Affine Loop Optimization Using Modulo Unrolling in Chapel

Aroon Sharma

April 27, 2014

Abstract

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.

This paper presents a loop optimization for message passing programs that use affine array accesses in Chapel, a PGAS parallel programming language. Each message in Chapel incurs some non-trivial run time overhead. Therefore, aggregating messages improves performance. 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. First pioneered by Barua et al for tiled architectures, we adapt modulo unrolling to the problem of compiling PGAS languages to message passing architectures. When applied to loops and data distributed cyclically or block cyclically, modulo unrolling can 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 very complex problem of automatic code generation of message passing code from a PGAS language such as Chapel. It also results in substantial performance improvement compared to the unoptimized 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

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 with its own set of processors, 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 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). 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 one that 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.

This paper presents a loop optimization for message passing code generation based on a technique called modulo unrolling. 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 architecutres that use message passing. It has since been modified to apply to such machines in this work. Modulo unrolling used here works by unrolling an affine loop by a factor equal to the number of locales of the machine. 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 remote array accesses into a single message before and after the loop.

We demonstrate the modulo unrolling loop optimization in practice by implementing it in Chapel. 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 that greatly simplify the implementation of modulo unrolling into the language.

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 blocksize number of elements 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 blocksize parameter.

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. Data redistribution may be used if elements of a data set are inserted or deleted. In particular, algorithms to redistribute data using a new blocksize exist for the Block Cyclic distribution [21]. If an application uses a dynamic data set with elements being appended to the end, a Cyclic or Block Cyclic distribution is superior to Block because new elements can just be 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.

Allowing a data set to be compatible with other PGAS languages might be an important consideration for a programmer. If a data set is going to be used by Chapel programs and programs written in other PGAS languages, Cyclic or Block Cyclic distributions may be the best choice to distribute the data. This is because other PGAS languages, such as UPC, support Cyclic and Block Cyclic distributions but not Block. 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.

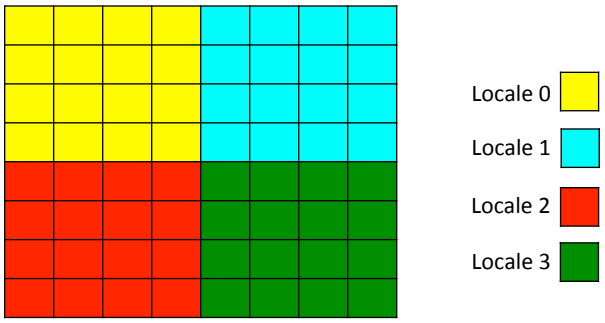

Figure 1: Chapel Block distribution.

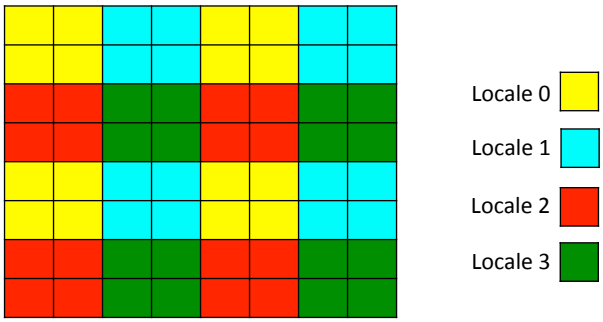

Figure 2: Chapel Block Cyclic distribution with a 2 x 2 blocksize parameter.

