Automatic Parallelization of Loops with Data Dependent Control Flow and Array Access Patterns

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By
Mahesh Ravishankar, B.Tech, M.S.
Graduate Program in Computer Science and Engineering

The Ohio State University
2014

Dissertation Committee:
Prof. P. Sadayappan, Advisor
Prof. Atanas Rountev
Prof. Gagan Agrawal
With the era of increasing clock speeds coming to an end, parallel computing architectures have now become main-stream. Due to the wide range of architectures available today that can be used to exploit parallelism, ranging from multicore CPUs, to GPGPUs, to distributed memory machines; adapting applications for efficient execution on all these architectures poses a significant challenge.

Scientific computing applications exhibit significant coarse-grained parallelism. Domain experts have exploited this to target distributed memory machines through the use of Message Passing Interface (MPI) libraries. Many such applications have been shown to scale to hundreds of thousands of processors. While powerful, programming in MPI is tedious and error prone, with significant portion of the parallelized application dedicated to managing communication and synchronization between the processes. Developing compiler techniques that can automatically generate parallel distributed memory versions of such codes is challenging due to the presence of data-dependent control flow and data access patterns. In this thesis we develop compiler algorithms for automatic parallelization of a class of computations common to many scientific applications. Under the inspector/executor paradigm, the generated code partitions and executes the computation in load-balanced manner, while reducing the communication costs. This approach is further enhanced by developing a framework capable of expressing both affine and non-affine parts of the code. This enables the
use of polyhedral compilation tools to analyze parts of the computation which can be completely characterized statically. The effectiveness of the developed approach is demonstrated on several benchmarks and real-world applications.

Image processing applications on the other hand exhibit significant fine-grained parallelism and are well suited for architectures with Single-Instruction Multiple Data (SIMD) processing units like the vector processing units on CPUs or special graphics processing units like the NVIDIA Kepler GPU [1]. Such architectures can be targeted either through hardware specific intrinsics (like SSE/AVX to target vector units), or through language support for writing device-specific code (like CUDA to target NVIDIA GPUs). State of the art compilers like ICC [2], Pluto [3], etc. can automatically implement architecture specific optimizations for programs written in low-level languages like C/C++. While effective, these approaches still require significant effort from the application developers. Programs written this way are less portable and difficult to maintain. Domain Specific Languages offer a convenient abstraction, allowing programmers to specify the computation at a high-level, while still allowing the compiler to use the semantics of operations to generate efficient code to target multiple architectures. In this thesis, we develop Forma, a DSL that provides a convenient syntax to specify many common image processing pipelines in a succinct manner. The compiler backend can generate code to target both multicore CPUs with SIMD units and NVIDIA GPUs, while making use of device specific features like texture units on GPUs. These ease of programming in Forma is demonstrated by porting complex image processing pipelines like Laplacian Pyramids [4] and Exposure Fusion [5]. The performance of the generated code is compared against a state of the art DSL for image processing, Halide [6].
ACKNOWLEDGMENTS

I would like to begin by expressing my deepest gratitude to my advisor, Prof. P. Sadayappan. I am indebted to him for taking me on as a graduate student in spite of my limited background in Computer Science, and for being patient with me till I got upto speed. I thank him for making time for me whenever I needed it, in spite of his busy schedule, and for all his technical input over the course of my graduate studies.

I would also like to thank Prof. Atanas Rountev for his invaluable inputs on formalization and presentation of all aspects of my graduate work. While there is still room for improvement, I have learnt a lot from his perspective. I would also like to thank Dr. Louis-Noël Pouchet, Prof. J. Ramanujam and Prof. Gagan Agrawal for their inputs and encouragement.

One of the best experiences I have had during my graduate studies was my stint at NVIDIA at Redmond in Summer of 2013. I would like to thank Vinod Grover and Justin Holewinski for their contributions to Forma along with all my colleagues, Sean Lee, Thibaut Lutz and Tyler Sorenson, for their help during my time there. I look forward to working with them over the coming years.

I would also like to thank other graduate student who have worked with me at OSU, specially, John Eisenlohr for his time and effort with parallelizing OLAM and Roshan Dathathri for his contribution towards the end. I thank my colleagues Qing-peng Niu, Kevin Stock, Martin Kong, Humayun Arafat, Naznin Fauzia, Venmugil
Elango, Pai-Wei Lai, Naser Sedghati, Arash Ashari, Tom Henretty, James Dinan and Sanket Tavarageri, for all their help. I would also like to thank Dr. Sriram Krishnamoorthy at PNNL for mentoring me during my stint there.

Finally, I won’t be able to put in words my gratitude for the support and encouragement of my wife, Haritha Mathsyaraja, my parents, Ravishankar and Seethalakshmi, my brother, Shashidhar and all the great friends I have made at OSU. A special shout-out to all members of the Columbus chapter of Association for India’s Development, who have inspired me by their selfless commitment to address the needs of others.
VITA


September 2007—June 2009 . . . . . . . . . . . . . . . . M.S, Mechanical Engineering, The Ohio State University, Columbus, OH, USA.

September 2007—August 2008 . . . . . . . . . . . . . . . . University Fellow, The Ohio State University, Columbus, OH, USA.

August 2008—June 2009 . . . . . . . . . . . . . . . . . . . . Graduate Research Associate, Dept. of Mechanical Engineering, The Ohio State University, Columbus, OH, USA.

June 2009—Present . . . . . . . . . . . . . . . . . . . . . . . . Graduate Research Associate, Dept. of Computer Science & Engineering, The Ohio State University, Columbus, OH, USA.

July 2011—September 2011 . . . . . . . . . . . . . . . . . . . Graduate Intern, IS&T Summer Design School, Los Alamos National Laboratory, Los Alamos, NM, USA.

May 2012—August 2012 . . . . . . . . . . . . . . . . . . . . Graduate Intern, CUDA Compiler Group, NVIDIA, Redmond, WA, USA.

PUBLICATIONS

Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev and P. Sadayappan,
Code Generation for Parallel Execution of a Class of Irregular Loops on Distributed Memory Systems. In The International Conference for High Performance Computing, Network, Storage and Analysis (SC), Salt Lake City, Utah, USA. November 16–20, 2012.

Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev and P. Sadayappan,
Automatic Parallelization of a Class of Irregular Loops for Distributed Memory Systems. Accepted in ACM Transactions on Parallel Computing.

Naznin Fauzia, Venmugil Elango, Mahesh Ravishankar, J Ramanujam, Fabrice Rastello, Atanas Rountev, Louis-Noël Pouchet and P. Sadayappan,

Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankar, Naznin Fauzia, Louis-Noël Pouchet, Atanas Rountev and P Sadayappan,

Jeff Willert, C.T.Kelly, D.A. Knoll, Han Dong, Mahesh Ravishankar, Paul Sathre, Michael Sullivan and William Taitano,

Giridhar Srinivas Murthy, Mahesh Ravishankar, Muthu Manikandan Baskaran and P.Sadayappan,
Optimal Loop Unrolling for GPGPU Programs. In International Parallel & Distributed Processing Symposium (IPDPS), Atlanta, GA. April 19–23, 2010.

**FIELDS OF STUDY**

Major Field: Computer Science and Engineering

Studies in High Performance Computing: Prof. P. Sadayappan
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xv</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xv</td>
</tr>
</tbody>
</table>

Chapters:

1. Introduction                               | 1    |
   1.1 Distributed Memory Parallelization of a Class of Irregular Loops | 4    |
   1.2 DSL for Image Processing Applications  | 8    |
   1.3 Document Structure                      | 10   |

2. Related Work                               | 12   |
   2.1 Run Time Dependence Analysis and Parallelization of Irregular Loops | 12   |
   2.2 Domain-Specific Languages and User-Guided Approaches for Parallelization | 16   |
   2.3 DSLs for Stencil and Image Processing Applications | 17   |
5. Distributed Memory Code-generation for Irregular-Outer Regular-Inner Computations

5.1 Motivation ........................................... 83
5.2 Representing Irregular Computation using Sets and Maps ......................... 85
  5.2.1 Integer Sets ..................................... 85
  5.2.2 Set Slicing ..................................... 87
  5.2.3 Integer Maps .................................... 89
5.3 Partitioning the Computation .................................. 90
  5.3.1 Partitioning the Iteration Space ................................ 90
  5.3.2 Data Partitioning and Ghost Communication ........................... 92
5.4 Syntactic Structure of Target Loops .................................. 94
5.5 Generation of Inspector and Executor Code .................................. 96
  5.5.1 Computing the Local Data Space of Arrays : Inspector Code 97
  5.5.2 Executing the Iteration Space Slices : The Executor Code ........... 103
5.6 Generating Array Index Expressions for the Executor Code ..................... 107
  5.6.1 Access Graphs .................................... 108
  5.6.2 Grouping Nodes of the Access Graph ................................. 111
  5.6.3 Modifying Index Expressions in the Executor Code .................... 114
5.7 Case Studies .......................................... 116
  5.7.1 Poisson Equation Solver on Multigrids ................................ 116
  5.7.2 Affine computations ................................ 119
  5.7.3 Irregular Computations .................................. 120

6. Forma : A DSL for Image Processing Pipelines .................................. 124

6.1 Syntax of Forma Programs ......................................... 124
  6.1.1 Stencil Specification .................................. 129
  6.1.2 Cropping and Padding Operations ................................ 133
  6.1.3 Compose Operation .................................... 134
  6.1.4 Vector Functions ..................................... 137
  6.1.5 Loops and Qualified Variables ................................ 137
6.2 Optimization Across Stages of Image Processing Pipelines ..................... 138
  6.2.1 Fusing Stages of a Pipeline ................................ 140
6.3 Code Generation .......................................... 143
  6.3.1 Targeting CPUs ..................................... 143
  6.3.2 Targeting GPUs ..................................... 144
  6.3.3 Interface to the generated code ................................ 145
6.4 Case Studies and Performance Evaluation ..................................... 146
  6.4.1 Simple Stencils ..................................... 146
  6.4.2 Camera Pipe ......................................... 147
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Typical grids used in computations</td>
<td>5</td>
</tr>
<tr>
<td>3.1 Syntactic structure of partitionable loops</td>
<td>21</td>
</tr>
<tr>
<td>3.2 Control flow and data-access patterns of iteration 2, 4, and 5, of loops i and k mapped to process 0</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Transformed iteration and data view</td>
<td>25</td>
</tr>
<tr>
<td>3.4 Syntactic structure of computation input to the detection phase</td>
<td>36</td>
</tr>
<tr>
<td>4.1 183.equake with ref input size</td>
<td>71</td>
</tr>
<tr>
<td>4.2 CG Kernel with hood.rb and tmt_sym.rb</td>
<td>73</td>
</tr>
<tr>
<td>4.3 P3-RTE on unstructured mesh</td>
<td>76</td>
</tr>
<tr>
<td>4.4 miniFE-1.1 with 100x100x100 grid</td>
<td>78</td>
</tr>
<tr>
<td>4.5 Impact of exploiting contiguity</td>
<td>79</td>
</tr>
<tr>
<td>4.6 OLAM atmospheric model</td>
<td>81</td>
</tr>
<tr>
<td>5.1 Syntactic structure of annotated loops</td>
<td>94</td>
</tr>
<tr>
<td>5.2 Indirection Array Access Graph</td>
<td>110</td>
</tr>
<tr>
<td>5.3 Comparison with Chapter 4</td>
<td>118</td>
</tr>
<tr>
<td>5.4 Comparison between the two proposed approaches</td>
<td>123</td>
</tr>
</tbody>
</table>
6.1 Grammar of Forma program . . . . . . . . . . . . . . . . . . . . . . . . . 126
6.2 Grammar of Forma programs - contd . . . . . . . . . . . . . . . . . . 127
6.3 Basic Exposure Fusion algorithm . . . . . . . . . . . . . . . . . . . . . . 136
6.4 DAG representation internal to Forma . . . . . . . . . . . . . . . . . . 140
6.5 Multiple passes of Fusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 142
6.6 Fused computation DAG for exposure fusion algorithm . . . . . . . . 143
6.7 Inputs used for the Exposure Fusion [5] . . . . . . . . . . . . . . . . . . 152
6.8 Effect of improper boundary conditions . . . . . . . . . . . . . . . . . . 153
A.1 Coarse and Fine Boxes used in AMR . . . . . . . . . . . . . . . . . . . . 161
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Irregular-Outer Regular-Inner benchmark : Poisson equation solver</td>
</tr>
<tr>
<td>5.2</td>
<td>Affine Computations: Time-tiled FDTD and Jacobi2D</td>
</tr>
<tr>
<td>5.3</td>
<td>Irregular Application : 3D BTE Solver</td>
</tr>
<tr>
<td>6.1</td>
<td>Performance of Simple Stencils</td>
</tr>
<tr>
<td>6.2</td>
<td>Performance of Camera Pipe (ms)</td>
</tr>
<tr>
<td>6.3</td>
<td>Performance of Laplacian Pyramid on CPU/GPU (ms)</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance of Exposure Fusion on CPU/GPU (ms)</td>
</tr>
</tbody>
</table>
## LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckForPartitionableLoop(\mathcal{A})</td>
<td>37</td>
</tr>
<tr>
<td>CheckSeparation(\mathcal{A},I_A^I_S)</td>
<td>38</td>
</tr>
<tr>
<td>CodeGenHyperGraph(\mathcal{H},ss)</td>
<td>49</td>
</tr>
<tr>
<td>GenGuards((e))</td>
<td>50</td>
</tr>
<tr>
<td>CodeToInitializeArrays(ss,L,\mathcal{C})</td>
<td>61</td>
</tr>
<tr>
<td>GenerateExecutor(D,L,\mathcal{C},ss)</td>
<td>64</td>
</tr>
<tr>
<td>CodeGenInspectorExecutor(\mathcal{P})</td>
<td>67</td>
</tr>
<tr>
<td>GenerateInspector(\mathcal{A})</td>
<td>97</td>
</tr>
<tr>
<td>GenerateExecutor(\mathcal{A})</td>
<td>106</td>
</tr>
<tr>
<td>GroupNodes(G)</td>
<td>112</td>
</tr>
<tr>
<td>ChangeGroupNum((v,\text{group},\text{processed_set}))</td>
<td>113</td>
</tr>
</tbody>
</table>
LIST OF LISTINGS

1.1 Sequential conjugate gradient computation. . . . . . . . . . . . . . . . 7
1.2 5-pt 2D Jacobi stencil . . . . . . . . . . . . . . . . . . . . . . . . . . 9
1.3 A sampling filter . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
1.4 An extrapolation filter . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.1 Parallel conjugate gradient computation. . . . . . . . . . . . . . . . . 26
4.1 Phase I of the inspector. . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.2 Phase II of the inspector. . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.3 Phase III of the inspector. . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.1 MultiGrid Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
5.2 Sample SpMV Kernel . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.3 After pre-processing Listing 5.1 . . . . . . . . . . . . . . . . . . . . . . 96
5.4 Inspector Code Snippet for Listing 5.1 . . . . . . . . . . . . . . . . . 98
5.5 Executor Code for Listing 5.1 . . . . . . . . . . . . . . . . . . . . . . 104
6.1 Hello World in Forma . . . . . . . . . . . . . . . . . . . . . . . . . . . 128
6.2 Stencils in Forma . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
6.3 Stencils with non-unit scaling factor . . . . . . . . . . . . . . . . . . . . 131
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4 Extrapolation</td>
<td>132</td>
</tr>
<tr>
<td>6.5 Crop and Padding Operations</td>
<td>133</td>
</tr>
<tr>
<td>6.6 Compose Expression</td>
<td>134</td>
</tr>
<tr>
<td>6.7 Custom Boundary Conditions</td>
<td>136</td>
</tr>
<tr>
<td>6.8 Vector Functions</td>
<td>137</td>
</tr>
<tr>
<td>6.9 Laplacian Pyramid Computation</td>
<td>139</td>
</tr>
<tr>
<td>6.10 Laplacian Filter</td>
<td>141</td>
</tr>
<tr>
<td>A.1 Original loop-nests of AMR</td>
<td>163</td>
</tr>
<tr>
<td>A.2 Inspector Code to compute footprint - Loop 1</td>
<td>168</td>
</tr>
<tr>
<td>A.3 Inspector Code to compute footprint - Loop 2</td>
<td>169</td>
</tr>
<tr>
<td>A.4 Executor Code - Loop 1</td>
<td>170</td>
</tr>
<tr>
<td>A.5 Executor Code - Loop 2</td>
<td>171</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Even before the advent of multicore processors, developers of scientific computing applications were interested in exploiting inherent parallelism within their applications to either reduce the running times (typically in the order of days) or to handle problems involving larger data-sets. Distributed memory machines, where multiple connected processing units, each with its own memory space, exchange data and synchronize with each other through messages evolved as the dominant mode for exploiting this parallelism. The Message Passing Interface (MPI) [7] provides an API that applications developers could use to target distributed memory machines, allowing them to scale their applications to hundreds of thousands of processes. This mode of parallelism can be classified as Single Program Multiple Data (SPMD), where the same program is run on multiple nodes, but they operate on the data that is resident in the local memory space of each node. Such an execution model not only shortened their execution times from days to a matter of hours, but also allowed them to solve problems at more realistic scales.

Similar to scientific computing, image processing applications are inherently parallel. For example a filter could be applied to all pixels of an image in parallel. The difference here is that the parallelism is more fine-grained than in scientific computing
applications. Vector processing units were designed to exploit this kind of parallelism under the Single Instruction Multiple Data (SIMD) paradigm, where a single operation can be performed on multiple data elements simultaneously. The latest Intel MIC architecture [8] uses the AVX instruction set [9] where a single instruction operates on 16 32-bit data elements (or 8 64-bit data elements). NVIDIA GPUs provide an alternate way of exploiting this parallelism. Each GPU has thousands of cores (2048 on the latest Kepler GPU cards [1]), with each core operating on either a single pixel or a subset of pixels. The CUDA programming language [10], which extends C++ with some keywords, allows application developers to write code in an SPMD fashion. The programmer describes the computation on each core, referred to as threads, while using fine-grained synchronization mechanisms between a subset of threads, called a thread-block.

The advent of multicore processors, has brought parallel computing to the forefront. Unlike the pre-multicore era where an increase in clock speed automatically allowed applications to run faster, compute intensive applications have to effectively utilize all cores on the machine to achieve good performance. The OpenMP programming model [11] allowed developers to exploit coarse-grain parallelism within their applications by providing a convenient interface to distribute the task amongst the different cores.

These different approaches that target different architectures pose a significant challenge for application developers. For each architecture, it is necessary to understand the various factors that are critical to achieve good performance. For example, one has to understand the performance implications of unaligned loads while targeting vector instruction units, or the cost of uncoalesced accesses from memory on a GPU.
In addition, significant time and effort would be required to port an application to a different architecture, making it harder to maintain.

Many solutions have evolved to address these problems. The most powerful approach is to let a compiler automatically generate code specialized for each architecture, from programs written in a general purpose language like C/Fortran. Significant progress has been made on this front. Compilers like ICC [2] can automatically generate code to utilize the vector processing units on modern CPUs. These and other research compilers like Pluto [3] and PolyOpt [12] can also generate code to exploit coarse-grained parallelism by targeting multiple cores on the CPU. NVCC [13] automatically generates code that can run on GPUs using a program specified in CUDA. Compilers like Cetus [14] and Pluto can also generate MPI code from annotated C codes.

Domain Specific Languages (DSLs) are another approach that allow application developers to target multiple architectures. Here, the language provides constructs that allow programmers to provide a high-level description of the computation. At the same time, the DSL compiler can use the semantics of these constructs to generate efficient code for different architecture. Several DSLs have been developed for applications in various domains, like tensor contraction [15], image processing [6,16], mesh-based application [17], iterative stencil computations [18–20], etc. Most of these DSLs use this high-level description to target multiple architectures.

This thesis addresses the following two problems. The first approach targets computations that are common in many scientific computing applications, and proposes a technique to automatically generate distributed memory code from a given sequential
program. The second approach proposes a new DSL Forma, targeting image processing applications. The syntax of this DSL makes it easy to specify complex pipelines in a succinct manner, and is capable of generating code to target both multicore CPUs, and NVIDIA GPUs. The following sections describes properties of the computations targeted by the above approaches and the challenges presented by them.

1.1 Distributed Memory Parallelization of a Class of Irregular Loops

Many scientific computing applications entail solving differential equations over physical domains, usually having arbitrary geometries. To solve these equations, the underlying physical domain is discretized using a grid of cells. Physical quantities of interest (like velocity and pressure in fluid dynamics simulations) are associated with centers of cells, or with centers of the face between cells, or at corners of cells known as vertices. To obtain values of these quantities, Finite-Volume [21] or Finite-Element [22] methods are used to convert the differential equation into a set of linear equations which can be represented by the following matrix-vector equation.

\[
A \vec{x} = \vec{b}
\]  

(1.1)

In equation (1.1), vector \(\vec{x}\) represents all unknown values being computed (either at centers of cells, faces or at vertices). The matrix, \(A\) is known as the co-efficient matrix and its values depend upon properties of the grid used to discretize the domain. This matrix is usually sparse, with the number of non-zero elements being of the order \(O(n)\), where \(n\) is the size of \(\vec{x}\). The value of vector, \(\vec{b}\) depends on the values of other physical quantities that are input to the differential equations being solved.
Equation (1.1) can be solved using a wide-variety of techniques. *Explicit* methods involve using an initial guess for all unknown values. The value at a point in the grid (cell center, face-center or vertex) is computed based on values at surrounding cell centers, face centers or vertices. Repeated sweeps over the entire grid result in the values converging to the solution of the differential equation. Jacobi and Gauss-Seidel are examples of such explicit methods. Here, the matrix $A$ is not stored explicitly in memory. *Implicit* methods are used when all unknown values are solved together. This is similar to computing $A^{-1}b$, with the matrix $A$ stored in memory. Since matrix inversion is expensive, several iterative methods, like Krylov subspace methods, are used instead. For modeling transient behavior, equation (1.1), is solved for every time-step with values at one time-step depending on values from the previous time-steps.

