as used in [3] provides static 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.

5. INTUITION BEHIND MESSAGE AGGREGATION WITH AN EXAMPLE

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. This section presents an example of a Chapel program with affine array accesses that can benefit from message aggregation. It also serves to present the intuition behind how modulo unrolling WU will be used in message aggregation.

The intuition behind why modulo unrolling is helpful for message aggregation in message passing machines is as follows. Message aggregation requires knowledge of precisely which elements must be communicated between locales. Doing so requires a statically disambiguated known locale for every array access, even when that array access refers to a

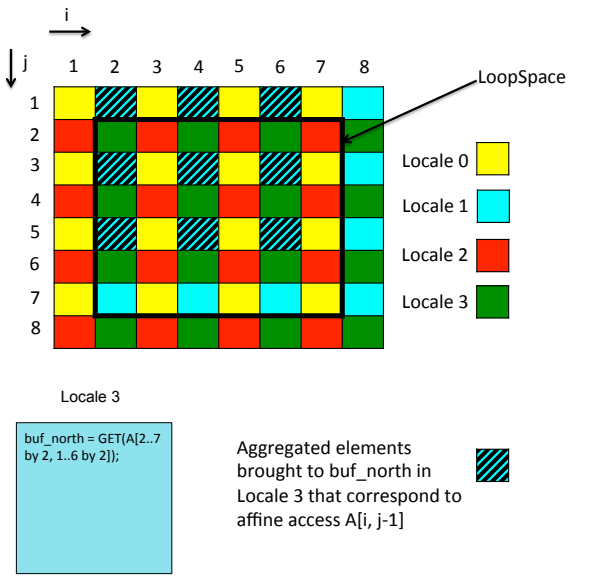

Figure 6: Illustration of message aggregation for the $A[i, j-1]$ affine array access of the Jacobi-2D relaxation computation with respect to locale 3. The region *LoopSpace* follows from Figure 5. The striped squares are the elements of A that have been aggregated. This same procedure occurs on each locale for each affine array access that is deemed to be remote for all iterations of the loop. For the whole 8 x 8 Jacobi-2D calculation, 144 remote gets containing one element each are necessary without aggregation, but only 16 remote gets containing nine elements each are necessary with aggregation.

varying address. For example, in a loop $A[i]$ refers to different memory addresses during each loop iteration. Modulo unrolling ensures such a known, predictable locale number for each varying array access. This enables such varying accesses to be aggregated and sent in a single message. We explain our method of doing so in Sections 6 and 7.

Consider the Chapel code for the Jacobi-2D 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. We assume that arrays A and A_{new} have already been distributed using a Cyclic distribution over four locales. 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 $A[i, j-1]$. 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 2, 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**. Therefore, these remote elements need to be transferred over to $A_{new}[i, j]$'s locale in four separate 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 many times.

We observe that message aggregation of remote data elements is possible over the entire loop for the Jacobi-2D example. Aggregation will reduce the number of times the message overhead is incurred during the loop. When the data is distributed using a Cyclic distribution, all array accesses (including remote accesses) exhibit a predictable pattern of locality.

Figure 6 illustrates this pattern in detail for loop iterations that write to locale 3. During these iterations ($(i, j) = (2, 2), (i, j) = (4, 2)$, etc.), there are two remote accesses from locale 1 and two remote accesses from locale 2. The remote accesses from locale 1 correspond to the $A[i, j+1]$, and $A[i, j-1]$ affine array accesses in Figure 5. If we highlight all the remote data elements corresponding to the $A[i, j-1]$ access that occur for loop iterations that write to locale 3, we end up with the array slice $A[2..7\ by\ 2, 1..6\ by\ 2]$, which contains the striped elements in Figure 6. This array slice can be communicated from locale 1 to a buffer on locale 3 before the loop executes in a single message. Then, during the loop, all $A[i, j-1]$ accesses can be replaced with accesses to the local buffer on locale 3.

The previous paragraph showed how aggregation occurs for the $A[i, j-1]$ affine array access on loop iterations that write to locale 3. This same procedure applies to the other three remote accesses for locale 3. In addition, this same procedure applies to loop iterations that write to the remaining locales. Finally, we claim that 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.

In this example, we chose to perform message aggregation with respect to the element that is written to during the loop. However, this is not always the best choice for all programs. To get better communication performance, we would like to assign loop iterations to locales with the most affine array accesses that are local. The result of this scheme is that elements that are written to during the loop may be the ones that are aggregated before the loop. If so, it is necessary to write these elements from the local buffers back to their remote locales. This is done in a single aggregate message after the loop body has finished.[1]

If arrays A and A_{new} are instead distributed using Chapel's Block or Block Cyclic distributions as shown in Figure 1 and Figure 3 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-2D computation decreases. For the Jacobi-2D 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 out-

[1]In Chapel, the programmer has some control over assigning loop iterations to locales. Therefore, our optimizations uses the programmer's assignment of loop iterations to locales when performing message aggregation.

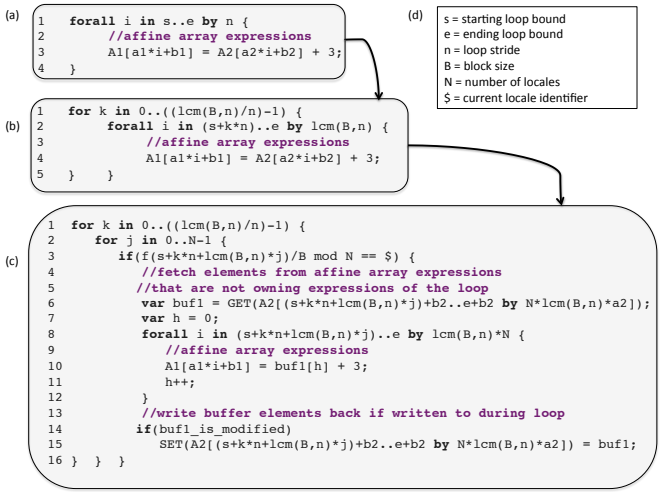

Figure 7: Steps to transform a parallel affine loop where the data is distributed cyclically or block-cyclically into an equivalent loop that performs message aggregation using modulo unrolling WU. (a) Original distributed parallel loop with two affine array accesses. (b) Loop after Block Cyclic transformation. After this step, the affine array accesses in loops with data distributed block-cyclically will be statically disambiguated. (c) Loop after the owning expression calculation and message aggregation steps. In line 6, remote array elements are communicated to a local buffer before the loop. The affine array access for A_2 is replaced with an access to the local buffer in line 10. In lines 14-15, elements in the local buffer are written back to the remote locale if they are written to during the loop. (d) Key of symbolic variables used in the transformations in parts a-c.

side of a block boundary. For these types of programs, a Cyclic or Block Cyclic distribution will perform better. Section 2 explained several reasons why the programmer may have chosen a Cyclic or Block Cyclic distribution.

