RICE UNIVERSITY
Code Generation for
Extreme Scale Parallel Systems
by
Karthik Srinivasa Murthy
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
Doctor of Philosophy
APPROVED, THESIS COMMITTEE:

John Mellor-Crummey, Chair
Professor of Computer Science and
Electrical and Computer Engineering

Keith Cooper
L. John and Ann H. Doerr Professor of
Computational Engineering

Vivek Sarkar
E.D. Butcher Chair in Engineering
Professor of Computer Science and
Electrical and Computer Engineering

Linda Torczon
Associate Chair, Department of
Computer Science

Peter Varman
Professor of Electrical and Computer
Engineering and Computer Science

Houston, Texas
March, 2017
Abstract

Power consumption and fabrication limitations are increasingly playing significant roles in the design of extreme scale parallel systems. These factors are influencing system designers to support higher on-node computing capability via throughput-optimized processors instead of latency-optimized processors. However, the inter-and intra-processor communication capabilities on such systems are not increasing at the same rate as the on-node computing capability. Consequently, achieving high performance requires careful orchestration of both single- and multiprocessor parallelism. This thesis shows that compiler technology and expressive programming model constructs can help applications more effectively exploit both forms of parallelism.

Compilers play an important role in harnessing short vector parallelism supported by cores in modern processors. Over last ten years, vector widths have increased dramatically from the 64-bit vectors supported by Intel’s Pentium MMX processor to the 512-bit vectors supported by Intel’s Knights Corner processor. However, the vectorization capabilities of state-of-the-art compilers are still immature, failing in the presence of complex control flow and data dependencies. This thesis presents compiler transformations that enable efficient vector parallelism in the presence of common kinds of complex dependencies.

To enable efficient multiprocessor parallelism, this thesis develops compiler technology to support sophisticated algorithms that minimize interprocessor communication. The class of .5D communication-avoiding algorithms was developed to address the inter-processor communication bottleneck. Mapping these algorithms to complex architectures efficiently is tedious for even expert programmers. To address this issue, this thesis presents the Maunam compiler, which generates efficient parallel code from a high-level, global-view sketch of a .5D algorithm that is expressed using symbolic data sizes and numbers of processors.
To mitigate the cost of communication for multiprocessor parallelism, this thesis develops a novel compiler transformation to overlap communication with computation for systolic computations. Additionally, to aid effective management of the completion of non-blocking communication, this thesis presents the cofence synchronization construct.
Acknowledgements

My family, advisor, professors at Rice, and my friends have been an integral part of the support and encouragement system that made this thesis a reality. I thank them profoundly.

I could not have imagined a better advisor than John. He has been a constant source of inspiration and support all these years. He has provided deep invaluable technical and professional advice. He has remained patient while I developed the skills needed to do good research. He has encouraged me to tackle hard problems. At the same time, he has helped me maneuver past many of the obstacles and setbacks that I encountered while tackling these hard problems. He has spent countless hours with me, just like he does with every student of his, to make sure that we generate the best work possible. I am eternally indebted to him.

Professor Keith Cooper and Dr. Linda Torczon have been gracious to lend a helpful ear whenever I want, technical or otherwise. Their constant words of encouragement made so many days so much less overwhelming. They gave me one of the biggest sources of happiness at Rice for me, working as a teaching assistant for COMP 412 for which I am truly grateful to them.

Professor Sarkar encouraged me to apply at Rice University. He has provided valuable technical advice on several projects. He has provided me with the same opportunities as his students. He and John referred me to one of the Codesign Summer Schools at Los Alamos National Laboratory, which remains the best work experience that I have had to-date. I thank him for always making the time for me.

Professor Varman has provided valuable feedback for this thesis. He made time for several of my presentations where I presented details on topics covered in this thesis. I thank him for his critical questions and feedback on the thesis.

I thank my collaborators, Sri Raj Paul, who collaborated on the design and im-
plementation of distributed phasers, Kuldeep Singh Meel, who collaborated on the complexity of operations on distributed phasers, Tiago Cogumbreiro, who helped write the paper on “Design and Verification of Distributed Phasers”. I thank Tim Harvey for our many discussions on vectorization over lunch. I thank Laksono Adhianto, Mark Krentel, Scott Warren, and Dung Nguyen for their support all these years.

I thank Deepak, Rishi, Priyanka, Milind, Prasanth, Dror, Kuldeep, Aarti, Priyanka, Shruti, Rajesh, Reshmy, Shams, Sagnak, Xu, and Chaoran for all the great memories at Rice University and Houston.

My family have come to my rescue umpteen times. Their constant supportive presence through the ups and downs of graduate school has made the ride much smoother. I thank Anuhya, Avani, Sohum, Ramya, Sharada, Srinivasa, Hyma, Indu, Mukul, Padma, Honnegowda, Sujatha, Ramamohan and Sravya for their support.

My work was supported in part by the DOE Office of Science Advanced Scientific Computing Research program through collaborative agreement DE-SC0008883 and by a gift from Texas Instruments. My research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.
# Contents

List of Illustrations x  
List of Tables xv  

1 Introduction 1  
1.1 Enabling Efficient Single-processor Parallelism 2  
1.2 Enabling Efficient Multiprocessor Parallelism 3  
1.3 Programming Model Constructs 4  
1.4 Thesis Statement 5  
1.5 Contributions 5  
1.6 Roadmap 6  

2 Related Work 7  
2.1 Generating High Performance Vector Code 7  
2.2 Generating High Performance Distributed Memory Code 13  
2.3 Hiding Communication Latency 16  
2.4 COFENCE 22  

3 dHPF Compiler Infrastructure 23  
3.1 dHPF Workflow 24  
3.2 Computation Partitioning for a Statement 26  
3.3 Usage of the Omega Library 26  

4 Generating High-performance Vector Code 28  
4.1 Handling Complex Data Dependences 29
5 Distributed Memory Code Generation 53

5.1 2.5D Matrix Multiplication ........................................... 56

5.2 Global-view 2.5D MM ................................................. 59
   5.2.1 Skew Communication ........................................... 61

5.3 Challenges ............................................................ 63
   5.3.1 Modulo Operations ............................................. 63
   5.3.2 Explicit Distributed Dimensions ............................... 65

5.4 Affinizing Modulo Operations ....................................... 66
   5.4.1 Simple Modulo Operations ..................................... 67
   5.4.2 Complex Modulo Operations ................................... 68
   5.4.3 Affinization ..................................................... 70
   5.4.4 Post-Affinization .............................................. 70
   5.4.5 Semantics of CSHIFT ......................................... 71

5.5 Reduction and Broadcast along Distributed Dimensions ....... 72
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Applicability to Other 5D CA algorithms</td>
<td>75</td>
</tr>
<tr>
<td>5.7</td>
<td>Experiments</td>
<td>76</td>
</tr>
<tr>
<td>5.8</td>
<td>Summary</td>
<td>81</td>
</tr>
<tr>
<td>6</td>
<td>Overlapping Communication with Computation</td>
<td>82</td>
</tr>
<tr>
<td>6.1</td>
<td>Motivation</td>
<td>82</td>
</tr>
<tr>
<td>6.2</td>
<td>EPAC Transformation</td>
<td>85</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Opportunity</td>
<td>85</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Description of the EPAC Transformation</td>
<td>90</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Safety</td>
<td>100</td>
</tr>
<tr>
<td>6.3</td>
<td>Experiments</td>
<td>100</td>
</tr>
<tr>
<td>6.4</td>
<td>Limitations</td>
<td>107</td>
</tr>
<tr>
<td>7</td>
<td>COFENCE</td>
<td>109</td>
</tr>
<tr>
<td>7.1</td>
<td>Motivation</td>
<td>110</td>
</tr>
<tr>
<td>7.2</td>
<td>Background</td>
<td>111</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Team</td>
<td>112</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Event</td>
<td>112</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Asynchronous Operations</td>
<td>113</td>
</tr>
<tr>
<td>7.3</td>
<td>Managing asynchrony in CAF 2.0</td>
<td>114</td>
</tr>
<tr>
<td>7.4</td>
<td>cofence API</td>
<td>116</td>
</tr>
<tr>
<td>7.5</td>
<td>cofence and Asynchronous Copy</td>
<td>118</td>
</tr>
<tr>
<td>7.6</td>
<td>cofence and Asynchronous Collectives</td>
<td>119</td>
</tr>
<tr>
<td>7.7</td>
<td>cofence and Shipped Functions</td>
<td>119</td>
</tr>
<tr>
<td>7.8</td>
<td>cofence and Events</td>
<td>120</td>
</tr>
<tr>
<td>7.9</td>
<td>cofence in Practice</td>
<td>123</td>
</tr>
<tr>
<td>7.10</td>
<td>Summary</td>
<td>123</td>
</tr>
<tr>
<td>8</td>
<td>Conclusions and Future Work</td>
<td>125</td>
</tr>
</tbody>
</table>
A 130

A.1 Transformational Intrinsic . . . . . . . . . . . . . . . . . . . . . . . 130

A.1.1 CSHIFT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130

A.1.2 SPREAD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131

A.1.3 SUM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131

A.1.4 TILT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132

Bibliography 133
Illustrations

3.1 Abstract dHPF compiler workflow to convert an input HPF program into an equivalent Fortran+MPI code. For a complete workflow and details on dHPF compiler framework, we refer the readers to Mellor-Crummey et al. [1].

3.2 Snippet of HPF program

4.1 GMG’s smooth kernel. Figure Credit: Basu et al. [2]

4.2 Performance analysis of the smooth kernel on a $400^3$ grid using Intel VTune performance profiler. The experiment was conducted on an Intel Haswell core. The kernel was compiled with `icc` version 17 and options `-O2 -g -restrict -xCORE-AVX2`. The above profile shows the memory system issues faced during the execution.

4.3 Memory access pattern in a 7-point 3D stencil.

4.4 Pseudo-code that depicts a one-dimensional stencil. $L_{i,j,k}$ and $U_{i,j,k}$ represent the lower and upper bounds of loop $i,j,k$, respectively. In this stencil, the elements of array $A^i$ are updated with the sum of elements of array $A$.

4.5 Pseudo-code that depicts a one-dimensional stencil before and after application of the semi-stencil algorithm along the $i$ dimension. $L_i$ and $U_i$ represent the lower and upper bounds of loop $i$. In this stencil, the elements of array $A^i$ are updated with weighted elements of array $A$; array $C$ contains the weights.
4.6 Memory access pattern in a 7-point 3D stencil after applying
semi-stencil transformation along the “j”-dimension. . . . . . . . . . . 41

4.7 Screen-shot from Intel VTune performance profiler showing the
performance analysis of the semi-stencil-transformed smooth kernel. . . 43

4.8 Screen-shot from Intel VTune performance profiler showing the
performance analysis of the semi-stencil-transformed smooth kernel,
which uses Cilk Plus notation. . . . . . . . . . . . . . . . . . . . . . . . 44

4.9 Screen-shot from Intel VTune performance profiler showing the
performance analysis of the original smooth kernel in Fortran. . . . . 45

4.10 Screen-shot from Intel VTune performance profiler showing the
performance analysis of the semi-stencil-transformed smooth kernel in
Fortran. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

4.11 A snippet of code from the Dilate kernel in the Leukocyte application. 46

4.12 Integer relation fed to Omega to eliminate the if-conditional involving
y in Dilate kernel. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

4.13 Count of vector instructions as reported by Intel Software
Development Emulator (SDE) for the original and if-eliminated
versions of Dilate kernel. . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

4.14 Performance of the if-eliminated version of Dilate kernel across
different input sizes and different compiler/hardware platforms. The
percentage reduction in time specified in each of the boxes above is
calculated as follows: \( \frac{T_{dilate\text{-}original} - T_{dilate\text{-}if\text{-}elim}}{T_{dilate\text{-}original}} \). . . . . . . . . . . . . . 50

4.15 Pseudo-code that depicts a two-dimensional stencil that is used in an
Image Integral kernel. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

4.16 Reference pattern for the computation of a single array element in a
2D in-place stencil that is used in an Image Integral kernel. . . . . . . 51
4.17 Factorization of a 2D in-place stencil, which is used in an Image Integral kernel, to two scan operations. Scan operations are easily vectorizable compared to the original 2D in-place stencil operation. 52

5.1 Workflow of Maunam. A snippet of Solomonik and Demmel’s 2.5D Matrix Multiplication [3] representing communication performed in the initial skew step is specified. Array ‘A’ represents one of the input matrices. Maunam reasons about communication partners when they are expressed either through TILT operations or through element-wise assignments with subscripts containing modulo operations. Maunam generates Fortran+MPI code. 55

5.2 2.5D Matrix Multiplication. $p$ represents the total number of processors, $c$ represents the depth of the virtual grid of processors. 58

5.3 Successive TILT s of data planes along the $j$ axis with respect to $k$ and $i$ axes achieves the required skew in 2.5D MM algorithm. The top half of the figure shows the TILT of a $j$ plane with respect to the $k$ axis. The bottom half of the figure shows the TILT of a $j$ plane with respect to the $i$ axis. 62

5.4 Conversion of a TILT operation into its equivalent modulo operation. 63

5.5 Global view expression of the shift communication in 2.5D MM expressed as element-wise assignments. 64

5.6 Solving for $s$ given $s = \text{mod}(j, m) + 1$. 67

5.7 Performance comparison of Maunam-generated 2.5D MM with Hand-coded 2.5D MM running on 128 cores performing 2.5D MM on $65536 \times 65536$ input matrices for different cores per socket. 77
5.8 Performance of Maunam-generated Fortran+MPI vs. hand-coded 2.5D MM code on a Cray XC30 supercomputer (Edison) for $65536 \times 65536$ input matrices. The hand-coded version employs overlap of communication with computation in the systolic phase of the 2.5D MM. ................................................................. 78

5.9 Performance comparison of Maunam-generated 2.5D MM with Hand-coded 2.5D MM for different sizes of the input matrices. The difference in performance achieved by Maunam-generated code and hand-coded version is $7 - 12\%$ of the machine peak. ................. 80

5.10 Time spent in the systolic phase waiting for communication completion increases with increase in size of input matrices. The graph shows times reported by CrayPat inside `MPI_Waitall` waiting for the `CSHIFT` communication to complete in the systolic phase. ... 80

6.1 HPF snippet contains a systolic loop where each processor sends and receives data in line 7 (S7), and consumes the received data in line 9 (S9). ................................................................. 84

6.2 Array definition-use patterns within a systolic loop that are addressed/not-addressed by EPAC transformation. All definitions above are covering definitions, uses before these definitions are upward exposed. $s, s', s''$ are index sets that access the unpartitioned dimensions of array $A$; $s' \subseteq s$ and $s'' \subseteq s$. $r, r'$ are index sets that access the unpartitioned dimensions of array $B$; $r' \subseteq r$. ■ indicates references to variables apart from $A$ and $B$. ......................... 87

6.3 Time spent in `MPI_Waitall` when multiplying matrices of rank $16384 \times 16384$. ................................................................. 103

6.4 Time spent in `dgemm` BLAS routine when multiplying matrices of rank $16384 \times 16384$. ................................................................. 104
6.5 Time spent in MPI\_Waitall when multiplying matrices of rank $2048 \times 2048$. 

6.6 Time spent in MPI\_Waitall when multiplying matrices of rank $65536 \times 65536$. 

6.7 Percentage of machine peak achieved and time spent in MPI\_Waitall when multiplying matrices of rank $65536 \times 65536$ on 4096 cores. The time taken in seconds for a single iteration of 2.5D MM is specified at the bottom of the bars in the left graph. The performance in TFlops/s is specified at the top of the bars in the left graph.

7.1 Execution timeline of an asynchronous operation indicating the different points of completion. *Finish, Phasers, cofence, and events* are different synchronization constructs that help achieve different points of completion.

7.2 Asynchronous operations and different stages of completion. *We discuss asynchronous broadcast as a representative example for asynchronous collectives.*

7.3 Cofence and asynchronous copy

7.4 Cofence and asynchronous broadcast. On a participating process image, say $q$, the cofence captures the arrival of broadcast data.

7.5 Cofence inside shipped functions

7.6 Sketch of the micro-benchmark to depict use of cofence and copy\_async. The size of the copied data is 80 bytes.

7.7 A micro-benchmark showing the advantage of using a cofence with copy\_async. The size of the copied data is 80 bytes, and the number of random processes to which data is sent in each iteration is 5.
Tables

2.1 Related work pertaining to vectorization in the presence of complex control flow and data dependencies. ........................................... 8
2.2 Related work pertaining to distributed memory code generation. .... 14
2.3 Related work pertaining to overlapping communication with computation. ................................................................. 20

4.1 Time taken in seconds for a single run of smooth kernel for different versions. ................................................................. 41

5.1 Time taken in seconds by a single iteration of hand-coded and Maunam-generated 2.5D MM on different number of cores for multiplying matrices of size: 65536 × 65536. .............................. 79
Chapter 1

Introduction

The computing power of a processor has increased a thousand-fold over the last 20 years [4]. However, increases have begun to slow because of thermal, fabrication, yield, and power consumption issues [5]. To continue the trend of delivering higher on-node computational capability, designs for extreme-scale parallel systems are shifting from latency-optimized processors that feature a few high-frequency cores to throughput-optimized processors that feature a larger number of low-frequency cores, higher thread counts, and wide-vector capabilities [6]. This shift significantly increases the available on-node parallelism. Harnessing the single-processor parallelism on such architectures is a challenge even to expert programmers.

Concurrently, the year-over-year rate of increase in the capability of communication pathways on extreme-scale systems is not on-par with on-node compute capability. The “Programming Abstractions for Data Locality” report [6] indicates that the cost of moving data operands within and between processors is and will continue to be orders of magnitude higher than the cost of performing an operation on them. Consequently, communication is and will be the primary power and performance bottleneck on such systems. Reducing communication and hiding communication latency on extreme-scale systems are at a minimum tedious and at worst difficult for programmers.

In this thesis, we show that compiler technology and expressive programming model constructs help applications address both challenges and deliver high perfor-
mance on extreme-scale systems. In this thesis, we present:

- compiler technology that enables efficient single-processor vector parallelism in the presence of complex control flow and data dependences,
- compiler technology that enables efficient multiprocessor parallelism by reducing communication and hiding communication latency, and
- programming model constructs that enable a succinct expression of different patterns of global data movement and that allow for effective management of asynchronous operations.

1.1 Enabling Efficient Single-processor Parallelism

Modern throughput-optimized processors feature a large number of low-frequency cores, threads, and wide-vector capabilities. Harnessing this parallelism is critical to achieving high performance. In this thesis, we focus on one aspect of on-node parallelism, namely short vector parallelism. Vector widths on a processor have increased dramatically from 64-bit vectors supported by Intel’s Pentium MMX processor (2001) to 512-bit vectors supported by Intel’s Knights Corner processor (2011). However, state-of-the-art compilers still fail to exploit vector parallelism efficiently in the presence of complex control flow and data dependencies. In this thesis, we present compiler transformations that enable high-performance vector parallelism in the presence of complex control flow and data dependencies.

To improve the performance of compiler-generated vector code for stencil calculations, we developed a compiler transformation that automatically identifies stencils and applies the “Semi-stencil” algorithm proposed by De La Cruz, Araya-Polo and Cela [7]. Their semi-stencil approach restructures the progression of a computation
along one or more stencil dimensions to improve its memory access pattern. We show that the improved access pattern enhances the performance of generated vector code.

To enable vectorization in the presence of complex control flow, we propose a compiler transformation that derives compact iteration spaces that avoid the need for control flow divergence. By employing these transformations, we achieve over a 30% reduction in time for a key smoothing kernel from a Geometric Multigrid code [8] and the Dilate kernel in the Leukocyte code [9] for reasonable input sizes on an Intel Ivy Bridge processor.

1.2 Enabling Efficient Multiprocessor Parallelism

Data movement between processors is a critical bottleneck for large scale parallel systems. The class of 5D communication-avoiding algorithms [3, 93, 94] were developed to address this bottleneck. These algorithms reduce communication and provide strong scaling in both time and energy. Mapping these complex algorithms to complex architectures efficiently is tedious for even expert programmers. Hence, we developed the Maunam compiler. Maunam generates efficient parallel code from high-level, global view sketches of 5D algorithms that are expressed using symbolic data sizes and numbers of processors. It supports the expression of data movement and communication through high-level global operations such as TILT—a cyclic skewing transformation of distributed multidimensional arrays—and CSHIFT—a circular shift of distributed arrays—as well as through element-wise copy operations. With element-wise copy operations, wrap-around communication patterns can be achieved using subscripts based on modulo operations. Maunam employs polyhedral analysis to reason about communication and computation present in a 5D algorithm. After partitioning data and computation, it inserts point-to-point and collective communi-
cation as needed. Maunam also analyzes data dependence patterns and data layouts to identify reductions over processor subsets. Maunam-generated Fortran+MPI code for 2.5D matrix multiplication running on 4096 cores of a Cray XC30 supercomputer achieves 59 TFlops/s (76% of the machine peak). Our generated parallel code achieves 91% of the performance of a hand-coded version.

Avoiding exposed communication latency is critical for high performance on modern supercomputers. One can hide communication latency by overlapping it with computation using non-blocking data transfers. In this thesis, we focus on overlapping communication with computation in systolic algorithms. Motivated by challenges that arise in algorithms including Communication Avoiding 2.5D Matrix Multiplication [3] and Gaussian Elimination, we present a novel compiler transformation that overlaps communication with computation even in the presence of an overlap-inhibiting data dependence between the communication and computation. Our transformation employs array Expansion, partial loop Peeling, loop Alignment, and array Contraction (EPAC) to convert an overlap-inhibiting data dependence into an overlap-amenable one. Applying EPAC to the systolic loop in 2.5D Matrix Multiplication, Maunam-generated Fortran+MPI code achieves 64 TFlops/s (81% of the machine peak) running on 4096 cores of a Cray XC30 supercomputer.