In cases where *structured grids* are used for the discretization, like the one shown in Figure 1.1a, explicit solution schemes reduce to the use stencils operations since a simple algebraic relationship can be used to access values at neighboring points.
When using implicit solution schemes, the matrix $A$ has non-zero entries only along certain diagonals. As a result, codes that use structured grids have a control flow and array access pattern that can be deduced at compile time, since they depend only on program parameters or loop iterators. Such computations are referred to as affine computations. The polyhedral model [25], that forms the basis of state of the art compilers like Pluto and PolyOpt, can accurately model the runtime behavior of these codes statically. Several previous work have proposed optimization techniques that use this model to generate efficient codes to target multicore CPUs [26], distributed memory systems [27], [28], [29], [30] and GPUs [31], [32].

On the other hand, many popular scientific computing applications use unstructured grids (like the one shown in Figure 1.1b) to discretize the underlying physical domain. For such cases, while using explicit solution scheme, the value at neighboring entities are accessed through explicit maps. In implicit schemes, the matrix $A$ has no fixed patterns for the position of non-zeros elements. Consequently, the control flow and data access pattern of such applications depends on data structures that are populated at run time. Traditional compiler optimization techniques are not applicable to such codes directly.

Listing 1.1 is an example of such a computation. It shows two loops from the Conjugate Gradient method for solving linear equations. Loop $k$ computes the values of array $x$. In loop $i$, array $y$ is computed by multiplying matrix $A$ and the vector stored in array $x$. Since the matrix $A$ is sparse, several schemes have been developed to store such matrices in memory. One such scheme, is the Compressed Sparse Row (CSR) format. It uses three arrays, (say A, ia and col) to store the information about the structure of the matrix and its values. If the matrix has $n$ rows,
while (!converged ){
  //...Other computation not shown...
  for( k = 0 ; k < n ; k++ )
    x[k] = ...;
  //...Other computation not shown...
  for( i = 0 ; i < n ; i++ )
    for( j = ia[i] ; j < ia[i+1] ; j++ ){
      xindex = col[j];
      y[i] += A[j]*x[xindex];
    }
  //...Other computation not shown...
}

Listing 1.1: Sequential conjugate gradient computation.

- Array ia is of size $n + 1$, and its values are such that the non-zero elements of the $i^{th}$ row of the matrix are stored from position $ia[i]$ to position $ia[i+1]-1$ of array $A$.

- Array col has the same size as $A$, and for every non-zero element in $A$, $col$ stores its column number in the matrix.

The bounds of the inner-loop at line 7 depends on the value stored in array $ia$. The data access pattern of array $x$ depends on the value stored in array $col$. Since the value of these arrays are known only at run time, purely static techniques do not have enough information about the computation and have to be conservative in the optimizations that they can apply [33]. Arrays $ia$ and $col$ that affect the control-flow and data access pattern of the code are refered to as indirection arrays. Note that the elements of array $col$ accessed itself depend on values stored in array $ia$. Such a pattern is refered to as multiple levels of indirection.

In this thesis we propose an approach to automatically transform codes, that might potentially have multiple levels of indirection, to target distributed memory machines using a combination of compile time and run time analysis. Such an approach is
commonly referred to as the inspector/executor technique. The compiler algorithms developed here automatically generate an *inspector* that analyzes the control flow and data access pattern of arrays at run time. This analysis is used to partition the iterations and the data structures of the computation. The compiler also generates an *executor* which executes the partitioned computation on each node, using the data structures generated by the inspector.

### 1.2 DSL for Image Processing Applications

Many image processing applications are developed as a series of filters applied to images. Many of these filters can be expressed as discrete convolution operations. Equation (1.2) shows a 2D convolution operator, where the function $in$ represents a 2D image such that $in(i, j)$ returns the value of the pixel at point $(i, j)$. $G$ represents the convolution operator, and $out$ is the result of applying the convolution operator on the input image.

$$
out(i, j) = \sum_{m=-p}^{p} \sum_{n=-q}^{q} G(m, n) \times in(i - m, j - n) \tag{1.2}
$$

Such convolution operations are commonly expressed as stencil operations. Listing 1.2 shows an example of a convolution operation, where $G = \{0, -1, 0; -1, 4, -1; 0, -1, 0\}$. This convolution operation can be expressed as a 5-pt 2D Jacobi stencil. From this specification, it is evident that applying this stencil along the edges of the input image is invalid. Therefore, boundaries need to be handled separately.

Laplacian Pyramids, introduced by Burt and Adelson [4], represent another important class of computation in image processing. Here, two image pyramids are constructed from an input image, namely the *Gaussian pyramid* and the *Laplacian*...
out(i,j) = 4*in(i,j) - in(i-1,j) - in(i,j-1) - in(i,j+1) - in(i+1,j);

Listing 1.2: 5-pt 2D Jacobi stencil

temp(x,y) = 0.25 * g_lower(2*x-1,y) + 0.5* g_lower(2*x,y) + 0.25* g_lower(2*x+1,y);

g_upper(x,y) = 0.25 * temp(x,2*y-1) + 0.5* temp(x,2*y) + 0.25* temp(x,2*y+1);

Listing 1.3: A sampling filter

Pyramid. The input image forms the base of the Gaussian pyramid, $g_0$. Successive levels, $g_l, l \in [1, L - 1]$, are constructed from the image at the lower level through the reduce or interpolation operation described by Equation 1.3

$$g_l(i,j) = \sum_{m=-p}^{p} \sum_{n=-q}^{q} R(i,j) \times g_{l-1}(2 * i - m, 2 * j - n) \quad \forall l \in [0, L] \quad (1.3)$$

Listing 1.3 shows an example of the reduce operation with $R = \{0.25, 0.5, 0.25; 0.5, 1, 0.5; 0.25, 0.5, 0.25\}$ expressed as a sequence of two operations. As a result of this operation, the size of an image at successive levels of the Gaussian pyramid are reduced by half along each dimension.

The Laplacian Pyramid involves using the reverse of the reduce operation known as the expand operation. Equation 1.4 describes this operation

$$h_l(i,j) = \sum_{m=-p}^{p} \sum_{n=-q}^{q} R(i,j) \times g_{l+1}\left(\frac{i - m}{2}, \frac{j - n}{2}\right) \quad \forall l \in [0, L - 1] \quad (1.4)$$

The Laplacian pyramid is computed by expanding each level of the Gaussian pyramid and subtracting it with its immediately lower level, i.e. $l_l(i,j) = g_l(i,j) - h_l(i,j) \forall l \in [0, L - 1]$. Listing 1.4 shows an example of the expand operation expressed
in terms of stencils, with the same $R$ as above. Depending on whether the point at
the output being evaluated is odd or even, different stencils are applied to the input
image.

The syntax of Forma allows application developers to express such filters in a con-
cise manner. The compiler uses this description to generate code that is customized
for both multicore CPUs with vector processing units, and GPGPUs. Tedious de-
tails of the computation like allocating buffers of appropriate sizes, handling boundary
conditions while applying a stencil, copying data to and from the device while tar-
getting GPUs, etc are handled automatically by the DSL compiler. Forma also allows
multiple filters to be chained together to express the entire image processing pipeline.
This high-level description of the pipeline allows the compiler to accurately capture
the producer-consumer relationships between different stages of the pipeline. Conse-
quently, inter-stage optimizations can be applied without being constrained by low
level implementation details.

1.3 Document Structure

This document is organized as follows. Chapter 2 discusses previous compiler ap-
proaches and DSLs developed for parallelization of irregular applications, along with
DSLs that target stencils and image processing applications. Chapter 3 provides an

Listing 1.4: An extrapolation filter

```
1 temp(2*x,y) = 0.75*image(x,y) + 0.25*image(x-1,y);
2 temp(2*x+1,y) = 0.75*image(x,y) + 0.25*image(x+1,y);
3 output(x,2*y) = 0.75*temp(x,y) + 0.25*temp(x,y-1);
4 output(x,2*y+1) = 0.75*temp(x,y) + 0.25*temp(x,y+1);
```
overview of the approach used for distributed memory parallelization of irregular applications developed in this thesis. It also describes the structure of program regions that are targeted for transformation along with a compiler algorithm to automatically detect them. Chapter 4 describes the compiler algorithm that implements the transformation to target distributed memory systems. The performance of the generated code is evaluated on several benchmarks and a real-world application. Chapter 5 describes a unified framework to represent both affine and non-affine parts of the codes. This representation is used to reduce overheads of run time inspection while preserving program properties of the affine parts of the code, which in-turn can be optimized by traditional compiler techniques. Chapter 6 describes the syntax and semantics of Forma programs, the optimizations implemented within the Forma compiler and compares the performance obtained for several complex image processing pipelines with Halide [34], another state of the art DSL for such applications.
CHAPTER 2

Related Work

2.1 Run Time Dependence Analysis and Parallelization of Irregular Loops

One of the earliest work on Inspector/Executor techniques was by Koelbel and Mehrotra [35]. The user was incharge of specifying the data and iteration distribution (like block, block-cyclic). The compiler framework would then generate an executor that split the iteration space into those that need only local data and those that access non-local data. The non-local iterations would predicate every array access to check if the accessed location is local or not. The inspector code computed the set of indices that are local to a process and those that need to be communicated to other processes.

Saltz et.al. [36] used the inspector/executor approach to analyze def-use patterns between iterations at run time, using a topological sort of iterations to create wavefronts of parallelism. In the executor code, busy waits were used to block the execution of iterations which access data to be received from other processes. This approach added considerable overhead to the executor code itself. Their follow-up work in Das et.al [37] and [38] used library/directive approach to partition data across processes,
manage communication of shared array elements and create local indirection arrays that allow the executor code to efficiently access appropriate elements in the local array. All the communication was done before the executor code, eliminating the need for busy waits. In [39] the same group developed compiler and runtime support for communication schedule generation of block structured applications, and could not target irregular applications. Hanxleden et.al. [40] proposed a data-flow analysis to eliminate redundant communication of array elements whose values haven't changed. Sharma et.al. [41] could handle adaptive applications, where the values in indirection arrays changed infrequently. The target application of all these approaches could have only one-level of indirection. In [42] and [43], Das et.al. proposed a slicing approach to handle codes with multiple levels of indirection. Here, a compile time analysis creates slices from the original code, such that each slice resolved one level of indirection. The distributed memory execution utilizes the owner-computes paradigm which might result in redundant computation and communication in cases where multiple iterations of a distributed loop update the value of the same data element, or the same iteration updates values of multiple data elements. Agrawal et.al. [44], developed inter-procedural analysis to compute optimal placements of code to generate communication schedules and to do the communication by developing an inter-procedural version of partial redundancy elimination. Lain et.al. [45, 46] proposed a technique that could exploit contiguity of accesses to reduce communication costs and inspector overheads to a limited extent. Zopetti et.al. [47] used the inspector/executor approach to target shared memory systems using a light-weight inspector. The approach relied on overlap of computation and communication for efficient parallel execution, but the
volume of communication was substantially high. On modern distributed memory machines this would be a considerable bottleneck.

Huo et.al. [48] explored the use of inspector/executor techniques to parallelize reductions on GPUs, for mesh-based applications and could handle one-level of indirection. This was extended in [49] to target heterogeneous CPU/GPU machines. Ravi et.al. [50] generalize this to execute general reduction-like computations on such architectures, but rely on a replication of all data structures that are updated using the reduction operators. ALTER [51] is a similar framework that allows user to specify reductions which are commutative and associative along with variables whose values depend on a previous snapshot of the execution, making use of lessons from database systems to parallelize reductions on shared memory systems.

The inspector/executor method can also be used to check for dependences at run-time. Rauchwerger et.al [52] developed techniques to track dependences arising due to irregular accesses to arrays by using shadow arrays. In absence of any dependences, the iterations of a loop would be executed in parallel on a shared memory machine. Chen et.al [53] tracked dependence in a similar way in context of a distributed memory system. The iterations of a loop are distributed across the processes, with a ticketing system used to stall iterations till their dependences are satisfied. Rus et.al. [54], tracked inter-iteration dependences through a combination of static and dynamic techniques. The compile time analysis computes a set of conditions that an inspector could check to prove that a loop is parallel. The fall back would be a complete run time analysis described in [52]. In [55] the authors propose an ordering of the conditions in increasing order of cost so as to reduce the overhead for the dependence analysis.
Basumallik and Eigenmann [56, 57] presented techniques for translating OpenMP programs with irregular accesses into code for distributed-memory machines. The translation scheme employed would result in some arrays being replicated on all processes. In addition, to satisfy cross-iteration dependences, these shared data-structures have to be communicated across all processes, resulting in significant communication volume. While the cost of this communication is reduced by identifying monotonic access pattern and employing communication-computation overlap, such an execution model would result in higher footprints and high communication volume of the generated distributed memory code.

Zhuang et.al. [58] use the inspector-executor model to track dependences and execute iterations whose dependence has been satisfied in parallel. The approach consists of using a slice of the original computation as a means to track dependence. A master thread executes the original computation. As the iterations are completed, future iterations that depend on these are now free to execute and are launched in parallel on worker threads. The master thread skips the execution of these iterations.

Ottoni et.al. [59] and Huang et.al [60], use a combination of control flow and data flow analysis to build a program dependence graph. This is used to build a DAG of strongly connected components. These are then executed in a pipelined manner to exploit coarse-grained parallelism. Building on this, Kim et.al. [61] developed a run time system that used profiling information to speculatively remove dependence edges between iterations of a loop, and to handle mis-speculations across multiple nodes of a cluster. Huang et.al. [62] proposed the DOMANY construct that generates loop whose iterations use fine-grained synchronization constructs to respect dependence. Interestingly, their approach is also capable to exploiting parallelism across multiple
invocation of the same loop. In [63], the authors use an approach similar to [36] to partition iterations of a loop into groups for execution on NVIDIA GPUs.

The sparse polyhedral framework [64–66] extends polyhedral techniques to handle irregular computations through use of uninterpreted function symbols (UFS). Such an abstraction was used to develop transformations like tiling for irregular computations, using the inspector/executor approach. The abstraction developed here can be used to represent irregular computations with multiple levels of indirection. Since the framework relies on polyhedral code-generation techniques, certain properties of uninterpreted function symbols have to be asserted at compile time (invertibility of UFS, for example). Such properties might not hold for many irregular applications. Venkat et.al. [67] use this abstraction to implement loop coalescing. Their compiler analysis was able to utilize asserted properties of UFS to simplify expressions involving these symbols.

### 2.2 Domain-Specific Languages and User-Guided Approaches for Parallelization

Several DSLs, and active libraries have also been developed that allow programmers to expose the parallelism inherent to the computation to the compiler/library. Following is a description of a few such approaches.

Liszt [17] is an example of such a DSL that provides semantics for user to specify computations over unstructured meshes, and has been demonstrated to target multiple architectures. But the semantics of Liszt allows specification of only one-level of indirection. Therefore, codes like CSR which deal with sparse-matrices are not readily expressible in Listz.
OP2 [68] provides a similar functionality through library support. The API provides means to specify the computation which is then executed through codes that are tuned to the target architecture. The OP2 framework assumes that the computation specified is parallel, but allows for updating values stored in arrays. They too support only one-level of indirection.

DAGuE [69], on the other hand, is a DSL which can be used to express the computation as a set of tasks with dependence between tasks to ensure the ordering. A run time system constructs the DAG of the computation and schedules tasks in a distributed memory framework while managing data-movement automatically.

Infrastructures like Cilk [70] and the Galois Framework [71], which targets SMP machines, provides constructs to let the user specify tasks and ordering between the tasks. This is used by the run time framework to schedule work on the different threads. It allows the user to specify a partitioning of the data, or to specify if the data is to be shared between the threads.

In all the above approaches the effort required from the application developer range from a re-implementation of the whole application (Liszt), to reimplementation of parts of the applications (OP2,Galois and DAGuE) to just providing hints to the compiler.

2.3 DSLs for Stencil and Image Processing Applications

Due to the wide-spread usage of stencil computations, many DSLs have been developed to target stencil computations. The Pochoir [19] stencil compiler embeds a DSL within C++. The DSL syntax can be parsed by standard C++ compilers using a template library, or by the Pochoir compiler that generates high-performance C++
code using cache-oblivious algorithms to generate efficient tiling. SDSL [18, 72] is a stencil DSL that addresses the issue of time-tiling using the approach described in [73] to target both SIMD architecture and GPGPUs. Patus [20] allows the programmer to decouple the stencil specification from the scheduling strategy for efficient execution. The programmer can also specify auto-tuning parameters for the compiler. Partans [74] is another stencil DSL that focuses on optimizations for multi-GPUs. Other DSLs like Diderot [16] and Spiral [75] raise the level of abstraction even higher by allowing the programmer to specify the computation in terms of mathematical operations. Most of these DSLs are either geared towards specification and optimization of individual stencil computations. While SDSL creates time-tiled code for a sequence of stencils, the focus of this DSL are computations that are iterative in nature. In this thesis we focus on image processing pipelines which are not iterative, but are best expressed as a Directed Acyclic Graph (DAG).

Halide [6] is a code-generation framework that targets the same class of application as this thesis. Embedded within C++, it provides a functional abstraction for images with the value of a pixel viewed as the result of a function application. Intermediate images are functions, and image processing pipelines are formed by composing expressions of functions. Halide uses many optimization techniques like spatial tiling and sliding-window optimizations to generate high-performance code on CPUs and GPUs, but relies on the user to specify the parallelism and tiling scheme. For complex applications, it can be quite challenging to specify an optimal schedule, especially across multiple functions. While Halide can auto-tune for various parameters like tile-sizes, unroll factors, etc. for many applications the search space can become intractable.
AlphaZ [76] is an interesting DSL that allows the user to setup producer-consumer relationships between points in an iteration space. While targeting a more general class of application, including linear algebra computations, it utilizes the polyhedral machinery to implement iteration space transformations that can be tuned for each target architecture. This DSL is able to target multicore CPUs, GPUs, and distributed memory machines. While semantics of Forma developed here are inspired by such abstractions, the dependences are not tracked at the level of individual iterations. Forma aims to capture dependence between different stages of a pipeline, and optimize across these stages.
CHAPTER 3

Partitioning Irregular loops for Distributed Memory Execution

In this chapter we describe the computation targeted for distributed memory parallelization. We describe a compiler approach that uses the inspector/executor paradigm to generates a parallel inspector code used to partition the computation at runtime in a load balanced manner while minimizing communication. None of the initial data structures of the original computation are replicated on any of the processes. While many of these concerns have been addressed by previous works, none of the above mentioned approaches handle the range of applications targeted by the techniques developed here. Unlike previous approaches, the executor code generated maintains program properties that affect the per-node performance of the distributed memory code. Finally we discuss the compiler algorithm to automatically detect code regions that are amenable to such transformations.

3.1 Targeted Computations

As described in Section 1.1, we target computations where the control flow and data access pattern depends that are themselves data-dependent (Listing 1.1). The arrays whose values are used to compute loop-bounds, conditionals or array index
expressions are referred to as indirection arrays. All other arrays used in the computation will be referred to as data arrays \((x, y, \text{ and } A \text{ in the example})\). Similarly, scalars can be classified as data scalars or indirection scalars. The latter are those whose values are directly or indirectly used to compute loop bounds, conditionals, or index expressions of arrays. All other scalars (apart from loop iterators referenced in the loop body) are treated as data scalars.

\[
\text{(Start)} ::= \langle \text{ElementList} \rangle
\]
\[
\langle \text{ElementList} \rangle ::= \langle \text{Element} \rangle | \{ \langle \text{ElementList} \rangle \langle \text{Element} \rangle \}
\]
\[
\langle \text{Element} \rangle ::= \langle \text{IAssignment} \rangle | \langle \text{DAssignment} \rangle | \langle \text{Loop} \rangle | \langle \text{If} \rangle
\]
\[
\langle \text{Loop} \rangle ::= \text{for}\ ' \langle \text{Iterator} \rangle ' = ' \langle \text{IExpr} \rangle ' ; ' \langle \text{Iterator} \rangle ' < ' \langle \text{IExpr} \rangle ' ; ' \langle \text{Iterator} \rangle ' ++ ' \}
\text{(ElementList)}
\]
\[
\langle \text{If} \rangle ::= \text{if} ' \langle \text{IExpr} \rangle ' \} \langle \text{ElementList} \rangle \text{else} \langle \text{ElementList} \rangle
\]
\[
\langle \text{IAssignment} \rangle ::= \langle \text{IScalar} \rangle \langle \text{AssignOp} \rangle \langle \text{IExpr} \rangle ' ;
\]
\[
\langle \text{DAssignment} \rangle ::= \langle \text{DScalar} \rangle \langle \text{AssignOp} \rangle \langle \text{DExpr} \rangle ' ;
\]
\[
\langle \text{IExpr} \rangle ::= \text{Side-effect-free expressions of} \langle \text{BasicIExpr} \rangle
\]
\[
\langle \text{DExpr} \rangle ::= \text{Side-effect-free expressions of} \langle \text{BasicDExpr} \rangle
\]
\[
\langle \text{BasicIExpr} \rangle ::= \langle \text{IScalar} \rangle | \langle \text{Iterator} \rangle | \langle \text{IArray} \rangle \text{' ['} \langle \text{IExpr} \rangle \text{' ]'}
\]
\[
\langle \text{BasicDExpr} \rangle ::= \langle \text{DScalar} \rangle | \langle \text{DArray} \rangle \text{' ['} \langle \text{IExpr} \rangle \text{' ]'}
\]
\[
\langle \text{AssignOp} \rangle ::= '=' | '+=' | '*=' | \ldots
\]

Figure 3.1: Syntactic structure of partitionable loops

Therefore, we target a class of computations that are more general than affine computations, where the loop bounds, conditionals of if's, and array-access expressions are affine functions of loop iterators and read-only program parameters. For such codes, the control flow and data access patterns can be fully characterized at
compile time. For computations like the one shown in Listing 1.1, affine techniques have to be conservative and over-approximate the data dependences and control flow. Here, we target a generalized class of computations, in which loop bounds, if conditionals, and array-access expressions are arbitrary functions of iterators, parameters, and values stored in read-only indirection arrays. Further, values in these indirection arrays may themselves be accessed through other indirection arrays, resulting in multiple levels of indirection.