6. MESSAGE AGGREGATION LOOP OPTIMIZATION FOR PARALLEL AFFINE LOOPS

This section describes our method to transform an affine loop that computes on cyclically or block-cyclically distributed data into an equivalent loop that performs message aggregation. As described in Section 2, our method is not meant for block distributed data. The proposed method is based on modulo unrolling [3], described in Section 4. Here we describe the method in pseudocode for simplicity and to show that this method is applicable to languages other than Chapel.

6.1 Modulo Unrolling Without Unrolling

Modulo unrolling increases code size because it unrolls loops by a factor equal to the number of locales (memory banks) on the system. However, we have devised an adaptation called modulo unrolling WU for message passing machines that does not increase code size. To understand it, consider that for parallel machines that use message passing, static disambiguation can be achieved by using the locale identifier without increasing the code size. Conceptually, an affine loop written in source code on a message passing machine where data is distributed cyclically among four locales such as:

```
forall i in 0..99 {
    A[i] = B[i+2];
}
```

becomes statically disambiguated using this observation as follows:

```
forall i in 0..99 by 4 {
    A[i+$] = B[i+2+$];
}
```

where $ represents the locale identifier. The above is the code that is run on each locale. This transformation is called *modulo unrolling without unrolling (modulo unrolling WU)* since, like modulo unrolling, it can be used for static disambiguation but on message passing machines instead of tiled architectures. Here, no unrolling of the loop is necessary.

Figure 7 shows how a generalized affine loop, expressed symbolically, can be transformed by our method in three steps: the Block Cyclic transformation (Figure 7a → Figure 7b), the owning expression calculation (described in Section 6.3), and the message aggregation (Figure 7b → Figure 7c).

As shown in Figure 7a, our method takes as its input a parallel **forall** loop that contains a number of affine array expressions in its loop body. Non-affine expressions are allowed in the loop body, but they are not optimized. The input loop shown in Figure 7a is defined by three explicit parameters: the starting loop bound s, the ending loop bound e, and the loop stride n. The input loop also contains two implicit parameters based on the data distribution. The number of locales the data is distributed over is N, and the block size, the number of consecutive array elements allocated to a single locale, is B. All five parameters are elements of $\mathbb{N}$. The output of the optimization is an equivalent loop structure that aggregates communication from all of the loop body's remote affine array accesses.

6.2 Block Cyclic Transformation

Modulo unrolling as described in [3] guarantees static disambiguation for data distributed cyclically but not for block-cyclically distributed data. However, we can think of a Block Cyclic distribution as B adjacent Cyclic distributions, each with a cycle size that is greater than N. In order to achieve static disambiguation for the Block Cyclic distribution, we must transform input loops with $B > 1$ into an equivalent loop with a loop step size that is a multiple of B.

Lines 1 and 2 of Figure 7b show this transformation. We replace the loop step size on line 1 of Figure 7a with the *least common multiple* of B and n in line 2 of Figure 7b. The intuition behind this new step size is that two successive loop iterations accessing the same position within a block will always be separated by a fixed stride length that is a multiple of the block size. To maintain the original meaning of the input loop, an outer **for** loop is added on line 1 of Figure 7b to handle iterations within each block, and the starting loop

bound on line 2 is written in terms of the outer loop variable k. After this transformation, all affine array accesses in the loop with be statically disambiguated. This transformation is a variant of the well-known strip mining transformation, which has been used for many other purposes in the literature.

The Cyclic and Block Cyclic distributions are closely related. Any Cyclic distribution can be thought of as a Block Cyclic distribution with $B = 1$. If we apply the transformation in Figure 7b to a loop with cyclically distributed data, we will end up with the original input loop in Figure 7a, which is already statically disambiguated after applying the transformation described in Section 6.1.

6.3 Owning Expression Calculation

There may be many affine array accesses in the input loop, each mapped to a single locale after static disambiguation. For the best communication performance, we must determine the *owning expression* for the loop, which is the most common affine array expression in the loop body. More formally, the owning expression is an affine function $f(i)$, where i is the loop's induction variable, that occurs statically the most number of times in the loop body. We can then use the owning expression to assign loop iterations to locales. Note that there may be instances where affine array expressions are found within control flow statements inside the loop body. Here, we will not know how many times each condiitonal block will execute at compile time. For these cases, we can use static profiling methods described in [21] to estimate the occurrences of affine array accesses within conditional blocks in the loop body.

As an example of how the owning expression is computed and used, consider that there are two affine array accesses in Figure 7b: $A_1[a_1i + b_1]$ and $A_2[a_2i + b_2]$. Each appears once in the loop body, so either expression can be chosen as the owning expression for the loop. For the remainder of Figure 7, we assume that $a_1i + b_1$ is the owning expression.

Line 3 of Figure 7c shows symbolically how the owning expression, which is an affine function of the loop induction variable i, is used to ensure that loop iterations are assigned to locales such that most of the affine array accesses are local. The argument to the owning expression f in line 3 represents the first loop iteration in each strip-mined portion created in the Block Cyclic transformation. We evaluate the owning expression at this loop iteration. This yields the array index that is most accessed during this loop iteration. The locale where this array index resides should be responsible for handling all iterations in this strip-mined portion because this way most of the loop body's affine array accesses will be local.