1.3 Programming Model Constructs

Additionally, effective management of the completion of non-blocking data transfers is necessary for hiding the communication latency. In this thesis, we develop the cofence synchronization construct. A cofence controls local data completion of implicitly synchronized asynchronous operations.
1.4 Thesis Statement

Managing data movement within- and across-nodes, and mapping control flow patterns to complex extreme-scale parallel architectures efficiently and achieving high performance is a daunting task for even expert programmers. By harnessing parallelism within and across compute nodes, compiler technology can help achieve high performance even in the presence of complex data and control flow. By employing the polyhedral framework for analysis, transformation, and code generation, compiler technology can generate high-performance vector and distributed memory code. By analyzing codes with systolic computations, compiler technology can hide communication latency along the critical path by overlapping communication with computation. Expressive programming constructs can help in the succinct expression and management of complex data movement on extreme-scale architectures.

1.5 Contributions

This dissertation makes the following contributions:

1. This thesis presents Maunam, the first compiler to generate distributed-memory parallel code from a high-level, global-view sketch of 5D algorithms that are expressed using symbolic data sizes and numbers of processors. Maunam analyses and optimizes SPMD systolic computations expressed in a global-view specification and produces efficient point-to-point and collective communication code using polyhedral analysis. This work appears in the paper, “Communication Avoiding Algorithms: Analysis and Code Generation for Parallel Systems”, which was presented at PACT 2015 [10].

2. This thesis develops a novel compiler transformation that hides communica-
tion latency by overlapping communication with dependent computation. This transformation composes the techniques of array Expansion, partial loop Peeling, loop Alignment, and array Contraction (EPAC) to achieve overlap. This work appears in the extended abstract, “A Compiler Transformation to Overlap Communication with Dependent Computation”, which was presented at PGAS 2015 [11].

3. This thesis describes co-fence synchronization construct, which enables a programmer to manage local data completion of implicitly-synchronized asynchronous operations [12]. This work appears in the paper, “Managing Asynchronous Operations in Coarray Fortran 2.0”, which was presented at IPDPS 2013.

4. This thesis develops compiler transformations to enable high-performance vector parallelism in a programmer’s input. These transformations are planned to be included in an industry-leading compiler infrastructure.

1.6 Roadmap

The rest of this thesis is organized as follows: Chapter 2 describes related work. Chapter 3 describes the dHPF compiler infrastructure used to implement the compiler transformations presented in this thesis. Chapter 4 describes the compiler transformations to enable high-performance vector parallelism. Chapter 5 describes the Maunam compiler, which generates efficient distributed memory code from a high level global-view sketch of 5D algorithms. Chapter 6 describes a compiler transformation to overlap communication with computation in systolic algorithms. Chapter 7 describes the co-fence synchronization construct. Chapter 8 presents conclusions.
Chapter 2

Related Work

In this section, we highlight literature relevant to the areas of generating high performance vector and distributed memory code.

2.1 Generating High Performance Vector Code

<table>
<thead>
<tr>
<th>Year</th>
<th>If-conversion, Iteration-set Splitting, Consensual Branches</th>
<th>Stencil Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td></td>
<td>Data-centric Multi-level Blocking [14].</td>
</tr>
<tr>
<td>1999</td>
<td></td>
<td>Time skewing: A Value-based Approach to Optimizing for Memory Locality [15].</td>
</tr>
<tr>
<td>2003</td>
<td>Improving Performance with Integrated Program Transformations [16]</td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>Generalized Index-Set Splitting [17]</td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td></td>
<td>Efficient Temporal Blocking for Stencil Computations by Multicore-aware WavefrontParallelization [20].</td>
</tr>
<tr>
<td>Year</td>
<td>Title</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2010</td>
<td>The Polyhedral Model is More Widely Applicable Than You Think [21]</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>Divergence Analysis with Affine Constraints [26]; Vectorization Past Dependent Branches Through Speculation [27]</td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>The Impact of SIMD-width on Control-flow and Memory Divergence [28]; Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures [29]</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1 : Related work pertaining to vectorization in the presence of complex control flow and data dependencies.

Table 2.1 highlights literature relevant to the generation of high performance vector code. We classify the related work into two categories: improving the performance of stencil computations, and handling of conditionals that hinder vectorization within loops. Below, we provide technical details on these references.
Improving the performance of Stencil Computations  Several techniques have been presented in the past to improve the performance of stencil computations. A popular technique is blocking. In blocking, the application of a stencil computation proceeds in blocks or submatrices, which promotes reuse of data cached at higher level caches of the memory hierarchy, instead of complete rows or columns. Lam et al. [13] discuss the performance impact of the blocking technique for different block sizes, stencil strides along different dimensions, and cache parameters. In contrast to blocking, our compiler transformation for stencil computations, discussed in Section 3.3, splits the application of a stencil on a single element into a forward and backward step to reduce loads and promote stores. Kodukula et al. [14] focus on data flow through the memory hierarchy rather than control flow to improve performance. They schedule the computation in the input program through data shackles. A “data shackle” represents a data object in the input program and its blocks. It also specifies the statement instances that reference the data blocks. The enumeration order for these data blocks can follow the program order specified in the input program or a different order to allow for optimizations as long as they respect the data dependences in the input program. In effect, their scheduling technique is constraining all accesses to a data block to occur once, i.e., when the corresponding data shackle is scheduled and thereby, promoting data reuse. In contrast, our compiler transformation does not schedule all operations on a data block at once.

Mccalpin and Wonnacott [15] show that loop skewing in conjunction with other transformation techniques helps to reduce the memory traffic in timed loops. They employ loop skewing with loop interchange and loop skewing with forward substitution along with a novel data flow analysis through “iteration space dataflow graph” to achieve better memory access patterns. They also reduce stores and loads per iter-
ation via forward substitution of computations from one loop nest to another within the same timed loop. In contrast, our transformation is applicable to both timed loops and non-timed loops, and does not employ skewing; skewing is employed as a precursor to help achieve tiling or parallelization in codes where data dependence prevents the application of tiling or parallelization. We also increase the number of stores in an iteration by orchestrating the backward movement of a portion of the computation from later iterations. Kamil et al. [19] show that both cache-aware and cache-oblivious approaches are useful, and that a cache-aware approach is better than a cache-oblivious approach due to sub-optimal optimizations applied on a cache-oblivious code. Their best results are on systems with explicitly-managed memory hierarchy. Our approach is cache-oblivious, in the sense that we rely on the caching mechanism and not specifically on the parameters of any level of cache. Our approach can benefit from an explicitly-managed memory hierarchy where stores are prioritized over loads.

Wellein et al. [20] describe a “pipelined wavefront parallelization” approach for stencil computations. Their approach schedules multiple wavefronts at the same time but spatially shifted in the computation domain. Their approach achieves the best performance on multicores where threads share at least the last level cache. In their approach, only one thread works on one wavefront. The thread on the foremost wavefront loads data from memory, and the threads working on subsequent wavefronts benefit from these loads. Additionally, only the thread working on the last wavefront stores data back to memory. Thus, they are able to employ the sharing of caches between cores in a multicore architecture to reduce the loads from and stores to the memory system. Our transformation exploits vectorization and reduces the loads from memory per thread. Henretty et al. [25] improve performance by
applying a non-linear data layout transformation to stencils that suffer from stream alignment conflicts. The intuition for their work is as follows: stream alignment conflicts require either costly inter-register data movement or additional loads from memory, and changing the data layout to make adjacent memory locations begin at aligned boundaries using a dimension-lifting transformation ameliorates this problem. Our transformation does not change the layout of the input data. Additionally, our transformation reduces the working set of a stencil along a dimension from \( L + R + 1 \) to \( \max(L, R) + 1 \), where \( L, R \) are the number of points/planes accessed in the left and right directions along the dimension, respectively. Their work does not achieve any such reduction.

Basu et al. [30] propose a compiler transformation that employs array common subexpression elimination to improve the performance of stencils by eliminating redundant floating point operations. By exploiting the symmetry in a stencil computation, they compute and store partial sums, which are reused to derive multiple outputs. While our transformation does calculate partial sums, it does not reuse them for calculating multiple output values. We do so to ensure that our transformation is applicable to both constant- and variable-coefficient stencils. In variable-coefficient stencils, it is difficult to identify common floating point subexpressions, in the event that they do exist.

Handling conditionals that hinder vectorization within loops Several techniques have been presented in the past to address the challenge of control flow within iteration spaces; control flow within loops results in ineffective vectorization and SIMT parallelization. Our strategy to address if-conditions, described in Section 4.5, employs polyhedral methods to derive compact iteration spaces at compile time that
account for the effect of affine conditionals. Lee et al. [29] address the challenge of conditionals within CUDA kernels by identifying “consensual” branches. Consensual branches are those that are taken only when all the threads within a CUDA warp have the same value. They employ both static and dynamic techniques to identify/group such branches. Han et al. [24] address this challenge by employing two techniques, iteration delaying and branch distribution. In the presence of a divergent branch in a loop, iteration delaying technique executes only one of the branches and delays the threads that execute the other branch for later iterations. Branch distribution reduces the size of divergent code paths by factoring out structurally similar code. Unlike the previous two techniques, our technique does not require any runtime support or architectural changes, e.g., ability to delay a subset of threads in a warp. Speculation is another important technique that is employed when the conditionals are too complex for static analysis. Sujon et al. [27] identify that a control hazard is a key hindrance preventing vectorization and propose a technique to improve code-vectorizable capability through speculation. Specifically within loops, they speculatively vectorize code paths assuming that these will be taken in consecutive loop iterations. If the speculation fails, they re-evaluate the previous vector-width of iterations using scalar operations, termed as “scalar restart”.

Employing the polyhedral model, Quillere et al. [31] propose a technique that splits the iteration spaces based on the constraints enforced from the affine-if conditionals present within these spaces. Vasilache et al. [18] propose a solution that does a depth-first traversal of the AST, summarizing the conditionals that constrain the span of each iteration space, and factorize them via polyhedral separation, namely intersection and difference. Our technique is similar to Vasilache et al. [18]. However, our technique targets affine conditionals involving auxiliary induction variables. We
argue that by targeting these conditionals in practice leads to fewer convex fragments of the original loop in the generated code than by targeting all affine conditionals.

### 2.2 Generating High Performance Distributed Memory Code

<table>
<thead>
<tr>
<th>Year</th>
<th>Languages and Compilers for Distributed Memory Code Generation</th>
<th>Polyhedral Frameworks and Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>Code Generation for Multiple Mappings [32]</td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>Compiler-Optimization of Implicit Reductions for Distributed Memory Multiprocessors [33]</td>
<td>Constraint-based Array Dependence Analysis [34]</td>
</tr>
<tr>
<td>2002</td>
<td>PICO: Automatically Designing Custom Computers [35]; Advanced Optimization Strategies in the Rice dHPF Compiler [36]</td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>Generation of Injective and Reversible Modular Mappings [37]</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Code Generation in the Polyhedral Model Is Easier Than You Think [38]</td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td>Static Macro Data Flow: Compiling Global Control into Local Control [39]</td>
<td></td>
</tr>
</tbody>
</table>
Table 2.2: Related work pertaining to distributed memory code generation.

Table 2.2 highlights literature relevant to the generation of high performance distributed memory code. Below, we provide technical details on few of these references that have synergy with Maunam.

**Languages and compilers for distributed-memory code generation**
Charisma [39] is a script-like language that enables programmers to develop SPMD programs by writing a high-level specification of global control and data flow. In Charisma, communication between objects is specified explicitly with “production” and “consumption” constructs. In contrast, our compiler automatically derives communication partners from a global view program. Bondhugula [41] describes a source-to-source compiler that employs polyhedral methods to analyze loop nests and generate MPI code for distributed memory parallel systems. A novel aspect of this compiler is that it does not use data distribution directives to guide parallel code generation. Like our compiler, Bondhugula’s compiler supports analysis and code generation for symbolic numbers of processors. Unlike our compiler, Bondhugula’s compiler only analyzes loop nests with affine subscripts and doesn’t support subscripts with modulo operations.

Kathail et al. develop a framework “PICO” (program in, chip out) that derives multiple system designs from an input C-based application, where the different designs tradeoff different amounts of cost for performance. The PICO-NPA is a subsystem.
that derives co-processors, also called Non-programmable Accelerators (NPA), for functions expressed in the input C-based application [45]. The user or a design exploration tool specifies a loop nest inside a C-based application for which the PICO-NPA needs to derive an NPA. The performance required of the derived NPA is specified in terms of the number of processors and their processing speed per iteration of the input loop nest. PICO-NPA transforms the loop nest in a sequence of phases. One of these phases is mapping the iterations of the loop nest to the specified number of processors. To implement this phase, PICO-NPA employs an implementation based on the SUIF compiler [46]. In this phase, it tiles the iterations in the input loop nest and assigns a tile to a processor. The key issue that it faces in assigning tiles to a processor is in ensuring that the memory demand from a tile does not cross the performance budget allocated to a processor. During tiling, mapping and scheduling the iterations among the processors, PICO-NPA faces the same challenges as our compiler such as identifying the extent of data referenced in a tile, identifying the definition for a specific array use, and communicate to and from local memory, in our case its remote memory and hence, additionally, we need to identify the remote processor. However, PICO-NPA restricts the loop bounds to be constant. It requires the user provide perfect loop nests as input. Our compiler does not have such restrictions and allows for loop bounds with symbolic variables.

Analysis of code containing non-affine expressions The Chill compiler supports polyhedral analysis of code with non-affine expressions [43]. Chill uses uninterpreted function symbols to represent non-affine loop bounds and uses the inspector-executor approach to cope with non-affine subscripts. Together, these techniques enable Chill to analyze and transform code for sparse matrix kernels. While Chill
could employ the inspector/executor approach to code with modulo operations in subscripts, our compiler’s affinization of modulo operations enables it to use polyhedral analysis to represent communication sets and relations between communication partners at compile time.

Lee and Fortes [37] generate modular mappings, which apply a modulo operation to a linear form with the modulus being a symbolic constant. Such mappings are invertible, injective and can be used to represent circular shift operations. Like our compiler, they are able to derive a reversible mapping if the span of a distributed dimension is the same as the modulus of a modulo operation. However, they cannot generate code for the complex modulo operation resulting from a TILT in 2.5D MM because it is not a modular mapping due to the presence of non-affine terms.

**Polyhedral code generators** Manuum uses the Omega Library [47, 32] for polyhedral code generation. Other polyhedral code generation frameworks include CLooG [38] and AlphaZ [42]. To the best of our knowledge, these code generators have little or no support for code generation in the presence of modulo operations. While CLooG provides limited support for modulo operations in a schedule [38], this support isn’t general enough to represent the modulo operations needed for TILT.

We are unaware of any data-parallel compiler that analyzes and generates collective communication over a subset of the dimensions of a processor grid.

### 2.3 Hiding Communication Latency
<table>
<thead>
<tr>
<th>Year</th>
<th>Communication-Computation Overlap</th>
<th>Storage Management / Loop Transformations / Overlap-Supporting Experiments</th>
<th>Overlap Related Transformations - Message Aggregation / Locality Enhancement / Software Pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>1989</td>
<td>Process Decomposition Through Locality of Reference [48]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1991</td>
<td>A Data Locality Optimization Algorithm [49]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1993</td>
<td>An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines [51]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1994</td>
<td>Techniques to Overlap Computation and Communication in Irregular Iterative Applications [52]</td>
<td>A new approach to array redistribution: Strip mining redistribution [53]</td>
<td></td>
</tr>
<tr>
<td>1995</td>
<td></td>
<td>Solving Linear Recurrences with Loop Raking [54]</td>
<td></td>
</tr>
<tr>
<td>Year</td>
<td>Research Area</td>
<td>Reference(s)</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------------------------------------------------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td>1996</td>
<td>Global communication analysis and optimization</td>
<td>[55]</td>
<td></td>
</tr>
<tr>
<td>1997</td>
<td>Data Centric Multilevel Blocking</td>
<td>[14]</td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>Automatic Storage Management for Parallel Programs</td>
<td>[56]</td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>Storage mapping optimization for parallel programs</td>
<td>[57];</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interarray Data Regrouping</td>
<td>[58]</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>Loop Transformation Algorithm for Communication Overlapping</td>
<td>[59]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Optimizing memory usage in the polyhedral mode</td>
<td>[60]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Synthesizing Transformations for Locality Enhancement of Imperfectly-nested</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Loop Nests</td>
<td>[61];</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Queues: A New Hardware/Software Approach to Efficient</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Software Pipelining</td>
<td>[62]</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Decoupled Software Pipelining with the Synchronization Array</td>
<td>[63]</td>
<td></td>
</tr>
<tr>
<td>Year</td>
<td>Title</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>HUNTING the Overlap</td>
<td>Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications [65]</td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>MPI-aware compiler optimizations for improving communication-computation overlap [66]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td>Automatically Generating Coarse Grained Software Pipelining from Declaratively Specified Communication [67]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td>Bamboo - Translating MPI applications to a latency-tolerant, data-driven form [68]; Exact Dependence Analysis for Increased Communication Overlap [69]</td>
<td>Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems [70]; Automatic generation of software pipelines for heterogeneous parallel system [71]</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td></td>
<td>Asynchronous MPI for the masses [72]</td>
<td></td>
</tr>
</tbody>
</table>
A well-known optimization for distributed memory parallel programs is to overlap computation with communication. Table 2.3 highlights literature relevant to overlapping communication with computation. Chakrabarti et al. [55] implement a novel algorithm in IBM’s pHPF compiler that performs a global analysis of communication required in a procedure to determine communication placement. By doing so, they are able to eliminate/reduce unnecessary communication and communication buffers. Hanxleden et al. [50] develop the GIVE-N-TAKE code placement framework to hide latency and eliminate redundant communication by placing send communications as early as possible and receive communications as late as possible. An interesting aspect of their framework is that by considering the side-effects of an earlier placement of communication or computation, they avoid placement of certain subsequent redundant communications, i.e., by relying on “production comes for free”. However,
in both the previous compiler algorithms, the earliest and latest positions for a communication satisfy the data dependence existing in the input program. Our compiler transformation breaks this dependence to exploit opportunities for overlap.

Pellegrini et al. employ the polyhedral model to apply loop fission on loops containing \texttt{MPI\_Send} and \texttt{MPI\_Recv} to obtain iterations that do not have a dependence on the communicated data [69]. Rogers et al. [48] employ loop distribution, loop jamming, and code motion to achieve communication-computation overlap by pipelining communication in certain loops when there is a true dependence. Ishiziaki et al. [59] employ loop tiling and loop interchange to achieve overlap of communication with computation in certain loops. Danalis et al. [66] employ LNO (loop nest optimization) and CCTP (communication computation tiling and pipelining) to partition the computation and communication into smaller portions and then overlap communication with computation. Strip mining of communication and computation increases latency along the critical path and some computations might not be amenable to this technique as well. Hence, our compiler transformation overlaps communication with dependent computation without splitting them into smaller portions.

Bamboo employs programmer annotation, namely, \texttt{olap}, to identify regions of code that contains communication to be overlapped with computation [68]. It relies on virtualization to pipeline communication and computation. They apply an MPI-communication reorder-rule, which effectively advances an MPI send communication along the backedge of a communication loop allowing for communication-computation overlap. This reorder rule is necessary to cope with the requirement of Tarragon, the underlying task graph library used by Bamboo to model the data dependencies.
2.4 COFENCE

The cofence construct is inspired by the SPARC V9 [75] MEMBAR instruction (for loads and stores to memory only), which has two variants: ordering and sequencing. The ordering MEMBAR allows precise control over what reorderings it will restrict. Ordering requirements are specified using a bit mask. For example, a MEMBAR #LOADLOAD requires that loads before the MEMBAR complete before any loads after the MEMBAR; this flavor of MEMBAR does not restrict the ordering of stores.

UPC's upc_fence [76] separates relaxed operations on shared data before and after the fence. In contrast, cofence enables a programmer to indicate that some kinds of reorderings are allowed across the fence, which provides greater flexibility for performance tuning. Also, the asynchrony support in UPC is limited to get and put operations with explicit completion guarantees.

The GASNet communication library [77] provides non-blocking get and put operations. It provides several routines to control the completion of implicit asynchronous operations. These routines, however, only capture global completion of asynchronous operations, and do not provide the finer grained direction control cofence provides. Fortran 2008 provides three primary synchronization constructs: SYNC ALL, SYNC IMAGES, and SYNC MEMORY. SYNC MEMORY prevents the movement of any FORTRAN statement that has side effects visible to other process images. cofence provides finer-grain control than SYNC MEMORY.
Chapter 3

dHPF Compiler Infrastructure

In this chapter, we introduce the Rice dHPF compiler infrastructure [1, 78, 79]. The dHPF infrastructure forms the platform on which the following thesis contributions are built: a) Maunam, a compiler to generate distributed-memory parallel code from a high-level, global-view sketch of 5D algorithms expressed using symbolic data sizes and numbers of processors, b) EPAC, a compiler transformation that hides communication latency by overlapping communication with dependent computation, and c) compiler transformations that help enhance the ability to leverage vector parallelism in a programmer’s input.

High Performance Fortran [80] provides a global view programming model for parallel programming. In this model, programmers write a single-threaded Fortran program augmented with compiler directives that describe the layout of data across an array of processors. The dHPF compiler infrastructure analyzes an HPF program and generates an equivalent Fortran+MPI program for execution on parallel systems. It reasons about communication and computation in a parallel program by translating it into sets and mappings described by constraints expressed in Presburger arithmetic. In the following sections, we describe the workflow of dHPF, briefly explain one of the steps in the workflow, i.e., determining computation partition for a statement, and present a note on the usage of Omega [47] – a library for manipulating integer sets and relations of integer tuples.
Figure 3.1: Abstract dHPF compiler workflow to convert an input HPF program into an equivalent Fortran+MPI code. For a complete workflow and details on dHPF compiler framework, we refer the readers to Mellor-Crummey et al. [1].

3.1 dHPF Workflow

Fig. 3.1 presents an abstract dHPF compiler workflow to convert an input HPF program into an equivalent Fortran+MPI code. Below, we briefly describe each step:

1. dHPF builds control flow, static single assignment, and dependence graph representations of the input program. These graphs are used for compile time reasoning about the runtime flow of values.

2. dHPF computes computation partitions for statements in the input program; a computation partitioning for a statement partitions the elementwise computations for data arrays manipulated by the statement among a set of processors. Computation partitionings are expressed as abstract mappings between statement instances and processors.