Figure 3.1 describes a grammar for the computations targeted by the approach described here. Section 3.5 describes the algorithm to detect computations that satisfy this grammar. Computations can consist of loops, conditional statements, and assignment statements. Each loop must have a unique iterator (represented by the terminal \langle \text{Iterator} \rangle), which is referenced only in the loop body and is modified only by the loop increment expression with unit increment. The terminals \langle \text{IArray} \rangle and \langle \text{DArray} \rangle represent indirection arrays and data arrays, respectively. Similarly, the terminals \langle \text{IScalar} \rangle and \langle \text{DScalar} \rangle represent indirection scalars and data scalars, respectively. Loop bounds, conditional expressions, and array index expressions are side-effect-free expressions built from iterators, indirection scalar variables, indirection array access expressions, and various operators (e.g., arithmetic and boolean) and functions (e.g., from libraries). An assignment could use the standard assignment operator, or an update operator of the form \text{op}=, where \text{op} is a commutative and associative binary operator. It is also assumed that no two arrays have overlapped regions of memory in the original computation. Asserting this at compile time is outside the scope of the present work.
Within this class, we target loops that are parallel, except for loop-carried depen-
dendencies due to reductions of data scalars or data array elements using operators
that are associative and commutative. Such loops will be referred to as partitionable
loops—they can be partitioned for parallel execution on distributed memory systems.
Loops $i$ and $k$ in Listing 1.1 are examples of such loops. If a partitionable loop is
nested inside another partitionable loop, only the outer loop is parallelized.

The proposed source-to-source transformation scheme is well suited for computa-
tions that have a sequence of partitionable loops enclosed within an outer sequential
loop (usually a time-step loop or a convergence loop), such that the control flow
and array access patterns are not modified within this loop. Such computations are
common in many scientific and engineering domains. Furthermore, with this code
structure, the inspector can be hoisted outside the outer loop.

3.2 Partitioning the Computation

Here we present an overview of the approach used to achieve distributed memory
parallelization of the targeted class of application, using Listing 1.1 as an example.

The goal of the transformation scheme is to parallelize the computation by parti-
tioning both the iterations of loops $i$ and $k$, and the data stored in arrays $A$, $x$ and
$y$ among the set of given processes. Figure 3.2 shows sample values for all arrays
in the computation where the value of $n$ is 6. Figure 3.3 illustrates this partitioned
execution; details of which are explained here. Listing 3.1 is the code that is executed
on each process during the parallel execution, i.e., the executor code.
3.2.1 Partitioning the Iterations

In Listing 1.1, there exists a producer-consumer relationship between the two loops due to array x. In a parallel execution of both loops, communication would be required to satisfy this dependence. The choice of iterations that are executed on a process affects the communication volume of the partitioned computation. Since the grouping of iterations depends on the iteration-to-data affinity, which can be computed only at run time. Section 3.4 discusses the partitioning scheme used to minimize the communication volume. This scheme might result in a non-contiguous set of iterations of the original computation mapped to a particular process. For now let us suppose that iterations 2, 4, and 5 (shown in dark gray) are chosen to be executed on process 0, and the remaining ones (shown in light gray) on process 1. The iterations mapped to a process are renumbered to be a contiguous sequence starting from 0. For example, iterations of loop i assigned to process 0 are renumbered to 0–2 (shown as the values of local iterator il in Figure 3.3).
3.2.2 Partitioning the Data

Once the iterations have been partitioned, the data arrays are partitioned such that each process has local arrays to store all the data needed to execute its iterations without any communication within the loop. In Figure 3.3, yl, A1, and xl are the local arrays on process 0 for data arrays y, A, and x.

The same data array element may be accessed by multiple iterations of the partitionable loop, which might be executed on different processes. Consider Figure 3.2, where x[1] and x[2] are accessed by both processes and are replicated on both, as shown in Figure 3.3. One of the processes is chosen as the owner of the data, and the
location of the data on another process is treated as a *ghost location*. For example, `x[2]` is owned by process 0, but process 1 has a ghost location for it. The ghost and owned locations together constitute the local copy of a data array on a process and represent all elements of the array touched by the iterations mapped to that process. Ghost elements for data arrays that are only read within the partitionable loop are set to the value at the owner before the start of the loop.

Elements of an array might receive updates from iterations of the partitionable loops executed on different processes. For example, consider a modification to Listing 1.1, where the statement at Line 9 was replaced with `x[col[j]] += A[j] * y[i];`.

This modified computation is still within the scope of targeted computations.
loop i is still a partitionable loop, with elements of the array x updated by multiple iterations of loop i. In a partitioned execution similar to the one shown in Figure 3.3, x[2] and x[1] are still owned by a process and have ghost locations on the other. To be consistent with the original execution, in the executor, these ghost locations are initialized to the identity element of the update operator (0 for “+=”, 1 for “*=”) before the loop execution. After the loop, their values are communicated to the owner where the values from all ghost locations are combined. For example, in the modified computation described above, if process 0 owns x[2], the ghost location for that element on process 1 would be initialized to 0 before the loop execution in the distributed memory code. After the loop, the value at the ghost location on process 1 is communicated to process 0 where it is combined with the value at the owned location. Therefore, the computation model is not strictly owner-computes. Since all update operations are associative and commutative, all iterations of the loop in the transformed version can be executed without any communication.

3.3 Data Structures for Executing of the Generated Partitions

After partitioning the computation, the individual partitions need to be executed in a manner consistent with the original sequential code. This is done by creating new arrays on each node which recreate the control flow and data access pattern. These arrays are allocated and populated by the inspector code.

3.3.1 Recreating Bounds of Inner Loops

The loop bounds of inner loops depend on values stored in read-only indirection arrays, surrounding loop iterators, or fixed-value parameters. Therefore, these bounds
can be precomputed by an inspector and stored in arrays in the local data space of each process. The sizes of these arrays would be the number of times an inner loop is invoked on that process. For example, in Figure 3.2, for iterations mapped to process 0, inner loop \( j \) is invoked once for every iteration of loop \( i \). Two arrays of size 3 would be needed to store the bounds of the \( j \) loop on process 0 (shown as \( lb \) and \( ub \) in Figure 3.3). Conditionals of \( ifs \) are handled similarly, by storing their inspected values in local arrays. The executor code uses the values in these arrays (\( lb \) and \( ub \) in Listing 3.1) to recreate the control flow of the iterations mapped to a process.

### 3.3.2 Recreating Data Accesses in the Transformed Code

Similar to the control flow, the data access patterns of the original computation needs to be recreated in the transformed version. Consider the expression \( col[j] \) used to access \( x \) in Listing 1.1. Since \( xl \) is the local copy of data array \( x \) on each process, all elements of \( x \) accessed by a process are represented in it. To access the correct elements in \( xl \), array \( col \) could be replicated on each process, and a map could be used to find the location that represents the element \( x[col[j]] \). Such an approach would need a map look up for every memory access and would be prohibitively expensive.

Similar to loop bounds, array-access expressions depend only on values stored in read-only indirection arrays, surrounding loop iterators, and constant parameters. Values of these expressions can be inspected and stored in arrays allocated in the local memory of each process. Further, the values stored are modified to point to corresponding locations in the local data arrays. The size of the array would be the number of times the expression is evaluated on a particular process. For example, the value of \( col[j] \) in Listing 1.1 is evaluated for every iteration of loop \( j \).
Figure 3.2, for iterations of \( i \) mapped to process 0, the total number of iterations of loop \( j \) is \( 2 + 3 + 2 = 7 \). An array, \( \text{access}_x \), of size 7 on process 0 is used to “simulate” the accesses to \( x \) due to expression \( \text{col}[j] \). The values stored in this array are modified to point to corresponding locations in \( x_1 \).

### 3.3.3 Optimizing Accesses from Inner Loops

The approach described previously would result in another array of the same size as \( \text{access}_x \) to recreate the access \( A[j] \). To reduce the memory footprint, we recognize that access expression \( j \) results in contiguous accesses to elements of \( A \), for every execution of loop \( j \). If the layout of the local array is such that elements that were accessed contiguously in the original data space remain contiguous in the local data space of each process, it would be enough to store the translated value of the access expression for only the first iteration of the loop. The rest of the accesses could be derived by adding to this value, the value of the iterator subtracted by the lower bound of the loop. For example, in Figure 3.2, for iterations mapped to process 0, the different invocations of the inner loop accesses the elements \( A[5:6] \), \( A[9:11] \) and \( A[14:15] \). These correspond to elements \( A_1[0:1] \), \( A_1[2:4] \) and \( A_1[5:6] \) in the partitioned view shown in Figure 3.3. The array \( \text{access}_A \) on process 0 stores the index of the first element of \( A_1 \) accessed within each sequence i.e. 0, 2 and 5. The other elements of the sequence are accessed within the loop by adding \((j - \text{lb}[\text{loop}_j])\) to this value, where \( \text{lb}[\text{loop}_j] \) is the lower bound for a given invocation of loop \( j \). In Listing 3.1, the variable \( \text{offset}_A \) is set to the value \((\text{access}_A[\text{loop}_j] - \text{lb}[\text{loop}_j]) \) before the loop. The expression \( A_1[\text{offset}_A + j] \) would be consistent with \( A[j] \) from the original computation.
Using such an approach reduces the size of the array needed to recreate the data access pattern for unit-stride index expressions to the number of invocations of the inner loop (3 in Figure 3.3), instead of the total number of iterations of the loop executed on a process.

### 3.3.4 Optimizing Accesses from the Partitionable Loop

For cases where elements of an array were accessed at unit-stride with respect to the partitionable loop in the original computation, it is desirable to maintain unit-stride in the transformed code as well. This can be achieved by placing contiguously in local memory all elements of the array accessed by the successive iterations on a process. For example, if iterations 2, 4, and 5 of loop k are mapped to process 0, elements of array x1 can be accessed by using iterator k1 if x1[0–2] correspond to x[2], x[4], and x[5]. The same could be done for y[i] in Listing 1.1.

If the same array is accessed elsewhere by another expression that is unit-stride with respect to an inner loop, the ordering of elements required to maintain this unit-stride in the transformed code may conflict with the ordering necessary to maintain unit-stride with respect to a partitionable loop. For example, in Figure 3.2, iterations 0, 1 and 3 of the i-loop access the elements A[0:1], A[2:4] and A[7:8] using the unit-stride expression, A[j]. In the partitioned view shown in Figure 3.3, these iterations are mapped to process 1. To maintain the contiguity of accesses, A[0:1], A[2:4] and A[7:8] have to be piecewise-contiguous in the local array A1 on process 1. Suppose that this loop also contained the expression A[i]. Since the iterations of the i-loop would be renumbered to be contiguous, to maintain contiguity of accesses from this loop, elements corresponding to A[0], A[1] and A[3] would have to be
contiguous in memory as well. This conflicts with the layout needed to maintain contiguous access from loop \( j \). In such cases, the accesses from the partitionable loop are not optimized, since using a separate array to recreate the access from the partitionable loop would have a smaller footprint.

A potential conflict could also arise if multiple partitionable loops access an array with unit-stride (like access \( y[i] \) and \( y[k] \) in Listing 1.1). To optimize all accesses it is necessary to partition all participating loops in a similar way to obtain a consistent ordering of the array elements. In the current approach, if it can be statically determined that the participating loops have the same loop bounds, then the partitioning process is modified to ensure that all the loops are partitioned identically (Section 3.4.3). In such cases, the unit-stride access from all partitionable loops are preserved in the executor code too (as shown in Listing 3.1), making it amenable to optimizations like vectorization. Otherwise, access arrays similar to the ones described in Section 3.3.2 are used to recreate the access from partitionable loops.

### 3.3.5 Executor Code

To execute the original computation on each process, local arrays are accessed for all data arrays, and values stored in local arrays are used for loop bounds, conditionals, and array-access expressions. In Listing 3.1, which is the executor code generated for Listing 1.1, loop bounds of partitionable loops are based on the number of iterations \( n_1 \) that are mapped to a process. The loop bounds of loop \( j \) are read from arrays \( lb \) and \( ub \). Accesses to local data arrays \( x_1 \) and \( A_1 \) are determined by values in arrays \texttt{access}_x and \texttt{access}_A. In addition, communication calls are inserted to satisfy the producer-consumer relationship due to array \( x \).
3.4 Partitioning the Computation for Load-balance and Communication Minimization

In order to achieve efficient distributed memory execution, the main factors to be considered are

- The computation must be partitioned such that each partitionable loop is executed in a load-balanced manner
- The amount of communication between processes has to be minimized.

Due to the use of ghost locations, communication is needed either before executing the iterations of the partitionable loop to initialize the ghosts within arrays read in the loop body, or after the loop execution to combine the values from all the ghost locations for data elements of arrays updated within the loop. Therefore, the amount of communication required is directly related to the number of ghost location needed for the distributed memory execution. Since a data element need not have any ghosts if all the iterations that access it are mapped to the same process, an accurate representation of the iteration-to-data affinity would allow the inspector to partition the computation while minimizing the number of ghosts. For example, consider the computation represented in Figure 3.2. Consider a case where both the loops i and k are block-partitioned amongst processes 0 and 1. The iterations mapped to process 0 access elements $x[0--3]$, while the iterations mapped to process 1 access elements, $x[0--5]$. 4 elements of array $x$ are accessed by both processes, resulting in 4 ghost locations. On the other hand the partitioned execution represented by Figure 3.3 results in only 2 ghost locations.
Similar to [77] and [78], we use a hypergraph to represent this affinity. A hypergraph $\mathcal{H}$ is defined by a set of vertices, $\mathcal{V}$ and a set of edges, $\mathcal{N}$ and is a generalization of a graph. An edge of a hypergraph, referred to as a hyperedge or a net, connects two or more vertices. The connected vertices are referred to as the pins of the net. The iteration-to-data affinity is captured by having vertices that represent iterations of the partitionable loops, and nets that represent individual array elements. The vertices that represent iterations that access the same data element form the pins of the corresponding net. While [77] used the hypergraph abstraction to reorder iterations to achieve better temporal locality, here we use a disjoint partitioning of the vertices of the hypergraph to compute the group of iterations of the partitionable loops to be executed on a process. Therefore the number of partitions is same as the number of processes, $P$ to be used. The partitioning is subjected to constraints that enforce the two requirements listed above. PaToH hypergraph partitioner was this constrained partitioning. Following is a description of these constraints adapted from [78].

### 3.4.1 Minimizing Communication

Since the communication cost depends on the number of bytes to be exchanged between the processes, each net $n \in \mathcal{N}$ is associated with a weight $c_j$, whose value is the same as the size in bytes of the data-type represented by the net (4 for integer and float types, 8 for doubles). For each partition $p \in P$, the set of nets that have pins in it can be divided into two disjoint subsets, a set of internal nets, $I_p$ that have pins only in $p$, and a set of external nets, $E_p$ that have pins in other partitions too. Each external net, $j \in E_p$ represents a data element that is accessed by more than one process. If $\lambda_j$ is the number of partitions in which $j$ has a pin, the number of ghost
elements needed for the corresponding data-element would be \( \lambda_j - 1 \). The metric \( \Pi_p \), which is the number of ghost elements for a partition \( p \), accurately captures the communication costs needed to execute the group of iterations mapped to a process.

\[
\Pi_p = \sum_{j \in E_p} c_j (\lambda_j - 1) \quad \forall p \in P
\]  

(3.1)

Since there is no communication needed for arrays that are only read within the computation, nets representing data-elements of such arrays are not removed from the hypergraph before partitioning it.

### 3.4.2 Achieving Load Balancing

To achieve load-balance across processes, the iterations of every partitionable loop have to be evenly distributed amongst them. This is done by using a multi-constrained partitioning scheme. Each vertex is associated with a vector of weights, \( \vec{w}_i \) of size equal to the number of partitionable loops. A vertex that represents an iteration of the \( k \)-th partitionable loop has the \( k \)-th element as 1, with all other elements being 0. The number of iterations of all the partitionable loops mapped to a partition can be computed by the vector sum of the weights of the vertices belonging to the partition. The quantity represents the weight of the partition, \( W_p \).

\[
W_p = \sum_{i \in p} \vec{w}_i \quad \forall p \in P
\]  

(3.2)

\( W_{avg} \), computed as \( W_{V}/P \), gives the number of iterations for all the partitionable loops that would be executed on a process in a perfectly load-balanced execution. Restricting the weight of every partition to be within a tolerance, \( \epsilon \) of \( W_{avg} \) ensures that iterations of the partitionable loop are distributed equally amongst the partitions. Therefore the constraints on the partitioning can be described by the system of vector
equations

\[ W_p \leq W_{avg}(1 + \epsilon) \quad \forall p \in P \]  

(3.3)

Partitioning the hypergraph under these constraints while minimizing \( \Pi_p \) for each partition achieves the desired partitioning of the original computation.

The model described above assumes that all iterations of a partitionable loop have similar execution times. It is possible to relax this assumption and have the inspector use heuristics such as total number of statement instances within a loop, to model the execution time of an iteration. Using this value for the non-zero element of the weight vector would better capture the requirements of load-balancing. In practice this assumption did not significantly affect the performance of the generated distributed memory code.

### 3.4.3 Modifications for Maintaining Unit-Stride from Partitionable Loops

To be able to optimize the accesses from partitionable loops as discussed in Section 3.3.4, it is necessary to ensure that multiple partitionable-loops are partitioned in a similar manner. If it can be statically determined that all these loops have the same loop-bounds, corresponding iterations from these loops are represented by the same vertex of the hypergraph, resulting in these loops being partitioned in an identical manner. While this approach might increase the communication costs due to the inexact representation of the iteration-to-data affinity, it is out-weighed by benefit of reduced memory overhead, and possible vectorization of the executor code.
(Start) ::= (ElementList)

(ElementList) ::= (Element) | '{' (ElementList) '}'

(Element) ::= (Assignment) | (Loop) | (If)

(Loop) ::= 'for' '(' (Iterator) '=' (Expr) ';' (Iterator) '<' (Expr) ';' (Iterator) '++') (ElementList)

(If) ::= 'if' '(' (Expr) ')' (ElementList) 'else' (ElementList)

(Assignment) ::= (LHSExp) (AssignOp) (Expr) ';'

(Expr) ::= Side-effect-free expressions of (BasicExpr)

(LHSExp) ::= (Scalar) | (Array) '[' (Expr) ']

(BasicExp) ::= (Scalar) | (Iterator) | (Array) '[' (Expr) ']

(AssignOp) ::= '=' | '+=' | '*=' | ...

Figure 3.4: Syntactic structure of computation input to the detection phase.

3.5 Identifying Partitionable Loops

The previous sections described an approach to achieve distributed memory parallelization of a class of computations described in Section 3.1. The algorithms that generated the inspector code and the executor code assume that the input code satisfied all the conditions described in that section. However, additional analysis would be needed to determine if a given computation is amenable to the transformations described previously. In this section we describe the compiler algorithm (Algorithms 1 and 2) that automatically detects such code regions. The analysis starts by assuming that the input code is consistent with the grammar described in Figure 3.4.

Algorithm 1 takes as input an AST corresponding to a (Loop) from this grammar, and determines whether the loop is partitionable. Section 3.5.1 discusses the analysis to detect indirection arrays and scalars; this analysis corresponds to line 2-19 in the
Algorithm 1: CheckForPartitionableLoop($A$)

Input: $A$: AST of loop satisfying the grammar in Figure 3.4
Output: is\_partitionable: Boolean flag set to true if $A$ is partitionable
$I_A$: Set of indirection arrays
$I_S$: Set of indirection scalars
$I$: Set of loop iterators

begin

$I_S = \emptyset$; $I_A = \emptyset$; $I = A$.Iterator; is\_partitionable = true;

foreach $s \in$ GetLoopStmts($A$.Body) do

$I = I \cup s$.Iterator;

foreach $s \in$ GetLoopStmts($A$.Body) do

$l = s$.LowerBound; $u = s$.UpperBound;

$I_A = I_A \cup$ GetArrayOfVariables($l$) \cup GetArrayOfVariables($u$);

$I_S = I_S \cup$ (GetArrayOfScalars($l$) \cup GetArrayOfScalars($u$) - $I$);

endfor

endfor

foreach $s \in$ GetIfStmts($A$.Body) do

$c = s$.Cond;

$I_A = I_A \cup$ GetArrayOfVariables($c$);

$I_S = I_S \cup$ (GetArrayOfScalars($c$) - $I$);

endfor

foreach $s \in$ GetAssignStmts($A$.Body) do

foreach $e \in$ GetArrayRefExprs($s$) do

$c = e$.IndexExpr;

$I_A = I_A \cup$ GetArrayOfVariables($c$);

$I_S = I_S \cup$ (GetArrayOfScalars($c$) - $I$);

endfor

endfor

worklist = $I_S$;

while not IsEmpty(worklist) do

$v = \text{RemoveElement}(\text{worklist})$;

foreach $s \in$ GetAssignStmts($A$.Body) do

if IsScalar($s$.LHS) \&\& $s$.LHS == $v$ then

$c = s$.RHS;

$I_A = I_A \cup$ GetArrayOfVariables($c$);

$S =$ GetArrayOfScalars($c$) - $I$;

foreach $u \in S$ do

if $u \notin I_S$ then

$I_S = I_S \cup \{u\}$; AddElement(worklist,$u$);

endfor

endfor

endfor

is\_partitionable = CheckSeparation($A,I_A,I_S$);

if is\_partitionable then

is\_partitionable = CheckForParallelism($A$.Body,$I_S,I_A,I$);

return [is\_partitionable,$I_A,I_S,I$];

else

return [false,$\emptyset$,$\emptyset$,$\emptyset$];

endfor
Algorithm 2: CheckSeparation($A, I_A, I_S$)

**Input**: $A$: AST of loop satisfying the grammar in Figure 3.4  
$I_A$: Set of indirection arrays  
$I_S$: Set of indirection scalars  

**Output**: $\text{is\_partitionable}$: Boolean flag set to true if variables in $I_A$ and $I_S$ are not used outside ⟨IExpr⟩s

1. **begin**
2. $\text{is\_partitionable} = \text{true}$;
3. **foreach** $s \in \text{GetAssignStmts}(A.\text{Body})$ **do**
4.  **if** $\neg \text{IsScalar}(s.\text{LHS}) \land s.\text{LHS}.\text{Array} \in I_A$ **then**
5.    $\text{is\_partitionable} = \text{false}$;
6.    **break**;
7.  **if** $\neg \text{IsScalar}(s.\text{LHS}) \lor s.\text{LHS} \notin I_S$ **then**
8.    $S = \text{GetAllScalars}(s.\text{RHS}, \text{IGNORE\_INDEX\_EXPR})$;
9.    **if** $S \cap I_S \neq \emptyset$ **then**
10.       $\text{is\_partitionable} = \text{false}$;
11.       **break**;
12.    $S = \text{GetAllArrays}(s.\text{RHS}, \text{IGNORE\_INDEX\_EXPR})$;
13.    **if** $S \cap I_A \neq \emptyset$ **then**
14.       $\text{is\_partitionable} = \text{false}$;
15.       **break**;
16. **return** $\text{is\_partitionable}$;

Algorithm. Section 3.5.2, corresponding to line 31 in the algorithm, checks the parallelism of the loop. The overall approach for identifying a sequence of maximal partitionable loops (by applying Algorithm 1 several times) is described in Section 3.5.3.