6.4 Message Aggregation

The final step of the optimization is to communicate the non-owned remote affine array accesses in a single message before the loop. Figure 7c shows this transformation. The loop nest starting on line 2 symbolically represents which loop iterations are assigned to the N locales on the system based on the owning expression calculation (line 3). The array access $A_2[a_2i + b_2]$ is non-owned and may either be entirely remote or entirely local. If entirely remote (as is assumed here), it will require communication. We compute its corresponding remote array slice in line 6 before communicating the entire array slice to a local buffer. Modulo unrolling guarantees that all elements in this array slice are remote with respect to a single locale on the loop iterations that they are used. So, they can be brought to the current locale \$ in one message. Now in lines 8-12, the affine array access $A_2[a_2i+b_2]$ can be replaced with an access to the local buffer. Lines 14-15 handle the case that elements brought over in bulk need to be written back to their remote locale.

6.5 Loops with Multi-Dimensional Array Accesses

The series of transformations described in this section and illustrated in Figure 7 all apply to one-dimensional arrays indexed by one loop induction variable. These transformations can also be generalized to apply to certain affine array accesses for multi-dimensional arrays. The intuition for this generalization is as follows. The input affine loop now contains m loop induction variables i_1, i_2, ... , i_m. Similarly, there are now m starting loop bounds, ending loop bounds, loop strides, and block sizes. The p^{th} block size is now the number of consecutive array elements allocated to a single locale in dimension p of the array, where $1 \leq p \leq m$. Each affine array access in the loop body now contains m affine array expressions where expression p is an affine function of i_p.

Under these assumptions, the transformations described in this section need only be applied to each loop induction variable independently. The owning expression calculation now produces an m-tuple of affine array expressions.[2] The results we collect in this work consider one-, two-, and three-dimensional array accesses.

7. ADAPTATION IN CHAPEL

The goal of this section is to present our adaptation in Chapel of the modulo unrolling WU optimization presented in Section 6. We also provide a basic understanding of zippered iteration and array slicing, two important features in Chapel used in the optimization's implementation.

7.1 Chapel Zippered Iteration

Iterators are 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. This is how Chapel can support other important language features such as scalar promotion and whole array assignment.

Figure 8b shows how the original code in Figure 8a can be rewritten to use zippered iteration [5] instead. Zippered iteration is a Chapel language construct that allows multiple

[2]In our adaptation of modulo unrolling WU in Chapel, the Cyclic distribution can apply the optimization to loops with multi-dimensional array accesses, but the Block Cyclic distribution is limited to one-dimensional array accesses because of the current limitations within Chapel's existing Block Cyclic implementation that are outside the scope of this work.

```
1  //(a) Parallel loop with affine array accesses
2  forall i in 1..10 {
3       A[i] = B[i+2];
4  }
5
6  //(b) Equivalent loop written using zippered iteration
7  forall (a,b) in zip(A[1..10], B[3..12]) {
8       a = b;
9  }
```

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

iterators of the same size and shape to be iterated through simultaneously. When zippered iteration is used, corresponding 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.

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.

7.2 Chapel 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. Line 7 of Figure 8b shows examples of two array slices of arrays A and B respectively.

In Chapel, 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 in the loop body is replaced with a corresponding array slice in the loop header, which produces the same elements as the original loop.

The example code in Figure 8 shows how regular and zippered iteration versions of the same program have different execution orders but the same result. There are two affine array accesses $A[i]$ and $B[i+2]$ in Figure 8a. The loop is

(a)

```
1  iter CyclicArr.these(param tag: iterKind, followThis, param fast: bool = false) var
2       where tag == iterKind.follower {
3
4  if arrSection.locale.id == here.id then local {
5       //call original fast follower iterator helper for local elements
6  } else {
7       //call original follower iterator helper for nonlocal elements
8       for i in followThis {
9            yield accessHelper(i);
10      }
11 } }
```

(b)

```
1  iter CyclicArr.these(param tag: iterKind, followThis, param fast: bool = false) var
2       where tag == iterKind.follower {
3
4  //check that all elements in chunk are from the same locale by examining each dim
5  for i in 1..rank {
6       if (followThis(i).stride * dom.whole.dim(i).stride %
7            dom.dist.targetLocDom.dim(i).size != 0) {
8            //call original follower iterator helper for nonlocal elements
9            for i in followThis {
10                yield accessHelper(i);
11           }
12 }    }
13 if arrSection.locale.id == here.id then local {
14      //call original fast follower iterator helper for local elements
15 } else {
16      //allocate local buffer to hold remote elements, compute source and destination
17      //strides, number of elements to communicate
18       chpl_comm_gets(buf, deststr, arrSection.myElems._value.theData, srcstr, count);
19       var changed = false;
20       for i in buf {
21            var old_i = i;
22           yield i;
23            var new_val = i;
24            if(old_val != new_val) then changed = true;
25       }
26       if changed then
27           chpl_comm_puts(arrSection.myElems._value.theData, srcstr, buf, deststr, count);
28 }    }
```

Figure 9: (a) Pseudocode for the unaltered Cyclic distribution follower iterator. The code only handles cases when a chunk of work is either completely local or remote. In the remote case in lines 6-11, remote data elements are accessed one at a time, resulting in multiple messages. (b) Pseudocode for the Cyclic distribution follower iterator that has been modified to perform modulo unrolling WU. Now, the code divides the remote case in (a) into two separate cases: remote from a *single* locale and remote from possibly multiple locales. If the chunk of work is remote from a single locale, we can perform message aggregation.

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 iterators. 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. The two loops in Figure 8 are equivalent and generate the same results, but they differ in their execution.

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 WU: 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

```
1   iter BlockCyclicDom.these(param tag: iterKind) var where tag == iterKind.leader {
3
4     //calculate blockcyclesize
5      var blockcyclesize = blocksize*numLocales;
6
7      //assign loop iterations to locales
8      coforall locDom in locDoms do on locDom {
9
10          //determine the index of the first element in the locDom
11         var start = locDom.myStarts.low;
12         var tasks = here.numCores;
13
14         //each core on a locale can handle its own chunk of work in parallel
15         coforall core in 0..tasks-1 do
16
17              //serialize the division of work in case there are
18              //more elements within a block than there are cores
19             for i in core..blocksize-1 by tasks {
20
21                  yield (start+i)..end by blockcyclesize;
22 } }          }
```

Figure 10: Pseudocode for the Block Cyclic distribution leader iterator that has been modified to perform modulo unrolling WU.

can then perform the aggregation of remote data elements according to the work that has been passed to it.