3. dHPF calculates the communication mappings for a statement employing a com-
putation partitioning. These communication mappings specify each processor’s role in communication required for a statement instance.

4. dHPF creates data movement placeholders, which are required to realize the communication.

5. dHPF refines computation partitions to reflect the additional processors that might be needed to execute a statement. It propagates computation partitions from statements to their dominators to ensure that a processor that needs to execute a statement executes every statement along the control flow path to that statement.

6. dHPF generates code skeleton containing statement placeholders that implements the computation partitionings using the Omega Library [47].

7. dHPF replaces the statement placeholders with statement instances from the input program. In all previous stages, virtual processor identities are employed. In this step, virtual identities are mapped to their physical processor identities.

8. dHPF replaces data movement placeholders with appropriate MPI primitives, i.e., collective and point-to-point primitives as necessary, to achieve the required communication.

9. dHPF recomputes the derived information to detect any errors.

For a complete workflow, we refer the readers to Mellor-Crummey et al. [1]. dHPF also performs communication optimizations such as: communication coalescing to reduce the total number of communications by grouping messages to different references or the same reference, communication vectorization to extract communication out of
loops to send larger messages instead of sending element-wise messages, and dataflow-based communication placement to initiate a communication as early as possible and delay the querying of communication completion to as late as possible.

3.2 Computation Partitioning for a Statement

In this section, we briefly explain step two from the workflow. A computation partitioning for a statement partitions the execution of the statement among a set of processors. A popular partitioning strategy is the “owner-computes” rule, which specifies that a computation is executed by the owner of computed value [48]. Consider the statement, S7, present in Fig. 3.2, the CP ON_HOME(q(i,j)), specifies that the instance of the statement in iteration i, j will be executed by the processor owning the array element q(f1(i),f2(j)), where f1(i) and f(j) are affine functions. The set of processors defined by the CP is identified by the subscript vectors, i, j, and the distribution of q reaching that statement, i.e., distribution specified by lines 3 and 4 in the snippet. dHPF supports both “owner-computes” rule and more sophisticated CPs. For example, the CP for a statement could be expressed as the owner of one or more arbitrary data references, i.e., \( \bigcup_{x \in 1,r} \text{ON_HOME}(q_x(f1(i),f2(j))) \). Also, each statement within a scope can have a different CP. dHPF can support a general class of partitionings since it uses an expressive representation based on sets and mappings described by Presburger arithmetic and enumeration algorithm for non-convex sets provided by the Omega library [47].

3.3 Usage of the Omega Library

dHPF uses the Omega Library to perform sophisticated optimizations and generate parallel code by expressing sets of processors, data elements and loop iterations along
with bidirectional mappings between pairs of these sets using Presburger formulae. The Omega Library determines the satisfiability of Presburger formulae by employing a modified form of Fourier-Motzkin elimination, known as the Omega Test [81]. For code generation, the Omega Library uses an algorithm based on Fourier-variable elimination [32]. To reduce the complexity of generated code, in many cases dHPF generates code skeletons for one loop level at a time rather than generating complete code skeletons for complex multi-level loops in a single step.
Chapter 4

Generating High-performance Vector Code

Over last ten years, vector widths have increased dramatically from the 64-bit vectors supported by Intel’s Pentium MMX processor to the 512-bit vectors supported by Intel’s Knights Corner processor. However, vectorizing capabilities of state-of-the-art compilers are still immature, failing in the presence of complex control flow and data dependencies. In this chapter, we present compiler transformations that enable high-performance vector parallelism in the presence of complex data and control flow dependencies.

In Section 4.1 to Section 4.4, we describe how to improve the performance of compiler-generated vector code for out-of-place stencil calculations by automatically identifying opportunities and applying the “Semi-stencil” algorithm proposed by De La Cruz, Araya-Polo and Cela [7]. In Section 4.5, we describe how to enable vectorization in the presence of complex control flow by proposing a compiler transformation that derives compact iteration spaces that void the need for control flow divergence. In Section 4.6, we discuss a compiler transformation to enable vectorization in a subset of in-place stencil computations by factorizing an in-place stencil into multiple scan operations.
4.1 Handling Complex Data Dependences

Partial Differential Equation (PDE) solvers are at the core of scientific computing. A common technique for solving PDEs is employing stencil computations, where the next value for an array element is based on some weighted computation of values in the element’s neighborhood. Stencil computations are difficult to perform efficiently on modern microprocessors because they are memory intensive. To motivate work on improving the vector performance of stencil computations, we focus on the challenge of enhancing the performance of vector code for a stencil employed for a smoothing calculation as part of a geometric multigrid solver developed by Williams et al. [8].

Fig. 4.1 shows a version of the smooth kernel written in C [2]. This smooth kernel employs an out-of-place stencil in $S_0$ with variable coefficients, and simple updates in $S_1$ and $S_2$. Vectorization of $S_0$ is straightforward because of the absence of any flow-, anti-, or output-dependences carried by the loop nest. However, employing an out-of-place stencil with variable coefficients results in a load-heavy stencil, i.e., a single iteration of $S_0$ writes one element while loading 13 array elements. Managing the movement of data through the memory hierarchy is critical to achieving high vector performance.

To better understand the time spent in $S_0$, we applied a top-down analysis using the general exploration feature of Intel’s VTune performance profiler [82, 83]. Our analysis helped identify that in a C version of the smooth kernel, the $S_0$ loop nest is completely vectorized. However, $S_0$ is significantly backend-bound memory-bound, as shown in Fig. 4.2. Specifically, 39.6% of clock cycles are spent waiting for data from the L1/L3 caches and DRAM. To generate a high-performance vector code, we need to reduce the time spent waiting for data.

To address these issues, De La Cruz, Araya-Polo and Cela propose restructur-
Additionally fuse smooth, residual and restriction together. Since typically there is a vertical communication-avoiding optimization, as it exposes reuse of variables the fusion of the restriction operator with the preceding residual operator in Figure 4, as it eliminates the redundant computation across time steps, a well-known horizontal communication. In our prior work on compiler-based optimization of GMG, we fuse the multiple smooths together, our prior work introduced communication-avoiding optimization. This allowed multiple applications of smooth per communication exchange that store data associated with redundant computation and the piecewise constant restriction used here is common in geometric multigrid, as grids are likely to exceed last-level cache. However, since execution time is dominated by the smooth operator, we would like to fuse smooth, residual and restriction together.

Figure 3: The Smooth operator.

Figure 4: The Residual operator.

Figure 3, employing array data-flow analysis to contract the compiler fused the Laplacian, Helmholtz and GSRB of the restriction operation is integral to geometric multigrid, which stores the output grid, and the second loop nest iterates through points in the input grid and accumulates their values in the output grid. The resulting new loops are then transformed output.

Figure 5: Example code for the Restriction operator.

Figure 4.1: GMG’s smooth kernel. Figure Credit: Basu et al. [2]
Figure 4.2: Performance analysis of the smooth kernel on a 400\(^3\) grid using Intel VTune performance profiler. The experiment was conducted on an Intel Haswell core. The kernel was compiled with `icc` version 17 and options `-O2 -g -restrict -xCORE-AVX2`. The above profile shows the memory system issues faced during the execution.

Their semi-stencil approach restructures the progression of a computation along one or more stencil dimensions to improve the memory access pattern. The semi-stencil restructuring is described in Section 4.2. Manually applying the semi-stencil transformation to a C version of the smooth kernel reduces the time waiting for data to 22%.

In this chapter, we describe a compiler transformation that automatically identifies opportunities for restructuring stencil operations into the semi-stencil form and safely performs the transformation. Section 4.2 describes the semi-stencil algorithm. Section 4.3 describes the opportunity, safety and details of our compiler transformation for applying the semi-stencil algorithm.
4.2 Semi-stencil Algorithm

In this section, we summarize the work of De La Cruz, Araya-Polo and Cela [7]. We start with the goal of the semi-stencil algorithm in Section 4.2.1, followed by a description of the semi-stencil approach in Section 4.2.2.

4.2.1 Goal

Consider the memory access pattern for a 7-point 3D stencil, shown in Fig. 4.3, that reads and writes different arrays; pseudo-code for this 3D stencil is present in Fig. 4.4. The stencil loads 7 elements of array A and writes 1 element of $A_l$ in every iteration of the 3-level perfectly nested loop $i, j, k$. Both arrays A and $A_l$ are stored in column-major order, i.e., data is continuously placed in memory along the “i”-dimension. The extent of the 3D stencil along each of the dimensions is one, i.e., computation of a single element, $A^l(i, j, k)$, requires a) three elements within the same column, $A(i-1, j, k)$, $A(i, j, k)$, and $A(i+1, j, k)$, b) two elements from adjacent columns, $A(i, j-1, k)$ and $A(i, j+1, k)$, and c) two elements from adjacent planes, $A(i, j, k-1)$ and $A(i, j, k+1)$. As a result, the working set of the stencil includes three planes of array A. Applying the semi-stencil algorithm along the “k”-dimension reduces the working set of the stencil from three to two planes of array A. It does so by breaking the stencil update into a forward and backward update. In this process, it trades potential load misses of the left plane with a store miss. Store latency is better hidden by the store buffer on most modern architectures*. Bhargava

*Stores are not immediately flushed to memory on most modern computer architectures. They are instead placed in store buffers and flushed out to memory when the buffer gets full or as needed by memory consistency.
and John [84] discuss the advantages of having store buffers that can house 16 – 64 active entries before pushing these entries to L1 or later stages of cache†.

4.2.2 Description

The central idea of the semi-stencil algorithm is to break the assignment in a stencil to two phases, forward and backward, which results in updating two points in one iteration of the stencil loop. Consider the one-dimensional stencil in Fig. 4.5. In this stencil, the elements of array $A^t$ are updated with weighted elements of array $A$; array $C$ contains the weights. The width of this stencil is $L+R+1$, i.e., $A(i)$, $L$ elements to the left of $A(i)$, and $R$ elements to the right of $A(i)$ are needed to calculate the $i^{th}$ element of $A^t$. In one iteration of loop $i$, one element of array $A^t$ is updated. The semi-stencil algorithm, applied along dimension $i$, breaks this update into two steps:

- **Forward Update:** In this step, element $A^t[i+L]$ receives values from $L$ elements beginning at $A[i]$. By doing so, a partial result for $A^t[i+L]$ has been computed.

†We do not know the sizes of the store buffers on the platforms where we performed our experiments.
\begin{verbatim}
  do k=L_k, U_k
     do j=L_j, U_j
       do i=L_i, U_i
         A_l(i,j,k) = A(i,j,k) + A(i+1,j,k) + A(i-1,j,k)
         + A(i,j-1,k) + A(i,j+1,k)
         + A(i,j,k-1) + A(i,j,k+1)
       enddo
     enddo
  enddo
\end{verbatim}

Figure 4.4: Pseudo-code that depicts a one-dimensional stencil. \(L_{i,j,k}\) and \(U_{i,j,k}\) represent the lower and upper bounds of loop \(i,j,k\), respectively. In this stencil, the elements of array \(A_l\) are updated with the sum of elements of array \(A\).

- Backward Update: In this step, element \(A_l[i]\) receives values from \(R\) elements beginning at \(A[i]\). This steps compliments the forward update and completes the computation for \(A_l[i]\); a partial value for \(A_l[i]\) was calculated during forward update in iteration \(i-L\).

As a result of these updates, the working set of the stencil is reduced to \(\max(L,R)+1\) along the “i”-dimension.

4.3 Semi-stencil Compiler Transformation

In this section, we describe the opportunity, safety and details of the semi-stencil compiler transformation implemented in the dHPF compiler.

4.3.1 Opportunity

The candidate perfect loop nests performing a stencil operation, i.e., single-element updates in each iteration, where semi-stencil can be applied along a specific dimension
!Before applying Semi-stencil

1 \textbf{do } i=L_i, U_i
2 A^i(i) = C(i-L) \cdot A(i-L) \cdot C(i-L+1) \cdot A(i-L+1) + \ldots + C(i) \cdot A(i)
3 + \ldots + C(i+R-1) \cdot A(i+R-1) + C(i+R) \cdot A(i+R)
4 \textbf{endo}

!After applying Semi-stencil

!prologue not presented for conciseness

5 \textbf{do } i=L_i, U_i-L
6 A^i(i) = A^i(i) + C(i) \cdot A(i) + \ldots + C(i+R-1) \cdot A(i+R-1) + C(i+R) \cdot A(i+R)
7 A^i(i+L) = C(i) \cdot A(i) + \ldots + C(i+L-1) \cdot A(i+L-1)
8 \textbf{endo}
9 !epilogue not presented for conciseness

Figure 4.5: Pseudo-code that depicts a one-dimensional stencil before and after application of the semi-stencil algorithm along the \textit{i} dimension. \textit{L}_i and \textit{U}_i represent the lower and upper bounds of loop \textit{i}. In this stencil, the elements of array \textit{A}^i are updated with weighted elements of array \textit{A}; array \textit{C} contains the weights.
of access are as follows:

- There should be no loop-carried flow-, anti- and output-dependences.

- There should be loop-carried input dependences; this indicates that there are excessive loads.

- All array index expressions must be SIV, i.e., single-induction variable, expressions. Index expressions must be of the form \( i \pm \text{sym} \pm c \), where \( i \) is a loop induction variable, \( \text{sym} \) is a variable that is loop-invariant inside the candidate loop-nest, and \( c \) is a constant. Further,
  
  - all index expressions for a specific dimension must refer the same induction variable \( i \).
  
  - all index expressions for a specific dimension can refer at most one \( \text{sym} \) variable.
  
  - a maximum of two different \( \text{sym} \) variables can be used across all the index expressions.

- The lvalue of the assignment performed in the candidate loop nest must be an array type.

4.3.2 Safety

The fundamental theorem of dependence states that “a reordering transformation that preserves every dependence in a program preserves the meaning of that program”. The semi-stencil transformation is designed for perfectly nested loop nests without flow dependence. By breaking a single update step into a forward and backward update, the semi-stencil transformation introduces an output and flow dependence. However,
these dependences are necessary to ensure that the complete value of the updated point is calculated. No anti-dependences are introduced due to the transformation. In conclusion, the transformation maintains all dependences and preserves the meaning of the program.

4.3.3 Details

Application of the semi-stencil transformation involves generating a prologue, transformed loop nest, and epilogue. In the description below, the original loop nest refers to the loop nest before the application of the semi-stencil algorithm, while the transformed loop nest refers to the loop nest after the application of the semi-stencil algorithm. Before describing the details, we define three entities:

1. Smallest Stencil Extent along i-dimension (\( \hat{S} \))
   
   The smallest stencil extent along \( i \) is the smallest offset along \( i \) used in the stencil. We obtain this extent by sorting the index expressions that index \( i \)-dimension. For example, in Fig. 4.5, the \( \hat{S} \) along dimension \( i \) is \(-L\). We can sort these expressions at compile time because all index expressions use constants, at most one symbolic variable, and a maximum of two different symbolic variables across all index expressions\(^4\), apart from the loop induction variable.

2. Forward Update Statement, FUS, for semi-stencil along i-dimension

   The forward update statement is a copy of the stencil assignment in the original loop that is obtained by

\(^4\)We can generate multiple versions of the transformed stencil code if we cannot ascertain whether a symbolic is \( \geq 0 \) or \( < 0 \).
• retaining only the operations on array elements that have offsets from $-1$ to $\hat{S}$ along $i$, and

• adding ($-\hat{S}$) to all index expressions that index $i$ in the lvalue and rvalues.

3. Backward Update Statement, BUS, for semi-stencil along $i$-dimension

The backward update statement is a copy of the stencil assignment in the original loop that is obtained by

• removing all the operations on array elements that have offsets from $-1$ to $\hat{S}$ along $i$, and

• converting the assignment to an update of the lvalue.

For the rest of this section, loop $j$ is the loop whose induction variable, $j$, governs the index expressions to the first non-contiguous dimension in the stencil. $L_j$, $U_j$ are the lower and upper bounds of the $j$ loop. $\hat{S}$ now refers to the smallest stencil extent along the $j$-dimension. We now describe the steps to generate the prologue, epilogue and the main loop in the transformed code.

**Generation of the prologue** The prologue to the semi-stencil transformed loop body captures the forward updates from the left boundary of a stencil, i.e., $\hat{S}$ elements to the left of the first application of the stencil. The following AST\(^8\) manipulations are performed.

• A copy of the stencil loop nest is inserted before the original loop nest.

• The prologue loop bounds are updated as follows:

\(^8\)Abstract Syntax Tree
The lower bound is set to $L_j$ and $\hat{S}$. The rationale for doing so is as follows: Consider $j_{L_j}$ as the first point along $j$ where the stencil is applied. The working set for $j_{L_j}$ includes $\hat{S}$ elements to the left of $j_{L_j}$. To accommodate the contributions from these $\hat{S}$ elements, we need to begin the prologue loop at $L_j-\hat{S}$.

The upper bound is set to $\min(L_j-1, U_j-\hat{S})$, i.e., the minimum of the original lower bound minus one and the difference of the original upper bound and the stencil extent along $j$. As discussed in the previous bullet, the working set for $j_{L_j}$ includes *stencil-extent* points to the left of $j_{L_j}$. The last point to the left of $j_{L_j}$ in the working set is $j_{L_j}-1$. However, to prevent the application of the stencil on elements beyond $U_j$, we need to end the prologue loop at the minimum of $j_L-1$ and $j_U-\hat{S}$.

- The assignment inside the loop nest is replaced by FUS.

**Generation of the semi-stencil transformed loop nest** The following AST manipulations on the original loop nest are performed.

- The upper bound of loop $j$ is reduced by $\hat{S}$. This prevents a forward update beyond the last point $U_j$.

- The assignment inside the loop nest is replaced by FUS and BUS.

**Generation of the epilogue** The epilogue captures the last few iterations of the backward update only. There are no forward updates in these iterations since a forward update in these iterations would update the points beyond $U_j$. The following AST manipulations are performed.
• A copy of the stencil loop nest is inserted after the original loop nest.

• The epilogue loop bounds are updated as follows:

  – The lower bound is set to \( \max(U_j - \hat{S}_j, L_j) \), i.e., the maximum of the difference between the original upper bound and \( \hat{S}_j \), and the original lower bound.

  – The upper bound remains unchanged.

• The assignment inside the loop nest is replaced by \texttt{BUS}.

4.4 Experiments

In this section, we discuss the application of the semi-stencil transformation on \( S_0 \) of the smooth kernel in Fig. 4.1 and discuss its performance. Based on our experiments manually applying the semi-stencil algorithm along different dimensions on problems with different data sizes and on various generations of Intel hardware, our compiler transformation applies the semi-stencil algorithm only on the first non-contiguous dimension of the stencil, e.g., “j”-dimension in Fig. 4.3. An illustration of this transformation is shown in Fig. 4.6. The indices of 	exttt{temp} updated in \( S_0 \) are indicated in red, while the indices of 	exttt{phi} read are indicated in blue. We conducted our experiments on an Intel Ivy-Bridge cluster at Rice University’s Center for Research Computing [85] and on a Cray XC40 supercomputer (Cori) at NERSC [86]; each compute node on the Cray XC40 has two Intel Haswell processors.

Experimental Methodology. We compiled original and semi-stencil-transformed smooth kernel with the Intel Compiler Suite 15.1. The execution time is the average time taken for a single iteration of the smooth kernel. We employed the Intel VTune
Figure 4.6: Memory access pattern in a 7-point 3D stencil after applying semi-stencil transformation along the “j”-dimension.

<table>
<thead>
<tr>
<th></th>
<th>Time Taken (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Smooth-kernel in C</td>
<td>0.615</td>
</tr>
<tr>
<td>Semi-stencil Transformed Smooth-kernel in C</td>
<td>0.425</td>
</tr>
<tr>
<td>Cilk Plus based Smooth kernel in C</td>
<td>0.406</td>
</tr>
<tr>
<td>Original Smooth-kernel in Fortran</td>
<td>0.497</td>
</tr>
<tr>
<td>Semi-stencil Transformed Smooth-kernel in Fortran</td>
<td>0.461</td>
</tr>
</tbody>
</table>

Table 4.1: Time taken in seconds for a single run of smooth kernel for different versions.
performance profiler to identify performance bottlenecks on the Intel Haswell core; we used the Haswell cores for performance profiling since we were unable to get access to a VTune installation with the required drivers on the Ivy-Bridge core. The semi-stencil transformation is implemented inside the dHPF compiler, which takes as input a well-formed High Performance Fortran code and generates MPI+Fortran as output. In this section, we discuss the performance of both hand-coded C versions of the smooth kernel and semi-stencil-transformed smooth kernel, and dHPF generated semi-stencil-transformed smooth kernel. Our observations are as follows:

- Applying the semi-stencil transformation along the first non-contiguous data dimension improves the performance of a C-version of smooth kernel by 30.81% on an Intel Ivy-Bridge core when working on a $400^3$ grid. The execution time for different versions of smooth kernel on a $400^3$ grid is present in Table 4.1. Applying the semi-stencil transformation along both non-contiguous data dimensions improves the performance of a C-version of smooth kernel by 29.1%, which is less than the improvement obtained by applying semi-stencil along the first non-contiguous data dimension.

- Using the “general exploration” feature of VTune, we observe that the DRAM latency accounts for 52.2% of the clock ticks in the original version. This is reduced to 8.7% of the clock ticks in the semi-stencil version. This highlights the effect of the semi-stencil transformation, which reduces the trips to DRAM and thereby, incurring less DRAM latency. This is shown in Fig. 4.2 and Fig. 4.7. These executions were on an Intel Haswell core when working on a $400^3$ grid.

- Employing a Cilk Plus array notation [87] in the original smooth kernel, only Intel compilers support the Cilk Plus notation, we observe that the performance
improves by 33%. While this improvement is higher than the improvement achieved by the semi-stencil version, our approach is superior since it is automatic and does not require the programmer to rewrite the code in a different form. Employing VTune performance profiler, we observe the Cilk Plus version is not significantly memory bound, as shown in Fig. 4.8. We have contacted the Intel compiler team to obtain a deeper understanding of the same.