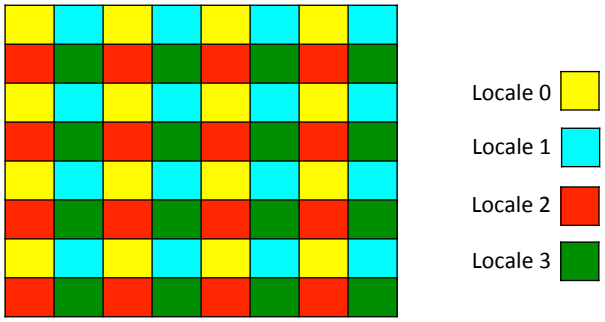

Figure 3: Chapel Cyclic distribution.

```
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 4: Chapel code for Jacobi computation.

2 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. For example, consider Chapel code for the Jacobi computation shown in Figure 4, 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**. Therefore, these remote elements are transferred over to $A_{new}[i, j]$'s locale in four individual messages every 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.

Since the data is distributed using a Cyclic distribution, we notice that the data is accessed in the same way every cycle. For example, on iteration $(2, 2)$, $A_{new}[2, 2]$ resides on locale 3, $A[2, 1]$ and $A[2, 3]$ reside on locale 1, and $A[1, 2]$ and $A[3, 2]$ reside on locale 2. If we look at iteration $(4, 2)$ which is an iteration in the next cycle, we see that $A_{new}[4, 2]$ also resides on locale 3, $A[4, 1]$ and $A[4, 3]$ reside on locale 1, and $A[3, 2]$ and $A[5, 2]$ reside

on locale 2. We can therefore bring in all remote data elements accessed by locale 3 to locale 3 before the loop executes and write them back to locales 1 and 2 after the loop finishes. 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 blocksize increases, the number of remote data accesses for the Jacobi computation decreases. For Jacobi, 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.

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 architecutre. 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 5. In Figure 5a 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 5b. 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 5c. 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.

Modulo unrolling, as used in [1] provides bank disambiguation and memory parallelism for tiled architecutures. 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.

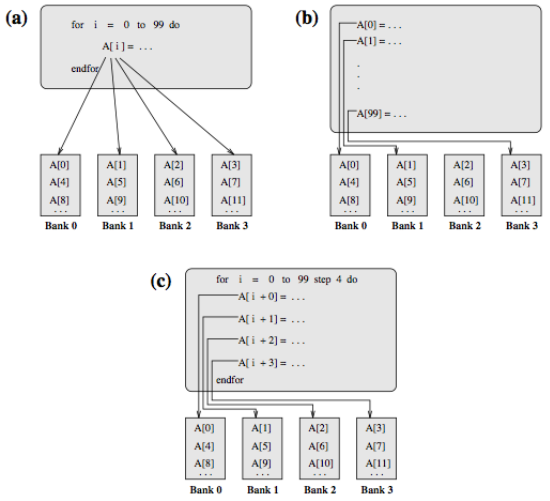

Figure 5: Modulo unrolling example. (a) Original sequential for loop. Array A is distributed using a Cyclic distribution. Each array access maps to a different memory bank on successive loop iterations. (b) Fully unrolled loop. Trivially, each array access maps to a single memory bank because each access 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.

4 Chapel 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, ... Arrays in Chapel support iteration by default.

Chapel allows multiple iterators of the same size and shape to be iterated through simultaneously. This is known as *zippered iteration* [4]. 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 6 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. Follower iterators generally resemble serial iterators such as $fibonacci(n : int)$ shown in Figure 6a.

4.1 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 succicnt 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. Consider the code fragment in Figure 7a. There are two affine array accesses $A[i]$ and $B[i+2]$ in Figure 7a. 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 to 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 7b, the same loop is written using zippered iteration. The loop induction variable i no longer needs to be specifed, 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.[1]

Because any parallel affine loop can be transformed to 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.

5 Cyclic Distribution with Modulo Unrolling

For the Cyclic distribution, only the follower iterator needs to be modified.[2]

[1] Question by Aroon: Is it true to say that any affine array access can be transformed using the way described here? What if the access was something like $A[2i + 3j]$?

[2] Aroon's comment: I am still working on a clear way to explain how modulo unrolling is implemented in both distributions without going into too much detail. Any suggestions on how I should do this?

(a)

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

(b)

```
Output:

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

Figure 6: (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 the variable j corresponds to the current loop iteration, i corresponds 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 6a.