### 3.5.1 Indirection Arrays and Indirection Scalars

The first step in the analysis of partitionable loops is to determine the variables corresponding to indirection arrays (⟨IArray⟩s) and indirection scalars (⟨IScalar⟩s). As mentioned before, these are variables whose values (or in case of arrays, the values stored in them) are, directly or indirectly, used to compute loop bounds, values of conditional expressions, or index expressions of array access expressions.
To identify such variables, all loop bounds, conditionals, and array index expressions are analyzed. All array variables appearing in them are added to the set of indirection array variables. All scalars appearing in such expressions are added to the set of indirection scalar variables. In addition, such scalars are also added to a worklist.

Following this, for every scalar in the worklist, the right-hand side expression of each assignment statement within the loop-body that assigns to this scalar is analyzed. All arrays appearing on this right-hand side are added to the set of indirection arrays. All scalars that appear on the right-hand side and are not already in the set of indirection scalars are added to that set and are also added to the worklist. Once all statements that assign to the scalar have been analyzed, it is removed from the worklist. These steps are repeated until the worklist is empty. The algorithm is guaranteed to terminate since a scalar is added to the worklist at most once, and for every iteration, an item is removed from the worklist. Upon termination, all indirection scalars and indirection arrays within the loop have been determined.

The remaining variables, apart from loop iterators, are categorized as data scalars (\langle DScalar \rangle s) or data arrays (\langle DArray \rangle s). For every such variable in the original computation, there is a corresponding variable in the transformed code that represents the local copy to be used by the executor. The grammar in Figure 3.1 does not allow the values in \langle IArray \rangle s to be modified. This is checked at lines 4-6 of Algorithm 2. Further the grammar does not allow the value stored in \langle IArray \rangle s or \langle IScalar \rangle s to be used in computation of values stored in \langle DArray \rangle s or \langle DScalar \rangle s. This is checked at lines 7-15, where every assignment in the target loop body is examined. If the left-hand side is a data scalar or a data array access expression, the right-hand side expression is
analyzed further. Ignoring all index expressions (i.e., \langle expr \rangle in \langle arr\rangle[(expr)]), any reference to a loop iterator, an indirection scalar, or an indirection array indicates that the loop is not a partitionable loop. Section 3.5.4 outlines how to handle cases where this property does not hold. The above analysis ensures that scalars and arrays within the computation can be separated into two disjoint categories, one whose values completely determine the control-flow or data-access patterns, and another which contains variables whose values are the inputs and outputs of the computation.

## 3.5.2 Parallelism of the Target Loop

The final property to be checked is that the target loop is parallel, except for dependences due to reductions. Since the transformation scheme generates a parallel inspector, the control-flow and data-access pattern for a given iteration of the target loop must not depend on any of the previous iterations. This can be ensured if

- All indirection arrays are read-only within the loop
- For any indirection scalar $s$ modified within the loop body, any use of $s$ reads a value that was written to $s$ in the same iteration, i.e. $s$ is privatizable within the partitionable loop

The second condition ensures that there are no inter-iteration dependences arising due to indirection scalars. Both properties can be easily checked statically.

To ensure that there are no dependences due to data arrays

- A data array variable \langle DArray\rangle can appear either only on the right-hand side of \langle DAssignment\rangle$s, or only on the left-hand side, but not both
• A ⟨DArray⟩ can appear on the left-hand side of multiple ⟨DAssignment⟩s, but all those statements must use the same ⟨AssignOp⟩

Finally, it has to be ensured that there is no dependence due to any data scalar. If a data scalar satisfies the same condition as the one for ⟨DArray⟩s listed above, then the inter-iteration dependence caused by updates to this scalar can be handled by the scheme described in Section 4.2.1. Scalars that do not satisfy this property might still not result in an inter-iteration dependence, if each use of the scalar reads a value that was assigned to it in the same iteration of the loop (similar to the property satisfied by ⟨IScalar⟩).

These constraints ensure that the only inter-iteration dependences are either output dependences due to data scalars/arrays assigned to by multiple iterations of the loop, or updates to such variable using associative and commutative update operators. A loop which satisfies all these properties is characterized as a partitionable loop.

3.5.3 Finding Sequences of Maximal Partitionable Loops

Algorithm 1 takes as input one loop from the grammar in Figure 3.4, and decides whether the loop is partitionable. Suppose we are given an AST based on this grammar, with several loops (disjoint and/or nested within each other). It is desirable to identify partitionable loops that are as large as possible, as well as sequences of such loops that can be optimized together (by inserting communications calls between pairs of consecutive loops). This sequence of loops has been characterized previously as chained loops in [79].
To identify such a sequence, the entire input program AST is assumed to be represented by the non-terminal ⟨ElementList⟩ in Figure 3.4. The nodes within the AST are recursively traversed. On encountering a ⟨Loop⟩, Algorithm 1 is used to determine if it is a partitionable loop. If it is, then the traversal skips all its children nodes and marks the loop for distributed memory parallelization. If it is not, then the loop-body is recursively analyzed. Once all the partitionable loops within the input program have been identified, an AST node that is a parent of all the marked loops are identified using the following criteria.

- There is no intersection between the variables marked as indirection arrays and data arrays across partitionable loops nested within the AST node. Since the inspector combines the analysis for all partitionable loops, updates to indirection arrays might change the control flow and data access pattern found by the inspector. The inspector would have to be re-executed to capture the modification.

- Indirection scalars are not live after the corresponding partitionable loop.

- No reference to indirection arrays appear nested within the AST node but outside of the partitionable loops. Since indirection arrays are not explicitly partitioned, such references violate the correctness of the generated executor code.

- No data arrays are referenced outside the partitionable loops but nested within the AST node. Since the partitioning of data is done based purely on accesses from the partitionable loops, such references might be invalid on a process.

Such an approach ensures that partitionable loops are not nested within each other, and that the generated inspector code can be placed above the AST node in
question. Presence of several loops (like time loop or convergence loop) between this AST node and the partitionable loop help amortize the cost of the inspector. If there is no such node in the AST that is a parent of all the partitionable loops, the analysis could be repeated by breaking up the sequence into sub-sequences. In the worst-case, a separate inspector would be generated for every partitionable loop, and placed right before the executor code for that loop. Such situations are not the use case for the approach described here, since the inspector cost is not effectively amortized.

3.5.4 Possible Generalizations

One of the restrictions described in Section 3.5.1 was that an assignment statement which assigns to a data scalar or data array element must not contain a reference to an indirection scalar or indirection array outside of index expressions. To relax this constraint, for every such occurrence, a new array is used which stores the value of the expression involving the indirection array/scalar. The inspector evaluates the part of the right-hand side that references the indirection scalar/array and stores it in a temporary array. This new array is treated as a read-only data array and used to replace the expression involving the indirection array/scalar in the executor code. Substituting the same expression in the original computation with a read from this new data array would result in the loop having a syntactic structure as shown in Figure 3.1. Therefore, all code-generation algorithms discussed earlier are still applicable. A similar restriction from Section 3.5.1 was that the value of a data array/scalar should not depend on the iterator value for the partitioned loop. This constraint could be removed, using a similar approach.
CHAPTER 4

Code generation scheme for Inspector and Executor for Distributed Memory Machines

4.1 Inspector/Executor: Run Time Functionality and Code Generation

The code for the inspector and the executor for a partitionable loop is generated by analyzing its loop body. Based on the description in Sections 3.2 and 3.3, the role of the inspector is to partition the computation at run time and to allocate/populate data-structures that are used by the executor on each process. This task can be broken down into three distinct phases:

- Phase I: Build and partition the hypergraph by analyzing the data elements touched by the iterations of all partitionable loops; allocate local copies for all data arrays based on the iterations assigned to each process.
- Phase II: Compute the sizes of the arrays needed to replicate the control-flow and array-access patterns.
- Phase III: Populate these arrays with appropriate values.
A description of how each of these tasks are completed for a single partitionable loop and the algorithm used to generate the code to perform these tasks are described in this section.

### 4.1.1 Phase I: Hypergraph Generation

#### Runtime Functionality

This phase of the inspector analyzes the computation and generates the corresponding hypergraph. For the original code shown in Listing 1.1, a portion of the inspector that generates the hypergraph is shown in Listing 4.1. The inspector code
for this phase contains only the \langle Assignment\rangle s, \langle Loop\rangle s and \langle If\rangle s from the original computation.

At the start of the computation, it is assumed that all arrays are block-partitioned across the processes. The approach can be easily adapted to other partitioning schemes for the initial data—e.g., cyclic and block-cyclic. Each process analyzes a block-partitioned subset of the original iterations, represented by \([k_{\text{start}},k_{\text{end}}-1]\) for loop \(k\) and \([i_{\text{start}},i_{\text{end}}]\) for loop \(i\) in Listing 4.1. Therefore each process computes only a part of the hypergraph. For each iteration of the partitionable loop executed on a process, a vertex is added to represent it in the hypergraph by calling the runtime function, AddVertex. This function takes as input a compile time integer identifier, which uniquely identifies the partitionable loop being analyzed, (e.g., \(id_{k_{\text{loop}}}\) for loop \(k\) at line 3) and returns a handle to the vertex added to the hypergraph.

For every data array element that is accessed by this iteration, the vertex returned previously is added as a pin to the corresponding net using the function AddPin. For example, to represent the elements of array \(y\) accessed by loop \(i\) of Listing 1.1 through the expression \(y[i]\) at line 9, this phase of the inspector calls the function AddPin(\(id_{y\_array},i,vi,\text{true}\)) at line 17 of Listing 4.1. The vertex \(vi\) represents the current iteration being inspected, and is added as a pin to the net representing the \(i\)-th element of array \(y\). The last argument, of boolean type, specifies that the element is accessed by an expression with a unit stride. Here \(id_{y\_array}\) is a unique compile time integer identifier for array \(y\).
It might not be possible to evaluate all \( (IExpr) \)s in the loop-body, since values of indirection arrays elements referenced in them might not be local to the process. Consequently, in the inspector code, statements containing such references are guarded by the function \texttt{is\_known}. This function returns true if the value of the indirection array element needed is known on the current process and false otherwise, with the element flagged as being requested. Multiple references to indirection arrays results in a conjunction of such guards, the construction of which is described later with the code-generation algorithm. After the block of iterations has been analyzed, the inspector code calls the function \texttt{DoneGraphGen} which returns true if any call to the function \texttt{is\_known} returned false. Before returning, this function also fetches the value of all the requested elements from the appropriate remote processes. The \texttt{do..while} loop around this phase of the inspector results in re-analysis of these iterations when this function returns true. This time around, the inspector analysis is able to proceed further since all the calls to \texttt{is\_known} that returned false previously, would now return true. The value of the element is retrieved via the function \texttt{get\_elem}. Due to multiple levels of indirection, the outer \texttt{do..while} loop executes as many iterations as the maximum depth of indirection (2 in case of 4.1). When \texttt{DoneGraphGen} return false, it means that all \( (IExpr) \)s have been analyzed on all the processes. In this phase, there is no communication due to the values of the data array elements, since these values are not used to index other arrays.

Since values assigned to indirection scalars might depend on indirection arrays, these values might not be known on a process either. A \textit{shadow scalar} is associated with every indirection scalar. If the right-hand side of an assignment to an indirection scalar cannot be computed on a process, the associated shadow scalar is set to false.
Statements that contain ⟨IExpr⟩s that use the values of these indirection scalars are also guarded to check the value of the corresponding shadow scalar. For example, in Listing 4.1 shadow_xindex is the shadow scalar corresponding to xindex. It is set to false at line 16 if the value of col[j] is not known on a process. Since this value is used by AddPin(id_x_array,xindex,vi,false) at line 20, this statement is guarded to check the state of shadow_xindex.

The portions of the hypergraph built by each process are combined to compute the complete iteration-to-data affinity. The hypergraph is partitioned P ways as described in Section 3.4, where P is the number of processes. Each process is assigned a unique partition representing the iterations to be executed on it. The iterations are renumbered such that they form a contiguous set on each process, while maintaining the relative ordering of the iterations mapped to that process.

**Code Generation at Compile Time**

The inspector code that achieves the functionality described above (e.g., the code in Listing 4.1) is generated automatically by the compiler. The compiler algorithm to generate the loop-body of the code to build the hypergraph is shown in Algorithm 3. The generation of the loop and the surrounding do..while loop is discussed later in Section 4.3.

The algorithm traverses the statements within the body of each partitionable loop. For an ⟨IAssignment⟩ in the original AST, an assignment statement is added to the inspector AST with the right-hand side modified to convert all array references to calls to function get_elem. This AST modification is performed by function ConvertArraysToFunctions at line 5. In addition, the statement is guarded by a conditional to check that the values of all array elements or scalars are known on the
Algorithm 3: CodeGenHyperGraph(\(\mathcal{H}, ss\))

\(\textbf{Input} : \: \mathcal{H} : \text{Hypergraph object} \)
\(\text{ss} : \text{Sequence of statements in the original AST} \)

\(\textbf{Output}: \: \mathcal{A}_H : \text{AST of the inspector code to generate hypergraph} \)
\(I : \langle \text{IScalar}\rangle s \text{ defined in } ss \)

\begin{algorithmic}
\State \(\mathcal{A}_H = \emptyset ; \) \(I = \emptyset ; \)
\EndFor
\If {IsIAssignment(s)} \then
\State \(b = \text{ConvertArraysToFunctions(s.RHS)} ; \)
\State \(l_H = \text{NewIfStmt()} \); \(l_H.\text{Cond} = \text{GenGuards(s.RHS)} ;\)
\State \(s_H = \text{NewAssignmentStmt(s.LHS,b)} ; \)
\State \(s_H.\text{Append(SetShadowScalar(s.LHS,true))} ;\)
\State \(I_L = \{ s.LHS \} ; l_H.\text{Then} = s_H ;\)
\ElseIf {IsDAssignment(s)} \then
\EndIf
\ElseIf {IsLoop(s)} \then
\State \(l = s.\text{LowerBound} ; u = s.\text{UpperBound} ;\)
\State \(b_l = \text{ConvertArraysToFunctions(l)} ;\)
\State \(b_u = \text{ConvertArraysToFunctions(u)} ;\)
\State \(s_H = \text{NewLoopStmt(s.Iterator,b_l,b_u)} ;\)
\State \(\mathcal{A}_H.\text{Body},I_L = \text{CodeGenHyperGraph}(\mathcal{H},s.\text{Body}) ;\)
\State \(l_H = \text{NewIfStmt()} ; l_H.\text{Then} = s_H ;\)
\State \(l_H.\text{Cond} = \text{NewAndCond(GenGuards(l),GenGuards(u)}) ;\)
\Else
\State \(c_H = \text{ConvertArraysToFunctions(s.Cond)} ;\)
\State \(s_H = \text{NewIfStmt()} ; s_H.\text{Cond} = c_H ;\)
\State \(\mathcal{A}_H.\text{Then},I_1 = \text{CodeGenHyperGraph}(\mathcal{H},s.\text{Then}) ;\)
\State \(\mathcal{A}_H.\text{Else},I_2 = \text{CodeGenHyperGraph}(\mathcal{H},s.\text{Else}) ;\)
\State \(l_H = \text{NewIfStmt()} ; l_H.\text{Cond} = \text{GenGuards(s.Cond)} ;\)
\State \(I_L = I_1 \cup I_2 ; l_H.\text{Then} = s_H ;\)
\EndElse
\EndIf
\EndIf
\EndFor
\Return \(\mathcal{A}_H, I \) ;
\end{algorithmic}
**Algorithm 4: GenGuards(e)**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>begin</strong></td>
</tr>
<tr>
<td>2</td>
<td>if <em>IsIScalar(e)</em> then</td>
</tr>
<tr>
<td>3</td>
<td><code>c = NewCheckEquality(e.ShadowScalar, true);</code></td>
</tr>
<tr>
<td>4</td>
<td>else if <em>IsArrayRefExp(e)</em> then</td>
</tr>
<tr>
<td>5</td>
<td><code>c_l = GenGuards(e.IndexExpr);</code></td>
</tr>
<tr>
<td>6</td>
<td><code>c_r = GenerateIsKnownFn(e.Array, e.IndexExpr);</code></td>
</tr>
<tr>
<td>7</td>
<td><code>c = NewAndCond(c_l, c_r);</code></td>
</tr>
<tr>
<td>8</td>
<td>else</td>
</tr>
<tr>
<td>9</td>
<td><code>c = ∅;</code></td>
</tr>
<tr>
<td>10</td>
<td>foreach <code>d ∈ e.Children</code> do</td>
</tr>
<tr>
<td>11</td>
<td><code>c = NewAndCond(c, GenGuards(d));</code></td>
</tr>
<tr>
<td>12</td>
<td><strong>return</strong> <code>c;</code></td>
</tr>
</tbody>
</table>

The expression to be used by the conditional is returned by function `GenGuards`. The generation of the guard expression constructed by `GenGuards` is described in Algorithm 4. The algorithm takes as input an `<IExpr>`. For expressions that are references to `<IScalar>`, the condition checks if the corresponding shadow scalar is set to true. For expressions that are `<IArray>` references, the condition contains two parts. The first part is a call to function `is_known`, with the arguments being the unique identifier for the array variable (computed at compile time), and the index expression. The second part is generated by recursively analyzing the index expression, to ensure that the index expression itself can be computed. These two parts are combined using a logical *and* operator. Since the latter condition has to be checked before the former one, it is set as the first operand in the *and* expression. The short-circuit evaluation of C/C++ ensures that the `is_known` function is called only when the index expression can be evaluated.
Since a ⟨Loop⟩ is executed only when the bounds are known, the loop iterator is always known within the loop body. Algorithm 4 returns ∅ for such an expression. For expressions that are not ⟨BasicIExpr⟩, all children of the ⟨IExpr⟩ are recursively evaluated and their guards are combined with the and operator.

Algorithm 3 sets the conditional expression returned by GenGuards as the condition of the guard statements. The statements to set the shadow scalar associated with the ⟨IScalar⟩ to true is added to the true branch of the guard statement at line 8 of Algorithm 3, along with a statement to set it to false in the false branch (at line 36 of the algorithm). Lines 11-16 of Listing 4.1 contain the code generated for the ⟨IStatement⟩ at line 8 of Listing 1.1.

For ⟨DAssignment⟩s in the original AST, for every reference to a data array, a call to function AddPin is generated by GenAddPinFn at line 15. Such a call takes as input (1) a compile time integer identifier for the data array, (2) the index expression, and (3) a boolean flag which indicates whether the current expression is unit-stride with respect to a surrounding loop. (The optimized handling of unit-stride accesses is discussed in Section 3.3.4.) The index expression used in the original AST is modified to convert all arrays references (all of which are ⟨IArray⟩s) to calls to get_elem. Every statement is guarded to check that the index expression can be evaluated on a process. This guard is again generated by GenGuards. Lines 17-20 of Listing 4.1 are the statements generated for the ⟨DStatement⟩ at line 9 of Listing 1.1.

Upon encountering a ⟨Loop⟩, the statement is replicated in the inspector AST, with references to indirection arrays in the bounds replaced with calls to get_elem. The loop body is generated by a recursive call to analyze the loop body in the original AST. This loop should be executed by the inspector only when the loop bounds
can be computed on a process. Therefore, the loop statement is guarded by conditional statements to check for this (generated by GenGuards). A similar approach is employed for (If) statements: references to indirection arrays in the conditional expression are replaced by calls to get_elem, and the branches are generated recursively. The statement is enclosed within guards to check that the conditional expression can be evaluated on a process.

It is possible that an indirection scalar is modified within an inner loop or within branches of conditional statements and used later within the partitionable loop. Such uses must be avoided when the loop/conditional statements were not executed due to the guards. Therefore, for all indirection scalars modified within the inner loop bodies or within branches of conditional statements, the shadow scalar must be set to false when the guard evaluates to false. The set of such scalars is returned by the recursive call that builds the loop body (at line 25) or the branches of the conditional statement (at lines 31, 32). The statements to set these variables to false are added to the false branch of the guard statement for the corresponding loop or conditional statement, at line 36 of Algorithm 3.