7.3 Implementation

Modulo unrolling WU is implemented into the Chapel programming language through the Cyclic and Block Cyclic distribution modules, as opposed to being implemented via traditional compiler passes. Specifically, the follower iterator is modified in the Cyclic distribution, and both the leader and follower iterators are modified in the Block Cyclic distribution. Because these modules are written in Chapel, the optimization can be expressed using Chapel's higher-level language constructs, such as zippered iteration and array slicing.

Figure 9 shows the Chapel pseudocode representation of the Cyclic follower iterator before and after it has been modified to perform modulo unrolling WU. Some coding details are left out for brevity. The follower iterator is responsible for carrying out the loop iterations that are passed to it by the leader iterator. Because the follower iterator has no knowledge about how the leader iterator divides up the loop iterations, this chunk of work can either be entirely local, entirely remote to a single locale, or spread across multiple locales. In Figure 9a, which shows the existing Cyclic follower iterator, the code only handles cases where the chunk of work *followThis* is completely local or remote from possibly multiple locales. If the chunk is local, a helper function responsible for yielding local elements is called on line 5. If the chunk is remote, each remote element is accessed individually with its own message, as shown in lines 6-11.

Figure 9b shows the Cyclic follower iterator modified to perform modulo unrolling WU. Now, the code has split up the remote case in the original follower iterator into two separate cases. First in lines 5-12, a chunk of work can be remote with elements coming from multiple locales. If so, then data elements are still accessed one at a time like before. Second in lines 13-28, a chunk of work can be remote with elements coming from a *single* locale. If so, then it is guaranteed that the elements within the chunk of work are separated by a fixed stride defined by the Cyclic distribution. This stride information is available from within the follower iterator. Therefore, we can aggregate all remote elements in that chunk into a single message.

The entire chunk of work, specified by the `arrSection` pointer, is communicated to the local `buf` in one message with the `chpl_comm_gets` call on line 18. Then, elements in this buffer are yielded back to the loop following zippered iteration semantics. The values in `buf` are compared before and after they are yielded in order to determine whether or not they were written to in the loop body. If so, a `chpl_comm_puts` call on line 27 is required to write all `buf` elements back to the remote locale.

The implementation of modulo unrolling WU into the Block Cyclic distribution is nearly identical to Figure 9 with one key addition: the Block Cyclic leader iterator is also altered so that chunks of work that it creates only contain elements that reside in the same position within a block. Figure 10 shows a Chapel pseudocode representation of the Block Cyclic leader iterator, with some coding details left out for brevity. Line 5 computes *blockcyclesize*, the product of the block size parameter and the total number of locales. Lines 8-22 assign loop iterations to locales according to how the leader's caller is distributed. Each object *locDom* referenced on line 8 represents the collection of ranges of elements of the leader's caller that reside on a single locale. Using this collection, the index of the first element of the first block of the leader's caller, called *start* is determined, as shown on line 11. Then, the leader iterator determines the offset within each block, denoted by i on line 19. The leader iterator also has knowledge of the total size of its caller, and this is denoted by *end*. Finally, the leader yields a range containing a statically disambiguated portion of its caller.

Section **??** described three steps necessary to perform modulo unrolling WU. As stated in Section 6.2, the *block cyclic transformation to ensure static disambiguation* is not required for the Cyclic distribution because a Cyclic distribution is equivalent to a Block Cyclic distribution with a block size parameter equal to 1, and this transformation is only required for Block Cyclic distributions with block sizes greater than 1. However, we explicitly perform the *block cyclic transformation to ensure static disambiguation* in the Block Cyclic leader iterator. Each range yielded by the Block Cyclic leader in line 21 of Figure 10 represents one strip mined portion of the transformed loop in Figrure 7.

Both the Cyclic and Block Cyclic leader iterators already assign loop iterations to locales according to the first item in the zippering by convention, so no owning expression calculation is necessary. Choosing to assign iterations based on the first item in the zippering is an accepted convention in the Chapel programming language that cannot be changed without needing to modify the leader iterators of other distributions not explored in this work. The only consequence of not directly implementing the owning expression calculation into the Cyclic and Block Cyclic leader iterators is that the loop will not minimize the number of remote data accesses per iteration.

Finally, for both the Cyclic and Block Cyclic distributions, the message aggregation step takes place in the modified follower iterators. Specifically, lines 15-28 of Figure 9 directly correspond to lines 3-6 and 14-16 in Figure 7.

We are currently in the process of contributing our source code implementation of modulo unrolling WU to the trunk repository of the Chapel compiler, maintained by Cray Inc. We are working very closely with the researchers at Cray to make this happen.

Name	Lines of Code	Input Size	Description	Elements per follower iterator chunk (Cyclic, Block Cyclic)
2mm	221	128 x 128	2 matrix multiplications (D=A*B; E=C*D)	4
fw	153	64 x 64	Floyd-Warshall all-pairs shortest path algorithm	2
trmm	133	128 x 128	Triangular matrix multiply	8
correlation	235	512 x 512	Correlation computation	16
covariance	201	512 x 512	Covariance computation	16
cholesky	182	256 x 256	Cholesky decomposition	16
lu	143	128 x 128	LU decomposition	8
mvt	185	4000	Matrix vector product and transpose	250
syrk	154	128 x 128	Symmetric rank-k operations	8
fdtd-2d	201	1000 x 1000	2D Finite Different Time Domain Kernel	16000
fdtd-apml	333	64 x 64 x 64	FDTD using Anisotropic Perfectly Matched Layer	4
jacobi1D	138	10000	1D Jacobi stencil computation	124, 249
jacobi2D	152	400 x 400	2D Jacobi stencil computation	2600
stencil9†	142	400 x 400	9-point stencil computation	2613
pascal‡	126	100000, 100003	Computation of pascal triangle rows	1563, 781
folding‡	139	50400	Strided sum of consecutive array elements	394

Figure 11: 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. All benchmarks are tested using the Chapel Cyclic distribution. Only *jacobi1D* and *pascal* are tested using the Chapel Block Cyclic distribution, with block sizes of 4 and 16 respectively. We also measure the maximum number of elements per follower iterator chunk of work for each benchmark to get a sense of how much aggregation is possible.

8. RESULTS

This section presents the results of four different experiments: a benchmark suite evaluation of modulo unrolling WU, a strong scaling experiment, a weak scaling experiment, and a block size variation experiment.