(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 7: (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.

6 Block Cyclic Distribution with Modulo Unrolling

7 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 8. Each benchmark is written in Chapel and contains loops with affine array accesses that have been transformed to use zippered iteration by hand, as discussed in 4.1. 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 Cylic distributions that have been modified to perform modulo unroling, as described in Sections 5 and 6. 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 reamined 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 9 and 10 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.

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

† Benchmark taken from Chapel Trunk test directory

‡ Benchmark developed from scratch

Figure 8: Benchmark suite.

Figures 11 and 12 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.

8 Related Work

This will be where the related section goes.[3]

[3] Aroon's comment: My related works section is still a work in progress. I think I'd like to address traditional methods for compiler optimization, polyhedral methods for compiler optimization, and PGAS

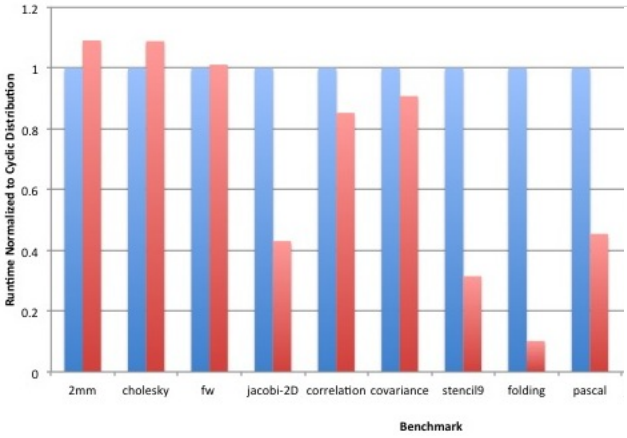

Figure 9: Cyclic runtime.

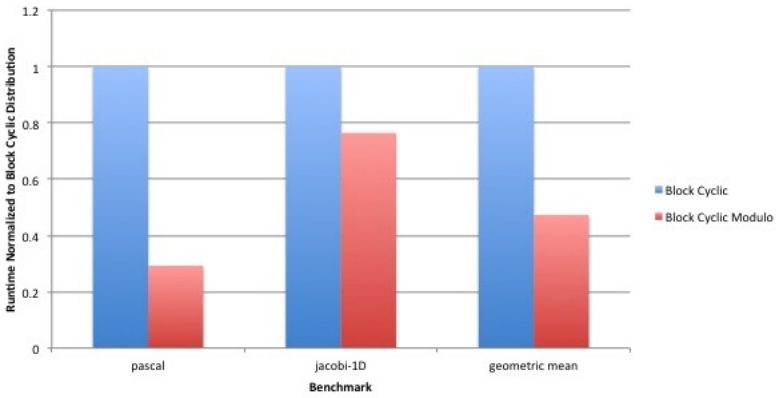

Figure 11: Block Cyclic runtime.

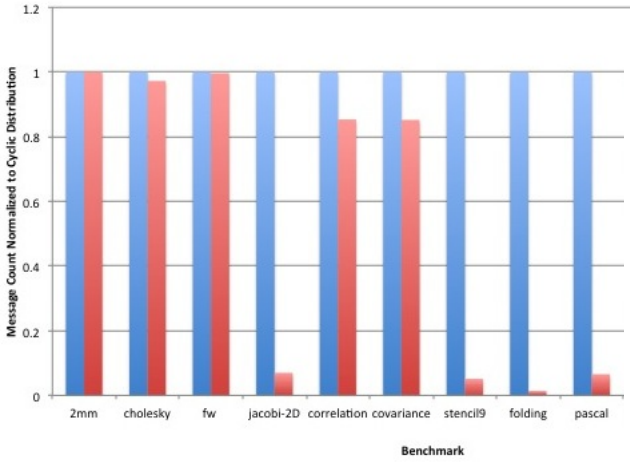

Figure 10: Cyclic message count.