• Applying our compiler transformation using dHPF, the generated semi-stencil-transformed smooth kernel outperforms the original Fortran version by 7.1%. We attribute the performance differences to the following factors:

  – Compared to the original C version, the original Fortran version is significantly faster (0.615s vs 0.497s). This indicates that the Fortran compiler does a better job optimizing the computation than the C compiler. Both the semi-stencil transformed smooth kernel and the input kernel spend

![Figure 4.7](image-url): Screen-shot from Intel VTune performance profiler showing the performance analysis of the semi-stencil-transformed smooth kernel.
more than 22% of clock ticks waiting for core resources (back-end core-bound) and more than 40% of clock ticks waiting for the Super Queue to issue requests to L2 cache as shown in Fig. 4.9, Fig. 4.10. This shows that the Fortran compiler is optimizing the computation differently than the C compiler, and that any subsequent optimization, e.g., semi-stencil transformation, has less scope for substantial improvement.

- Currently, we do not generate aligned information for arrays in the generated code. As a result, we observe that the time taken for a single iteration of smooth kernel by the semi-stencil-transformed Fortran version is higher than that semi-stencil-transformed C version. We intend to generate alignment information for arrays in the future.
Figure 4.9 : Screen-shot from Intel VTune performance profiler showing the performance analysis of the original smooth kernel in Fortran.

Figure 4.10 : Screen-shot from Intel VTune performance profiler showing the performance analysis of the semi-stencil-transformed smooth kernel in Fortran.
4.5 Handling Control Flow Dependences

The presence of control flow divergence hinders vectorization. Reasoning about the safety, feasibility, and profitability of vectorization on multiple code paths encountered due to control flow divergence is difficult. Several techniques have been presented in the past to address the challenge of divergent control flow. They can be classified into techniques based on speculation [88], if-conversion, and iteration space reduction [89]. Speculation requires significant changes to the code structure, e.g., addition of compensation code that needs to be executed when the speculation fails. If-conversion requires the support for predicated execution. In this section, we develop a compiler transformation that aims to generate convex and compact loops that are devoid of control flow divergence. By employing this transformation, we achieve a 36% reduction in time for the Dilate kernel in Leukocyte for reasonable input sizes on an Intel Ivy Bridge processor.

The Leukocyte application from the Rodinia Benchmark Suite detects and tracks the movement of white blood cells in the blood stream [9]. The first step in the Leukocyte code is to detect white blood cells. The detection process includes three steps: compute the Gradient Inverse Coefficient of Variation (GICOV) score across a range of ellipses for each pixel in an image of the blood stream, dilate a matrix

```c
for(el_j = 0; el_j < strel_n; el_j++) {
    y = col - el_center_j + el_j;
    if( (y >= 0) && (y < img_n) ) {
        if(c_strel[el_i*strel_n + el_j] != 0 && t_img[(x*img_n)+y] > max)
            max = t_img[(x * img_n) + y];
    }
}
```

Figure 4.11: A snippet of code from the Dilate kernel in the Leukocyte application.
of GICOV scores to find local maxima, and apply a contour algorithm to determine the shape of the cell. From a performance perspective, the detection process takes 10% – 15% of the total time taken for a single run of Leukocyte, e.g., detection and tracking of cells in a single frame, where frame size 250 × 600, on an Intel IvyBridge core. The dilation kernel takes 28.8% of the total time for detection.

In this section, we focus our efforts on improving the performance of the second step, namely, dilate a matrix of GICOV scores. Fig. 4.11 presents a snippet of code from the Dilate kernel. In this snippet, y is an auxiliary induction variable, i.e., a value derived from the loop induction variable, loop invariant variables, and constants. y is derived from induction variable, el_j, and loop invariants, col, el_center. The conditional involving y affects the vector performance negatively because

- every iteration of the loop el_j is executed even though the conditional is false in some of them, and
- the conditional has a vector cost.

Since the conditional involving y is affine in nature, if we can account for its effect at compile time, then we will be able overcome these negative effects. In this section, we propose a technique to eliminate affine if conditionals involving auxiliary induction variables.

4.5.1 If Elimination: A Compiler Transformation

Our proposed compiler transformation statically accounts for the constraints enforced from affine if-conditionals involving auxiliary induction variables. We do so by intersecting these affine if-conditionals with the iteration span of the loop and generating
Optimized DilateKernel

• Aim: Produce loop bounds that reduce control overhead

• $\Omega$ — generate code from the above polyhedral representation

new loop bounds. Traditional vectorization techniques are then applied to the resulting code snippets. We propose the following steps:

- Identify auxiliary induction variables in the loop.

- Identify affine conditionals involving these auxiliary induction variables.

- Create an integer relation that is a conjunction of the affine conditionals and the loop span of the induction variable.

- Employ a polyhedral solver, e.g., $\Omega$, to solve the relations to obtain the new loop bounds for the induction variable.

The integer relation developed for Fig. 4.11 is present in Fig. 4.12. In Fig. 4.12, the $g_1$ relation represents the known quantities. “codegen” is used to solve the relation and generate a C-loop nest that respects the relation.

4.5.2 Experiments

We perform experiments on a single core of an Intel Ivy-Bridge processor cluster, NOTS, and on a single node of an IBM Power 755 cluster, BlueBioU, at Rice University’s Center for Research Computing. We executed both the original and improved versions of Dilate kernel. We make the following observations:
1. By eliminating the if-condition based on $y$, we are reducing the iteration span of the $el_j$ loop. We are also reducing vector operations that were used to perform the conditional. This is reflected in the decreased count of instances of different vector operations used in the loop as reported by Intel SDE. These numbers are presented in Fig. 4.13. We see a reduction in $vmaskmovps$, which was needed to selectively move contents of $c\_strel$ array from memory based on the success of the conditional. Hoisting the if-condition eliminates the need for $vpand$ and $vpcmpeqd$ that were used to execute the condition.

2. As shown in Fig. 4.14, we achieve a range of 24%-55% reduction in time, i.e., performance improvement across a range of image sizes. The improvement is observed on all platforms and across different compilers. These results indicate that each of the state-of-the-art compilers could benefit from hoisting guards involving affine expressions of induction variables or auxiliary induction vari-

### Figure 4.13: Count of vector instructions as reported by Intel Software Development Emulator (SDE) for the original and if-eliminated versions of Dilate kernel.

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Optimized</th>
<th>Optimized/Original</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLDMXCSR</td>
<td>1</td>
<td>1</td>
<td>1.000</td>
</tr>
<tr>
<td>VMASKMOVPS</td>
<td>9768000</td>
<td>4510000</td>
<td>0.462</td>
</tr>
<tr>
<td>VMOVD</td>
<td>3168483</td>
<td>1056481</td>
<td>0.333</td>
</tr>
<tr>
<td>VMOVQDQU</td>
<td>10800</td>
<td>10800</td>
<td>1.000</td>
</tr>
<tr>
<td>VPADD</td>
<td>4887600</td>
<td>3600</td>
<td>0.001</td>
</tr>
<tr>
<td>VPAND</td>
<td>9768000</td>
<td>-</td>
<td>--------</td>
</tr>
<tr>
<td>VPBLENDQVB</td>
<td>4884000</td>
<td>4510000</td>
<td>0.923</td>
</tr>
<tr>
<td>VPCMPQEOQD</td>
<td>4884001</td>
<td>4510001</td>
<td>0.923</td>
</tr>
<tr>
<td>VPCMPGTD</td>
<td>9768000</td>
<td>-</td>
<td>--------</td>
</tr>
<tr>
<td>VPMSAOF</td>
<td>5940000</td>
<td>5566000</td>
<td>0.937</td>
</tr>
<tr>
<td>VPSHUFQD</td>
<td>1584003</td>
<td>1584001</td>
<td>1.000</td>
</tr>
<tr>
<td>Image Size in Pixels</td>
<td>ICC</td>
<td>GCC</td>
<td>XLC</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>120 × 120</td>
<td>40%</td>
<td>24%</td>
<td>55%</td>
</tr>
<tr>
<td>360 × 360</td>
<td>36%</td>
<td>42%</td>
<td>53%</td>
</tr>
<tr>
<td>1024 × 1024</td>
<td>33%</td>
<td>49%</td>
<td>53%</td>
</tr>
</tbody>
</table>

Figure 4.14: Performance of the if-eliminated version of Dilate kernel across different input sizes and different compiler/hardware platforms. The percentage reduction in time specified in each of the boxes above is calculated as follows: \( \frac{T_{\text{dilate-original}} - T_{\text{dilate-if-elim}}}{T_{\text{dilate-original}}} \).

```plaintext
1 do j=L_j, U_j
2   do i=L_i, U_i
3     A(i,j) = A(i,j) + A(i-1,j) + A(i,j-1) + A(i-1,j-1)
4   enddo
5 enddo
```

Figure 4.15: Pseudo-code that depicts a two-dimensional stencil that is used in an Image Integral kernel.

4.6 Future Work: Enhancing Vector Parallelism in In-place Stencils

Consider the 2D stencil shown in Fig. 4.15, which is used in an Image Integral kernel. This 2D stencil is an in-place stencil, where the input to the stencil is the same array as the one used to capture the result of the stencil. As a result there exist
Figure 4.16: Reference pattern for the computation of a single array element in a 2D in-place stencil that is used in an Image Integral kernel.

three flow dependencies, as shown in Fig. 4.16, value of \( A(i,j) \) depends on the newly computed values of \( A(i-1,j) \), \( A(i,j-1) \), and \( A(i-1,j-1) \). These flow dependences hinder vectorization.

4.6.1 Enabling Vectorization in an In-place 2D stencil

Metzger [90] approaches the optimization of such operations by recognizing and replacing snippets of input code with idioms from a catalogue; these idioms have well known high-performance mappings. However, such approaches are limited in how accurately a catalogue of idioms captures the syntactic and semantics structure of the idioms; too accurate and an opportunity might be missed, and too loose might lead to an incorrect application of an idiom.

Recently, techniques have been introduced to address partial stencil operations
**Integral image calculation** $\equiv$ Scan(L→R) + Scan(T→B)

**Inter-iteration Dependencies**

\[
a(i)(j) = a(i)(j) + a(i)(j-1)
\]

\[
a(i)(j) = a(i)(j) + a(i-1)(j)
\]

Figure 4.17: Factorization of a 2D in-place stencil, which is used in an Image Integral kernel, to two scan operations. Scan operations are easily vectorizable compared to the original 2D in-place stencil operation.

by converting them into multiple scan operations [91]. Vectorization of scan operations has been widely studied, and well-known scan kernels exist for different vector hardware. By combining these techniques, one could factor the stencil in Image Integral kernel into semantically equivalent simpler scan kernels. This is illustrated in Fig. 4.17, where we convert the 2D stencil into a pair of scans, a top-to-bottom scan followed by a left-to-right scan. When we performed this transformation manually, we achieved an 80% reduction in time for the Image Integral kernel for reasonable input sizes on an Intel Ivy Bridge processor. In future, we plan to focus on implementing this technique. We also plan to explore the generality of such a technique to other multidimensional in-place stencils.

---

*For mapping a scan to its equivalent high-performance mapping, we can pursue an approach similar to Metzger [90].*
Chapter 5

Distributed Memory Code Generation

The computing power of a processor has increased a thousand fold over the last 20 years [4]. As processing power continues to increase as predicted by “Moore’s law,” communication is increasingly becoming the principal power and performance bottleneck. The “ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems” [92] opines that it is easier to solve the power-consumption problem for intra-node computation than it is to solve the problem of inter-processor communication. The problem of minimizing inter-processor communication is referred to as horizontal communication-avoidance [40].

To address the problem of horizontal communication avoidance, the BeBOP group at UC Berkeley developed the .5D class of Communication-Avoiding (.5D CA) algorithms [3, 93, 94]. These .5D CA algorithms are reformulations of existing algorithms that reduce inter-processor communication and provide strong scaling in both time and energy on large numbers of processors by exploiting more than the minimal amount of on-node memory required.

Automatically deriving .5D CA algorithms from sequential code is well beyond the scope of compilers today. As a first step to spur the adoption of these algorithms and help automate the development of CA libraries, we are developing Maunam—a compiler based on the dHPF compiler infrastructure [1] that employs polyhedral methods for program analysis and transformation to generate efficient parallel code for .5D CA algorithms.
From a high-level, global view sketch of a .5D CA algorithm expressed using symbolic data sizes and numbers of processors, Maunam mechanically generates a parallel implementation complete with partitioned computation and necessary communication. To enable a succinct global-view expression of the data movement inside .5D CA algorithms, Maunam supports array transformational intrinsics including CSHIFT, SPREAD, and a novel TILT operation. Sum reductions are supported through the SUM intrinsic. Maunam also supports analysis and code generation for wrap-around communication expressed as an element-wise assignment of an array indexed by one or more subscripts that contain modulo operations. Fig. 5.1 shows a snippet from the 2.5D matrix multiplication (2.5D MM) [3] written in a global view style that illustrates skewing an input matrix in two ways: TILT intrinsics or element-wise assignment.

Efficient parallel code generation for such global view specifications requires sophisticated analysis and code generation. Maunam extends the polyhedral analysis and optimizations in the dHPF compiler infrastructure to support code generation for .5D CA algorithms. From HPF directives that specify a data partitioning, Maunam automatically derives a computation partitioning that co-locates computation with data it manipulates. Maunam performs communication analysis and code generation by manipulating polyhedra described using constraints expressed in Presburger arithmetic. Maunam employs the Omega Library [47] to manipulate and generate code from these polyhedra. Maunam then inserts MPI [95] point-to-point and collective communication as needed.

To our knowledge, Maunam is the first compiler to generate distributed-memory parallel code for .5D CA algorithms from a global view specification. This chapter describes novel compiler technology that
Figure 5.1: Workflow of Maunam. A snippet of Solomonik and Demmel’s 2.5D Matrix Multiplication [3] representing communication performed in the initial skew step is specified. Array ‘A’ represents one of the input matrices. Maunam reasons about communication partners when they are expressed either through TILT operations or through element-wise assignments with subscripts containing modulo operations. Maunam generates Fortran+MPI code.
analyses and optimizes SPMD systolic computations expressed in a global-view specification,

produces efficient point-to-point communication code for TILT and CSHIFT operations using a novel method for polyhedral analysis and optimization of wrap-around communication, and

analyses and generates collective communication for broadcasts and reductions along some or all dimensions of a processor grid.

Maunam employs this compiler technology to generate high performance Fortran+MPI code. Maunam-generated code for 2.5D MM running on 4096 cores of a Cray XC30 supercomputer at NERSC achieves 59 TFlops/s, which is 76% of the machine peak and 91% of the performance of a hand-coded version of 2.5D MM.

The rest of the chapter is organized as follows. Section 5.1 briefly describes the 2.5D MM algorithm. Section 5.2 describes our high-level, global view representation of 2.5D MM. Section 5.3 describes the challenges Maunam faces handling 5D CA algorithms. Section 5.4 describes how Maunam affinizes modulo operations found in 5D CA algorithms. Section 5.5 describes how Maunam identifies and generates reductions on subsets of processors. Section 5.6 describes the utility of Maunam to generate code for other 5D CA algorithms. Section 5.7 discusses the performance achieved by Maunam-generated Fortran+MPI code. Section 5.8 presents a summary of this work.

5.1 2.5D Matrix Multiplication

The 2.5D Matrix Multiplication (2.5D MM) algorithm devised by Solomonik and Demmel [3] describes a family of distributed-memory matrix multiplications ranging
from Cannon’s algorithm [96] to 3D matrix multiplication [97]. Solomonik and Demmel combine the skew and systolic phases of Cannon’s matrix multiplication with the broadcast and reduction phases of 3D matrix multiplication to develop an innovative 2.5D matrix multiplication algorithm. This algorithm exploits larger on-node memories to reduce inter-processor communication bandwidth and latency along the critical path.

**Algorithmic Sketch of 2.5D MM**  
**Input:** $A, B$ matrices of rank $n \times n$ are distributed on the front plane of a virtual 3D processor grid of size $\sqrt{p/c} \times \sqrt{p/c} \times c$, where $p$ represents the total number of processors and $c$ represents the depth of the processor grid.  
**Output:** $R = A \times B$ is a matrix of rank $n \times n$ distributed on the front plane of processors.

As shown in Fig. 5.2, the 2.5D MM algorithm employs a logical 3D processor grid of size $\sqrt{p/c} \times \sqrt{p/c} \times c$. $n/\sqrt{p/c} \times n/\sqrt{p/c}$ tiles of $A$ and $B$ are assigned to each of the processors in the front ($k = 1$) plane. Each processor in the grid holds a tile of array $R$ of size $n/\sqrt{p/c} \times n/\sqrt{p/c}$. At any point in time, the tiles of $R$ on a plane of processors along the $k^{th}$ dimension contain a partial result of the matrix multiplication. The algorithm proceeds as follows:

- Each $(i, j, 1)$ processor broadcasts its tiles of $A$ and $B$ to the processors $(i, j, 2..c-1)$.

- Tiles of the $A$ and $B$ matrices are skewed along the $j$ and $i$ dimensions of the processor grid, respectively. Details of the skewing are shown in the alignment step of Listing 5.1.
Figure 5.2: 2.5D Matrix Multiplication. $p$ represents the total number of processors, $c$ represents the depth of the virtual grid of processors.

- The processors perform multiple steps of a systolic computation. In each step, each processor participates in a circular shift of A and B tiles along its row and column of the processor grid respectively, multiplies the A and B tiles it possesses, and adds the result to its local tile of R.

- A sum reduction of the R tiles along the $k$ dimension yields the final matrix product R, distributed in tiles across the $k = 1$ plane.

Solomonik and Demmel show that their 2.5D MM algorithm reduces memory bandwidth by a factor of $\sqrt{c}$ and latency by a factor of $\sqrt{c^3}$ along the critical path. In this chapter, we use 2.5D MM as a representative of the .5D class of algorithms to showcase the capabilities of Maunam.
5.2 Global-view 2.5D MM

In this section, we describe a global view sketch of 2.5D MM that we provide as input to Maunam. Solomonik and Demmel [3] describe 2.5D MM in a Single Program Multiple Data (SPMD) form using message passing. In this model, data tiles and the processors owning and operating on them are explicit. A natural way to sketch 2.5D MM in global view form is by using arrays that have both data and processor dimensions. Listing 5.1 presents a global view sketch of 2.5D MM. The algorithm relies on several intrinsic functions. The Fortran \texttt{MATMUL} intrinsic multiplies two input matrices and returns the result; the \texttt{CSHIFT}, \texttt{SPREAD}, \texttt{SUM}, and \texttt{TILT} data movement intrinsics are defined in the Appendix.

2.5D MM operates on a 3D grid of processors. The HPF \texttt{processors} directive specifies the logical processor grid for the algorithm. The value of $\sqrt{\frac{p}{c}}$ need not be a compile time constant since Maunam accepts symbolic data and processor sizes. The \texttt{align} and \texttt{distribute} directives describe the data layout. The “*” subscript in the first three array dimensions in the \texttt{align} directives replicates these dimensions across all processors. The align directive specifies a 1:1 mapping of elements in the last three dimensions of each array onto elements of the virtual processor grid \texttt{t} defined by the \texttt{template} directive. Since the template \texttt{t} and the processor grid \texttt{proc} have identical extents in all dimensions, the 3D block distribution specified in the \texttt{distribute} directive amounts to a 1:1 mapping between elements of the virtual processor grid and the physical processor grid.

Our global view sketch employs two planes of the A and B arrays, indexed by \texttt{cur} and \texttt{next} in the third dimension. The \texttt{cur} plane is used as the source of \texttt{CSHIFT} communication and the \texttt{next} plane is used as the destination. Employing \texttt{cur} and \texttt{next} planes enables communicated values to be received directly into the data arrays,
Listing 5.1: A sketch of the global view 2.5D MM code compiled by Maunam. The program is written in HPF extended with the TILT intrinsic.

```fortran
double precision :: B(n,n,2,√p/c,√p/c,c)

double precision :: A(n,n,2,√p/c,√p/c,c)

double precision :: R(n,n,√p/c,√p/c,c)

! HPFS processors proc(√p/c,√p/c,c)

! HPFS template tmpl(√p/c,√p/c,c)

! HPFS align B(*,*,*,:,:) with tmpl(:,:)

! HPFS align A(*,*,*,:,:) with tmpl(:,:)

! HPFS align R(*,*,:, :) with tmpl(:,:)

! HPFS distribute tmpl(block,block,block) onto proc

!!! Initialize A,B,R

!!! Broadcast of A,B

A(:,cur,:, :) = SPREAD (A(:,cur,:,1),6,c)

B(:,cur,:, :) = SPREAD (B(:,cur,:,1),6,c)

!!! Alignment of A,B

A(:,next,:, :) = TILT(TILT(A(:,cur,:, :) ),5,6,√p/c^3),5,4,-1)

B(:,next,:, :) = TILT(TILT(B(:,cur,:, :) ),4,6,√p/c^3),4,5,-1)

cur = next

next = 3-cur

R(:, :, :) = MATMUL(A(:,cur,:, :) ,B(:,cur,:, :))

!!! Systolic Shift + Multiply

do l = 1, √p/c^3-1

A(:,next,:, :) = CSHIFT(A(:,cur,:, :) ,-1,5) ! rotate tiles left along rows

B(:,next,:, :) = CSHIFT(B(:,cur,:, :) ,-1,4) ! rotate tiles up along columns

cur = next

next = 3-curt

R(:, :, :) = R(:, :, :) + MATMUL(A(:,cur,:, :) ,B(:,cur,:, :))

endo
doo

!!! Reduce

R(:, :, 1) = SUM(R(:, :, :) , 5)
```
avoiding copies from communication buffers.

In their SPMD description of 2.5D MM, Demmel and Solomonik describe the circularly skewed alignment and circular shift communications using modulo operations on processor indices. Our equivalent global view sketch expresses these communications through TILT and CSHIFT operations. Alternatively, communication can be expressed through element-wise assignments indexed with subscripts containing modulo operations, analogous to those that appear in the SPMD 2.5D MM.

The algorithm specifies that each processor multiplies its local A and B matrix tiles. The sketch in Listing 5.1 uses Fortran’s MATMUL intrinsic to do so.