To support the optimizations of accesses from partitionable loops as discussed in Section 3.3.4, it might be necessary to ensure that multiple partitionable loops are partitioned the same way. To enforce this, the loop bounds of all such loops are checked at compile time. If they are the same, a single compile time identifier is used to represent all of them. Therefore, at run time, AddVertex would map corresponding iterations of all these loops to the same vertex. In cases where the loop bounds are not the same, the accesses to data arrays that are unit-stride with respect to the partitionable loops in the original code are recreated through use of access arrays.
4.1.2 Initializing Local Data Arrays

Initially all the data arrays used in the original computation were assumed to be block-partitioned. At the end of phase I, the inspector has built the hypergraph that represents the iteration-to-data affinity. This graph is then partitioned by using external hypergraph partitioning libraries such as PaToH [80], under the constraints described in Section 3.4. This gives a partitioning of the iterations of the partitionable loops. The data is partitioned as follows: for a net that has all its pins in the same partition, the corresponding data element is assigned to the same process. If a net has pins in different partitions, the element is assigned to the process that executes the majority of the iterations that access this data. All other processes have a ghost location for that element. The local copy of the array consists of the elements that a process owns and the ghost locations for elements owned by other processes. This is done for all data arrays in the computation. For example, arrays \( y_1, x_1 \) and \( A_1 \) of Listing 3.1 are allocated at this time.

As described in Section 4.1.1, expressions that result in unit-stride accesses to a data array are identified at compile time, but the actual elements accessed by such expressions are known at run time (using the value of the last argument of \( \text{AddPin} \)). If \( \text{nprocs} \) is the number of processes, for every local array on every process, the inspector maintains the following metadata

- **local_array_size** : The size of the local array on a process, include ghost elements and owned elements

- **n_unitstride_accesses** : The number of elements of the local array that were accessed using unit-stride expressions

53
• \texttt{l\_to\_g} : An array of size \texttt{local\_array\_size} that stores the global index of the corresponding element in the local array.

• \texttt{nghost\_elements} : An array of size \texttt{nprocs+1}, such that 
\texttt{nghost\_elements[p+1]} - \texttt{nghost\_elements[p]} gives the number of elements of the local array that are owned by process \texttt{p} and ghosted on the current process.

• \texttt{ghost\_elements} : An array of size \texttt{nghost\_elements[nprocs]}, such that 
\texttt{ghost\_elements[nghost\_elements[p]]} to 
\texttt{ghost\_elements[nghost\_elements[p+1]]-1} store positions of all elements in the local array that are ghosted from process \texttt{p}.

• \texttt{nowned\_elements} : An array of size \texttt{nprocs+1}, such that 
\texttt{nowned\_elements[p+1]} - \texttt{nowned\_elements[p]} gives the number of elements of the local array that are owned by the current process and are ghosted on process \texttt{p}.

• \texttt{owned\_elements} : An array of size \texttt{nowned\_elements[nprocs]}, such that 
\texttt{owned\_elements[nowned\_elements[p]]} to 
\texttt{owned\_elements[nowned\_elements[p+1]]-1} store positions of all elements in the local array that are owned by a process and are ghosted on process \texttt{p}.

The elements of the local array are laid out such that positions 0 to \texttt{n\_unitstride\_accesses-1} of the local array store values of elements that were accessed through unit-stride expressions, in increasing order of their index in the original computation. Positions \texttt{n\_unitstride\_accesses} to \texttt{local\_array\_size-1}
of the local array stores the values of the remaining elements of the array accessed by the iterations mapped to a process, in increasing order of their index in the original computation. This scheme ensures that elements accessed contiguously in the original computation remain contiguous in the local memory of each process after partitioning. By construction, values in $l_{to_g}[0]$ to $l_{to_g}[n_{unitstride_accesses}-1]$ are in ascending order and values from $l_{to_g}[n_{unitstride_accesses}]$ to $l_{to_g}[local_array_size-1]$ are in ascending order.

4.1.3 Phase II: Computing the Sizes of Local Access Arrays

Runtime Functionality

The next phase of the inspector determines the sizes of access arrays: arrays that are used to

- store loop bounds of inner loops,
- store the results of conditionals, and
- store the indices of accessed data array elements.

The sizes of these arrays depend on the expressions they represent. Array-access expressions that are unit-stride with respect to a surrounding loop would need an array of size equal to the number of invocations of that loop. For expressions that are not unit-stride with respect to any surrounding loop, loop-invariant analysis is performed to determine the innermost loop with respect to which the value of the expression changes. In the worst case, this might be the immediately surrounding loop. The size of the array needed to represent these expressions is the total number of iterations of that loop across all iterations of the partitioned loop mapped to a
process. The size of arrays that store the bounds of an inner loop are the same as the number of invocations of the loop. For an array needed to store the values of a conditional, the number of times the if statement is executed should be counted.

Listing 4.2 shows the code for this phase of the inspector for the running example. Each process analyzes only those iterations that are mapped to it after the partitioning process. The number of invocations of inner loop \(j\) is tracked via counter `loop_j`. Counters `body_*` track the total number of times a loop body is executed. For conditional statements, `then_*` and `else_*` counters track the number of times the true or false branch of the statements are taken, with `if_*` counting the number of times the conditional is evaluated. All access array used in the executor have sizes equal to the value of one of these counters at the end of this phase.

In addition to this counting, since the iterations mapped to a process may be different from those analyzed while building the hypergraph, this phase of the inspector also ensures that all values of indirection arrays needed for subsequent analysis have been prefetched. Therefore, as in Section 4.1.1, all inner loops, conditionals, and assignments are guarded to check if the values of indirection scalars and indirection array elements are known. Again, this phase is completed only after all levels of indirections have been resolved.

Based on the values of these counters, the access arrays that would be needed to recreate the control flow and data accesses patterns are allocated in the local memory of every process.

**Code Generation at Compile Time**

The code generation for the loop-body of this phase is similar to the approach from Phase I. Only statements that affect the control-flow and array-access patterns
are considered. The differences of the code-generation algorithm of this phase from Algorithm 3 are

- Bounds of the loops generated for this phase that correspond to the partitionable loop are same as the original computation and its body is enclosed in an if statement that checks if the iteration is to be executed on the current process.

- Statements to increment counters loop_*, body_* are introduced without and within corresponding loop-bodies, respectively. Statements to increment if_* are inserted before the corresponding conditional statement generated for this
phase. The then_* and else_* are inserted in the generated true and false branches respectively.

- The statements that add pins and vertices to the hypergraph are removed.

The code-generation scheme described here results in some statements, that can be classified as dead-code, like the assignment to xindex at line 12 of Listing 4.2. Since the proposed algorithm is a source-to-source transformation, we rely upon the host compiler to identify and remove such statements.

### 4.1.4 Phase III: Initializing Local Access Arrays

**Runtime Functionality**

The access arrays are initially populated with the sequence of values of the corresponding expression in the original computation. Listing 4.3 shows the code generated for this phase of the inspector, for the example in Listing 1.1. Each process again
analyzes the iterations mapped to it after partitioning. For expressions that are unit-stride with respect to a surrounding inner loop, the index of the element accessed by the first iteration of every invocation of the loop is stored in the array that represents the index expression. For all other expressions, the values for all iterations are stored in arrays. The values of the loop bounds of all inner loops, as well as the results of conditionals, are also stored in arrays in this phase of the inspector.

**Code Generation at Compile Time**

To generate the code for this phase, once again the loop-body generated for each partitionable loop is enclosed within a conditional statement to analyze only the iterations mapped to a process. The body itself is generated by using Algorithm 5.

As was done for Phases I and II, ⟨IAssignment⟩s are replicated in the inspector AST with references to indirection arrays on the right-hand side of the original statement replaced with calls to function `get_elem`. For example, the right-hand side of `xindex=col[j]` at line 8 in Listing 1.1 is modified to `get_elem(id_col,j)` (line 10). There is no need for any guards in this phase since the previous phase ensured that values of all indirection array elements needed for the analysis have been prefetched.

For ⟨DAssignment⟩s, all expressions used to access data array elements are considered. For every such expression, an assignment is generated to store the array index accessed. The array that is to be used to store the value of the index expression is retrieved by function `GetAccessArray` at line 10. If the index expression used is a unit-stride expression with respect to a surrounding loop, `GetCounterVariable` returns the `loop_*` counter associated with that loop. If not, it returns the `body_*` counter associated with the outermost loop for which the expression is not loop invariant. In the presence of conditional statements between the ⟨Loop⟩ AST node for that loop
and the AST node for the ⟨DAssignment⟩, the function returns the \texttt{then\_\_\_\_} or \texttt{else\_\_\_\_} counter associated with the closest intervening conditional statement, depending on whether the ⟨DAssignment⟩ is in the true or the false branch, respectively.

Further, if the index expression is unit-stride with respect to a surrounding loop, the statement is enclosed within an \texttt{if} statement (generated by \texttt{GenIfFirstIter}) which checks if the value of the loop iterator is the same as that stored in the lower-bound array. For example, at line 12 in Listing 4.3, \texttt{access\_A} stores the value of expression \texttt{j} used to access array \texttt{A} at line 9 of Listing 1.1, and is enclosed within an \texttt{if} statement that is true for the first loop iteration.

Upon encountering a ⟨Loop⟩ node, statements to store the current lower/upper bounds of the loop in arrays are added to the inspector AST. Following this, a loop statement is added, with bounds modified to read from the array locations which were assigned to. The body of the loop is generated by processing recursively the loop body in the original computation. ⟨If⟩ statements are handled similarly. Statements to store the value of the conditional are added to the inspector AST, followed by a new conditional statement whose branches are computed by recursively traversing the true and false branches in the original code.

Having populated all access arrays with the original values of the expressions they represent, these values are now modified to point to the corresponding locations in the local copies of the data arrays being accessed using the meta-data stored for the target data array (Section 4.1.2). For global index \texttt{g} in the access array, first a binary search is done over elements 0 to \texttt{n\_unitstride\_accesses-1} of \texttt{l\_to\_g}. The returned position gives the modified value to be stored. If \texttt{g} is not found, then a binary search over elements \texttt{n\_unitstride\_accesses} to \texttt{local\_array\_size-1} is
Algorithm 5: CodeToInitializeArrays($ss, L, C$)

Input: $ss$ : Sequence of statements in the original AST
       $L$ : Access arrays for index expressions, loop bounds, and conditional values
       $C$ : Counter variables

Output: $A_P$ : AST of inspector code to populate the access arrays

begin
    $A_P = \emptyset$ ;
    foreach $s \in ss$ in order of appearance do
        if IsStatement($s$) then
            $b = \text{ConvertArraysToFunctions}(s.\text{RHS})$ ;
            $l_P = \text{NewAssignmentStmt}(s.\text{LHS}, b)$ ;
        else if IsDStatement($s$) then
            $l_P = \emptyset$ ;
            foreach $d \in \text{GetDataArrayExprs}(s)$ do
                $a = \text{GetAccessArray}(L,d.\text{Array},d.\text{IndexExpr})$ ;
                $c = \text{GetCounterVariable}(C,d.\text{IndexExpr})$ ;
                $b = \text{ConvertArraysToFunctions}(d.\text{IndexExpr})$ ;
                $e = \text{NewArrayRefExpr}(a,c) ; s_P = \text{NewAssignmentStmt}(e,b)$ ;
                if IsUnitStride($d.\text{IndexExpr}$) then
                    $l = \text{GetLoop}(c) ; s_P = \text{GenIfFirstIter}(c,s_P)$ ;
                    $l_P.\text{Append}(s_P)$ ;
                else if IsLoop($s$) then
                    $l = s.\text{LowerBound} ; u = s.\text{UpperBound}$ ;
                    $b_l = \text{ConvertArraysToFunctions}(l) ; b_u = \text{ConvertArraysToFunctions}(u)$ ;
                    $a_l = \text{GetLowerBoundArray}(L,s) ; a_u = \text{GetUpperBoundArray}(L,s)$ ;
                    $c = \text{GetLoopCounterVariable}(C,s)$ ;
                    $e_l = \text{NewArrayRefExpr}(a_l,c) ; e_u = \text{NewArrayRefExpr}(a_u,c)$ ;
                    $l_P = \text{NewAssignmentStmt}(e_l,b_l)$ ;
                    $l_P.\text{Append}(\text{NewAssignmentStmt}(e_u,b_u))$ ;
                    $s_P = \text{NewLoopStmt}(s.\text{Iterator},e_l,e_u)$ ;
                    $s_P.\text{Body} = \text{CodeToInitializeArrays}(s.\text{Body}, L, C)$ ;
                    $c_u = \text{GetBodyCounterVariable}(C,s) ; s_P.\text{Body}.\text{Append}(\text{NewIncrementStmt}(c_u))$ ;
                    $l_P.\text{Append}(s_P)$ ;
                    $l_P.\text{Append}(\text{NewIncrementStmt}(c_u))$ ;
                else if IsIf($s$) then
                    $c_P = \text{ConvertArraysToFunctions}(s.\text{Cond})$ ;
                    $a = \text{GetConditionalArray}(L,s)$ ;
                    $c = \text{GetConditionalCounterVariable}(C,s)$ ;
                    $e = \text{NewArrayRefExpr}(a,c) ; l_P = \text{NewAssignmentStmt}(e,c_P)$ ;
                    $s_P = \text{NewIfStmt}() ; s_P.\text{Cond} = e$ ;
                    $s_P.\text{Then} = \text{CodeToInitializeArrays}(s.\text{Then}, L, C)$ ;
                    $c_t = \text{GetThenCounterVariable}(C,s) ; s_P.\text{Then}.\text{Append}(\text{NewIncrementStmt}(c_t))$ ;
                    $s_P.\text{Else} = \text{CodeToInitializeArrays}(s.\text{Else}, L, C)$ ;
                    $c_e = \text{GetElseCounterVariable}(C,s) ; s_P.\text{Else}.\text{Append}(\text{NewIncrementStmt}(c_e))$ ;
                    $l_P.\text{Append}(s_P)$ ;
                    $l_P.\text{Append}(\text{NewIncrementStmt}(c_e))$ ;
                else
                    $A_P.\text{Append}(l_P)$ ;
            end
        end
    end
end
return $A_P$ ;
used to get the value to be stored. The partitioning of the data ensures that the value
\( g \) is present in \( l_{\text{to}} g \).

4.2 Executor Code

The previous section describes the run time functionality of the inspector code
and discussed the algorithms that generate the inspector code for a partitionable
loop. Here we discuss the execution and generation of the \textit{executor} code. After
completion of all phases of the inspector, the loop iterations and data arrays have
been partitioned among the processes. All access arrays have been initialized with
values that point to the appropriate locations in the local data arrays.

The executor code is similar in structure to the original code. All counter vari-
ables are first reset to 0. The lower and upper bounds of the partitioned loops are
set to 0 and the number of assigned iterations, respectively. The executor code is
generated by using Algorithm 6 to modify the original code in place. \( I\text{Statement}\)s
are removed since the control-flow and array-access patterns are explicitly represented
through access arrays. For \( D\text{Statement}\)s, all accesses to data arrays are replaced
with accesses to the corresponding local data arrays. The index expressions used to
access these arrays are also modified as necessary.

For expressions that are unit stride with respect to an inner loop, the index ex-
pression is the sum of the loop iterator and the value stored in an offset variable.
This offset variable is initialized to the value stored in the access array associated
with the index expression, subtracted with the lower bound of the loop (variable
\texttt{offset\_A} at line 16 in Listing 3.1). Since the access array stores the location of
the first element of the array accessed within the loop, adding the iterator value to
this offset allows accessing the local arrays in a manner consistent with the original computation, and in a contiguous manner. Such an expression allows for subsequent optimizations such as vectorization and prefetching, which rely on this property. To the best of our knowledge, no previously proposed compiler approaches for I/E code generation ensure this highly-desirable property.

Upon encountering ⟨Loop⟩s and ⟨If⟩s, the bounds/conditionals modified to read from arrays that were populated in Phase III of the inspector.

4.2.1 Communication Between Processes

Once the executor code has been generated for all partitionable loops, calls to perform the communication of ghost values between the processes are added to executor AST. The execution model assumes that before the execution of a partitionable loop, the value of a data element is known only at the owner process. Communication is required to initialize all ghost elements for data arrays read within the loop, before its execution. Calls to the run time to perform this communication are inserted before the partitionable loop in the executor.

For arrays whose elements are updated within the partitionable loop (using a commutative and associative operator), the ghost elements of these arrays have to be initialized to the identity element of the update operator, before the loop execution. At the end of the loop execution, the ghost elements contain partial contributions to the data element. These partial contributions from all ghost locations are communicated to the owner process, where they are combined. A call to the run time to initialize the ghost elements are inserted before the partitionable loop in the executor. The calls to combine the partial contributions are inserted after the loop. As a result,
Algorithm 6: GenerateExecutor($D,L,C,ss$)

Input: $D$ : Local data arrays
$L$ : Access arrays for index expressions, loop bounds, and conditional values
$C$ : Counter variables

InOut: $ss$ : Sequence of statements in the original AST

begin
  foreach $s \in ss$ in order of appearance do
    if IsIAssignment($s$) then
      RemoveStatement($s$);
    else if IsDAssignment($s$) then
      foreach $d \in GetDataArrayRefExprs(s)$ do
        ReplaceWithLocalArray($d$.Array,$D$);
        $c$ = GetCounterVariable($C,d$.IndexExpr);
        $a$ = GetAccessArray($L,d$.Array,$d$.IndexExpr);
        $e$ = NewArrayRefExpr($a,c$);
        if IsUnitStride($d$.IndexExpr) then
          $l$ = GetLoopStatement($c$);
          $o$ = NewVariable();
          $el$ = NewArrayRefExpr(GetLowerBoundArray($L,l$),$c$);
          $sg$ = NewAssignmentStmt($o$,NewSubtractExpr($e,el$));
          InsertBefore($l$,sg);
          ReplaceExpression($d$.IndexExpr,$el$);
        else
          ReplaceExpression($d$.IndexExpr,$e$);
        end if
      end foreach
    else if IsLoop($s$) then
      $l$ = $s$.LowerBound;
      $u$ = $s$.UpperBound;
      $al$ = GetLowerBoundArray($L,s$);
      $au$ = GetUpperBoundArray($L,s$);
      $c$ = GetLoopCounterVariable($C,s$);
      $el$ = NewArrayRefExpr($al$,$c$);
      $eu$ = NewArrayRefExpr($au$,$c$);
      GenerateExecutor($D,L,C,s$.Body);
      $cb$ = GetBodyCounterVariable($C,s$);
      $sp$.Body.Append(NewIncrementStmt($cb$));
      $s$.Append(NewIncrementStmt($c$));
    else
      $a$ = GetConditionalArray($L,s$);
      $c$ = GetConditionalCounterVariable($C,s$);
      $e$ = NewArrayRefExpr($a,c$);
      GenerateExecutor($D,L,C,s$.Then);
      $cb$ = GetThenCounterVariable($C,s$);
      $sp$.Then.Append(NewIncrementStmt($cb$));
      GenerateExecutor($D,L,C,s$.Else);
      $cb$ = GetElseCounterVariable($C,s$);
      $sp$.Else.Append(NewIncrementStmt($cb$));
      $s$.Append(NewIncrementStmt($c$));
    end if
  end foreach
end
after the communication is complete, the owner process has the correct value for the data element.

The communication pattern for both the read-updates and write-updates is similar to the MPI_Alltoallv collective, where each process sends a separate message of potentially different size to every other process. For the read-updates, for every array read within the loop, each process packs the values of the elements owned by it and ghosted on another process into a temporary buffer. The position of these elements in the local memory space of a process is obtained from the metadata information maintained by the inspector for the arrays, specifically nowned_elements and owned_elements (Section 4.1.2). After the communication, the metadata information in the arrays nghost_elements and ghost_elements are used to unpack the values received into the corresponding ghost locations on a process. For the write updates, the metadata information in nghost_elements and ghost_elements associated with the arrays updated in the loop are used to pack the values from ghost elements into a temporary buffer. The information in nowned_elements and owned_elements are used to unpack the data and update the corresponding owned elements on a process.

As the number of partitions increases, every process has to communicate with only a small number of other processes. As a result the communication costs can be further reduced by using point-to-point communication instead of the MPI_Alltoallv collective. A further improvement could be obtained by using one-sided communication APIs like ARMCI_Put provided by ARMCI [81] library.

For computations where elements of an array are assigned to (instead of updated through an operator like += or *=), there might be output dependences for a particular
data element. To resolve this dependence, the inspector tracks the last iteration of the partitionable loop that assigned to a data element. During the combination phase, the owner process selects the value received from the process that executed that iteration.

For arrays that are accessed at unit-stride from the partitionable loop in the original code (for example, \( y[i] \) in Listing 1.1), and when this unit-stride is maintained in the executor code as well (Section 3.3.4), it can be asserted statically that unique array elements are accessed by the different iterations of the partitionable loop. Setting the process that executes the iteration as the owner of the data elements eliminates the need for communication due to such accesses.

Lastly, to handle updates to \((DScalar)\)s within a partitionable loop, the generated executor code initializes the value of the scalar to the identity of the update operator used on all processes except process 0. After the loop execution, an `MPI_Allreduce` is used combine the values from all processes and communicate the resulting value to every process.

### 4.3 Overall Code Generation Algorithm

While the previous section described the algorithms to generate the three phases of the inspector and the executor for a single partitionable loop, this section outlines the complete approach used to generate the inspector/executor code for a sequence of partitionable loops that are targeted for distributed memory parallelization. Algorithm 7 describes this process.