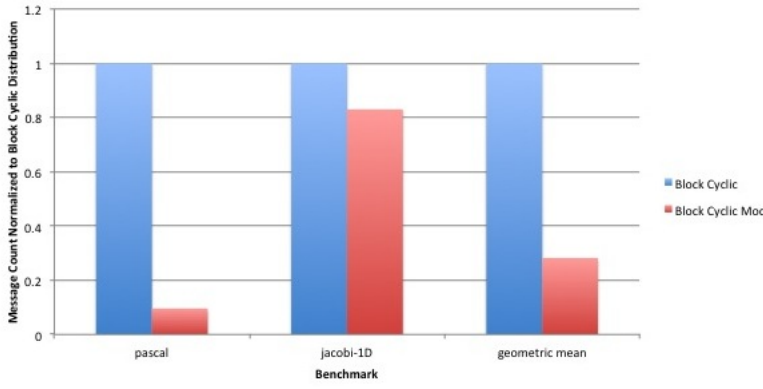

Figure 12: Block Cyclic message count.

[3] [6] [5] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [19] [20] [22] [4] [2] [18]

References

[1] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant 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] Dan Bonachea. Proposal for extending the upc memory copy library functions and supporting extensions to gasnet, version 2.0. 2007.

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

[4] Bradford L Chamberlain, Sung-Eun Choi, Steven J Deitz, and Angeles Navarro. User-defined parallel zippered iterators in chapel. 2011.

[5] Bradford L Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W Derrick Weathersby. Factor-join: A unique approach to compiling array languages for parallel machines. In *Languages and Compilers for Parallel Computing*, pages 481–500. Springer, 1997.

[6] Bradford L Chamberlain, C Lin, Sung-Eun Choi, L Snyder, C Lewis, and W Derrick Weathersby. Zpl's wysiwyg performance model. In *High-Level Parallel Programming Models and Supportive Environments, 1998. Proceedings. Third International Workshop on*, pages 50–61. IEEE, 1998.

[7] Daniel Chavarría-Miranda and John 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.

methods for compiler optimizations. The citations listed in this section are present here just so they would appear in the bibliography. In the final paper, we might not include all of them.

[8] Jack W Davidson and Sanjay Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 125–132. IEEE Computer Society Press, 1995.

[9] Michèle Dion, Cyril Randriamaro, and Yves Robert. Compiling affine nested loops: How to optimize the residual communications after the alignment phase? *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING (JPDC*, 38:176–187, 1996.

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

[11] Manish Gupta and Prithviraj Banerjee. Automatic data partitioning on distributed memory multiprocessors. Technical report, 1991.

[12] Sandeep K. S. Gupta, SD 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.

[13] Andrew S Huang, Gert Slavenburg, and John Paul Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*, pages 200–210. IEEE, 1994.

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

[15] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of parallel and distributed computing*, 13(2):213–221, 1991.

[16] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J Ramanujam, P Sadayappan, and Nicolas Vasilache. Loop transformations:

convexity, pruning and optimization. In *ACM SIGPLAN Notices*, volume 46, pages 549–562. ACM, 2011.

[17] 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.

[18] Alberto Sanz, Rafael Asenjo, Juan López, Rafael Larrosa, Angeles Navarro, Vassily Litvinov, Sung-Eun Choi, and Bradford 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.

[19] Kuei-Ping Shih, Jang-Ping Sheu, and Chih-Yung Chang. Efficient address generation for affine subscripts in data-parallel programs. *The Journal of Supercomputing*, 17(2):205–227, 2000.

[20] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, Ramakrishna Upadrasta, et al. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*, 2010.

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

[22] Wen-Hsing Wei, Kuei-Ping Shih, Jang-Ping Sheu, et al. Compiling array references with affine functions for data-parallel programs. *J. Inf. Sci. Eng.*, 14(4):695–723, 1998.