5.2.1 Skew Communication

The TILT operation, defined in Appendix A.1.4, enables a succinct global-view expression of the skew communication present in 2.5D MM. We use it to skew a multi-dimensional matrix along a coordinate axis. In this section, we describe the handling of TILT operations in detail.

The global view 2.5D MM shown in Listing 5.1 uses the following TILT expression to skew the A matrix on a 3D grid of processors:

\[ A(\cdot,\cdot,next,\cdot,\cdot,\cdot) = \text{TILT}(\text{TILT}(A(\cdot,\cdot,cur,\cdot,\cdot,\cdot),5,6,\sqrt{\frac{p}{c^3}}),\ 5,\ 4,\ -1) \]

The inner TILT specifies that the 5\textsuperscript{th} dimension of array A should be translated with respect to the 6\textsuperscript{th} dimension by \( \sqrt{\frac{p}{c^3}} \). The outer TILT specifies that the 5\textsuperscript{th} dimension of the array resulting from the first TILT should be translated with respect to the 4\textsuperscript{th} dimension by \(-1\). Fig. 5.3 illustrates the effect of each TILT.

Maunam handles TILT operations by converting them into an element-wise copy loop that expresses the wrap-around data movement using subscripts containing modulo operations. Since TILT operations are an injective mapping between array po-
Figure 5.3: Successive TILTs of data planes along the $j$ axis with respect to $k$ and $i$ axes achieves the required skew in 2.5D MM algorithm. The top half of the figure shows the TILT of a $j$ plane with respect to the $k$ axis. The bottom half of the figure shows the TILT of a $j$ plane with respect to the $i$ axis.
sitions and are circular in nature (elements shifted out one end of a dimension are shifted in at the other end of the dimension), converting these operations into a copy loop is straightforward. This is illustrated in Fig. 5.4. In the illustration, the 5th dimension of array A is TILT’ed with respect to the 4th dimension which introduces two variables $i, j$ that represent these dimensions. The skewed value of $j$ is equal to $\text{mod}(j + (-1) \ast i, \sqrt{p/c})$. TILT is a circular skew operation and hence requires a modulo operation. The array declaration for A provides the span of the 5th dimension, i.e., $1 : \sqrt{p/c}$. Handling for CSHIFT is similar.

5.3 Challenges

In this section, we present two primary challenges addressed by Maunam.

5.3.1 Modulo Operations

The shift communication in the systolic phase of 2.5D MM of the input matrix, A, is present below:

$$s = \text{mod}(j, \sqrt{p/c}) + 1$$
\begin{verbatim}
do k= 1,c
  do i= 1,√(p/c)
    do j= 1,√(p/c)
      A(:,next,1,modulo(j,√(p/c))+1,k) = A(:,1,j,k)
    enddo
  enddo
enddo
\end{verbatim}

Figure 5.5: Global view expression of the shift communication in 2.5D MM expressed as element-wise assignments.

\[ P_{i,j,k} \text{ sends } A_{ir} \text{ to } P_{i,s,k} \]

Fig. 5.5 shows a global view expression of the circular shift communication using element-wise assignment.

Consider the problem of communication code-generation for this shift communication. Since communication partners share the same \( i \) and \( k \) index values, the primary compilation challenge for generating communication is to reason about the relationships between values of \( j \) and \( s \). A compiler that generates two-sided communication must determine every iteration \( j \) in which a processor functions as a sender and every iteration \( s \) in which the processor functions as a receiver. A compiler that generates one-sided communication faces a different but equally difficult problem: it must determine every iteration \( j \) in which a processor must perform a put and whether the processor must wait for one or more remote put operations to complete before it may perform its put. Unlike send and receive, which perform both data movement and synchronization, get and put only move data. When a put must follow receipt of the same data item, the compiler must introduce synchronization to wait for the put to complete to preserve the data dependence. A barrier synchronization at the end of
Regardless of whether generating one-sided or two-sided communication, a compiler generating a circular shift communication must reason about the modulo operation that maps $j$ to $s$, and the inverse relation mapping $s$ to $j$. Modulo operations are non-affine because of their periodicity and are difficult to represent precisely in polyhedral analysis frameworks. While the Omega Library does not support modulo operations directly, one may introduce them into constraints using uninterpreted function symbols [34] or by using existential quantifiers [81]. Such solutions, which circumvent the modeling of the modulo operation, generate inefficient communication because in the absence of a relation tying a value of $j$ to $s$ and $s$ to $j$ at compile time, communication partners cannot be determined precisely. Without static knowledge of the pairing between communication partners, a compiler must generate conservative code in which a processor loops through all iterations of $j$ to determine the iterations in which it must function as a sender, receiver, or both. The parallel performance of such loops scales poorly as the number of processors increases. As described in Section 5.4, Maunam addresses this challenge by affinizing modulo operations.

5.3.2 Explicit Distributed Dimensions

The global view expression of 2.5D MM in Listing 5.1 reduces the span of distributed dimensions to singletons on every processor. Because the extents of corresponding template and processor dimensions are equal, the block size in each HPF distribute directive reduces to 1. This mapping makes it easy to express collective communication operations such as reductions and broadcast on a subset of processor dimensions. Section 5.5 sketches the approach used by Maunam to detect and generate MPI collective code for reductions on subsets of processors.
5.4 Affinizing Modulo Operations

Maunam represents the data movement implied by TILT and CSHIFT operations along distributed dimensions (and nested compositions thereof) as an element-wise assignment of a distributed global-view array. Fig. 5.5 shows an element-wise assignment that is equivalent to a single CSHIFT. In this global view form, each unique tuple of indices in the distributed dimensions represents a processor in the processor grid.* In a TILT and CSHIFT along distributed dimensions and their nested compositions, every processor sends and receives data once when each distributed dimension reduces to a singleton on each processor. To avoid generating code that loops over the entire range of a processor dimension and checking whether a processor is supposed to send a message, Maunam synthesizes constraints in Presburger arithmetic that enable our polyhedral code generator to identify symbolically at compile time the single iteration in which a processor will perform a send. (Handling for receives is similar.) To do so, Maunam must represent modulo operations as constraints expressed using Presburger arithmetic. To enable that, Maunam applies a novel strategy to affinize modulo operations that arise from TILT and CSHIFT.

In this section, we, first, describe how Maunam affinizes a simple modulo operation. We then describe how Maunam affinizes a complex modulo operation that results from an arbitrary nesting of CSHIFT and TILT operations.

*With the declaration for A and the HPF directives shown in Listing 5.1, the span of each distributed dimension indexed in the global view expression shown in Fig. 5.5 reduces to a singleton on each processor.
\[ -2m \leq j \leq -m - 1 \implies s = j + 2m + 1 \] and
\[ -m \leq j \leq -1 \implies s = j + m + 1 \] and
\[ 0 \leq j < m \implies s = j + 1 \] and
\[ m \leq j < 2m \implies s = j - m + 1 \] and
\[ 2m \leq j < 3m \implies s = j - 2m + 1 \] and

... Figure 5.6 : Solving for \( s \) given \( s = \text{mod}(j, m) + 1 \).

5.4.1 Simple Modulo Operations

Consider the modulo operation \( s = \text{mod}(j, m) + 1 \). We refer to such a modulo operation as a simple modulo operation since it has only one linear term in its first operand. Based on the definition of a modulo operation, the value of \( \text{mod}(j, m) \) is constrained to the closed interval \([0, m - 1]\). The value of \( s \) is therefore constrained to the interval \([1, m]\). As illustrated in Fig. 5.6, for an unconstrained value of \( j \), there exist an infinite number of implications to convert \( j \) to \( s \) based on the interval in which \( j \) falls. To accurately model the relationship between \( s \) and \( j \) in a polyhedral-framework for unconstrained values of \( j \), we would need all of these implications.

However, when using modulo operations in distributed dimensions to represent wrap-around communication in global view algorithms such as the one in Listing 5.1, \( s \) and \( j \) are used to index the same distributed dimension. Hence, the modulus, \( m \), and the upper bound of the span of the distributed dimension, \( j \), are the same. Since \( j \in [1, m] \) and the coefficient of \( j \) is 1, only two implications are needed to relate the possible values of \( s \) and \( j \): \( 0 \leq j < m \implies s = j + 1 \) and \( m \leq j < 2m \implies s = j - m + 1 \).
5.4.2 Complex Modulo Operations

Consider an assignment expression that represents an arbitrarily nested set of TILT and CSHIFT operations along distributed dimensions. While such a nesting could represent a communication that skews an array along multiple distributed dimensions, we can analyze data movement along each distributed dimension independently. Without loss of generality, we consider data movement along distributed dimension $i$. We generalize the modulo operation appearing in the left hand side of the resulting array assignment operation, similar to Fig. 5.5, to be of the following form:

$$\text{mod}(i \pm t_1 \pm ... \pm t_n, m) \mid t_k = S_k j_k \lor t_k = S_k, k \in [1, n]$$

where each $j_k$ is an induction variable representing a different distributed dimension, and $S_k$ is an integer constant or symbolic variable. Determining the number of implications needed to model this complex modulo operation requires determining the upper and lower bound of the expression $i \pm t_1 \pm t_2 \pm ... \pm t_n$. This problem is hard at compile time because of the presence of symbolics. Hence, Maunam employs an alternate strategy to model these complex modulo operations based on the following property of modulo operations: $\text{mod}(a \pm b, m) = \text{mod}(\text{mod}(a, m) \pm \text{mod}(b, m), m)$.

Maunam repeatedly applies this rewrite rule until each modulo operation has, as its first operand, either $i$ or $t_k, k \in [1, n]$ or it has two modulo operations combined with a $+$ or $-$. Here, we show the first two of $n$ rewriting steps:

$$\text{mod}(i + t_1 \pm t_2 \pm ... \pm t_n, m)$$

$$= \text{mod}(\text{mod}(i, m) \pm \text{mod}(t_1 \pm ... \pm t_n, m), m)$$

$$= \text{mod}(\text{mod}(i, m) \pm \text{mod}(\text{mod}(t_1, m) \pm \text{mod}(t_2 \pm ... \pm t_n, m), m)$$

We reduce such modulo expressions into a finite set of implications using three rules.
First, each modulo operation with only a constant or symbolic variable as the first operand can take values from \([0 : m - 1]\). We can model this term with a formula using an existential quantifier: \(\exists_\alpha \alpha \in [0 : m - 1]\).

Second, each modulo operation with \(S_k j_k\) as the first operand is handled as follows:

**Case 1: \(S_k\) is a constant**  \(S_k + 1\) implications are sufficient to model this modulo operation. In particular, if \(S_k\) is positive then \(S_k + 1\) implications are sufficient that are based on ranges \([0, m - 1], [m, 2m - 1], ..., [S_k m, (S_k + 1)m - 1]\). If \(S_k\) is negative, then \(S_k\) implications are sufficient that are based on ranges from \([-S_k m, -(S_k - 1)m - 1], [-2m, -m - 1], ..., [-m, -1]\).

**Case 2: \(S_k\) is a symbolic variable**  In general, Maunam cannot reason about the number of implications needed to model a modulo operation containing a term \(S_k j_k\) where \(S_k\) is a symbolic variable. In general, Maunam may need to generate a conservative loop to identify at runtime the iterations in which a processor functions as a sender or receiver or both. However, in TILTs, CSHIFTs and their compositions, Maunam replaces a modulo operation containing the term \(S_k j_k\), where \(S_k\) is a symbolic variable, with a formula using an existential quantifier: \(\exists_\alpha \alpha \in [0 : m - 1]\). Doing so enables the Omega polyhedral code generator to understand that these relations represent one-to-one mappings and generate efficient code.

Finally, modulo operations with a first operand consisting of two modulo operations combined with a + or − can take values between \([-m + 1 : m - 1]\) or \([0 : 2m - 2]\) depending upon whether the modulo operations are combined with a + or −. As described in section 5.4.1, only two implications are needed to model such modulo operations.
5.4.3 Affinization

After Maunam reduces a modulo operation to a finite list of implications, it converts each implication to a disjunction of Presburger formulae as shown below:

\[ p \implies q \equiv \neg(p) \lor q \]

Below is a relation representing the values of \( s \) when \( s = \text{mod}(j, m) + 1 \). In the relation, the result of the modulo operation is represented by an existentially quantified variable \( \alpha \).

\[ \{ [s] : \exists (\alpha : s = \alpha + 1 \land (\neg(0 \leq j < m) \lor (\alpha = j)) \land (\neg(m \leq j < 2m) \lor (\alpha = j - m)) \} \]

5.4.4 Post-Affinization

After the modulo operation is affinized and incorporated into constraints expressed using Presburger formulae, the communication code generation follows the steps detailed in Mellor-Crummey et al. [1]. A brief summary is as follows: Maunam identifies statements that need communication and determines where these communications should be performed. It then identifies the sets of processors and data involved in each communication and generates the appropriate message-passing code to perform the communication. The analysis and code generation steps are as follows:

- Maunam determines the local (owned) and non-local (non-owned) sections of an array referenced in an executable statement. Maunam must introduce communication for non-local sections accessed by each processor. To determine the local and non-local sections, Maunam
  - uses an array’s affinized subscript expressions to build a relation that expresses the array elements accessed by the loop nest on each processor,
– employs the align and distribute directives to build a relation that identifies the elements of the array owned by each processor, and
– composes the relations to determine the local and non-local elements of the array referenced.

• Maunam creates computation partitionings that may read or write non-local data. Each processor must communicate data it owns to others that read it. Similarly, each processor that writes data must send new values back to their owners. Maunam uses a combination of dependence and data flow analysis to determine the placement for both types of communications.

• Maunam determines the identities of communication partners by composing a relation representing the non-local data elements referenced with the inverse of the relation that specifies the data elements owned by each processor.

• Maunam generates MPI_ISend, MPI_IRecv and MPI_Wait primitives for point-to-point communication.

5.4.5 Semantics of CSHIFT

As described in Appendix A.1, the CSHIFT is a transformational fortran intrinsic operation. A fortran intrinsic operation guarantees the evaluation of all arguments prior to invoking the function. Hence, irrespective of whether the same array is provided as an input argument to CSHIFT and/or as the destination of the CSHIFT operation, the CSHIFT communication involves sending each processor’s data to a communication partner before overwriting it with incoming data. Consider the shift communication presented in Listing 5.1. Here, the send and receive occur through different buffers,
i.e., there does not exist a data dependence between the send and receive communications. Hence, its straightforward to realize the \texttt{CSHIFT} communication. However, if the same array was used as input to the \texttt{CSHIFT} operation and as the destination, additional buffer management is necessary such as storage expansion, array expansion or strip mining the communication. We pursue array expansion. The details on the safety and opportunity for array expansion will be discussed in the context of overlapping communication and computation in the subsequent chapter.

5.5 Reduction and Broadcast along Distributed Dimensions

Consider the reduction in the final step of 2.5D MM. In this step, the final result of the matrix multiplication is accumulated in the $R$ tiles held by the front plane of processors by gathering the partial results for $R$ from the back planes of processors. There are $\sqrt{p/c} \times \sqrt{p/c}$ independent reductions that occur along independent lines in the $k$ dimension to obtain this result. Similarly, the broadcasts of $A$ and $B$ in the first step of 2.5D MM occur along independent lines in the $k$ dimension as well. By extending reduction recognition techniques developed by Lu and Mellor-Crummey [33], Maunam identifies reductions and broadcasts along some or all dimensions of the processor grid, creates the necessary processor subsets and performs collectives over these processor subsets.

Our global view sketch of 2.5D MM presented in Listing 5.1 reduces the span of the distributed dimensions (3–5) of $R$ to singletons on every processor. A unique tuple of indices in these distributed dimensions represents a single processor in the processor grid. By extension, a specific distributed dimension represents a specific processor grid dimension. Access patterns along these distributed dimensions help determine the processor grid dimensions along which a collective occurs. Below, we discuss
these access patterns in the context of reduction and broadcast collectives. In these
descriptions, only the \( n \) distributed dimensions of arrays \( R1 \) and \( R2 \) are depicted, \( C_t \)
refers to a scalar variable or integer constant, and \( \ominus \) represents a reduction operator.

**Listing 5.2: Reduction(s) of \( R2 \)**

\[
\begin{align*}
\text{do } j_1 &= L_1, U_1 \\
\text{do } j_2 &= L_2, U_2 \\
\ldots \\
\text{do } j_n &= L_n, U_n \\
R1(i_1, i_2, \ldots, i_n) &= R1(i_1, i_2, \ldots, i_n) \ominus R2(j_1, j_2, \ldots, j_n) \\
\text{enddo} \\
\ldots \\
\text{enddo}
\end{align*}
\]

\[
\forall t \in [1, n] \ i_t = j_t \lor i_t = C_t \\
\forall t \in [1, n] \ L_t = \text{LBound}(R1, t) \text{ and } U_t = \text{UBound}(R1, t)
\]

**Reduction** The pattern in Listing 5.2 depicts a reduction(s) that performs a \( \ominus \)-
reduction of \( R2 \), and accumulates the result in \( R1 \). The reduction occurs along the
distributed dimensions \( t \mid i_t = C_t \). The root of the reduction, i.e., the processor that
accumulates the result, is derived from the values of \( C_t \) in the dimensions reduced.

**In-place Reduction** In an in-place reduction, the root of the reduction supplys
a single buffer that will both provide an input to the reduction and receive the re-
sult. Maunam has limited support for in-place reduction recognition. Specifically, it
recognizes in-place reduction candidates that arise from a \text{SUM} operation. The \text{SUM}
operation used in Listing 5.1 is recognized as an in-place reduction.
Broadcast

\[ \forall_{j_1, \ldots, j_n} R1(j_1, j_2, \ldots, j_n) = R1(i_1, i_2, \ldots, i_n) \mid \forall_{t \in [1, n]} i_t = j_t \lor i_t = C_t \]

The above pattern depicts a broadcast(s) that occurs along the distributed dimensions \( t \mid i_t = C_t \). An element-wise assignment operation derived from a SPREAD operation\(^\dagger\) represents this pattern as well.

Maunam performs the following three steps to identify and generate collective communication code.

**Identify reduction and broadcast candidates**  Reduction candidates are identified by utilizing a data dependence graph, Static Single Assignment (SSA) form, and the Abstract Syntax Tree (AST) of the program. A reduction is characterized by a loop-carried read-after-write (RAW) flow-dependence, a loop-carried write-after-read (WAR) anti-dependence, and a loop-carried output dependence. In addition, there should be no intermediate consumption of the reduced result, i.e., all the dependences mentioned previously must be self-dependences. This information is captured in the data dependence graphs. SSA form is employed to obtain the definitions of variables used in the reduction operands. The AST is used to inspect the expressions used in the array subscripts of the reduction operands. It is also used to obtain the reaching alignment and distribution information of the operands used in the reduction. Maunam identifies broadcast operations by examining assignments for access patterns presented previously.

**Identify the processor dimensions along which a reduction or broadcast occurs**  Maunam identifies the distributed dimensions of the array operands used in

\(^\dagger\)The appendix describes the SPREAD operation.
the broadcast or reduction candidates through the reaching `align` and `distribute` directives. It then uses the AST to determine the span of the distributed dimensions referenced in these operands by examining the subscript expressions. Based on whether the span is the entire range or a single value, Maunam determines the distributed dimensions along which a reduction or broadcast occurs.

**Generate collective communication** Maunam generates a call to the collective, `MPI_Comm_split [95]`, to create the necessary subsets of processors. The `color` value used in `MPI_Comm_split` is the linearized value of a processor’s indices in the dimensions along which the broadcast/reduction does not occur. Maunam employs primitives such as `MPI_Reduce`, `MPI_Allreduce` and `MPI_Bcast` to perform the collectives. Maunam memoizes the processor subsets created for a collective communication and employs them for subsequent collective communications over these same subsets.

### 5.6 Applicability to Other 5D CA algorithms

In the literature, 5D CA algorithms, including 2.5D MM and 2.5D LU, are expressed in an SPMD style in which data partitions and the processor working on each data partition are explicit. When such algorithms are recast in a global view form, there is a natural explicit 1:1 association between some array dimensions and dimensions of a processor grid. Maunam’s analysis of modulo operations and reductions exploits only general knowledge of bounds on subscripts and array dimensions; it has no special domain-specific knowledge of the 2.5D MM algorithm. For that reason, techniques described in this chapter can be applied to analyze communication in other global view programs, including other 5D CA algorithms.
5.7 Experiments

In this section, we discuss the performance achieved by Maunam-generated Fortran+MPI code for 2.5D MM. We conducted our experiments on a Cray XC30 supercomputer (Edison) at NERSC. Each compute node in the system contains two 2.4 GHz 12-core Intel Ivy Bridge processors [98]. Intel’s Ivy Bridge processor supports “Turbo Boost,” which can increase a core’s frequency when the number of active cores, power, current, and temperature are below their limits. In our experiments, we compute the frequency-adjusted machine peak for the #cores used per compute node. We employ the CrayPat [99] performance analysis toolkit to identify a good mapping of MPI processes to available cores (MPICH_RANK_ORDER), and to measure the time spent in different portions of the code.

Previously, Georganas et al. [100] evaluated the performance of a hand-coded Unified Parallel C (UPC)-based version of 2.5D MM on a Cray XE6. Since their version is not publicly available, we developed a hand-coded 2.5D MM version in C+MPI that employs the same optimizations as Georganas et al. [100]. Our hand-coded 2.5D MM: 1) employs the Intel Math Kernel (MKL) dgemm routine to perform local matrix multiplication, 2) overlaps communication with computation in the systolic phase of the algorithm, and 3) employs MPI_Comm_split to create subsets of processors, and uses MPI_Bcast and MPI_Reduce in the broadcast and reduction phases of 2.5D MM.

Experimental Methodology  We compile both Maunam-generated and hand-coded 2.5D MM with the GNU compiler suite version 4.9.0 and link with the Intel Math Kernel Library’s BLAS routines. We determine the execution time as follows:

- Multiple iterations of the 2.5D MM algorithm are executed in a single run of the experiment. Initialization of input matrices is performed only-once before these
Figure 5.7: Performance comparison of Maumam-generated 2.5D MM with Hand-coded 2.5D MM running on 128 cores performing 2.5D MM on $65536 \times 65536$ input matrices for different cores per socket.

iterations. All cores measure the total time taken for these iterations excluding initialization.