First the code to allocate the data-structures used to build the hypergraph is generated (line 2). Next, the Phase I code for all partitionable loops are generated. For every partitionable loop, a new loop is created with bounds from the original
Algorithm 7: CodeGenInspectorExecutor($\mathcal{P}$)

InOut : $\mathcal{P}$ : Sequence of partitionable loop ASTs
Output: $\mathcal{A}_I$ : Code for the inspector

begin
$\mathcal{H}$ = InitHyperGraph() ; $\mathcal{A}_I$ = $\emptyset$ ; $\mathcal{A}_H$ = $\emptyset$
foreach $p \in \mathcal{P}$ in order do
    $[l_H,u_H]$ = BlockIterationBounds($p$.LowerBound,$p$.UpperBound) ;
    $P_1$ = NewLoopStmt($p$.Iterator,$l_H,u_H$) ;
    $P_1$.Body = GenAddVertex($\mathcal{H}$,$p$.Iterator) ;
    $[L_1,S_1]$ = CodeGenHyperGraph($\mathcal{H}$,$p$.Body) ;
    $F_1$ = InitShadowScalars($S_1$) ; $P_1$.Body.Append($F_1$) ;
    $P_1$.Body.Append($L_1$) ; $\mathcal{A}_H$.Append($P_1$) ;
    $\mathcal{A}_H$ = NewDoWhile("DoneGraphGen()",$\mathcal{A}_H$) ; $\mathcal{A}_I$.Append($\mathcal{A}_H$) ;
    $\mathcal{A}_I$.Append(CodeToPartitionIterations($\mathcal{H}$)) ;
    $D$ = CodeToAllocateLocalData($\mathcal{H}$) ; $\mathcal{A}_I$.Append($D$) ;
    $C$ = DeclareCounterVariables($\mathcal{P}$) ; $\mathcal{A}_I$.Append($C$) ;
    $\mathcal{A}_C$ = $\emptyset$
    foreach $p \in \mathcal{P}$ in order do
        $\mathcal{A}_C$.Append(InitCounterVariables($p$)) ;
        $L_2$ = NewIfStmt() ; $L_2$.Cond = GenerateIsHome() ;
        $[L_2$.Then,$S_2]$ = CodeToGetAccessArraySizes($p$.Body,$C$) ;
        $F_2$ = InitShadowScalars($S_2$) ; $P_2$.Body.Append($F_2$) ;
        $P_2$.Body = $L_2$ ; $\mathcal{A}_C$.Append($P_2$) ;
        $\mathcal{A}_C$ = NewDoWhile("DoneCounters()",$\mathcal{A}_C$) ; $\mathcal{A}_I$.Append($\mathcal{A}_C$) ;
        $I$ = CodeToAllocateAccessArrays($C$) ; $\mathcal{A}_I$.Append($I$) ;
        $\mathcal{A}_P$ = $\emptyset$
        foreach $p \in \mathcal{P}$ in order do
            $L_3$ = NewIfStmt() ; $L_3$.Cond = GenerateIsHome() ;
            $L_3$.Then = CodeToInitializeArrays($p$.Body,$I$,$C$) ;
            $P_3$.Body = $L_3$ ; $\mathcal{A}_P$.Append($P_3$) ;
            $\mathcal{A}_P$.Append(CodeToRenumberAccessArrays($C,D,I$)) ;
            $\mathcal{A}_I$.Append($\mathcal{A}_P$) ;
        foreach $p \in \mathcal{P}$ in order do
            nlocal = InsertBefore($p$,InitNLocalIters($p$)) ;
            ModifyLowerBound($p$,0) ; ModifyUpperBound($p$,nlocal) ;
            GenerateExecutorCode($D,I,C,p$.Body) ;
            InsertCommunicationCode($D,I,p$.Body) ;
        return $\mathcal{A}_I$ ;

loop modified to be block-partitioned (line 4). A call to the function $\text{AddVertex}$ is added to the body of the new loop (line 6). The rest of the loop body is generated by traversing the AST of the original partitionable loop as described in Algorithm 3.
Statement to initialize the shadow scalars associated with the indirection scalars to false are added to the generated loop-body (line 8). After all partitionable loops have been processed, all the generated inspector loops are enclosed within a do..while loop with the conditional being a call to DoneGraphGen (line 10).

Next, the code to partition the hypergraph and to allocate local arrays is appended to the inspector AST at lines 11 and 12, respectively. For every partitionable loop, the code to compute the sizes of the access arrays, as described in Section 4.1.3, is generated as follows. A new loop is generated with the same loop bounds as the original loop. The body for this loop is a conditional statement to check whether the current iteration is local to a process (line 18). The true branch of this conditional statement is generated by analyzing the body of the partitionable loops using function CodeToGetAccessArraySizes (line 19), which implements the functionality described in Section 4.1.3. Once all partitionable loops have been processed, the loops generated in this phase are enclosed within a do..while loop with the conditional being a call to function DoneCounters (line 22).

The code to allocate all access arrays is appended to the inspector AST at line 23. Following that, the loops for Phase III of the inspector code for all the partitionable loops is generated. The bounds of the generated loop is same as the original loop. The loop body is generated using Algorithm 5. As was done for Phase II, this loop-body is enclosed within a conditional to execute only those iterations mapped to a process. Finally, the code to modify the values stored in access arrays is appended to the inspector AST (line 30).

The executor code is generated by modifying in-place the bounds of the original partitionable loop. The code to retrieve the number of iterations to be executed on
a process is inserted before the loop (line 33). The upper-bound of the partitionable
loop is set to this value, with the lower-bound set to 0 (line 34). The loop-body is
modified to use the local data structures on each process as described in Algorithm 6.
Following this, the code to perform the communications of ghost values (described in
Section 4.2.1) is inserted before and after every partitionable loop (line 36).

4.4 Evaluation

The techniques and algorithms outlined in Sections 4.1, 4.2 and 4.3 were imple-
mented using the ROSE compiler infrastructure [82]. To find code regions that are
amenable to the transformation described, PolyOpt [12] was modified to be able to
extract parts of the program that satisfy the grammar shown in Figure 3.4. By de-
default, PolyOpt can recognize code regions that are affine, also known as static control
parts or SCoP. This was modified to allow references to arrays within loop-bound
expressions, conditionals and array index expressions. The current implementation
targets C-based codes, but the technique can be used to target codes in other similar
languages like Fortran.

The hypergraph partitioner, PaToH [78] was used for partitioning the hypergraph
built by the inspector at run time. While it supports multi-constraint hypergraph
partitioning, it is sequential. Since the generated inspector code is parallel, this se-
quential step would add considerable overhead to the inspection times during the
distributed memory execution. While there are no parallel hypergraph partitioners
that support multi-constraints partitioning yet, there do exist parallel graph parti-
tioners that offer this functionality. ParMetis [83] is one such, state of the art, parallel
graph partitioning library. To use this in the inspector, the hypergraph generated by
the inspector was converted to a graph by adding an edge between every pair of vertices belonging to the same net. The weight of an edge in the graph was set to the sum of the weights of all the nets the two connected vertices belonged to. This graph representation is less accurate than the hypergraph representation, and might result in higher communication cost in the executor. To evaluate the impact of using a representation of the iteration-to-data affinity to partition the computation, the execution times using a simple block partitioning scheme was also measured.

All experiments were performed on a cluster of Intel Xeon E5360 processors with 4 cores per node at a clock speed of 2.53 GHz, connected over an Infiniband. MVAPICH-1.9 was used for MPI communication with GA-5.2 used for the one-sided ARMCI communication. All benchmarks/applications presented here were compiled using the Intel Compiler-13.1. All running times reported in this section are average of 5 runs. Very limited variation was found in the times over the different runs. All performance improvements are measured with respect to the sequential execution times of the original code.

4.4.1 Benchmarks

We first present the performance of the generated code from benchmarks with data dependent control flow and array access expressions. Each of the following benchmarks have a sequence of partitionable loops enclosed within an outer time or convergence loop, with the control-flow and array-access pattern within the loop remaining the same for every iteration of that outer loop. Function calls within the loops of interest were inlined manually so that the input to the prototype compiler falls within the grammar described in Figure 3.4.
183.equake

This benchmark is part of the SPEC2000 benchmark suite [84], and simulates seismic wave propagation in large basins using an unstructured 3D grid of tetrahedrons. It consists of a sequence of partitionable loops enclosed within an outer time loop. The SPEC ref data size was used for the evaluation, which uses a grid containing 30169 tetrahedral elements. A manual MPI implementation of this benchmark was also developed for evaluation purposes. Figure 4.1a shows that the performance
of the generated executor code for all three partitioning schemes is considerably better than the performance of the manual MPI implementation. The peak performance is achieved for 32 processes with the partitions generated by PaToH and ParMetis achieving the best execution time. Due to the small problem size used for evaluation, as the number of partitions increase, the ratio of ghost cells to the total footprint of the computation increases. This results in increased communication costs as the number of processes increase. For more than 64 processes the communication cost dominates the execution time resulting in a loss of performance.

Figure 4.1b shows that the overhead of the inspector while using ParMetis or block partitioning is negligible, but with PaToH, the sequential nature of the partitioner adds considerable overhead. Figure 4.1c shows the break-up of the inspector times. Phase I and Phase II corresponds to parts of the inspector described in Sections 4.1.1 and 4.1.3, respectively. The cost of the call to the partitioner (PaToH and ParMetis) are shown separately. Phase III shows the time to allocate the local data arrays (Section 4.1.2) and to execute the last phase of the inspector (Section 4.1.4). With PaToH, the inspector cost is dominated by the partitioner. With Metis, the cost to build the hypergraph, and partitioning it are almost equal due to the parallel graph partitioner. The cost for the Phase II and III of the inspector turn out to be negligible in this case due to the small problem size used.

Conjugate Gradient

The conjugate gradient (CG) method to solve linear system of equations consists of five partitionable loops within a convergence loop. Two sparse matrices, hood.rb and tmt_sym.rb, from the University of Florida Sparse Matrix Collections [85], stored in CSR format were used as inputs for evaluation. While hood has 220542 rows and
Figure 4.2: CG Kernel with \textit{hood.rb} and \textit{tmt\_sym.rb}
9895422 non-zero elements, \textit{tmt\_sym} has more (726713) rows and lesser (5080961) non-zero elements. The structure of the latter is such that the non-zero elements fall along diagonals of the matrix.

Figures 4.2a and 4.2d show that the executor code achieves good scaling overall with better than ideal scaling between 8 and 32 processes, due to the partitions becoming small enough to fit in caches. Using block-partitioning gives good performance with \textit{tmt\_sym} but not for \textit{hood}. Due to the structure of the latter, block partitioning results in a larger number of ghosts cells and therefore higher communication costs, demonstrating the need for modeling the iteration-data affinity. The inspector overheads reduce the overall speed-up achieved, as shown in Figures 4.2b and 4.2e. This cost could be further amortized in cases where the linear system of equations represented by the matrices are solved repeatedly, say within an outer time loop, with the same non-zero structure. Such cases are common in many scientific applications.

The performance of the executors was also compared to a manual implementation using PETSc [86], which employed a block-partitioning of the rows of the matrix. For \textit{hood}, Figure 4.2a shows that the performance of the generated executor code while using PaToH and ParMetis out-performs the manual PETSc implementation. The performance of the latter drops off due to the same reason the performance of the block-partitioned scheme drops off. With \textit{tmt\_sym}, the generated executors perform on par with the manual implementation for all three partitioning schemes (Figure 4.2d) upto 128 processes.

The break down of the inspector overheads for each of the matrices with different partitioning schemes are shown in Figures 4.2c and 4.2f. It is interesting to note
that the execution time of Phase III of the inspector is lower when using Metis than
with Block. Since this phase populates the indirection arrays needed by the executor,
lesser ghosts results in lesser computation. At the same time, the block partitioning
scheme has lesser overhead for Phase II of the inspector since the iterations of the
partitionable loop analyzed in this phase are same as those analyzed in Phase I. The
values of indirection arrays have already been prefetched, reducing the amount of
communication required for this phase.

**P3-RTE Benchmark**

This benchmark solves the radiation transport equation (RTE) [87] approximated
using spherical harmonics on an unstructured physical grid of 164540 triangular cells.
Finite-Volume Method is used for discretizing the RTE with Jacobi method used for
solving the system of equations at each cell center [88]. The different partitionable
loops iterate over cells, faces, nodes, and boundaries of the domain, and are enclosed
within a convergence loop.

Figure 4.3a compares the executor times for the three schemes with a manual MPI
implementation, which uses a partitioning of the underlying physical grid to partition
the computation. Since the partitioning scheme used by the auto-generated code
groups together iterations that touch the same data elements, the generated partitions
are similar to those generated by the scheme used by the manual MPI implementation.
As a result, the executor code while using PaToH or ParMetis achieves performance
comparable to the manual MPI implementation up to 32 processes. Since the block-
partitioning scheme does not do this, the performance of the corresponding executor
code achieves poor performance.
Past 32 processes, the manual implementation continues to achieve scalable performance by replicating some of the computation on multiple processes, significantly reducing the communication costs. Since our scheme strictly partitions the computation across processes without any replication, the generated executor code has to perform five collective communications instead of two such communications performed by the manual implementation. Automatically identifying computations which when replicated significantly reduce communication costs would be an interesting avenue to explore. Figure 4.3b shows that the inspector overhead is negligible even when using the sequential PaToH partitioner.
miniFE-1.1

This benchmark is form the Mantevo suite of mini-applications developed by Sandia National Laboratories [89]. It uses an implicit finite-element method over an unstructured 3D mesh to solve a system of equations. A problem size of 100 points along each axis was used for the evaluation. The execution times shown here are for a single solution of the system of equations. The inspector in this case was placed immediately before a convergence loop that surrounds the partitionable loops. The typical use of such computations is to repeatedly solve the system within an outer time-stepping loop. If the underlying mesh is static, the inspector can be hoisted out of this time loop too. The suite also provides a manual MPI implementation of the computation which was used for comparison.

Figure 4.4 compares the running times for the executors (using ParMetis, PaToH, and block-partitioning) with the execution time of the manual MPI implementation. Up to 128 processes, the performance of the auto-generated executor is on par with the manual implementation. At higher processes, the communication costs in the manual implementation is reduced by overlap of communication and computation. While the current auto-generated code doesn’t do this automatically, it can be incorporated into the current code-generation scheme by static analysis of the def-use relationship of arrays between the different partitionable loops.

Figure 4.4b and 4.4c show the impact of the inspector overheads on the total speedup achieved and a break down of the inspector execution time. Since the actual running time of the original sequential application is not very high (32.22s) even for the large problem size chosen here, the cost of the inspector dominates the overall running time. With PaToH, the cost of the sequential hypergraph partitioner is
almost 10 times the total execution time of the sequential code. With Metis, the cost of partitioning reduces drastically. The total inspector cost is now dominated by the cost of allocating local data-structures. A similar trend can be observed with the block-partitioning scheme. It should however be noted that solving the system of equations for a mere 100 timesteps would amortize the cost of the inspector, even if the sequential hypergraph partitioner were used.
4.4.2 Impact of Exploiting Contiguity

The code-generation scheme described in this paper takes special care to preserve the contiguous accesses present in the original code within the generated executor code. The advantage of this are,

- At most one read from an indirection array is needed for every set of contiguous access to data arrays reducing the total memory request made by the executor code
- The size of the indirection array needed is significantly reduced resulting in a small footprint of the executor
The resulting array access expression might enable future compiler optimizations like memory prefetching which could further improve the executor code performance.

Figure 4.5 shows the improvements in total run time of the executor with and without this optimization applied to inner-loops alone for all the benchmarks where this optimization was applied. The benefit of the optimization is especially important in benchmarks that have a high footprint, like the CG kernel with \texttt{hood.rb} and miniFE-1.1 (Figures 4.5c and 4.5b), resulting in a 25\% reduction in running times. Since the footprint for CG kernel with \texttt{tmt\_sym.rb} is almost half of that with \texttt{hood.rb}, the maximum benefit for the former occurs at 8 processes and for the latter at 16 processes. As the number of processes used increases, the footprint of the computation on a process fits in some level of cache, reducing the benefit of the optimization. Similarly, for \texttt{183.equake} which has a small footprint to begin with, the benefit is within 5\%.

4.4.3 Application: OLAM

The previous section showed the performance obtained from automatic transformations of codes that are representative of the compute-intensive parts of a wide-variety of scientific computing applications. To evaluate the effectiveness of the approach in a real-world application, we applied these techniques to parallelize OLAM (Ocean, Land, and Atmosphere Modeling) \cite{90}, a software package used for climate simulations of the entire planet written in Fortran 90. It employs finite-volume methods of discretization to solve for physical quantities such as pressure, temperature, and wind velocity over an 3D unstructured grid consisting of 3D prisms covering the
surface of the earth. Physical quantities are associated with centers of prisms and prism edges. The input grid used contained 155520 prisms. For evaluation purposes, the atmospheric model simulation module of OLAM was targeted for distributed memory parallelization. This module consists of 13 partitionable loops surrounded by a sequential loop that typically executes hundreds of thousands of iterations, completely amortizing the inspector cost. Here we report the execution times for 30000 iterations, using PaToH for partitioning the computation. Since the current implementation targets C codes and does not incorporate inter-procedural analysis, it was not possible to automatically generate the distributed memory code for this complex application. Therefore, we manually implemented the code that would be generated by a full-fledged compiler, using the same run time library for partitioning and communication as the benchmarks discussed previously.

We compared the performance of our implementation with a reference MPI implementation developed by domain experts. The latter uses an efficient domain decomposition scheme to partition the computation across the MPI processes. Figure 4.6
shows that up to 32 processes, the performance of the code generated by the transformation scheme (including inspector time) is on par, if not better, when compared to the manual MPI implementation. Past that, the efficient domain decomposition scheme used by the manual MPI implementation results in fewer ghosts and therefore, lower space and communication overheads. These factors contribute to a better than ideal scaling achieved by the manual implementation. Note that the I/E version still achieves ideal scaling.
CHAPTER 5

Distributed Memory Code-generation for Irregular-Outer Regular-Inner Computations

5.1 Motivation

It is common in many scientific computing applications that indirections are used only at outer loop levels, while being affine at the inner loop levels. An example of such a computation is shown in Listing 5.1. The code there is representative of applications like Chombo [91], where the physical domain is discretized using one or more boxes. Each box is further discretized using structured grids. Boxes at the coarsest level of refinement span the entire physical domain. Boxes at finer levels of refinement are colocated with a unique box that uses a coarser level of refinement. Listing 5.1 uses two levels of refinement with the finer level using a resolution twice that of the coarser level along each dimension. The control flow and data access pattern for an iteration of the outer loop \( k \) depends on indirection arrays like \texttt{fine_boxes}, \texttt{ftoc}, \texttt{start_x}, etc. At the same time, the value of all expressions of the form \((\text{Array})[\ldots]\) in loop bounds and array index expressions depend only on the value of the outer loop iterator. Standard compiler techniques like loop invariant code motion can be employed to bring the computation within loop \( i \) into a form that can be recognized
#pragma parallel
for ( k = 0; k < nf; k++ ){
    fbox_ID = fine_boxes[k];
    cbox_ID = ftoc[k];
    for( i = start_y[fbox_ID]/2; i < end_y[fbox_ID]/2; i++)
        for( j = start_x[fbox_ID]/2; j < end_x[fbox_ID]/2; j++){
            int fpt_y = 2*i - start_y[fbox_ID];
            int fpt_x = 2*j - start_x[fbox_ID];
            int cpt_y = i - start_y[cbox_ID];
            int cpt_x = j - start_x[cbox_ID];
            phi[cbox_ID][cpt_y][cpt_x] =
                ( phi[fbox_ID][fpt_y][fpt_x] + phi[fbox_ID][fpt_y][fpt_x+1] + phi[fbox_ID][fpt_y+1][fpt_x] + phi[fbox_ID][fpt_y+1][fpt_x+1] ) / 4.0;
        }
    }
}

Listing 5.1: MultiGrid Example

by polyhedral compilers like PolyOpt [12]. Such a pattern of computations can be classified as *Irregular-Outer Regular-Inner*

Further, in Listing 5.1, the outermost loop, \( k \) is parallel and can be targeted for distributed memory execution. As explained previously, for effective parallelization of the computation both the iterations and the data need to be partitioned across processes. The use of indirection arrays prevent purely static techniques from accurately computing the data elements touched by iterations of the partitioned loop, hampering them from effectively partitioning the data. The inspector/executor approach to distributed memory parallelization described in Chapter 4 could be used to effectively partition the data, but has the following limitations,

- It fails to leverage polyhedral compilation techniques to gather as much information as possible at compile time. This leads to unnecessary inspector
overhead, since the inspector code collects information about the computation that is available statically.

- The generated executor code contains artifacts that hamper further optimization of the generated executor code resulting in degraded on-node performance.

In this chapter, we describe a framework that extends the abstraction used for affine codes to represent computations similar to the code in Listing 5.1. The developed framework addresses both the issues highlighted above, reducing both the inspector costs and generating executor code that has a structure similar to the original code, allowing for better on-node performance.

5.2 Representing Irregular Computation using Sets and Maps

In this section, the various data structures used to represent computations are introduced. The two fundamental data structures used to represent computations and communications are *sets of integer points* and *maps between two sets of integer points*. These data structures are similar to the one developed within the Omega Calculator [92]. While the data structures there were developed to express affine computations, here they are extended to allow reasoning of computations that have a mix of irregular and regular program regions.

5.2.1 Integer Sets

A convenient way to represent the dynamic executions of a statement surrounded by *for* loops is to associate an integer point in a multi-dimensional space for each
instance of the statement, such that its coordinates represent the value of the surrounding loop iterators for that instance. This is a classical concept in affine compilation, referred to as the *iteration domain* of a statement. A similar set of integer points is used to describe the set of data elements accessed. Here, the coordinates of each point captures the value used in the subscript function of an array reference.