8.1 Benchmark Suite Evaluation

To demonstrate the effectiveness of modulo unrolling WU in the Chapel Cyclic and Block Cyclic distributions, we present the results of our benchmark suite evaluation. We have composed a suite of sixteen parallel benchmarks shown in Figure 11. Each benchmark is written in Chapel and contains loops with affine array accesses that use zippered iterations, as discussed in Section 7.2. This ensures that the leader and follower iterators where modulo unrolling WU is implemented are called. 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. Thirteen of the sixteen benchmarks are taken from the Polybench suite of benchmarks [1] and are 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 WU. *folding* is the only benchmark in our suite that has strided affine array accesses.

To evaluate improvements due to modulo unrolling WU, we ran our benchmarks using the Cyclic and Block Cyclic distributions from the trunk revision 22919 of the Chapel compiler as well as the Cyclic and Block Cyclic distributions that have been modified to perform modulo unrolling WU, as described in Section 7. We measure both runtime and message counts for each benchmark and report the normalized measurements with respect to the existing Chapel Cyclic and Block Cyclic distributions. We also compute the geometric means of all normalized runtimes and message count numbers for both distributions to get a sense of how much improvement, on average, modulo unrolling WU provided for our benchmark suite.

Data was collected on the ten-locale Golgatha cluster at the Laboratory for Telecommunication Sciences in College Park, Maryland. Each computing node on the cluster is comprised of two 2.93 GHz Intel Xeon X5670 processors, with 24 GB of RAM. The nodes are connected via an InfiniBand network communication link. Benchmarks *fdtd-apml*, *syrk*, *lu*, *mvt*, and *trmm* were run using eight of the ten locales because these programs drew too much power and did not complete execution during data collection when all ten locales were used. The remaining benchmarks were run on ten locales.

When evaluating modulo unrolling WU used with the Block Cyclic distribution, we only ran two benchmarks (*jacobi1D* and *pascal*) out of our suite of sixteen because of limitations within the original Chapel Block Cyclic distribution. Many of our benchmarks operate on two or three-dimensional arrays and are written using array slicing. Both array slicing of multi-dimensional arrays and array slicing containing strides for one-dimensional arrays are not yet supported in the Chapel compiler's Block Cyclic distribution. Implementing such features remained outside the scope of this work. There was no limitation when evaluating modulo unrolling WU with the Cyclic distribution, and all sixteen benchmarks were tested. Once these missing features are implemented in the Chapel compiler, then our method will apply to all of our benchmarks using the Block Cyclic distribution.

Figure 12 compares the normalized runtime numbers for the Cyclic and Block Cyclic distributions with and without modulo unrolling WU. For ten out of the sixteen benchmarks, we see reductions in runtime when the modulo unrolling WU optimization is applied to the Cyclic distribution. Both benchmarks tested with the Block Cyclic distribution with modulo unrolling WU show reductions in runtime. On average, modulo unrolling WU results in a 36 percent decrease in runtime for Cyclic and a 53 percent decrease in runtime for Block Cyclic.

Figure 13 compares the normalized message count numbers for the Cyclic and Block Cyclic distributions with and without modulo unrolling WU. For the Cyclic distribution, nine out of the sixteen benchmarks show reductions in message count 15 percent or greater. Both benchmarks tested with Block Cyclic with modulo unrolling WU show reductions in message count greater than 15 percent. On average, modulo unrolling WU results in a 64 percent decrease in message count for Cyclic and a 72 percent decrease in message count for Block Cyclic.

The final column in Figure 11 shows the maximum number of data elements per follower iterator chunk of work for each benchmark. These numbers, measured experimentally, give us a sense of how many data elements can be aggregated into

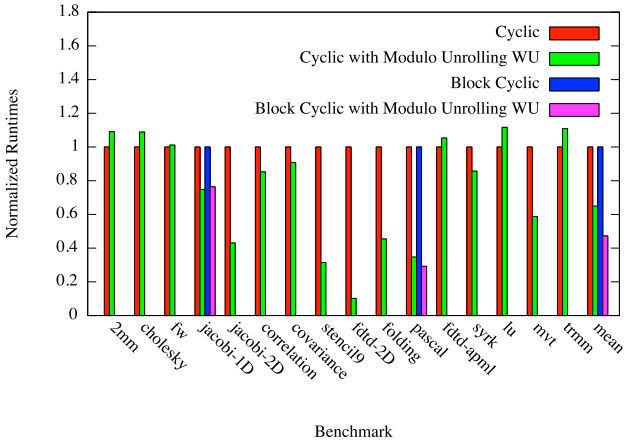

Figure 12: Runtime data collected for our suite of benchmarks. Each measurement is normalized to the benchmark's runtime using the original Chapel Cyclic and Block Cyclic distributions. Measurements below 1 indicate that benchmarks that use modulo unrolling WU with the specified Chapel distribution run faster. The last set of bars reports the geometric means of all sixteen normalized runtimes per distribution.

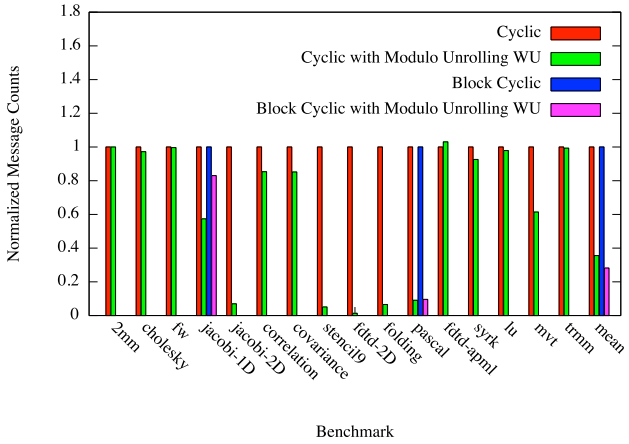

Figure 13: Message count data collected for our suite of benchmarks. Each measurement is normalized to the benchmark's message count using the original Chapel Cyclic and Block Cyclic distributions. Measurements below 1 indicate that benchmarks that use modulo unrolling WU with the specified Chapel distribution run using fewer messages. The last set of bars reports the geometric means of all sixteen normalized message counts per distribution.

a single message using modulo unrolling WU. The results of this experiment show that programs with chunks of work each containing more than a few hundred data elements see a significant runtime and message count improvement when using modulo unrolling WU over the original Chapel distributions.