- The average execution time is computed by dividing the sum of time-per-iteration reported by each of cores by the number of cores.

- The number of iterations is 10 for experiments on 128, 256, 512 cores, while the number of iterations is 50 for experiments on $1K, 2K, 4K$ cores$^\dagger$.

Strong scaling experiments on $65,536 \times 65,536$ input matrices. We use 171 compute nodes on a Cray XC30, which enables us to measure scaling up to 4,096 cores. In experiments with fewer cores, e.g., 128, we employ one core per socket whereas in experiments with larger number of cores, e.g., 4,096, we employ 12 cores per socket. As Fig. 5.7 shows, experiments with fewer cores per socket achieve higher

$^\dagger$Small numbers of cores take several minutes to complete one iteration.
Figure 5.8: Performance of Maunam-generated Fortran+MPI vs. hand-coded 2.5D MM code on a Cray XC30 supercomputer (Edison) for $65536 \times 65536$ input matrices. The hand-coded version employs overlap of communication with computation in the systolic phase of the 2.5D MM.

performance. Hence, in our scaling study, we start with a spread configuration of 1 core per socket rather than a packed configuration.

Fig. 5.8 shows strong scaling results on input matrices of size $65,536 \times 65,536$ with double-precision elements. The time taken in seconds is reported in Table 5.1. Maunam-generated 2.5D MM achieves 59 TFlops/s on 4096 cores, which translates to 76% of the machine peak. Our hand-coded version of 2.5D MM achieves 83% of the machine peak. Thus, our Maunam-generated code achieves 91% of the performance of the hand-coded version. We compared the performance of Maunam-generated code with the hand-coded version for input matrices of different sizes. Fig. 5.9 shows that Maunam-generated code achieves $7 - 12\%$ less of the fraction of machine peak than

---

5Strong scaling experiments hold the data size constant while varying the number of cores.
Table 5.1: Time taken in seconds by a single iteration of hand-coded and Maunam-generated 2.5D MM on different number of cores for multiplying matrices of size: $65536 \times 65536$.

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>Hand-coded 2.5D MM</th>
<th>Maunam-generated 2.5D MM</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>195</td>
<td>199</td>
</tr>
<tr>
<td>256</td>
<td>99.6</td>
<td>103</td>
</tr>
<tr>
<td>512</td>
<td>50.5</td>
<td>52.5</td>
</tr>
<tr>
<td>1024</td>
<td>27.5</td>
<td>28.8</td>
</tr>
<tr>
<td>2048</td>
<td>14.5</td>
<td>15.5</td>
</tr>
<tr>
<td>4096</td>
<td>8.67</td>
<td>9.41</td>
</tr>
</tbody>
</table>

Performance measurements of both the Maunam-generated and hand-coded versions using CrayPat showed that Maunam-generated code spends more time waiting for `CSHIFT` communication to complete in the systolic phase than does the hand-coded 2.5D MM. Fig. 5.10 shows that the time waiting for `CSHIFT` communication to complete increases with size of input matrices. The hand-coded version spends less time waiting for its `CSHIFT` communication to complete since it overlaps its communication with local multiplication.

Fig. 5.8 shows that the efficiency of Maunam-generated code decreases from 86% to 76% of peak as the number of cores increases from 128 to 4096. A portion of this decrease can be attributed to the overhead of creating processor subsets using `MPI_Comm_split`. On 4096 cores, Maunam-generated code spends 8% of its time in `MPI_Comm_split`. Several strategies could be employed to reduce this overhead:

- `MPI_Comm_split` can be implemented more efficiently [101, 102]. We can employ these techniques in Maunam’s runtime library to create the necessary subcommunicators instead of relying on the `MPI_Comm_split`. 
Figure 5.9: Performance comparison of Maunam-generated 2.5D MM with Hand-coded 2.5D MM for different sizes of the input matrices. The difference in performance achieved by Maunam-generated code and hand-coded version is 7 – 12% of the machine peak.

Figure 5.10: Time spent in the systolic phase waiting for communication completion increases with increase in size of input matrices. The graph shows times reported by CrayPat inside MPI_Waitall waiting for the CSHIFT communication to complete in the systolic phase.
• We can use point-to-point communication to implement a reduction instead of creating process subsets using MPI_Comm_split. This strategy can be employed when the number and size of reductions performed is insufficient to amortize the cost of creating sub-communicators.

5.8 Summary

In this chapter, we presented Maunam—a compiler that automatically derives a parallel implementation of a program for distributed memory systems from a single-threaded global view specification.

Analysis and code generation for 2.5D MM with Maunam required several new capabilities. We developed a novel approach for affinizing modulo operations so that Maunam’s polyhedral framework could reason about wrap around communication patterns that arise with TILT and CSHIFT. In addition, we extended Maunam’s analysis and code generation for reductions to enable reductions across any or all dimensions of a multi-dimensional processor grid. The polyhedral analysis techniques we describe in this chapter are generally applicable and suitable for reasoning about communication and computation in PGAS languages, such as Chapel [44]. In the next chapter, we describe how we extended Maunam’s support for communication generation to enable communication and computation overlap in the systolic phase of 2.5D MM.
Chapter 6

Overlapping Communication with Computation

Hiding communication latency is essential to achieve scalable high performance on current and future parallel systems. In this chapter, we present a novel compiler transformation that overlaps communication with computation to hide communication latency. Unlike prior work, our technique is able to achieve this overlap even in the presence of an overlap-inhibiting data dependence between communication and computation steps in a loop iteration. We do so by transforming the data dependence into an overlap-amenable one. To achieve this overlap, the Maunam compiler transforms the code by employing array expansion, partial loop peeling, loop alignment, and array contraction (EPAC). This transformation is useful for optimization of systolic, communication avoiding algorithms. This chapter describes a transformation strategy first described in a PGAS 2015 extended abstract entitled “A Compiler Transformation to Overlap Communication with Dependent Computation” [11].

6.1 Motivation

Interprocessor communication of a value is several orders of magnitude more expensive than computation of a value. For that reason, strategies to hide inter-processor communication latency are important for efficiently mapping programs onto distributed-memory architectures. One well-known strategy for hiding communication latency is to overlap communication with computation. In this strategy, communication la-
tency is hidden by: a) employing non-blocking communication primitives, b) initiating non-blocking communication before an independent computation, and c) querying completion of pending communication after the computation. Manually overlapping communication with computation in complex codes is error prone. To address this problem, we describe compiler analyses and transformations that achieve this overlap automatically.

Efforts to overlap communication with computation include: a) using programmer hints [67, 68] to identify code regions where communication and computation can be overlapped, and b) splitting communicated data into portions that are dependent and independent from the computation, and overlapping only the communication of independent portions with computation, e.g., Pellegrini et al. [69]. Neither of these strategies can overlap communication with computation when the data communicated and data generated/consumed in the computation are identical. Such patterns of data usage occur in systolic, communication avoiding algorithms, e.g., 2.5D Matrix Multiplication [3].

Figure 6.1 presents a code snippet written in HPF [103]. In this snippet, communication is immediately followed by dependent computation. The !HPF$ directives specify that array A is block-distributed across the processors; the first dimension of A is unpartitioned, whereas the second dimension is distributed. In the snippet, the t loop performs a systolic computation. In each iteration of this loop, each processor receives a complete column of A from its left neighbor in S7 through a $\text{CSHIFT}^*$ operation, and consumes the received column in S9. The challenges to generate the communication in S7, and to overlap this communication with the computation in

\*CSHIFT operation is a transformational intrinsic function in Fortran that performs a circular shift of the elements of an array in the specified dimension.
double precision :: A(n,p)
!HPF$ processors proc(p)
!HPF$ template tmpl(p)
!HPF$ align A(*, :) with tmpl(:)
!HPF$ distribute tmpl(block) onto proc

do t = 1, c
    !processor receives A from left neighbor
    A(:, :) = CSHIFT(A(:, :) , -1, 2))
    !processor consumes received A
    ... = f(A(:, :) , ...)
enddo

**Figure 6.1**: HPF snippet contains a systolic loop where each processor sends and receives data in line 7 (S7), and consumes the received data in line 9 (S9).

S9 are as follows:

1. The semantics of \texttt{CSHIFT} operation in S7 dictate that each processor sends its column of A before receiving a new column.

2. The consumption of the received column of A in S9 creates a loop-independent flow dependence between S7 and S9. This dependence prevents trivial overlap of communication in S7 with the computation in S9.

Strip mining communication [53] is a powerful technique that can achieve overlap for such data usage patterns. However, this technique increases communication latency on the critical path. Furthermore, the applicability of a strip mining transformation depends on whether the computation can be partitioned. In this chapter, we present a technique that achieves overlap without increasing the latency along the critical path or partitioning the computation.

In the rest of this chapter, we describe EPAC — a compiler transformation that
automatically overlaps communication with dependent computation in systolic loops by applying array Expansion, partial loop Peeling, and loop Alignment, followed by array Contraction.

6.2 EPAC Transformation

This section describes the intuition behind the EPAC transformation, and then discusses both the opportunity for applying the EPAC transformation as well as the details of applying the EPAC transformation. The section concludes with a discussion of the safety of the transformation.

The EPAC transformation employs array expansion to transiently break anti- and output-dependences carried by a systolic loop. It employs loop peeling and alignment to move the source of the overlap-inhibiting flow-dependence into the previous iteration of the systolic loop. This ensures that the source and sink of the flow-dependence are now at a dependence distance of at least one. The conversion of a loop-independent flow dependence into a backward loop-carried dependence satisfies a critical property: The loop-independent dependences involving the source and the sink are at most input dependences, thereby, allowing for overlap if either the source or the sink requires communication.

Finally, EPAC employs array contraction to check the additional storage introduced in array expansion.

6.2.1 Opportunity

Fig. 6.2 describes dependence patterns in systolic loops that are addressed by EPAC. The opportunity to apply EPAC transformation is as follows:

1. The loop must be systolic in nature. A systolic loop is one that exhibits the
same pattern of communication and computation in every iteration. This is recognized through dependence patterns:

- every definition in the loop body has an output dependence on itself carried by the systolic loop
- every loop-independent flow- and anti-dependence has a complimentary loop-carried anti- and flow-dependence, respectively, through the same data reference that is carried by the systolic loop.

2. All uses of a variable must be derived from a covering definition†, except for upward exposed uses of the variable. All variable references must be either a use or an upward exposed use‡.

3. There must be no dependence cycles involving two or more variables. The third case in Fig. 6.2 provides an example of a dependence cycle created through a loop-independent flow dependence between the first and second statement via reference A and a loop-carried flow-dependence between the second and first statement via reference B. Using EPAC, we can convert the loop-independent flow dependence via A into an overlap-amenable loop-carried flow dependence. However, doing so would violate the dependence between the first and second statement via reference B.

†A covering definition for a variable is the first definition encountered along the control flow path in the loop. A loop can have a set of different covering definitions for a variable because of multiple code paths through the loop.

‡Variable references can be both a use and an upward exposed use because of multiple code paths through the loop.
### Figure 6.2: Array definition-use patterns within a systolic loop that are addressed/not-addressed by EPAC transformation.

All definitions above are covering definitions, uses before these definitions are upward exposed.  $s, s', s''$ are index sets that access the unpartitioned dimensions of array $A$; $s' \subseteq s$ and $s'' \subseteq s$.  $r, r'$ are index sets that access the unpartitioned dimensions of array $B$; $r' \subseteq r$.  ■ indicates references to variables apart from $A$ and $B$.

<table>
<thead>
<tr>
<th>Case</th>
<th>Array Def-Use</th>
<th>Loop-independent Dependence of Interest</th>
<th>Overlap Transformation [Expansion, Peeling, Alignment, Contraction]</th>
<th>Array Expanded, Contracted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A(s) = $</td>
<td>Flow dependence between $A(s)$ and $A(s')$</td>
<td>EPAC</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>$A(s) = $</td>
<td>Anti dependence between $A(s')$ and $A(s)$</td>
<td>EAC</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>$A(s) = B(r')$</td>
<td>-</td>
<td>Dependence cycle exists, EPAC/EAC is not applied</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>$A(s) = $</td>
<td>Flow dependence between $A(s)$ and $A(s')$</td>
<td>1 overlapped with 2&amp;3 using EPAC</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>$B(r) = $</td>
<td>Anti dependence between $B(r')$ and $B(r)$</td>
<td>1&amp;2 overlapped with 3 using EAC</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>$A(s) = $</td>
<td>Anti dependence between $A(s')$ and $A(s)$</td>
<td>1 overlapped with 2&amp;3 using EAC</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>$A(s) = $</td>
<td>Flow dependence between $A(s)$ and $A(s')$</td>
<td>1&amp;2 overlapped with 3 using EPAC</td>
<td>A</td>
</tr>
</tbody>
</table>
4. The definition-use (def-use) chain for a variable inside the systolic loop must match either of the cases 1, 2, 4, 5 in Fig. 6.2.

- In case 1, the definition of variable \( A \) is a covering definition, while the subsequent use accesses a subset of the indices accessed in the covering definition. While the pattern shows only one use, it represents all subsequent uses of the covering definition. Here, the loop-independent flow-dependence between the covering definition and subsequent use prevents any possible overlap. However, using EPAC, we overlap the communication/computation in the covering definition with the communication/computation in the use.

- In case 2, the upward exposed use of array \( A \) accesses a subset of the indices that are accessed in the covering definition. Here, the loop-independent anti-dependence between the upward exposed use and covering definition prevents any possible overlap. However, using EAC (EPAC with no peeling), we overlap the communication/computation in the upward exposed use with the communication/computation in the covering definition.

- In case 4, the def-use chain for variable \( A \) fits case 1, but the def-use chain for variable \( B \) fits case 2. In such cases, when there are common statements between the def-use chains of two variables and the def-use chains map to different cases, 1 and 2, then we overlap the communication and computation involving only one of the variables. As discussed in Section 6.4, overlap for both variables is possible in this context and is a small extension to our current work.

When there are no common statements and the def-use chains of two vari-
ables match different patterns, then we do overlap the communication and computation of both variables.

- Case 5 depicts a combination of cases 1 and 2. Application of EPAC or EAC depends on which of the statements requires communication. In cases when both EPAC and EAC can be applied, we default to EPAC.

**Identifying opportunity**  
Steps for identifying opportunities for applying EPAC or EAC are as follows:

- We identify three sets of references for each variable referenced in a systolic loop using the control flow graph and static single assignment (SSA) form; set of upward exposed uses, set of covering definitions, and set of uses.

- Utilizing these sets, we identify the pattern that can be applied to each variable based on Fig. 6.2. Based on the pattern, the statements referencing the variable are classified into either the EPAC source, EPAC sink, EAC source or EAC sink groups. These variables are candidates for array expansion and contraction.

  - EPAC source group for pattern 1 includes the covering definition while EPAC sink group includes the use. If there are upward exposed uses and EPAC is being applied, then those upward uses are placed in the EPAC source group.

  - EAC source group for pattern 2 includes the statements containing the upward exposed uses while EAC sink group includes the covering definition.

- Other statements in the loop are handled as follows:

  - A statement having a dependence relation with a statement in one of
these four groups, i.e., EPAC source/EPAC sink/EAC source/EAC sink, is placed in that statement’s group.

– A statement not having a dependence relation with any statement in one of the four groups is not placed in any group and remains unchanged.

– This step is performed repeatedly until a fixed point is reached.

• For EPAC/EAC to be applied, no statement should be in more than one group. If the same statement is in more than one group, then applying the transform will violate a dependence. Hence, we require that no statement be in more than one group.

6.2.2 Description of the EPAC Transformation

To overlap communication with computation in systolic loops with the characteristics described previously, we, first, employ array expansion and contraction to manage the extra space needed to achieve the overlap, as explained in Section 5.4.5. Next, we employ partial-loop peeling and loop alignment. Finally, we employ code motion to determine the placement for initiation and completion of communication within the systolic loop. We describe this process in detail in the following sections.

Array Expansion and Contraction

Consider the example present in Fig. 6.1. A straightforward way to realize the semantics of CSHIFT, refer to Section 5.4.5 for additional details, is to create two copies of the column of A on each processor, i.e., cur and next. In each iteration of t, a processor sends its cur column, receives and computes with the next column, and swaps the values of cur and next at the end of the iteration. However, this strat-
Listed 6.1: Before Array Expansion of A

1 double precision :: A(n,p)
2 ! HPF$ processors proc(p)
3 ! HPF$ template tmpl(p)
4 ! HPF$ align A(*, :) with tmpl(:)
5 ! HPF$ distribute tmpl(block) onto proc
6 do t = 1, c
7 ...
8 enddo

Listed 6.2: After Array Expansion of A

1 double precision :: A(n,0:c,p)
2 ! HPF$ processors proc(p)
3 ! HPF$ template tmpl(p)
4 ! HPF$ align A(*, :, :) with tmpl(:)
5 ! HPF$ distribute tmpl(block) onto proc
6 do t = 1, c
7 ...
8 enddo

ey does not remove the overlap-inhibiting dependence between $S_7$ and $S_9$. Hence, EPAC transformation uses array expansion and contraction. Our strategy for array expansion and contraction is influenced by Lefebvre and Feautrier [56] and Cohen and Lefebvre [57], who previously used array expansion and contraction to improve available parallelism.

Array expansion of the candidate variables, identified during EPAC opportunity, includes the following steps:

- We add a new non-distributed dimension to the variable involved in the overlap-inhibiting dependence. The span of this dimension in our implementation is $0 : c$, where $c$ is the upper bound of the systolic loop. Listing 6.1, Listing 6.2 indicate the change in array A before and after array expansion. The references to A will be updated in the subsequent steps.

- We update the variable references with appropriate indices along the new dimension inside the systolic loop.

  - Covering definitions write to the index that is equal to the current value
Listing 6.3: Before update of the references to $A$ in the loop $t$.

```fortran
1 double precision :: A(n,0:c,p)
2 ! HPF$ processors proc(p)
3 ! HPF$ template tmpl(p)
4 ! HPF$ align A(*,*, :) with tmpl(:)
5 ! HPF$ distribute tmpl(block) onto proc
6 do t = 1, c
7 ... 
8 ... 
9 ... 
10 enddo
```

Listing 6.4: After update of references to $A$; but before array contraction.

```fortran
1 double precision :: A(n,0:c,p)
2 ! HPF$ processors proc(p)
3 ! HPF$ template tmpl(p)
4 ! HPF$ align A(*,*, :) with tmpl(:)
5 ! HPF$ distribute tmpl(block) onto proc
6 do t = 1, c
7 next = t
8 A(:,t,:)=CSHIFT(A(:,t-1,:),-1,3))
9 ... = f(A(:, t,:), ...)
10 enddo
```

of the systolic-loop induction variable. If the loop is not normalized, then in contrast to using the systolic-loop induction variable, we can use an auxiliary loop induction variable that is initialized to 1 and is incremented in every iteration of the systolic loop.

- Upward exposed uses of the variable employ the definition from the previous iteration of the systolic loop. This is illustrated in the upward exposed use of $A$, in $S7$ of Listing 6.4, using $t-1$ as index into the new dimension.

As a result of array expansion, we manifest $c + 1$ copies of the original array, where $c$ is the span of the systolic loop. By making all copies manifest with the array expansion, we simplify the application of a peeling+alignment transformation. We reduce the number of copies to 2 during array contraction, i.e., after the application of peeling+alignment.

- We update variable references before the systolic loop to write and read from
index 0 of the new dimension in case of EPAC and index 1 of the new dimension in case of EAC.

- We update variable references after the systolic loop to write to and read from the new dimension at index \( \text{next} \). \( \text{next} \) is a scalar variable that captures the index of the new dimension used by the covering definition in every iteration of the systolic loop.

**Array contraction** While we manifest multiple copies of the original array, we do not need that many to overlap communication with computation. This is because we convert a loop-independent flow-dependence between an array definition and its subsequent use to a loop-carried flow-dependence with dependence distance of one. After applying partial loop peeling and loop alignment, we contract away excess copies by applying the contraction modulo (also called expansion degree) [56], i.e., using a modulo operation on the index for the new dimension where the modulus is set to the number of iterations between the source and sink of a loop-carried dependence. Our implementation uses a modulus of two. However, as indicated in Section 6.4, one can increase the modulus to greater than two to allow for greater overlap. Additionally,

- We replace every index into the new dimension, \( s \), inside the systolic loop with a modulo operation, \( \text{modulo}(s, 2) \).
- We reduce the value of \( \text{next} \) to \( \text{modulo}(\text{next}, 2) \).

**Partial Loop Peeling and Loop Alignment**

After array expansion, for case 1 in Fig. 6.2, the systolic loop still holds a loop independent dependence between the source and sink that prevents overlap. For
example, in Fig. 6.1 computation instance $S_9(t)$ is dependent on communication instance $S_7(t)$. To remove this loop independent dependence, we employ loop peeling and loop alignment. Our peeling and alignment technique is influenced by the CCTP transformation developed by Danalis et al. [66] that strip mines communication to achieve overlap.

**Applying Loop Peeling** During peeling, we perform the following steps:

- We hoist the first instance of every statement in the EPAC source group outside the loop. To do so, we create a copy of every statement in the EPAC source group and insert it before the loop. We enclose these copies inside an if condition that checks whether the loop is executed at least once; this is done by determining whether the upper bound is greater than or equal to the lower bound. We illustrate this step in Listing 6.5, Listing 6.6. For copies of covering definitions present in this source group, we do additional AST manipulations:
  - We update index expressions of the new dimension in the left-hand side of the definition to be 1, e.g., definition of $A$ in $S_7$ of Listing 6.6.
  - We update index expressions of the new dimension in the right-hand side of the definition to be 0.