**Definition 1 (Integer Set)** An integer set \( S \) is a set of integer points \( \bar{x} \in \mathbb{Z}^d \). The dimensionality of the set is noted \( d \), its cardinality is noted \( \#S \). With \( \bar{x} : (i_1, \ldots, i_d) \) we note:

\[
S : \{(i_1, \ldots, i_d) \mid \text{constraints on } i_1, \ldots, i_d\}
\]

Standard operations on sets can be used, such as difference \( \setminus \), intersection \( \cap \) and union \( \cup \), as well as computing its image through a map/function. However, computing the result of these operations at compile time depends on the structural properties of the set. There exists numerous sub-categories of integer sets, each of which have different properties regarding the ability to compute them at compile time. When the constraints are conjunction of affine inequalities involving only the variables \( i_i \) and parameters (constants whose value is not known at compile time), then the set is a polyhedron and all operations can be computed statically. When the set is defined using disjunctions of affine inequalities then it is a union of convex polyhedra. When the set is the intersection of a polyhedron and an integer affine lattice then it is a \( \mathbb{Z} \)-polyhedron [93], which can also be computed statically using the Integer Set Library [94]. When it involves affine inequalities of variables, parameters and functions whose value depends only on the value of their arguments, some operations may be computable at compile time with the Sparse Polyhedral Framework [65] using
1 for (i = 0; i < N; ++i)
2     for (j = lb[i]; j < ub[i]; ++j)

Listing 5.2: Sample SpMV Kernel

uninterpreted functions, or using an inspector. When the set is arbitrary, an inspector
is needed.

5.2.2 Set Slicing

One key operation to reason about distributing a computation is slicing, that is, taking a particular subset of a set. Intuitively, an integer set slice $S_I$ is a subset of $S$ that is computed using another integer set $I$.

**Definition 2 (Integer Set Slice)** Given an integer set $S$, the integer set slice $S_I$ is $S_I = S \cap I$.

Slicing can be used to extract polyhedral subsets from an arbitrary integer set. To achieve this we focus on a particular kind of slicing where the set $I$ is a polyhedron made only of affine inequalities of the variables and parameters not occurring in the original computation but which we introduce for modeling purpose. For example, the iteration domain for the statement $S$ in Listing 5.2, i.e., the set of dynamic instances of $S$ can be written:

$$S = \{(i, j) \mid 0 \leq i < N \land lb[i] \leq j < ub[i]\}$$

The expressions $lb[i]$ and $ub[i]$ are not parameters: they may take different values for different values of $i$. However, for a given $i$, these expressions are constants and can be viewed as parameters. Let us now define the slice $I_1$, for $p_1 \in Z$, as $I_1 =$

87
\{(i, j) \mid i = p_1\}. The set \(\mathcal{I}_1\) is a set of two-dimensional integer points, with the first dimension set to a fixed but unknown value. The second dimension is unrestricted, this polyhedron is in fact a cone. The slice \(\mathcal{S}_{\mathcal{I}_1}\) is defined as follows:

\[
\mathcal{S}_{\mathcal{I}_1} = \mathcal{S} \cap \mathcal{I}_1 = \{(i, j) \mid 0 \leq i < N \land lb[i] \leq j < ub[i] \land i = p_1\}
\] (5.1)

This set now necessarily models a single point (e.g., a single loop iteration) along the first dimension. To model two different iterations of the outer loop, one can simply introduce another parameter \(p_2 \in \mathbb{Z}\), with \(p_1 \neq p_2\) for a slice \(\mathcal{I}_2\) where \(i = p_2\), to get:

\[
\mathcal{S}_{\mathcal{I}_2} = \{(i, j) \mid 0 \leq i < N \land lb[i] \leq j < ub[i] \land i = p_2\}
\]

Consequently, a set containing two arbitrary but different iterations of the outer loop is simply the union \(\mathcal{S}_{\mathcal{I}_1} \cup \mathcal{S}_{\mathcal{I}_2}\) with \((p_1 > p_2) \lor (p_1 < p_2)\) as additional constraints. Two consecutive iterations can be modeled the same way, with \(p_2 = p_1 + 1\).

In addition to the ease of modeling of subsets of loop iterations (typically arising from the iterations of the partitionable loop(s) to be executed on a processor), a key property on the computability of these subsets has arisen: when fixing \(i\) to a unique value, the subset obtained is now a standard polyhedron. This is because the expressions \(lb[i]\) and \(ub[i]\) are necessarily constants for a fixed value of \(i\). We can now view them as parameters, noted \(lb[p_1]\) and \(lb[p_2]\) for instance, and observe that a (union of) slice(s) containing a single iteration of the outer loop is now a (union of) classical polyhedra, which can be manipulated at compile time using for instance ISL. We do not require the use of uninterpreted functions, and enable polyhedral optimization on the computation to be executed on a particular processor, at the sole expense of using extensively unions of convex sets.
5.2.3 Integer Maps

The second data structure used associates, or maps, integer points to other integer points. A typical use is to represent data elements accessed by a loop iteration.

Definition 3 (Integer Map) A map $M$ defines a function from an integer set of dimension $d$ to another integer set of dimension $e$, written as:

$$M : \{(i_1, ..., i_d) \rightarrow (o_1, ..., o_e) \mid \vec{o} = M(\vec{i})\}$$

where $\vec{i} : (i_1, ..., i_d)$ and $\vec{o} : (o_1, ..., o_e)$

In a manner similar to integer sets, the tractability of $M$ depends on the form of the function $M$. If $M$ is a matrix of integer coefficients (e.g., an affine function), and is applied to a polyhedral set, then the output of the map (that is, the set of points which are the image of the input set by the map) can be computed statically, as a union of polyhedral sets. For example, consider the map representing the relationship between the iteration space and the data space for the reference $A[j]$, expressed as $D_A : \{(i,j) \rightarrow (o_1) \mid o_1 = j\}$. To represent the set of data accessed by the entire computation for this reference, we note $D_A(S) : \{(o_1) \mid o_1 = j \land \vec{x} \in S\}$. Here $\vec{x} \in S$ is only a notation shortcut for the inequalities on $i$ and $j$ defining $S$. For a particular slice $S_{xi}$, this set is polyhedral. Therefore, the set of distinct data elements accessed by this reference can be computed at compile time. A code scanning exactly this set can then be generated, using for instance the CLooG polyhedral code generator [95].

The map for the reference $B[col[j]]$ is $D_B : \{(i,j) \rightarrow (o_1) \mid o_1 = col[j]\}$. Here, since $col[j]$ is not an affine function, its value for different $j$ is not known at compile time. Consequently, the data space $D_B(S) : \{(o_1) \mid o_1 = col[j] \land \vec{x} \in S\}$ will require an
inspector to be properly computed. We remark that in our formalism, we will express the data spaces of each reference to model the data that needs to be communicated, and determine the need of an inspector by simply observing if the sets and maps of interest are polyhedral sets and affine maps. If not, then we use an inspector to compute them.

We conclude with the definition of the data space of an array which is simply the union of the data spaces touched by each reference to this array.

Definition 4 (Data space for an array) Given an array $A$ and a collection of $n$ references to it $D_A^1, ..., D_A^n$. To each reference $k$ is associated an iteration set $S_{D_A^k}$. The set of distinct array elements accessed by the computation is $F_A : \bigcup_{i=1}^n D_A^k(S_{D_A^k})$.

5.3 Partitioning the Computation

The partitioning of the iterations and the data across nodes can be described in terms of the quantities defined in Section 5.2.

5.3.1 Partitioning the Iteration Space

The set $K^q$ defines the set of iterations of a partitionable loop that are to be executed on a particular processor $q$. Depending on how the distribution scheme is determined (cyclic, block-cyclic, etc. or using run-time hypergraph partitioning) the set $K^q$ of iterations of the partitioned loop(s) executed by $q$ may be a consecutive subset of the loop iterations (thereby defined using affine inequalities) or an arbitrary, non-consecutive subset such as with hypergraph partitioning [77].

First, a slice of the original iteration domain, $S_I$, can be constructed by computing an intersection of the original iteration space $S$ with a set $I_p$. In set $I_p$, iteration
space dimensions whose iterators are used in the index expression of array accesses appearing in constraints of the various sets and map descriptions are fixed to a newly introduced parameter \( p \) (e.g., \( i = p \) if the iterator \( i \) appears in a loop bound expression such as \( lb[i] \)). This allows all such expressions to be treated as parameters for a slice. All such dimensions will be refered to as the *irregular dimensions* of the computation. The outermost dimension is always set as irregular. If the number of such dimensions is \( r \), the set \( I_p \) is a cone of same dimensionality as \( S \), with the values of the \( r \) irregular dimensions each set to a newly introduced vector of parameter \( \vec{p} \in \mathbb{Z}^r \). The set \( C \) contains all the different \( \vec{p}_i \) needed to cover the full iteration space of the irregular dimensions with one distinct \( \vec{p}_i \) per distinct iteration \( i \) of the irregular loop(s), with the property that \( \vec{p}_i \neq \vec{p}_j \) for \( i \neq j \). The original complete iteration space is therefore the union of all slices,

\[
S = \bigcup_{\vec{p} \in C} S_{\vec{p}}
\]

In the parallel execution, each process executes a subset of the slices from the original computation, defined using \( \vec{p} \in C^q \subseteq C \). The local iteration space \( S^q \) is simply the union of all slices executing on \( q \),

\[
S^q = \bigcup_{\vec{p} \in C^q} S_{\vec{p}}
\]

The set \( C^q \) is constructed from \( K^q \) by taking one distinct distinct \( \vec{p}_i \in C \) per distinct element \( k \in K^q \), and adding the constraint that \( p^1_i = k \) (\( p^1 \) is the value of the first dimension of \( \vec{p} \)). A key observation is that if the computation is affine and the outermost loop is block partitioned then the set \( K^q \) is known at compile-time, and the set \( C^q \) is reduced to a single parameter \( \mathcal{C}^q = \{ \vec{p} : b_1 \leq p^1 < b_1 + B \} \) where \( b_1 \) is
a newly introduced parameter and $B$ is the block size. Consequently the slice $I_p$ will contain $B$ iterations, and the above union can be fully computed statically.

5.3.2 Data Partitioning and Ghost Communication

In our execution model, the data is partitioned amongst the processes such that each process has all data needed to execute the set of iterations of the partitionable loop(s) mapped to it. If $D_A$ is an integer map used to represent accesses to array $A$, for each slice of the iteration space, the elements of array $A$ accessed by it can be computed as,

$$F_{A,I_p} = D_A(S_{I_p})$$  \hspace{1cm} (5.4)

Eq (5.4) represents a slice of the local data space of array $A$. A union of these slices gives the local data space on a process.

$$F^q_A = \bigcup_{\vec{p} \in C^q} D_A(S_{I_p})$$  \hspace{1cm} (5.5)

For affine computations, when the outer loop is block-partitioned, $C^q$ can be defined as $C^q = \{p_1 : b_1 \leq p_1 < b_1 + B\}$. Consequently, $F^q_A$ can also be computed statically.

In general, the same data element might be accessed by iteration mapped to two different processes. In such cases, one of the processes is assigned as the owner of the element and the location of the element on the other processes are treated as ghosts. The location at the owner contains the correct value of the data element, with ghost locations storing a snapshot of the value at the owner. Since the partitioned loops are parallel, these elements are either read from or are updated through commutative and associative operations. The loop itself can be executed in parallel without any communication as long as
• The ghost locations corresponding to elements that are read within the loop are updated with the value at the owner before the loop execution.

• The ghost locations corresponding to elements that are updated within a loop are initialized to the identity of the update operator used (0 for ‘+’, 1 for ‘*’) before the loop execution. After the loop execution, the ghost locations contain partial contributions to the final value and are communicated to the owner process where values from all ghost locations are combined.

To setup the communication between processes, we define a set \( O^q_A \) which contains the all elements of array \( A \) that are owned by process \( q \). This set could either be decided at compile time (using block or cyclic distribution of array elements), or could be computed based on run time analysis that uses the iteration-to-data affinity, similar to Section 4.1.2. Since each array element has a unique owner, \( O^q_A \cap O^{q'}_A = \emptyset \) if \( q \neq q' \). Note that the choice of the set \( O^q_A \) does not change the communication volume as long as \( O^q_A \subseteq F^q_A \) \( \forall 0 \leq q < N \).

On a process \( q \), the set of ghost locations for array \( A \) which are owned by process \( q' \) can be computed as follows:

\[
G^{q,q'}_A = F^q_A \cap O^{q'}_A
\]  \hspace{1cm} (5.6)

This gives the elements of array \( A \) on process \( q \) that are

• Received from process \( q' \) if \( A \) is read within the partitioned loop

• Sent to process \( q' \) if \( A \) is written within the partitioned loop.

To complete the setup for communication, we also need to compute the set of all ghost locations on process \( q' \) that are owned by process \( q \). This can be computed as:

\[
O^{q,q'}_A = F^{q'}_A \cap O^q_A
\]  \hspace{1cm} (5.7)
\( O_{A}^{q,q'} \) gives the elements of array \( A \) on process \( q \) that are

- Sent to process \( q' \) if \( A \) is read within the partitioned loop
- Received from process \( q' \) if \( A \) is written within the partitioned loop

Computing \( O_{A}^{q,q'} \) requires computing the data space for iteration space slices mapped to process \( q' \) on process \( q \). Since this process has to be repeated for all \( q' \in [0, N - 1] - q \), this requires enumerating all the iterations space slices in \( \mathcal{C} \) on all the processes. To avoid this, since \( G_{A}^{q,q'} = O_{A}^{q',q} \), each process computes only \( G_{A}^{q,q'} \) and communicates this information to process \( q' \) for all \( q' \in [0, N - 1] - q \). Process \( q' \) uses this information to compute \( O_{A}^{q',q} \).

### 5.4 Syntactic Structure of Target Loops

\[
\begin{align*}
(\text{Start}) & ::= (\text{Loop}) \\
(\text{Loop}) & ::= \text{'for'} \ ('(' \text{Iterator} \ ' = ' \text{IExpr} \ ') ; (' \text{Iterator} \ '<' \text{IExpr} \ ') ; (' \text{Iterator} \ '++' ')') \text{(ElementList)} \\
(\text{ElementList}) & ::= (\text{Element}) | ('\{' \text{ElementList} \text{Element} \ '}') \\
(\text{Element}) & ::= (\text{IAssignment}) | (\text{DAssignment}) | (\text{Loop}) | (\text{If}) \\
(\text{If}) & ::= \text{'if'} \ ('(' \text{IExpr} \ ')') \text{(ElementList)} \text{'else'} \text{(ElementList)} \\
(\text{IAssignment}) & ::= \text{(IScalar) = (IExpr) ;'} \\
(\text{DAssignment}) & ::= \text{(DScalar) (AssignOp) (DExpr) ;'} \\
(\text{IExpr}) & ::= \text{Affine expressions of (BasicIExpr)} \\
(\text{DExpr}) & ::= \text{Side-effect-free expressions of (BasicDExpr)} \\
(\text{BasicIExpr}) & ::= \text{(IScalar) | (Iterator) | (IArray) '[' (IExpr) ']'} \\
(\text{BasicDExpr}) & ::= \text{(DScalar) | (DArray) '[' (IExpr) ']'} \\
(\text{AssignOp}) & ::= \text{‘=’ | ‘+=’ | ‘*=’ | ...}
\end{align*}
\]

Figure 5.1: Syntactic structure of annotated loops
Loops that are parallel are annotated with \texttt{#pragma parallel}, as shown in line 1 of Listing 5.1. Annotated loops cannot be nested within each other. Dependence between the iterations of an annotated loop can only be through associative and commutative reduction operations. Further, all loops are assumed to have unit increments. Figure 5.1 shows the syntactic structure of loops targeted in this chapter. The difference between this grammar and 3.1 is that \langle IExpr \rangle s are affine expressions involving \langle BasicIExpr \rangle s.

The input code is pre-processed to make the subsequent transformations easier. \langle IAssignment \rangle statements are commonly used as place holders for expressions used in index expression and loop-bounds, for example, statements that assign to \texttt{fID}, \texttt{cID}, \texttt{fy}, \texttt{fx}, \texttt{cy} and \texttt{cx} in Listing 5.1. The pre-processing step described here removes as many \langle IAssignment \rangle statements as possible. All \langle IExpr \rangle s within the annotated loop are analyzed. For every \langle IScalar \rangle used in them, the set of reaching definitions of this scalar is obtained. All reaching definitions are necessarily from \langle IAssignment \rangle s. If there is a single reaching definition, then the current reference is replaced with the expression on the right-hand side of that \langle IAssignment \rangle. Once all \langle IExpr \rangle s have been analyzed, dead-code elimination can be used to remove all \langle IAssignment \rangle s whose definitions are not used. This process is continued till no more \langle IAssignment \rangle s are removed. Listing 5.3 shows the result of this pre-processing step on Listing 5.1.

Following the above pre-processing, loops within the annotated parallel loop are marked as being \textit{regular} or \textit{irregular}. The computation within regular loops are model using sets and maps where constraints used in their definitions are affine. A loop is marked irregular if

- its loop body contains an \langle IAssignment \rangle statement.
for ( k = 0; k < nf; k++ ){
    for(i=start_y[fine_boxes[k]]/2;i<end_y[fine_boxes[k]]/2;i++)
        for(j=start_x[fine_boxes[k]]/2;j<end_x[fine_boxes[k]]/2;j++){
            phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]] =
                (phi[fine_boxes[k]][2*i-start_y[fine_boxes[k]]]
                 [2*j-start_x[fine_boxes[k]]]
                 + phi[fine_boxes[k]][2*i-start_y[fine_boxes[k]]+1]
                 [2*j-start_x[fine_boxes[k]]]
                 + phi[fine_boxes[k]][2*i-start_y[fine_boxes[k]]]
                 [2*j-start_x[fine_boxes[k]]+1]
                 + phi[fine_boxes[k]][2*i-start_y[fine_boxes[k]]+1]
                 [2*j-start_x[fine_boxes[k]]+1]) / 4.0;
        }
    }

Listing 5.3: After pre-processing Listing 5.1

• its loop body contains an ⟨If⟩ statement.

• its iterator is used in ⟨IArray⟩⟨(IEexpr)⟩.

• any loop nested within it is marked as irregular.

By construction, computation within regular loops can be expressed using affine constraints since all expressions of the form ⟨IArray⟩⟨(IEexpr)⟩ can be treated as parameters for a given value of the iterators of the outer irregular loops. All ⟨IScalar⟩s within loops that are regular can also be treated as parameters.

5.5 Generation of Inspector and Executor Code

Based on the framework described in Sections 5.2 and 5.3, the partitioning of the computation might need run time analysis to compute the local data space for arrays. The steps to generate an inspector code when required, and an executor code is discussed in this section. For presentation purposes, each annotated loop is assumed to be perfectly nested. Imperfectly nested loops can be handled by considering each
Algorithm 8: GenerateInspector($A$)

Input: $A$: AST of the annotated parallel loop
Output: $A_I$: AST of the inspector

begin

[\[N,R\] = FindIrregularDimensions($A$) ;
$A_I = \emptyset$ ;
if $N \neq \emptyset$ then

$A_I = \text{MakeCopy}(A)$ ;
InsertCheckLocalIteration($A_I$) ;
$P = \text{InsertTemporaryVariables}(A_I,N,R)$ ;
$I = \text{ComputeAffineIterationSpace}(A_I,N,R,P)$ ;
forall the $a \in \text{Arrays}(A)$ do

$D_a = \text{ComputeAccessMap}(A_I,N,R,P,a)$ ;
$F_a = \text{ComputeImage}(D_a,I)$ ;
$F_O^a = \text{ProjectOutInnerDimensions}(F_a)$ ;
if IsMultiDimensional($a$) then

$F_I^a = \text{ParameterizeOuterDimension}(F_a)$ ;
InsertCodeToComputeBounds($A_I,F_I^a,F_O^a$) ;
InsertCodeForExactUnion($A_I,a,F_O^a$) ;
forall the $e \in \text{ArrayIndexExpression}(a,A_I)$ do

$o = e.\text{OuterDimension}$ ;
if $\neg \text{IsOfDesiredForm}(o)$ then

InsertCodeToCreateTrace($o$) ;
else if IsNonAffineIterator($o$) then

InsertCodeToTraceContiguousAccess($o$) ;
RemoveRegularLoopsAndStatements($A_I,R$) ;
GenerateGuardsForIndirectionArrayAccesses($A_I$) ;
return $A_I$ ;
end

statement to be perfectly nested within its surrounding loops with different statements embedded within the same iteration space during code-generation.

5.5.1 Computing the Local Data Space of Arrays: Inspector Code

Algorithm 8 describes the generation of the inspector code. For a given annotated loop AST, function $\text{FindIrregularDimensions}$ marks a loop as being irregular if its value is used in index expressions of indirection array accesses. Listing 5.4 shows
for ( k = 0 ; k < nf ; k++ )
if( get_home(loop_k) == myid ) {
    // t1= start_y[fine_boxes[k]];
    t1 = get_elem(id_start_y,get_elem(id_fine_boxes,k));
    // t2= start_x[fine_boxes[k]];
    t2 = get_elem(id_start_x,get_elem(id_fine_boxes,k));
    // t3= end_y[fine_boxes[k]];
    t3 = get_elem(id_end_y,get_elem(id_fine_boxes,k));
    // t4= end_x[fine_boxes[k]];
    t4 = get_elem(id_end_x,get_elem(id_fine_boxes,k));
    // t5= start_y[ftoc[k]];
    t5 = get_elem(id_start_y,get_elem(id_ftoc,k));
    // t6= start_x[ftoc[k]];
    t6 = get_elem(id_start_x,get_elem(id_ftoc,k));
    // t7= fine_boxes[k];
    t7 = get_elem(id_fine_boxes,k);
    // t8= ftoc[k];
    t8 = get_elem(id_ftoc,k);

    // Record access to outer dimension for phi[cID][cy][cx]
    p_outer = t8;
    update_access(id_phi,p_outer);
    // Record Lexmin/Lexmax for the inner dimensions
    lexmin_dim1 = -t5 + t1;
    lexmin_dim2 = -t6 + t2;
    lexmax_dim1 = -t5 + t3 - 1;
    lexmax_dim2 = -t6 + t2;
    update_lexmin(id_phi,p_outer,lexmin_dim1,lexmin_dim2);
    update_lexmax(id_phi,p_outer,lexmax_dim1,lexmax_dim2);
    ....
}
the inspector code generated for Listing 5.1 using Algorithm 8. Since we target computations that are irregular-outer regular-inner, if a particular loop is irregular, all its surrounding loops are also marked as irregular. The annotated loop is marked as irregular, unless the computation within the loop is affine and its partitioning is decided statically. No inspector is needed in such cases. Presence of one or more irregular loops, the iteration space slices mapped to a process are known only at runtime. While the data space for a single slice can be computed statically using Eq (5.4), the union of these slices is evaluated at runtime by the inspector.

