# Affine Loop Optimization Using Modulo Unrolling in CHAPEL

Aroon Sharma

April 15, 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. First pioneered by Barua et al for tiled architectures, we adapt modulo unrolling to the problem of efficiently 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 ten parallel benchmarks on a ten-locale cluster show that on average, modulo unrolling used with Chapel's Cyclic distribution results in 69 percent fewer messages and a 30 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.

How a program's data is distributed and the program's data access patterns determine the degree of message aggregation that is possible. 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 parameter is specified and this number of elements is allocated to consecutive array indices in a round robin fashion.

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. This information is used by the compiler 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. 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, 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 implementation 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.

## 2 Related Work

[3] [6] [5] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [19] [20] [21] [4] [2] [18]

# 3 Modulo Unrolling

This section describes modulo unrolling.

# 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.

[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] 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.