- We remove the last instance of all the statements in the EPAC sink group to after the loop. To do so, we do similar AST manipulations as mentioned above, creating a copy of the statement and inserting it in an if block that determines whether the systolic loop is executed at least once. The index into the added new dimension is updated to `next` in references on the right hand side of these statements. We illustrate this step in Listing 6.7, Listing 6.8.
Listing 6.5: Before partial loop peeling; Listing 6.6: After partial loop peeling of source array contraction has not been applied of the overlap-inhibiting dependence; array contraction has not been applied yet.

```fortran
1 double precision :: A(n,0:c,p)  
2 ! HPF$ processors proc(p)  
3 ! HPF$ template tmpl(p)  
4 ! HPF$ align A(*,*,:) with tmpl(:)  
5 ! HPF$ distribute tmpl(block) onto proc
6 if(c .ge. 1) then  
7   A(:,1,:) = CSHIFT(A(:,0,:), -1,3))
8 endif  
9 do t = 1, c  
10   next = t  
11   A(:,t,:) = CSHIFT(A(:,t-1,:), -1,3))  
12   ... = f(A(:, t,:), ...)  
13 enddo
```

- We reduce the upper bound of the systolic loop by one. For example, in Fig. 6.1, we peel the first instance of the source, i.e., $S_7(1)$, and the last instance of the sink, i.e., $S_9(c)$.

**Applying Loop Alignment**  The application of loop alignment differs based on whether EPAC/EAC is applied.

- In EPAC transformation, after loop peeling, we reverse the program order of the source and sink of the dependence between the covering definition and its subsequent uses. This reversal does not change the lexicographic order of the dependence since the source group and sink group statements are in different iterations after peeling. The effect of the reversal is the movement of the covering definition along the back edge from a subsequent iteration to the current itera-
Listing 6.7: After partial loop peeling of source of the overlap-inhibiting dependence; array contraction has not been applied yet.

```fortran
   double precision :: A(n,0:c,p)
   ! HPF$ processors proc(p)
   ! HPF$ template tmpl(p)
   ! HPF$ align A(*,*,:) with tmpl(:)
   ! HPF$ distribute tmpl(block) onto proc
   if(c .ge. 1) then
       A(:,1,:) = CSHIFT(A(:,0,:),1,3))
   endif
   do t = 1, c
     ...
   enddo
```

Listing 6.8: After partial loop peeling of sink of the overlap-inhibiting dependence; array contraction has not been applied yet.

```fortran
   double precision :: A(n,0:c,p)
   ! HPF$ processors proc(p)
   ! HPF$ template tmpl(p)
   ! HPF$ align A(*,*,:) with tmpl(:)
   ! HPF$ distribute tmpl(block) onto proc
   if(c .ge. 1) then
       A(:,1,:) = CSHIFT(A(:,0,:),1,3))
   endif
   do t = 1, c-1
     ...
   enddo
   if(c .ge. 1) then
     ... = f(A(:,next,:),...)
   endif
```
Listing 6.9: Before loop alignment; array contraction has not been applied yet.

```fortran
double precision :: A(n,0:c,p)
!
HPF$
processors proc(p)
!
HPF$
template tmpl(p)
!
HPF$
align A(*,*,:) with tmpl(:)
!
HPF$
distribute tmpl(block) onto proc
!
if(c .ge. 1) then
A(:,1,:) = CSHIFT(A(:,0,:), -1, 3))
endif
!
do t = 1, c-1
...
!
do t = 1, c-1
...
!
do t = 1, c-1
...
!
if(c .ge. 1) then
...
endif
```

Listing 6.10: After loop alignment; array contraction has not been applied yet.

```fortran
double precision :: A(n,0:c,p)
!
HPF$
processors proc(p)
!
HPF$
template tmpl(p)
!
HPF$
align A(*,*,:) with tmpl(:)
!
HPF$
distribute tmpl(block) onto proc
!
next = 0
!
if(c .ge. 1) then
next = 1
A (:,1,:) = CSHIFT(A (:,0,:), -1, 3))
endif
!
do t = 1, c-1
...
!
do t = 1, c-1
...
!
do t = 1, c-1
...
!
if(c .ge. 1) then
...
endif
```

tion. Hence, alignment dictates that the index expressions of the new dimension be increased by one in all statements of the source group. The statements of the sink group remain unchanged. This is illustrated in Listing 6.9, Listing 6.10.

In Listing 6.10, S14 writes to $t+1$ index of the new dimension of array $A$ while reading from index $t$ of the new dimension of array $A$. In contrast, as shown in Listing 6.9, the same statement, i.e., S10, was writing to $t$ of the new dimension of array $A$ before alignment.

- In EAC transformation, after array expansion, the dependence between the source group and sink group is loop-carried. During alignment, all statements
in both the source and sink groups are updated such that the index expression of the new dimension is increased by one.

**Code Motion to realize the overlap** After array expansion, loop peeling and loop alignment, and array contraction; we employ code motion so that the placeholders for the initiation of communication, namely `MPI_Isend` and `MPI_Irecv` needed for either the covering definition, use or upward exposed use, float to the beginning of the loop. The completion of the communication, namely `MPI_Waitall`, float to the bottom of the loop.

Applying to Fig. 6.1, we align $S_7(t+1)$ and $S_9(t)$ in the same iteration by: a) peeling the first instance of the source of the dependence, i.e., $S_7(1)$, and the last instance of the sink of the dependence, i.e., $S_9(c)$, and (b) reversing the textual order of $S_7$ and $S_9$, converting the forward dependence to a backward dependence. Finally, we complete the loop alignment transformation so that iteration $t$ of the transformed loop executes the $t^{th}$ instance of $S_9$ and the $t+1^{st}$ instance of $S_7$. With the absence of a loop independent dependence, the communication for the `CSHIFT in $S_7(t+1)$ can be initiated before the computation $S_9(t)$, and the completion can be queried after $S_9(t)$. This is illustrated in Listing 6.12. In Listing 6.12, `CSHIFT_BEGIN_COMM` functions as a placeholder for `MPI_Isend` and `MPI_Irecv`, which are needed to initiate the communication for `CSHIFT`. Additionally, one can note that `CSHIFT_BEGIN_COMM` has not floated to the top of the systolic loop, but is positioned as the third statement. This is because by floating to the top, it violates the loop independent flow dependence between the definitions of `cur`, `next` and their uses as indices into the new dimension of array $A$, which is used as an argument to `CSHIFT`. Similarly, `CSHIFT_END_COMM` functions as a placeholder for `MPI_Waitall`, which is needed to complete the communication for
Listing 6.11: After array contraction

```plaintext
double precision :: A(n,0:1,p)
! HPF$ processors proc(p)
! HPF$ template tmpl(p)
! HPF$ align A(*,* , :) with tmpl(:)
! HPF$ distribute tmpl(block) onto proc
cur = next = 0
if(c .ge. 1) then
  next = 1
  A(:,next,:) = CSHIFT(A(:,cur,:), -1, 3))
endif
do t = 1, c-1
  cur = modulo(t, 2)
  next = modulo(t+1, 2)
  ... = f(A(:, cur,:),...)
  A(:,next,:) = CSHIFT(A(:,cur,:), -1, 3))
enddo
if(c .ge. 1) then
  ... = f(A(:, next,:),...)
endif
...
```

Listing 6.12: Applying code motion after EPAC to overlap communication with computation

```plaintext
double precision :: A(n,0:1,p)
! HPF$ processors proc(p)
! HPF$ template tmpl(p)
! HPF$ align A(*,* , :) with tmpl(:)
! HPF$ distribute tmpl(block) onto proc
cur = next = 0
if(c .ge. 1) then
  next = 1
  A(:,next,:) = CSHIFT(A(:,cur,:), -1, 3))
endif
do t = 1, c-1
  cur = modulo(t, 2)
  next = modulo(t+1, 2)
  A(:,next,:) = CSHIFT_BEGIN_COMM(A(:,cur,:), -1, 3)
  ... = f(A(:, cur,i),...)
  A(:,next,:) = CSHIFT_END_COMM(A(:,cur,:), -1, 3)
enddo
if(c .ge. 1) then
  ... = f(A(:, next,:),...)
endif
```
6.2.3 Safety

In this section, we prove that EPAC compiler transformation is safe.

**Theorem 6.2.1.** *EPAC compiler transformation is safe.*

*Proof.*

The fundamental theorem of dependence states that “any reordering transformation that preserves every dependence in a program preserves the meaning of that program”. While this is a sufficient condition, a necessary condition is that all flow dependences must be preserved. Our EPAC transformation maintains the lexicographic order between the source and sink endpoints of any flow dependence. As stated before, the reversal of the source and sink of the flow dependence in EPAC after loop peeling is legal since the source of the dependence is from a subsequent iteration, i.e., is at a distance of one from the sink. The anti- and output-dependences carried by the systolic loop are removed by array expansion, but re-introduced by array contraction. These dependences are false dependences occurring as a result of reuse of space and not reuse of value. Hence, their elimination does not affect the meaning of the program. EPAC transformation preserves the flow-, anti-, and output-dependences, and thereby, preserves the meaning of the program. EPAC transformation is safe. 

6.3 Experiments

In this section, we discuss the performance achieved by EPAC-generated Fortran+MPI code for a modified version of 2.5D MM. This version of 2.5D MM is illustrated in Listing 6.13. The changes from the version discussed in the previous
chapter are: 1) there are only two unpartitioned dimensions for \(A\) and \(B\), since our compiler transformation does array expansion and adds the third unpartitioned dimension, further reducing the programmer’s burden, 2) a multiplication before the systolic loop is now after the systolic loop, and 3) the textual order of the shifts of input matrices and the multiplication are reversed within the loop. After applying our compiler transformation, shifts of the input matrices \(A, B\) are overlapped with the matrix multiplication, i.e., \texttt{matmul}, in every iteration of the systolic loop, 1 : \(\sqrt{(p/c^3)} - 1\). If we had used the unmodified version of 2.5D MM, then we would have been able to achieve overlap of communication with computation in 1 : \(\sqrt{(p/c^3)} - 2\), i.e., one less, number of iterations of the systolic loop. We conducted our experiments on a Cray XC30 supercomputer (Edison) at NERSC. Each compute node in the system contains two 2.4 GHz 12-core Intel Ivy Bridge processors [98]. We employ the Cray-Pat performance analysis toolkit [99] to measure the time spent in different portions of the code.

**Experimental Methodology.** We compile both Maunam-generated code without-EPAC and Maunam-generated code with-EPAC 2.5D MM with the Intel Compiler Suite 15.1 and link with the Intel Math Kernel Library’s BLAS routines. The compiler options include “-O3 -mkl=sequential -opt-matmul”. We determine the execution time as follows:

- Multiple iterations of the algorithm are executed in a single run of the experiment. Initialization of input matrices is performed only-once before these iterations. All cores measure the total time taken for iterations following initialization.

- The average execution time is computed by dividing the sum of time-per-
Listing 6.13: Modified Version of 2.5D MM

1 double precision ::A(n,n,\sqrt{p/c},\sqrt{p/c},c)
2 double precision ::B(n,n,\sqrt{p/c},\sqrt{p/c},c)
3 double precision ::R(n,n,\sqrt{p/c},\sqrt{p/c},c)
4 ! HPF$ processors proc(\sqrt{p/c},\sqrt{p/c},c)
5 ! HPF$ template tmpl(\sqrt{p/c},\sqrt{p/c},c)
6 ! HPF$ align B(*,*,:,:,:) with tmpl(:,,:,:)
7 ! HPF$ align A(*,*,:,:,:) with tmpl(:,,:,:)
8 ! HPF$ align R(*,*,:,:,:) with tmpl(:,,:,:)
9 ! HPF$ distribute tmpl(block,block,block) onto proc
10 !!! Initialize A,B
11 R(:,,:,:,:) = 0
12 A(:,,:,:,:) = SPREAD(A(:,,:,:,:1),5,c)
13 B(:,,:,:,:) = SPREAD(B(:,,:,:,:1),5,c)
14 A(:,,:,:,:) = TILT(TILT(A(:,,:,:,:),4,5,\sqrt{p/c^2}),4,3,-1)
15 B(:,,:,:,:) = TILT(TILT(B(:,,:,:,:),3,5,\sqrt{p/c^2}),3,4,-1)
16 do l = 1, \sqrt{p/c^2}-1
17 R(:,,:,:,:) = R(:,,:,:,:) + MATMUL(A(:,,:,:,:),B(:,,:,:,:))
18 A(:,,:,:,:) = CSHIFT(A(:,,:,:,:),1,4)
19 B(:,,:,:,:) = CSHIFT(B(:,,:,:,:),1,3)
20 enddo
21 R(:,,:,:,:) = R(:,,:,:,:) + MATMUL(A(:,,:,:,:),B(:,,:,:,:))
22 R(:,,:,:,:1) = SUM(R(:,,:,:,:),5)
Figure 6.3: Time spent in MPI\_Waitall when multiplying matrices of rank 16384 × 16384.

iteration reported by each core by the number of cores.

- The number of iterations is 10 for multiplication of matrices of rank 16384 × 16384 and greater, and is 30 for multiplication of matrices of rank 2048 × 2048.

- We use 128 compute nodes and perform experiments using up to 1024 cores.

Figures Fig. 6.5, Fig. 6.3, Fig. 6.6 and Fig. 6.4 present the Craypat-collected time spent in the computation (BLAS) and the communication (MPI\_Waitall) for multiplying two 2,048 × 2,048, 16,384 × 16,384, and 65,536 × 65,536 ranked matrices using the 2.5D MM algorithm. When multiplying smaller matrices, the execution is communication bound. Hence, applying our compiler transformed code results in significant performance benefits, e.g., 29.9% of the total time is spent in MPI\_Waitall when multiplying 2,048 × 2,048 ranked matrices on 512 cores before applying our
Figure 6.4: Time spent in `dgemm` BLAS routine when multiplying matrices of rank $16384 \times 16384$.

Figure 6.5: Time spent in `MPI_Waitall` when multiplying matrices of rank $2048 \times 2048$. 

Figure 6.6: Time spent in MPI_Waitall when multiplying matrices of rank 65536 × 65536.

Figure 6.7: Percentage of machine peak achieved and time spent in MPI_Waitall when multiplying matrices of rank 65536 × 65536 on 4096 cores. The time taken in seconds for a single iteration of 2.5D MM is specified at the bottom of the bars in the left graph. The performance in TFlops/s is specified at the top of the bars in the left graph.
compiler transformation, whereas the time spent in MPI\_Waitall is reduced to 18.9% of the total time after applying our compiler transformation. When multiplying medium to large matrices, the execution is compute bound. This is indicated by the time spent in the BLAS kernels shown in Fig. 6.4. For such matrices, applying our compiler transformation reaps benefits reducing time spent in MPI\_Waitall from 7.9% of the total time to 3.1% of the total time on 512 cores when multiplying matrices of size 16,384 × 16,384. We see a similar reduction in time spent in MPI\_Waitall from 5.1% of the total time to 2.9% of the total time on 1,024 cores when multiplying matrices of size 65,536 × 65,536. We achieve a total of 64 TFlops/s when multiplying matrices of rank 65536 × 65536 on 4096 cores, whereas the hand-coded version achieves 66TFlops/s. We attribute the difference of 2TFlops/s to the following: the local matmul and the subsequent update to result array R in the systolic loop are done through a single call to the BLAS DGEMM routine in case of the hand-coded version, whereas, in case of Maunam-generated EPAC-overlapped version these are done in two separate steps.

No improvement for 256 core execution  We observe that the time spent in MPI\_Waitall is the same for a 256-core execution. In a 256-core execution, the processor grid employed for 2.5D MM algorithm is 8 × 8 × 4, i.e., c = 4. In this case, the systolic loop undergoes 1 iteration, i.e., \(\sqrt{(p/c^3)} - 1\). With only one iteration of the systolic loop, there are two multiplications performed by each processor. Only one of these multiplications is overlapped with communication. In case of 512-core execution, where the processor grid employed is 16 × 16 × 2, the systolic loop undergoes 7 iterations. There are a total of eight multiplications performed by a processor, and seven of these multiplications are overlapped with communication. Thus, we observe
a performance improvement in the 512-core executions across different input matrix sizes, whereas we observe very less to no improvement with the 256-core executions. The performance improvement due to overlap is, expectedly, more impactful when the systolic loop executes multiple iterations.

### 6.4 Limitations

In this section, we discuss the limitations of our compiler transformation and lay out initial steps to overcome the same.

**Applying EPAC followed by EAC in pattern 5 of Fig. 6.2** Currently, we apply either EPAC or EAC to achieve the overlap of statements 1 and 2 or statements 2 and 3, respectively. However, after applying EPAC, we can pursue array expansion of variable B to remove the anti-dependence between statements 2 and 3. Following this removal, we can overlap the communication/computation in statements 2 and 3 since they read and write different entries of B. We can, finally, apply array contraction to manage the additional space needed for B. This application of EPAC or EAC will help achieve a greater amount of overlap.

**Increasing the contraction modulo to > 2** To overlap multiple rounds of communication with multiple rounds of computation, we can orchestrate the movement of statements along $k > 2$ back edges of the systolic loop and use a contraction modulus of $k$. Doing so could be significantly profitable if a) the communication has a high latency cost associated with it, and b) the interconnect has sufficient bandwidth to carry multiple rounds of communicated data at the same time. Additionally, this orchestration would require that manuam’s communication generation be able to place
matching initiation and completion of CSHIFT operations in different iterations.
Chapter 7

COFENCE

In recent years, there has been a surge of interest in partitioned global address space (PGAS) languages. Coarray Fortran is one such language. It has its advocates because it enables users to explicitly place communication and partition computation to parallelize algorithms that current compilers for global view languages can’t, e.g., a Fast Fourier Transform algorithm [104]. In Coarray Fortran, synchronization is explicit and synchronization primitives are part of the language. In the previous two chapters, we described analysis and cogeneration from a global view parallel language, and we hid communication latency by overlapping communication with computation using non-blocking data transfers. In this chapter, we describe cofence synchronization construct, first described in an IPDPS 2014 paper entitled “Managing Asynchronous Operations in Coarray Fortran 2.0” [12], that enables one to manage local data completion of implicitly-synchronized asynchronous operations in a PGAS language. We implement the cofence construct in Coarray Fortran 2.0 (CAF 2.0) [105, 106, 107]—a partitioned global address space language that provides a rich set of asynchronous operations for avoiding exposed latency including asynchronous copies, function shipping, and asynchronous collectives.
Fig. 7.1: Execution timeline of an asynchronous operation indicating the different points of completion. *Finish, Phasers, cofence, and events* are different synchronization constructs that help achieve different points of completion.

### 7.1 Motivation

Asynchronous operations are a key component of PGAS languages for large-scale parallel systems. They are necessary for hiding the latency of synchronization and data movement.

Asynchronous operations are useless without completion guarantees. Figure 7.1 shows four forms of completion worth considering for an asynchronous operation: *initiation completion, local data completion, local operation completion, and global completion*. For an asynchronous operation $A$ initiated by image $i$, we define these four completion points as follows:

- $A$ is *initiation complete* on image $i$ when $A$ has been queued up for execution;
- $A$ is *local data complete* on image $i$ when any inputs of $A$ on $i$ may be overwritten, and any outputs of $A$ on $i$ may be read.
- $A$ is *local operation complete* when any pair-wise communication for $A$ involving
image $i$ is complete, at which point data on $i$ touched by $A$ can be read or modified.

- $A$ is globally complete when $A$ is local operation complete with respect to all participating images.

To address the need for a flexible synchronization construct that captures local data completion of implicitly synchronized asynchronous operations, we developed the cofence construct. Inspired by the SPARC V9 [75] MEMBAR instruction (for loads and stores to memory only), cofence provides optional arguments enabling a programmer to relax the requirements for a subset of the implicitly synchronized asynchronous operations.

The rest of the sections are as follows: Section 7.2 briefly introduces asynchronous operations in CAF 2.0, followed by the memory model of CAF 2.0 in Section 7.3. Section 7.4 describes the cofence API. Section 7.5, Section 7.6, Section 7.7, Section 7.8 explore subtle interactions between cofence, asynchronous copies and collectives, function shipping, and events. Section 7.9 presents performance results, which show that local data completion rather than local operation completion yields superior performance for a producer-consumer scenario.

### 7.2 Background

Although features of CAF 2.0 are described elsewhere [105, 106], we briefly review key features here to make this chapter self-contained and provide a context for discussing cofence, which is the focus of this chapter.
7.2.1 Team

A team is a first-class entity that represents a process subset in CAF 2.0. Teams in CAF 2.0 serve three purposes: a) the set of images in a team serves as a domain onto which shared distributed data objects, known as coarrays, may be allocated; b) a team provides a name space within which process images can be indexed by their relative rank; c) a team provides an isolated domain for its members to communicate and synchronize collectively.

All process images initially belong to team_world; new teams are created by invoking team_split.

7.2.2 Event

Events in CAF 2.0 serve as a mechanism for managing explicit completion of asynchronous operations and pair-wise coordination between process images. Like other variables, events to be accessed remotely are declared as coarrays. Local events (those not in a coarray) are only useful for managing completion of locally-initiated operations.

Events can be passed as optional parameters to asynchronous operations, which use them to initiate or signal completion of various forms. The occurrence of an event can also be signaled explicitly using event_notify. An event_wait operation blocks execution of the current thread until an event has been posted.
7.2.3 Asynchronous Operations

Asynchronous Copy

CAF 2.0 provides a predicated asynchronous data copy. The \texttt{copy\_async} API, as shown below, copies \texttt{srcA} on image \texttt{p2} to \texttt{destA} on image \texttt{p1}.

\begin{verbatim}
copy_async(destA[p1],srcA[p2],preE,srcE,destE)
\end{verbatim}

An asynchronous copy may proceed after its (optional) predicate event \texttt{preE} has been posted. Notification of event \texttt{srcE} indicates that read of data \texttt{srcA} is complete; \texttt{srcA} can be overwritten on \texttt{p2}. Notification of event \texttt{destE} indicates that the data has been delivered to \texttt{destA} on \texttt{p1}. \texttt{p1} and \texttt{p2} may either be local or remote images. \texttt{preE}, \texttt{srcE}, and \texttt{destE} each may either be a local event or an event located on a remote image.