Some detailed observations on Figures 12 and 13 follow. For six benchmarks that were run using the Cyclic distribution with modulo unrolling WU, runtimes were actually slightly slower and message count numbers either slightly increased or decreased by under 15 percent. Following Figure 11, all six of these benchmarks contain follower iterator chunks of work with few data elements. This suggests that, although modulo unrolling WU is applicable to these benchmarks, there are not enough elements to aggregate within each chunk of work to see performance improvments from message aggregation. Unlike individual remote data memory accesses (RDMA) that normally occur during each loop iteration, the strided bulk communication primitives `chpl_comm_gets` and `chpl_comm_puts` that are used in the optimization are not hardware optimized and are slower than RDMA when few data elements are being transferred. However, as the number of data elements per follower iterator chunk increases, we reach a point where the strided bulk communication primitives are faster than individual RDMA transfers. The six benchmarks that used the Cyclic distribution with modulo unrolling WU ran slower because of this overhead.

Another observation is that the Chapel distributions using modulo unrolling WU use more memory than the originals. The optimization yields elements directly from the local buffer that stores the aggregate message instead of yielding one remote element at a time. This increase in memory overhead is not unique to our scheme – any method that aggregates messages will necessary use more memory for the aforementioned reason. Although we did not directly measure peak memory usage, this means that each locale on our computer needs at least enough memory to fit the number of elements per follower iterator chunk for each benchmark (see Figure 11). This amount of memory is strictly a minimum because it is conceivable for multiple aggregate messages to take up space on a single locale at once, since follower iterator chunks are yielded in parallel. For very large data sets, this behavior could limit the cache performance that we would get when running the original distribution's follower iterator.

In Chapel, the size of a follower iterator chunk of work used in a parallel zippered loop is determined by many factors including the program's input size, the number of locales that the data is distributed over, the block size parameter (when data is distributed using the Block Cyclic distribution), and most importantly, the number of elements in each object of the zippering. This last factor is closely related to the algorithm of a particular program, and our benchmark suite evaluation has already tested how modulo unrolling WU performs for a variety of algorithms. The remaining subsections of Section 8 illustrate precisely how changing the parameters of input size, number of locales, and block size affect the performance improvements of modulo unrolling WU.

8.2 Strong Scaling Experiment

This experiment tests how strong scaling affects the performance improvements of modulo unrolling WU with the Chapel Cyclic and Block Cyclic distributions. Focusing on the following benchmarks – *pascal*, *folding*, *jacobi2D*, and *fdtd-2d* – problem sizes stay fixed according to Figure 11,

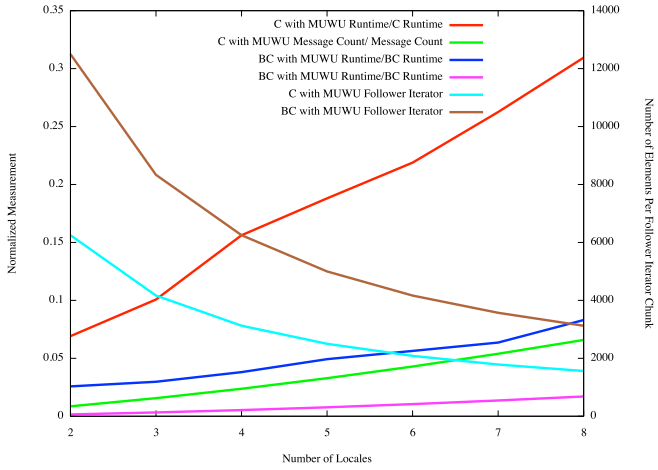

Figure 14: *pascal* strong scaling results. For the Block Cyclic results, the block size parameter was 4.

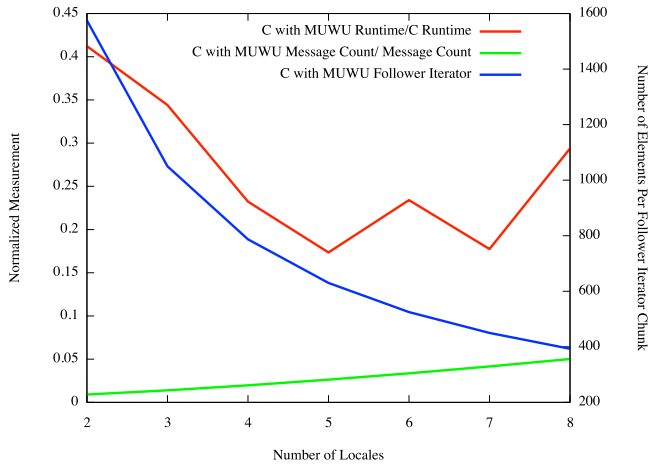

Figure 15: *folding* strong scaling results.

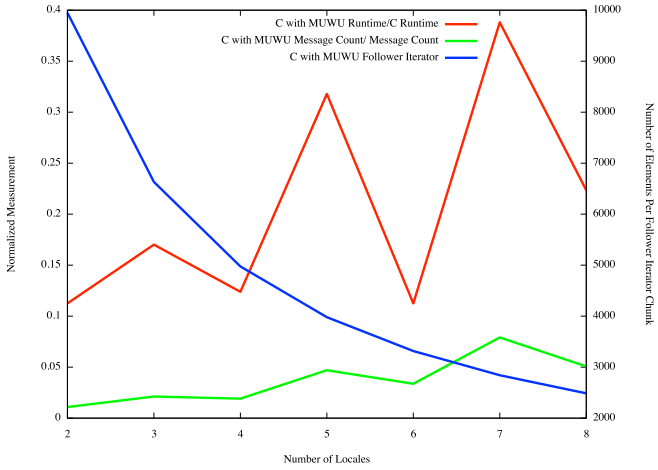

Figure 16: *jacobi2D* strong scaling results.