Asynchronous Collectives

CAF 2.0 asynchronous collectives enable one to overlap collective communication with computation. Our vision for asynchronous collectives in \texttt{CAF 2.0} includes support for \texttt{alltoall}, \texttt{barrier}, \texttt{broadcast}, \texttt{gather}, \texttt{reduce}, \texttt{scatter}, \texttt{scan}, and \texttt{sort}. The following line of code shows an example of an asynchronous broadcast.

\begin{verbatim}
teamm\_broadcast\_async(A(:,),root,myteam,srcE,localE)
\end{verbatim}

Like asynchronous copy, asynchronous collectives have optional \texttt{event} parameters. In the example above, \texttt{srcE} indicates local data completion; \texttt{localE} indicates local operation completion.
7.3 Managing asynchrony in CAF 2.0

With the spectrum of parallel programmers ranging from domain scientists mapping applications onto clusters to computer scientists trying to tune applications for maximal performance, we believe it is necessary to provide expressive language-level mechanisms for coordinating communication that suit developers with varying needs and levels of expertise. In CAF 2.0, we provide two models for coordinating asynchrony: *implicit completion* and *explicit completion*.

Using *implicit completion*, the casual programmer relinquishes the ability to manage completion of an individual asynchronous operation. When initiated with implicit completion (i.e., without *event* variables), the completion of an asynchronous operation is not known until the next *cofence* or the end of an enclosing *finish* block.

Using *explicit completion*, advanced programmers can use *event* variables to acquire completion of an individual asynchronous operation until the latest possible moment. The former is intended as the default synchronization model and the latter is intended for tuning the performance of particularly costly communications.

For performance, CAF 2.0 uses a relaxed memory model [108]. Relaxed memory models do not guarantee a strict order between operations on data and synchronization objects unless these operations are ordered by synchronization. The relaxed model applies to all asynchronous operations, coarray read/write, and event notify/wait. Other operations that have local side effects follow a *data-race-free-0* model [109]; in short, operations that access process-local variables are free to be reordered unless they are (a) *SYNC* statements (as defined in the Fortran 2008 standard), and/or (b) violate data or control dependences. The guiding principles for the CAF 2.0 relaxed memory model are:
<table>
<thead>
<tr>
<th>Operation type</th>
<th>Image’s role</th>
<th>Local data completion (cofence)</th>
<th>Local operation completion (event_wait)</th>
<th>Global completion of operations with implicit synchronization (finish)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous broadcast*</td>
<td>Root</td>
<td>Data buffer can be safely modified</td>
<td>All pair-wise communication involving this image is complete</td>
<td>Data is ready on every participating image</td>
</tr>
<tr>
<td></td>
<td>Participant</td>
<td>Data may be read</td>
<td>All pair-wise communication involving this image is complete; Data can be safely modified</td>
<td>Data is ready on every participating image</td>
</tr>
<tr>
<td>Asynchronous copy</td>
<td>Reading from a local buffer</td>
<td>Source buffer may be written</td>
<td>Destination buffer may be read</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Writing to a local buffer</td>
<td></td>
<td>Destination buffer may be read</td>
<td></td>
</tr>
<tr>
<td>Spawn</td>
<td>Initiator</td>
<td>Arguments to the spawn operation are evaluated; local data passed as arguments may be written</td>
<td>Spawn is complete on the target image</td>
<td>Any asynchronous operation with implicit completion initiated by this spawn is globally complete</td>
</tr>
</tbody>
</table>

**Figure 7.2**: Asynchronous operations and different stages of completion.

*We discuss asynchronous broadcast as a representative example for asynchronous collectives.

**Processor consistency** This principle dictates that all process images see writes of a single process image in the same order. The writes of different process images may be seen in different orders by different process images [109].

**Different forms of consistency at synchronization points** Different synchronization operations guarantee different levels of completion. Below, we outline the semantics associated with `finish`, `event_notify`, `event_wait`, and `cofence`.

- `finish` guarantees global completion of implicitly completed asynchronous operations.
- `cofence` guarantees local data completion of implicitly completed asynchronous operations.

When an `event` is notified as a side effect of an asynchronous operation, it guar-
antees local operation completion of that operation. For example, consider the code snippet below depicting a broadcast by process image P0, and a subsequent event_wait on event doneE.

team_broadcast_async(A(:,), P0, myteam, srcE, doneE)
call event_wait(doneE)

When doneE is notified on P0, the role played by P0 in the broadcast is complete, i.e., all stages of the broadcast in which P0 participates are complete. However, notification of doneE does not imply that the broadcast is complete on other process images.

event_wait and event_notify obey acquire and release semantics. event_notify has release semantics. This principle allows the execution of operations that follow an event_notify in the program to begin before it, provided that data and control dependence constraints are not violated. event_wait has acquire semantics. This principle allows a process image to wait for an event earlier than it occurs in program order, provided that it does not violate data and control dependences. A summary of the asynchronous operations, and the different points of completion for each of the operations is described in Fig. 7.2.

7.4 cofence API

The cofence statement enables a programmer to demand local data completion of implicitly-synchronized asynchronous operations. An asynchronous operation A is local data complete on the initiating image i, when 1) any inputs of A on i may be overwritten and 2) any outputs of A on i may be consumed.

The cofence API is as follows:
The construct takes two optional arguments. The first argument specifies which class of asynchronous operations (i.e., write/read) with implicit synchronization appearing in statement instances encountered before a cofence statement may defer completion until after the cofence statement instance. The second argument specifies which class of asynchronous operations with implicit completion, whose statement instances are encountered after a cofence statement instance, may be initiated before the cofence statement instance completes. Depending upon the argument values passed, a cofence allows reads, writes, or both to pass across in the specified direction. If an asynchronous operation both reads and writes local data, then a cofence that allows either a read or write to pass across may not have any practical effect if the unconstrained action (e.g., a read) must occur before a constrained action (e.g., a write). The optional arguments enable one to relax requirements for implicit completion for performance tuning, without requiring a programmer to manage completion of each asynchronous operation explicitly.

Currently, Communication substrates, e.g., GASNet, used by parallel languages ensure that a non-blocking communication operation, involving a read from a local buffer, is local data complete before the call returns. This completion semantics makes it difficult to perform computation between initiation completion and local data completion. To circumvent this limitation, one can offload asynchronous communication calls to a communication thread so that the main thread can perform useful work as soon as it initiates a communication. This strategy of using an extra communication thread might not be advantageous on systems with few threads per node, e.g.,

*Here, we use the term statement instances rather than statements to accommodate the case in which a cofence and asynchronous communication with implicit completion are present in a loop.*
Blue Gene/P. However, emerging systems such as Blue Gene/Q and Intel MIC have a substantial number of hardware threads per node. On such platforms, dedicating a small number of threads on each node for communication could be beneficial. In the following sections, we detail the interactions between asynchronous operations in CAF 2.0 and cofence.

### 7.5 cofence and Asynchronous Copy

The semantics of local data completion w.r.t. an asynchronous copy operation are:

- On the initiating process, if the source buffer of the copy is local, local data completion of the copy operation ensures that the source buffer can be safely modified.

- On the initiating process, if the destination buffer of the copy is local, local data completion of the copy operation ensures that the destination buffer is ready to be consumed.

Figure 7.3 shows an example in which both cofence and copy_async constructs are used. The asynchronous copy operation at line 1 copies outbuf(i) on the initiating
image to coarray inbuf on image succ. The operation is local data complete when outbuf(i) can be modified on the initiating image without potentially affecting the value written to inbuf(i). The point to note is that this completion does not include whether the copy operation is complete on succ, i.e., whether inbuf(i) is available for consumption. The cofence at line 8, however, allows the copy_async operation at line 5 to pass downwards, i.e., it may complete after the cofence. At the same time, the cofence at line 8 ensures local data completion of the copy_async operation at line 6.

7.6 cofence and Asynchronous Collectives

Consider the code fragment in Figure 7.4. This example shows the use of an asynchronous broadcast operation along with cofence. The cofence on line 5 guarantees that process image p can reuse buffer buf without affecting the correctness of the broadcast operation. However, the cofence at line 5 allows other asynchronous operations writing other local data to move across unconstrained. In this snippet of code, the root image of the broadcast (p) is using the time between local data completion and local operation completion to ready the buffer for the next round of broadcast. On other process images involved in the broadcast, e.g., process image q, the cofence on line 10 captures the arrival of broadcast data on q.

7.7 cofence and Shipped Functions

When a cofence is used inside a shipped function, it should only capture the completion of implicit asynchronous operations launched by the shipped function, i.e., with dynamic scoping. Accordingly, the cofence on line 3 in Figure 7.5 does not guarantee local data completion of the asynchronous operation in line 6, but only of
```fortran
if(my_image_rank .eq. p) then
  ...
  call broadcast_async(buf, p)
  ...
  cofence(WRITE, WRITE)
  ! buf can be safely overwritten
  buf = ... 
else 
  call broadcast_async(buf, p)
  cofence()
  ! read buf
endif
```

Figure 7.4: Cofence and asynchronous broadcast. On a participating process image, say q, the cofence captures the arrival of broadcast data.

the asynchronous operation in line 2. A cofence on line 9, present after the spawn, captures the completion of argument evaluation (if any) for the spawn function, and does not provide any guarantee w.r.t. execution of the spawned function. In other words, if any local buffers are passed as arguments to the spawn then they can be safely modified only after the cofence.

7.8 cofence and Events

Events contain an implicit cofence. In this section, we describe the semantics of two event mechanisms: event wait and event notification. The semantics are as follows:

An event_notify acts as a release operation An event_notify by a process image p indicates that p has completed a certain set of updates to shared data of interest to other process image(s). Since, in general, its not possible to identify the
subroutine foo()
  copy_async(...)  
  cofence()
end subroutine

program  
  copy_async(...)  
  finish
    spawn foo()[random_process_image]
  cofence()
end finish
end program

Figure 7.5 : Cofence inside shipped functions

updates of interest to other process image(s), e.g., q, who might be waiting on the event, the event_notify should prevent operations from moving downwards. On the other hand, p does not give any guarantee to q when it executes operations following the event_notify. Thus, the event_notify can be porous to operations that appear afterward, i.e., these operations can be initiated before the event_notify. Hence, we implement the event_notify as a release operation [110]. We use epochs (inspired by MPI’s access epochs [111]) to allow for such an implementation.

An event_wait acts as an acquire operation An event_wait indicates that a process image q cannot execute any operations after the event_wait until a notification for that event arrives. Since it does not give any guarantee about operations before, we have modeled the event_wait as an acquire operation, i.e., operations before an event_wait can be initiated or completed after the event_wait.
Figure 7.6: Sketch of the micro-benchmark to depict use of cofence and copy_async. The size of the copied data is 80 bytes.

Figure 7.7: A micro-benchmark showing the advantage of using a cofence with copy_async. The size of the copied data is 80 bytes, and the number of random processes to which data is sent in each iteration is 5.
7.9 cofence in Practice

We evaluated the utility of cofence using a producer consumer micro-benchmark. We used a Cray XK6 Supercomputer (Jaguar) at Oak Ridge National Laboratory for our experiments. Figure 7.6 shows a sketch of our micro-benchmark. In the micro-benchmark, we await the completion of asynchronous copies using either a cofence, an event_wait, or a finish. In the benchmark variant using a cofence, the producer (process 0) uses local data completion to determine that it can reuse the src buffer. It is able to prepare the src buffer for the next round without waiting for delivery of the buffer contents to the destination process image. The benchmark variant using event_wait waits for each copy to be delivered to its destination process image. This variant incurs a delay proportional to the communication latency. The variant using finish waits for global completion of all copies and incurs a cost proportional to $O(\log p)$ communication latencies. Figure 7.7 shows the cost of these three synchronization strategies. The cofence variant is the fastest, 35% reduction in time on 1024 cores when compared to the finish variant. The event variant is slower because it must wait for data to be delivered to the destination image before execution can proceed. The finish variant is the slowest because it enforces global completion.

7.10 Summary

In this chapter, we presented cofence, a synchronization construct that enables one to manage local data completion of implicitly-synchronized asynchronous operations in PGAS languages. We explored subtle interactions between cofence, events, asynchronous copies and collectives, and function shipping. We justified its presence in a relaxed memory model for CAF 2.0. Our cofence micro-benchmark shows that for a
producer-consumer scenario, using local data completion rather than local operation completion yields superior performance.
Chapter 8

Conclusions and Future Work

With the anticipated end of Moore’s law within the next decade [112], improvements in the software stack, namely, programming model, compiler technology, and application-level algorithms are necessary. This thesis describes improvements needed in two areas, compiler technology and programming model.

Compiler Technology  The cost of data movement is significantly higher than the cost of a floating point operation on large scale parallel systems as indicated in the Programming Abstractions for Data Locality (PADAL) report [6]. In this thesis, we claimed that by employing polyhedral methods for analysis, transformation and code generation, we could generate high-performance distributed memory code. To support this claim, we developed Maunam—a compiler that automatically derives a high-performance parallel implementation of a program for distributed memory systems from a single-threaded global view specification. Analysis and code generation for 2.5D communication avoiding algorithms, e.g., communication avoiding 2.5D matrix multiplication, with Maunam required several new capabilities. We developed a novel approach for affinizing modulo operations so that Maunam’s polyhedral framework could reason about wrap around communication patterns. In addition, we extended Maunam’s analysis and code generation for collectives to enable collectives across any or all dimensions of a multi-dimensional processor grid.

We identified the cause of the performance gap between Maunam-generated code
and the hand-coded version to be Maunam’s inability to overlap communication with dependent computation in systolic loops. To overcome this performance gap, we developed EPAC—a compiler transformation that automatically overlaps communication with dependent computation in systolic loops by applying array expansion, partial loop peeling, and loop alignment, followed by array contraction. We were able to do so without increasing the latency on the critical path or fragmenting the computation. Applying EPAC, Maunam achieved 64 TFlops/s (81% of the machine peak) running on 4096 cores of a Cray XC30 supercomputer. Code generated by Maunam from a high level global view representation achieves 97% of the performance of a hand-coded SPMD version.

**Global Communication Optimization** We believe that additional opportunities exist for applying EPAC to achieve communication-computation overlap if we extend the window of overlap to beyond systolic loops, e.g., multiple systolic loops within the same application. Currently, even within systolic loops, we lose certain opportunities by requiring array uses to be uses of the covering definition. To capture these additional opportunities, we would need to extend our work to use either the GIVE-N-TAKE code placement framework by Hanxleden et al. [50] or the global analysis and placement of communication algorithm developed by Chabrabarti et al. [55].

With increasing hardware vector lengths, techniques that improve vectorization in common cases are especially important. In this thesis, we claimed that by employing polyhedral methods for analysis, transformation and code generation, we could generate high-performance vector code. To support this claim, we automatically identified and applied the “Semi-stencil” algorithm proposed by De La Cruz, Araya-Polo and Cela [7] to improve the performance of compiler-generated vector code for stencil
calculations. Applying the semi-stencil transformation on the smooth kernel grid improved the performance of a C-version of the smooth kernel by 30.81% on an Intel Ivy-Bridge core working on a 400³ grid. We also developed a compiler transformation that derives compact iteration spaces that reduce the need for control flow divergence to enable efficient vectorization. We showed that applying this transformation on Dilate kernel of Leukocyte application achieves a 24%-55% reduction in time across a range of image sizes. We showed the improvement on many architectural platforms and across different state-of-the-art compilers.

**Factoring in-place stencils into simple semantically equivalent kernels** We also introduced a solution to factor a 2D in-place stencil that appears in an Image Integral kernel into semantically equivalent composition of two scans. Since scans are easily vectorizable, we were able to achieve an 80% reduction in time for a reasonable input image size. However, we still need to explore the generality of this technique to multidimensional in-place stencils.

**Polyhedral transformation vs Abstract Syntax Tree manipulation** While we implemented our transformations through abstract syntax tree (AST) manipulations, we have realized that these transformations could have been done during code generation from Presburger relations using Omega [47]. Doing so would have greatly reduced the implementation complexity in crafting the prologue and epilogue loops of the semi-stencil.

**Counting solutions in the polyhedral model** We have successfully observed performance improvements on applying our transformations to the applications discussed in this thesis. However, in general, it is important to consider profitability metrics before applying the transformations for other applications, e.g., determining the number of convex loops created by eliminating the affine-if-conditionals and their
span prior to performing if-elimination. To do so, we have to count solutions to the
presburger relations developed at compile time. Crauss et al. [113] discuss one strat-

ey to obtain symbolic counts for presburger relations by employing Pick’s theorem
and Ehrhart polynomials. We plan to include this mechanism in the near future.

Programming Model In this thesis, we claimed that expressive programming con-
structs are necessary to describe different patterns of global data movement and to
allow for effective management of non-blocking operations. To support this claim,
we proposed a relaxed memory model for CAF 2.0, a Partitioned Global Address
Space language, that provided processor consistency guarantee for all operations ex-
cept synchronization operations, which provided varying levels of relaxed consistency.
In this model, we defined cofence. cofence is a synchronization construct that en-
ables one to manage local data completion of implicitly-synchronized asynchronous
operations. We explored subtle interactions between cofence, events, asynchronous
copies and collectives, and function shipping. We showed that using cofence in a
producer-consumer scenario, i.e., using local data completion rather than local oper-
ation completion, yields superior performance.

We exposed succinct communication constructs such as TILT, CSHIFT, SUM and
SPREAD in HPF that help express complex data flow patterns. We automatically
derived high-performance point-to-point and collective communication interactions
for each of these constructs. Thus, we have shown that automatically realizing the
communication needed for expressive programming constructs that capture complex
data flow patterns is possible. We argue that these constructs improve programmer
productivity since combining the expression of a complex data movement pattern
with distributed-memory communication increases code complexity.
The gap between complex parallel system designers, whose goal is to improve power and thermal efficiency, and the programmer, whose goal is to achieve high application performance with minimal effort, is bridged only by expressive programming models and sophisticated compiler technology. Enhancements in programming models and compiler technology are the need of the hour and especially so, with the anticipated demise of Moore’s law in the coming decade.
Appendix A

A.1 Transformational Intrinsic Functions

Here, we present definitions of transformational intrinsic functions used by Maunam in the global view version of 2.5D MM.*

A.1.1 CSHIFT

CSHIFT(a, s[, d]) is a transformational intrinsic function introduced in Fortran 90 that performs a circular shift on the elements of array a of rank $n \geq 1$ along dimension d by s places.

- a is an array of type integer, real, or complex.
- s is a scalar or an array of type integer. If s is an array and a is of shape $(r_1, r_2, ..., r_n)$, then s must be of shape $(r_1, r_2, ..., r_{d-1}, r_{d+1}, ..., r_n)$. A positive shift value indicates a left shift.
- Optional parameter d is a scalar of type integer with a value in the range 1..n. If d is omitted, its value defaults to 1.
- The return value of CSHIFT has the same type and rank as a. If s is a scalar, then each rank one section along dimension d is circularly shifted by s. If s

*Maunam’s implementation currently lacks support for an array-based shift specification for CSHIFT and a mask for SUM. Neither of these features is needed for 2.5D MM.
is an array, then each rank one section $a(i_1, ..., i_{d-1}, i_{d+1}, ..., i_n)$ is circularly shifted by $s(i_1, ..., i_{d-1}, i_{d+1}, ..., i_n)$.

A.1.2 SPREAD

SPREAD($a$, $d$, $c$) is a transformational intrinsic function introduced in Fortran 90 that assembles $c$ copies of $a$ into an array.

- $a$ is an array of rank $n \geq 0$ of type integer, real, or complex. A scalar is treated as a rank zero array.
- $d$ is a scalar of type integer with a value in the range $1..n + 1$. $d$ indicates the dimension of the result along which $a$ should be replicated.
- $c$ is a scalar of type integer that indicates the number of copies to create along dimension $d$ of the result.
- The return value of SPREAD has the same type as $a$.

A.1.3 SUM

SUM($a[, d[, m]]$) is a transformational intrinsic function introduced in Fortran 95 that performs a sum reduction of the elements of an array $a$ along one or all dimensions, according to a mask $m$.

- $a$ is an array of type integer, real, or complex.
- Optional argument $d$ is a scalar of type integer with a value in the range $1..n$, where $n$ is the rank of $a$.
- Optional parameter $m$ is of type logical and is either a scalar or an array of the same shape as $a$. If $m$ is omitted, its value defaults to true.
• The return value of \texttt{SUM} has the same type as \texttt{a}. If \texttt{d} is omitted, then \texttt{SUM} returns a scalar that is the sum of all elements in \texttt{a}. Otherwise, \texttt{SUM} returns an array of rank \( n - 1 \), where \( n \) is the rank of \texttt{a}, with a shape congruent to that of \texttt{a} with dimension \texttt{d} omitted. The sum includes elements where \texttt{m} evaluates to true.

\textbf{A.1.4 TILT}

\texttt{TILT}(\texttt{a}, \texttt{d}, \texttt{s}, \texttt{c}) is a transformational intrinsic function that performs a wrap-around skewing transformation on rank-one sections along dimension \( t \) of an array of rank \( n \geq 2 \). Our \texttt{TILT} intrinsic was inspired by a similar operation proposed for Rubik [114]—a tool to map application tasks to processor topologies.

• \texttt{a} is an array of type \texttt{integer}, \texttt{real}, or \texttt{complex}.

• \texttt{d}, a scalar of type \texttt{integer} with a value in the range 1 .. \( n \), indicates the dimension of \texttt{a} along which the skewing transformation will occur.

• \texttt{s}, a scalar of type \texttt{integer} with a value in the range 1 .. \( n \), \( s \neq d \), indicates the dimension with respect to which dimension \texttt{d} will be skewed.

• \texttt{c}, a scalar of type \texttt{integer}, is the skew coefficient.

• The return value has the same type and rank as \texttt{a}. \texttt{TILT} independently skews each 1D array section

\[
\texttt{a}(i_1, \ldots, i_{d-1}, : , i_{d+1}, \ldots, i_n)
\]

of the input array with respect to its coordinate \( i_s \) in dimension \texttt{s}. Element \( i \in [1, \text{size}(\texttt{a}, \texttt{d})] \) of the result for this section is \( \texttt{a}(i_1, \ldots, i_{d-1}, 1 + \text{modulo}(i + i_s \ast \texttt{c}, \text{size}(\texttt{a}, \texttt{d})), i_{d+1}, \ldots, i_n) \).
Bibliography


[110] Intel Corporation, “IA64 volume 2: System architecture, revision 2.3.”