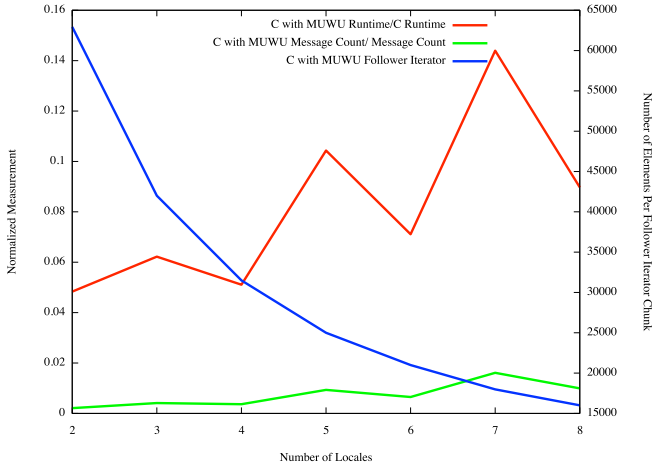

Figure 17: *fdtd-2d* strong scaling results.

but the number of locales varies from two to eight as we measure runtime and message counts relative to the existing Chapel distributions. We also measure the number of elements per follower iterator chunk as a function of the number of locales.

We choose this subset of benchmarks because they achieved the greatest runtime and communication performance improvements in Section 8.1, while still representing one- and two-dimensional data structures and strided and non-strided data access patterns. In this experiment, *pascal* is the only benchmark used to measure strong scaling for the Chapel Block Cyclic distribution.

Figures 14 - 17 show the strong scaling results for the four benchmarks. We observe the same two trends in each figure. First, the number of elements per follower iterator chunk is inversely proportional to the number of locales that the data is distributed over. This is directly related to the fact that there will be a fewer number of data elements distributed on each locale as more locales are present on the system. Second, as the number of locales increases, both the normalized runtime and message count measurements for modulo unrolling WU increase. This means that the performance improvement gap between modulo unrolling WU and the existing Chapel data distributions gets smaller as the number of locales increases. We attribute the decrease in performance improvement of modulo unrolling WU to be caused by fewer elements available to be aggregated as the number of locales increases.

In Figures 16 and 17, which correspond to the strong scaling results for the *jacobi2D* and *fdtd-2d* benchmarks respectively, we observe spikes in the normalized runtime and message count measurements for some numbers of locales, even though the overall trend still increases when more locales are added. Both the *jacobi2D* and *fdtd-2d* benchmarks happen to operate on two-dimensional arrays and contain nearest neighbor computation. When distributing data cyclically in Chapel, by default the pattern that the Cyclic distribution tries to assign elements to locales is as “square” or “rect-

angular" as possible. For example, in Figure 2, the two-dimensional array is distributed over four locales using a 2x2 pattern. However, with four locales, this same array could have been distributed using a 4x1 or 1x4 linear pattern. It turns out that for benchmarks similar to *jacobiD* and *fdtd-2d*, using a "rectangular" pattern will result in more remote data accesses per loop iteration than a linear pattern, leading to a higher improvement when message aggregation is applied. The spikes we see in Figures 16 and 17 occur when we distribute the data over a prime number of locales. In this case, the only pattern to use to distribute a two-dimensional array over a prime number n locales is a linear pattern of nx1 or 1xn. Note that it is possible to specify such patterns explicitly in Chapel for composite number of locales.

8.3 Weak Scaling Experiment

This experiment tests how weak scaling affects the relative performance of modulo unrolling WU with the Chapel Cyclic and Block Cyclic distributions. The same benchmarks used to measure strong scaling in Section 8.2 are also used here to measure weak scaling. Varying the input size of our benchmarks and keeping the number of locales constant at eight, we measure the runtimes and message counts normalized to the existing Chapel distributions. Once again, *pascal* is the only benchmark used to measure weak scaling for the Chapel Block Cyclic distribution.

Figures 18 - 21 show the weak scaling results for the four benchmarks. We observe some similar trends throughout each benchmark. First, normalized message counts are inversely proportional to the input size of the benchmark, and this is evident for both the Cyclic and Block Cyclic distributions. As input size increases, we observe that the message count measurements continue to increase when modulo unrolling WU is not used because each remote data access requires its own message. When modulo unrolling WU is used, message count measurements increase until a maximum and then stop increasing, due to aggregation, even when input size continues to increase. Normalized runtime also appears to be inversely proportional to input size, but a given benchmark's normalized runtime for a particular input size is not predictable. Finally, as input size increases, so does the number of elements per follower iterator chunk for both the Cyclic and Block Cyclic distributions, implying that larger input sizes do indeed create a greater opportunity for message aggregation. The number of elements per follower iterator chunk increases linearly for the *pascal* and *folding* benchmarks and quadratically for the *jacobi2D* and *fdtd-2d* benchmarks.

8.4 Block Size Variation Experiment

In this experiment, we focus on the two benchmarks *pascal* and *jacobi1D*, where the Chapel Block Cyclic distribution can be used to distribute data, in order to assess the effect of block size on the runtime and communication performance of modulo unrolling WU. We vary the block size parameter (keeping input sizes constant according to Figure 11 and the number of locales constant at eight) as we measure runtimes and message counts relative to the existing Chapel distributions.

Figures 22 and 23 show the results of the block size variation experiment for the *folding* and *jacobi1D* benchmarks, respectively. For both benchmarks, we observe that as block size increases, the relative performance improvement of mod-

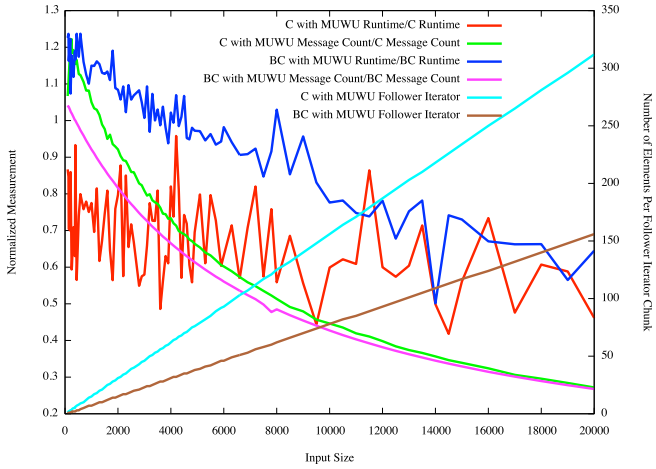

Figure 18: *pascal* weak scaling results. For the Block Cyclic results, the block size parameter is 16.

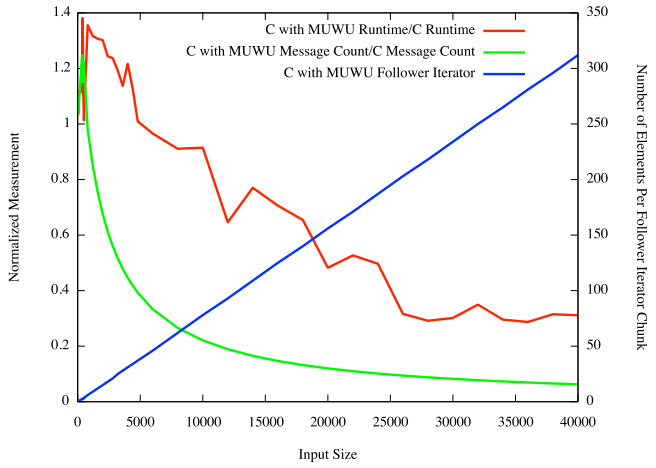

Figure 19: *folding* weak scaling results.

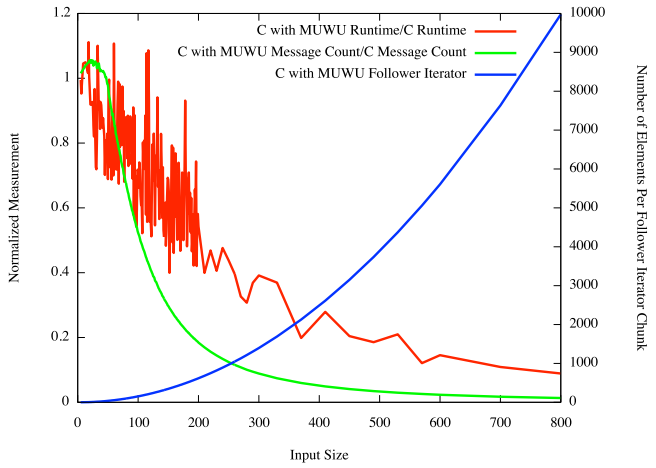

Figure 20: *jacobi2D* weak scaling results.

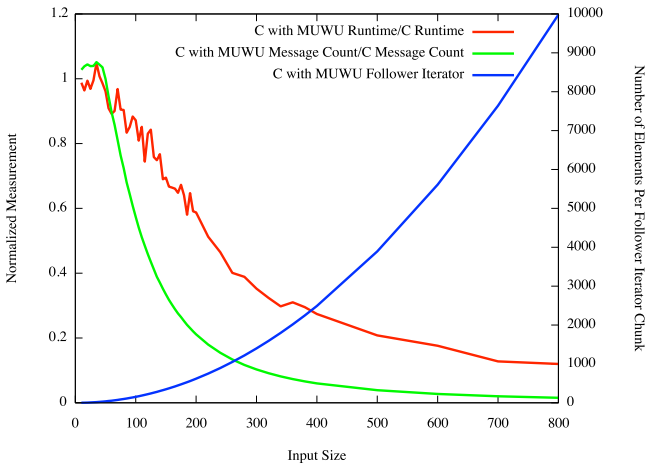

Figure 21: *fdtd-2d* weak scaling results.

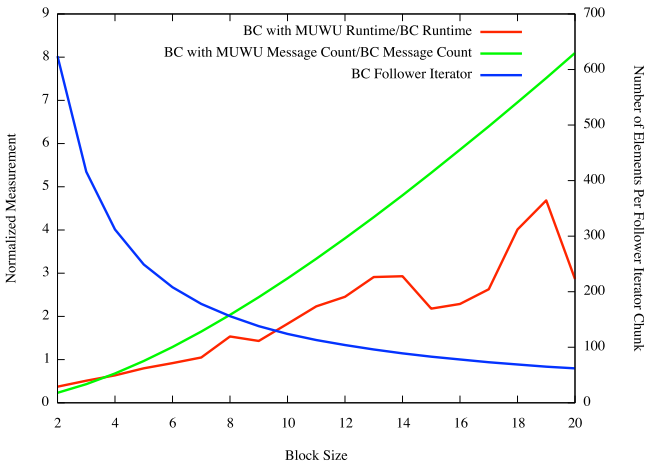

Figure 23: *jacobi1D* block size variation results.

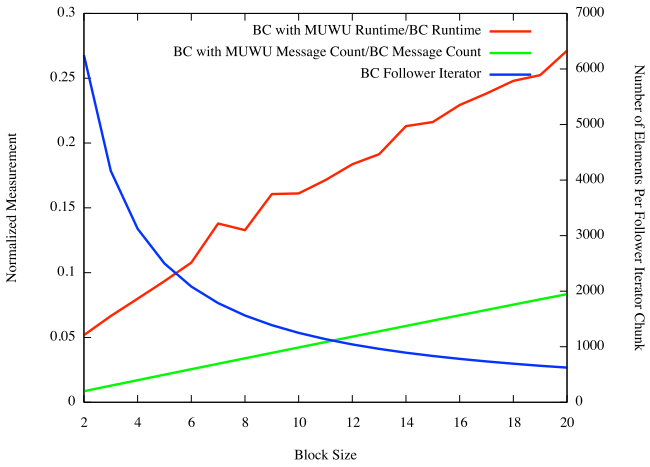

Figure 22: *folding* block size variation results.

ulo unrolling WU decreases. This is because increasing block size generally decreases the number of elements per follower iterator chunk, which lowers the amount of aggregation possible.

9. REFERENCES

[1] Polybench/C- The Polyhedral Benchmark Suite. `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`.

[2] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar. Communication optimizations for distributed-memory x10 programs. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1101–1113. IEEE, 2011.

[3] 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.

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

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

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

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

[14] L. Prylli and B. Tourancheau. Fast runtime block

cyclic data redistribution on multiprocessors. *Journal of Parallel and Distributed Computing*, 45(1):63–72, 1997.

[15] 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.

[16] 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.

[17] S. D. Sung-Eun Choi. Chapel: Distributions and Layouts. `http://chapel.cray.com/tutorials/DC2010/DC08-DISTRIBUTIONS.pdf`.

[18] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, et al. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

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

[20] 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.

[21] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11. ACM, 1994.

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