The AST of the inspector code is constructed by first replicating the AST of the annotated loop. At line 6 of Algorithm 8, the loop body of the outermost loop is enclosed within a conditional that executes only those iteration that belong to set $K^q$. All indirection array accesses are invariant with respect to the loops that constitute the regular portions of the computation. Therefore, the value of all such expressions can be stored in temporary variables just before the outermost loop that corresponds to a regular dimension of the iteration space. Line 7 inserts such statements into the inspector AST and replaces indirection array accesses with the corresponding temporary variable. Lines 4-18 of Listing 5.4 shows these assignment statements. The point in the inspector AST immediately after these statements enumerates all elements of $\vec{p} \in C^q$, and can therefore analyze all iteration space slices mapped to a process.

The iteration space slice representing the regular portion of the AST can now be expressed using affine constraints and is computed at line 8. For Listing 5.1, a slice would be,

$$I_P := \{(k, i, j)|k = p1 \land t1/2 \leq i < t3/2 \land t2/2 \leq j < t4/2\}$$
For each array, the integer map representing accesses to it is built at line 10. For a particular reference, such as \( \text{phi}[\text{cID}][\text{cx}][\text{cy}] \) in Listing 5.1, this map would be

\[
D_{\phi i} := \{(k, i, j) \rightarrow (l, a, b) | l = t8 \land a = i - t5 \land b = j - t6 \}
\]

Such a map is built for each access of the array. A union of these maps is computed statically and is applied to the iteration space slice, \( I_p \) at line 11 of Algorithm 8. The result represents a slice of the data space for an array (Eq 5.4). For example,

\[
D_{\phi i}(I_p) := \{(l, a, b) | l = t8 \land t1/2 - t5 \leq a < t3/2 - t5 \land t2/2 - t6 \leq b < t4/2 - t6 \}
\]

(5.8)

The code to compute the union of data space slices is added at line 16. The simplest solution would be to use a bounding box approach [96]. For each slice, the inspector can compute the lexicographic minimum and maximum index of an array accessed by it. The union can be computed by taking the minimum of lexicographic minimums, and maximum of all lexicographic maximums for all the slices executed on the process. The size of the local array would be equal to the difference between this computed maximum and minimum. Elements of the local array can be accessed in a manner consistent with the original computation by just replacing all index expression for an array with the computed lexicographic minimum. Since the elements of arrays accessed on each process need not be contiguous, such an approach would allocate more space than required.

Alternatively, the union of data space slices can be computed at runtime by maintaining, for each array, a set of elements accessed on a process. For all 1D arrays, line 16 inserts code to add elements of the set computed at line 11 to this set.

100
For large multi-dimensional arrays, computing an exact union of all elements accessed is very expensive. This cost can be reduced by recognizing that accesses to such arrays also follow the irregular-outer regular-inner pattern. For example, the access $\phi_{cID}[cy][cx]$ in Listing 5.1 results in the outer dimension being accessed using indirections, but for a given iteration of loop $k$, a rectangular patch of the inner dimensions of the array is accessed (Eq (5.8)). Therefore, for a multi-dimensional array, the union of data space slices on a process is computed as follows,

- The exact union of the set of all the outer most indices of arrays accessed by each slice is computed.
- For each index of the outer dimension, a bounding box approach is used to compute the union for all the inner dimension indices touched for an outer dimension index.

Since an exact union is computed only for the outer dimension indices, the cost of union is drastically reduced.

To use this approach, the inner dimensions of the data space slice computed at line 11 is modified to parameterize the outer dimension (line 14). This can be done applying the following map to the data space slice

$$ PO = \{(o_1, o_2, \ldots, o_e) \rightarrow (o_2, \ldots, o_e) \mid o_1 = p_{outer}\} $$

This expression now computes the set of indices of inner dimensions accessed for every outer dimension index, $p_o$ of the array. The lexicographic minimum and maximum of the resulting expression is computed statically at line 15. The inspector employs the bounding box approach for the inner dimensions by computing the minimum of all lexicographic minimums and maximum of all lexicographic maximums for every outer dimension index of the array accessed on a process. Lines 24-27 shows the
inspector code that computes the lexicographic minimum and maximum for each inner dimension for the access \( \phi[cID][cy][cx] \) in Listing 5.1. This is then used to compute the bounding box at lines 28-29 of the inspector code in Listing 5.4. The set of these outer dimension indices can be computed by projecting out the inner dimensions of the data space slice computed at line 11. Line 16 inserts code to compute the exact union of such sets for all data space slices on a process. Lines 21-22 correspond to the code generated to record the outer-dimension index accessed using \( \phi[cID][cy][cx] \). The code for other array access expressions is not shown in Listing 5.4 for brevity.

If an index expression used to access 1D arrays, or the outer most dimension of multi-dimensional arrays, are of the form,

\[
\langle\text{MapExp}\rangle ::= \langle\text{Array}\rangle'['\langle\text{MapExpr}\rangle']' | \langle\text{Iterator}\rangle
\]  

(5.9)

Section 5.6 describes an approach that allows the generated code to recreate the accesses in a manner consistent with the original computation, without having to generate a trace of such index expressions. If not of this form, the algorithm reverts to the trace-based approach (Sections 3.3.2 and 3.3.3) to recreate this expression in the executor code. Note that the inspector will never have to trace values of inner dimension index expressions of multi-dimensional arrays, significantly reducing the size of traces generated. Since the regular parts of the computation have been modeled statically they can be removed from the inspector (line 23). Having computed the local data space, the inspector allocates an array of size equal to this space and populates it with values from the original array in lexicographic order.

Since the inspector code is only incharge of enumerating the values of \( \vec{p} \in C^q \), the affine loops and statement within the loop are removed from the inspector AST.
Similar to Section 4.1.1, the indirection arrays used in the original computations are also assumed to be initially block-partitioned across nodes. Therefore, a process might not have all the indirection array elements necessary to compute loop-bounds and array index expressions used within the inspector code. Line 24 uses Algorithm 4 to generate conditionals that guard loop bounds and statements containing array access expressions. The loops and statements are executed only when all ⟨IExpr⟩s in them can be evaluated on a process.

Once the local data space of all arrays have been computed, the sets \( O_{q,q'} \) and \( G_{q,q'} \) on each process needs to be computed for all of them as described in Section 5.3.2. The set \( O_q \) is computed using the iteration-to-data affinity (Section 4.1.1), when the partitioned loop has a non-affine iterator, or is defined statically to be block distributed when the partitioned loop is affine.

### 5.5.2 Executing the Iteration Space Slices: The Executor Code

Algorithm 9 shows the steps involved in generating the executor code. Listing 5.5 shows the generated executor code when Listing 5.1 is used as input. The generated code executes the iteration space slices mapped to each process in the same order as the original computation. Similar to Algorithm 8, the executor AST is initialized to be a copy of the original AST. Since the iteration space slices are described using affine inequalities, polyhedral code-generation tools like CLooG [95] can be used to generate the code for a particular slice (line 6). This allows the manipulation of the computation represented by each slice using transformations previously developed within the polyhedral model like [97,98]. While currently no such transformations have been implemented in our framework, this approach allows the easy integration...
of these approaches. The CLooG generated code is used to replace the loop-nest corresponding to the regular parts of the input AST. Lines 13-19 shows the regular parts of the executor code. It is evident that its structure is similar to the regular parts of the original code in Listing 5.1.

For fully affine computations, since the loop nest generated at line 6 replaces the entire executor code, the loop bounds of the outermost loop are modified statically to execute only a portion of the iteration space on each process. Since the data space of all arrays on a process can also be computed statically, this is used to allocate the local arrays. All array access expressions are modified to refer to these local arrays.

In Listing 5.5, \( \phi_1 \) is the local array corresponding to array \( \phi \) in Listing 5.1. A similar naming convention is used for all arrays used. The index expressions are replaced by the original expressions subtracted with the lexicographically smallest
index of the array accessed on a process. Lines 9-15 correspond to the code-generation scheme for purely affine computations with static outer loop partitioning.

For an input AST with one or more irregular iterators, the function \textit{ReplaceOuterLoopBounds}, modifies the bounds of the outer-most loop to iterate from 0 to \(|K^q|-1\), where \(|K^q|\) is computed by the inspector. Since the original value of the iterator is needed within the loop body, the inspector creates a temporary array, \texttt{local\_k}, that stores the elements of \(K^q\) in increasing order. All references to the outer-most loop iterator, \(k\), are replaced with \texttt{local\_k[k]}. Section 5.6.3 describes cases where this temporary array can be eliminated. The loop bound expressions of inner loops are left as is, so that their iterators assume the same values as the original computation.

For array access expressions of the form \langle\textit{MapExpr}\rangle, Section 5.6.3 describes the corresponding index expression to be used in the executor code. For array access expressions that are not of this form, the index expression is modified the same way as Section 4.2. For inner dimensions of expressions used to access multidimensional arrays, the expression in the executor is obtained by subtracting the lexicographic minimum of the inner dimensions accessed for the given index of the outermost dimension on that process. In Listing 5.5, the array \texttt{lexmin\_phi\_dim1} and \texttt{lexmin\_phi\_dim2} store the lexicographic minimum index for each outer dimension index accessed on a process for the array \texttt{phi}.

Once the executor code has been generated, communication calls necessary to exchange the values of ghost locations are inserted before and after the executor AST.

Appendix A describes a sample AMR computation along with the inspector and executor code generated using Algorithms 8 and 9.
Algorithm 9: GenerateExecutor($\mathcal{A}$)

**Input:** $\mathcal{A}$: AST of the annotated parallel loop

**Output:** $\mathcal{A}_E$: AST of the executor

1. begin
2.  $[N,A] = \text{FindIrregularDimensions}(\mathcal{A})$;
3.  $\mathcal{A}_E = \text{MakeCopy}(\mathcal{A})$;
4.  $P = \text{InsertTemporaryVariables}(\mathcal{A}_E,N,A)$;
5.  $I = \text{ComputeAffineIterationSpace}(\mathcal{A}_E,N,A,P)$;
6.  $\mathcal{A}_{\text{new}} = \text{GenerateLoopNests}(I)$;
7.  $\text{ReplaceWithLocalArrayReferences}(\mathcal{A}_E)$;
8.  if $N = \phi$ then
9.     $\text{PartitionIterationSpace}(\mathcal{A}_{\text{new}})$; $\text{ReplaceLoops}(\mathcal{A}_E,A,\mathcal{A}_{\text{new}})$;
10.    $\text{forall the } a \in \text{Arrays}(\mathcal{A})$ do
11.       $D_a = \text{ComputeAccessMap}(\mathcal{A}_E,N,A,P,a)$;
12.       $F_a = \text{ComputeImage}(D_a,I)$; $L_{\text{min}} = \text{ComputeLexMins}(F_a)$;
13.       $\text{forall the } i \in \text{ArrayIndexExpression}(a,\mathcal{A}_E) \text{ do}$
14.          $i_{\text{new}} = \text{NewSubtractExpression}(i,L_{\text{min}})$;
15.          $\text{ReplaceExpression}(i,i_{\text{new}})$;
16.    else
17.     $\text{ReplaceLoops}(\mathcal{A}_E,A,\mathcal{A}_{\text{new}})$;
18.     $\text{ReplaceOuterLoopBounds}(\mathcal{A}_E)$;
19.    $\text{forall the } a \in \text{Arrays}(\mathcal{A})$ do
20.       $\text{forall the } e \in \text{ArrayIndexExpression}(a,\mathcal{A}_E) \text{ do}$
21.          $o = e.\text{OuterDimension}$;
22.          if $\neg \text{IsOfDesiredForm}(o)$ then
23.             $\text{InsertCodeToReadTrace}(o)$;
24.          else if $\text{IsNonAffineIterator}(o)$ then
25.             $\text{ModifyContiguousAccess}(o)$;
26.          if $\text{IsMultiDimensional}(a)$ then
27.             $i = e.\text{InnerExpressions}$;
28.             $l_{\text{min}} = \text{GetLexminValueExpression}(a,o,i)$;
29.             $i_{\text{new}} = \text{NewSubtractExpression}(i,l_{\text{min}})$;
30.             $\text{ReplaceExpression}(i,i_{\text{new}})$;
31.    return $\mathcal{A}_E$;

return $\mathcal{A}_E$;
5.6 Generating Array Index Expressions for the Executor Code

This section describes the modifications to array index expressions that allow the executor code to access elements of the local arrays in a manner consistent with the original computation. In Section 4.2 the index expressions were modified to read from arrays that stored the trace of elements accessed. In this section, we identify array access expressions that are common in many applications, and outline a strategy that eliminates the need to store and use such trace arrays.

It is helpful to recognize that indirection arrays represent an encoding of a map from one set of entities in the computation to another. For example, in Listing 5.1 the array ftoe represents a map from a boxes at finer levels of refinement to the boxes at coarser levels of refinement colocated with it. The outer loop k iterates over the fine boxes in the computation and uses the array ftoe to obtain the index for the corresponding coarse box. Similarly, in unstructured grid computations, indirection arrays are used to represent a map from a face to the two cells adjacent to it, and so on. Further, the same map could be used to access multiple arrays, all of which contain data associated with a particular entity. In Listing 5.1, the array fine_boxes is used to access elements of start_y, end_y, start_x and end_x, all of which contain information associated with a particular box of the computation. Multiple levels of indirection represent a composition of such maps.

Partitioning the computation, results in data related to a subset of the different entities of the original computation being accessed on each process. For example, partitioning the computation in Listing 5.1, results in each process accessing information associated with a subset of all boxes used in the computation. As a result,
the arrays \texttt{start\_y}, \texttt{end\_y}, \texttt{start\_x} and \texttt{end\_x} have similar local data spaces and their elements are also laid out in the local memory of the process in a similar way. Therefore, a local indirection array, say \texttt{fine\_boxes\_l}, could be created to access the local versions of all these arrays in the executor code. In effect, the local indirection array encodes the same mapping as the original code, but in terms of a local numbering of entities on a process.

Use of indirection arrays as maps result in array index expressions being of form described by the non-terminal \texttt{⟨MapExpr⟩} (Eq (5.9)). \texttt{⟨Array⟩}s used in \texttt{⟨MapExpr⟩} in the original code can be replaced with local indirection arrays in the executor code, eliminating the need to store the trace of accesses resulting from such expressions. The creation of such local indirection arrays can be viewed as a two-step process,

1. Create a local buffer that contains all the elements of the original indirection array accessed on a process. This is done by the inspector code generated in Section 5.5.1.

2. Change the values stored in the local indirection arrays to point to the corresponding local positions of elements of the target array accessed. This process is described in Section 5.6.1.

5.6.1 Access Graphs

The information about arrays that are accessed by an indirection array, and the different levels of indirection used in the original code is represented using a Directed Acyclic Graph (DAG), referred to as access graph. Nodes in this graph either represent indirection arrays or loop iterators. For every array access expressions of the form \texttt{⟨Array⟩[(Iterator)]}, an edge is added from the node representing the iterator, and
the node representing the array. By construction, iterators used here correspond to irregular dimensions of the iteration space. For array access expressions of the form $\langle\text{Array}\rangle_1[\langle\text{Array}\rangle_2[\langle\text{MapExpr}\rangle]]$, an edge is added from the node representing $\langle\text{Array}\rangle_2$ to the node representing $\langle\text{Array}\rangle_1$.

Expressions of the form $\langle\text{Array}\rangle[\langle\text{MapExpr}\rangle]$s might also appear in loop bounds and inner dimensions of multi-dimensional arrays. Arrays used in such expressions are also indirection arrays, but the values of these arrays are not modified before they are used in the executor. To capture such a use of an indirection array an edge is added from the node representing the $\langle\text{Array}\rangle$ and a special node, $\text{global}$, added to the access graph. The node $\text{global}$ has no successors. Figure 5.2 represent the access graph built for the computation in Listing 5.1. In cases where the access graph built in this fashion has cycles, edges are removed from the access graph till no cycle exists. The expressions corresponding to the edges removed are recreated in the executor using the trace approach used. However, we havent come across computations where such a scenario exists.

All indirection arrays of the computations have one or more immediate successors in the access graph. Nodes that have multiple immediate successors represent indirection arrays used to access multiple arrays. Values of the local indirection array can be modified to access the corresponding local arrays if the local data space computed by Eq (5.5) for all of them are the same. One way to do this is it to be equal to the union of all their individual local data space. As explained previously, in many scientific computing applications, the arrays accessed using the same indirection array store information regarding a particular entity of the computation. Therefore, the local data space of these arrays would be similar. The modified data space would at most
represent only a slight over-estimation. The decision about which arrays need to have identical local data spaces can be made by grouping nodes that represent arrays in the access graph into disjoint sets, such that immediate successors of an array node belong to the same set.

A trivial solution would be to add all array nodes to the same set. This would lead to a significant over-estimation of the local data space. To avoid this trivial solution, the sets can be created such that no node and its immediate predecessor belong to the same group. Finally, a node which is an immediate predecessor to the *global* node represents an indirection arrays whose values are not modified in the executor. Such indirection arrays cannot be used to access other arrays in the executor. Therefore, the grouping is done such that no array node is added to the same set as the global node.
In summary, the grouping of array nodes has to satisfy the following three conditions,

1. All successors of an array node have to belong to the same set.

2. No node should be in the same set as the global node.

3. An array node and its immediate successors must not belong to the same set.

In general, it might not be possible to create such groups for all possible access graphs. Modifications can be made to the access graph so that these properties can be enforced. For nodes in the DAG that have multiple immediate successors, one of which is the global node, a new node is added to the DAG with the same predecessors. The global node is added as an immediate successor of the new node, and is removed as a successor of the original node. This operation represents duplication of the corresponding indirection array in the executor, the values in one copy are modified to access the elements of local arrays, and the other has its values unchanged. A similar strategy is used when all immediate successors of a node cannot be added to the same set. If successors of a node belong to two separate groups, values of one copy of the indirection array can be modified to access elements of one group and the other copy modified to access elements of the other group.

5.6.2 Grouping Nodes of the Access Graph

Algorithm 10 presents a scheme to group array nodes into sets while satisfying the above requirements, adding new nodes whenever necessary. Each node is assumed to be associated with two fields, 1) \textit{group}, which denotes the set which the node belongs to, and 2) \textit{Type} which can either be \textit{ArrayNode}, \textit{LoopNode} or \textit{GlobalNode}. For all the
Algorithm 10: GroupNodes($G$)

**Input**: $G = (V,E)$: An access graph with group number of all nodes set to $-1$

**Output**: $G = (V,E)$: Graph with nodes added into groups

```
begin
ngroups = 0; done = False;
while ¬ done do
    done = True;
    foreach $v$ in ReverseTopologicalOrder($G$) do
        group = Unknown; Unchanged_set = φ;
        foreach $s$ in $v$.successors do
            if group = Unknown then
                group = $s$.group;
            else if group ≠ $s$.group then
                processed_set = $v$;
                if ChangeGroupNum($s$, group, processed_set) then
                    Unchanged_set = Unchanged_set ∪ $s$;
        if $v$.Type = ArrayNode then
            if Unchanged_set ≠ φ then
                $r$ = Copy($v$); $V$ = $V$ ∪ $r$;
                $r$.predecessors = $v$.predecessors;
                $r$.successors = Unchanged_set;
                $v$.successors = $v$.successors - Unchanged_set;
                done = False; Break;
            $v$.group = ngroups; ngroups = ngroups + 1;
        return $G$;
```

nodes, the field group is initially set to Unknown to indicate that the node is not part of any set.

The access graph is traversed in reverse topological order. For every node encountered, all its immediate successors have already been assigned to groups. The algorithm tries to add all its immediate successors to the same group as its first immediate successor by calling the function ChangeGroupNum for each of them. This function, described in Algorithm 11, takes as input, the node $v$ whose group number has to be changed and the group number to change to.
Algorithm 11: ChangeGroupNum(v,group,processed_set)

<table>
<thead>
<tr>
<th>Input</th>
<th>v : Node in the graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>Group number to be added to</td>
</tr>
<tr>
<td>Output</td>
<td>True if the node was added to the group, False otherwise</td>
</tr>
</tbody>
</table>

1. begin
2. if v.Type = GlobalNode then
3. return False;
4. if v.group = Unknown then
5. v.group = group;
6. return True;
7. foreach p in (v.predecessors - processed_set) do
8. if p.Type = ArrayNode ∧ p.group ≠ Unknown then
9. if p.group ≠ group then
10. processed_set.insert(p);
11. foreach s in p.successors do
12. if ¬ChangeGroupNum(s,group,processed_set) then
13. return False;
14. else
15. return False;
16. v.group = group;
17. return True;

In function ChangeGroupNum, if the input node v, is the global node the function returns False to signify it couldnt be added to the group. If the node has already been assigned to a group, then at least one of its immediate predecessors has already been visited by Algorithm 10. For all such previously visited predecessors, the function returns Unknown if any of them are also assigned to the same group. Otherwise, the node v can be assigned to the group if all the immediate successor of the previously visited immediate predecessors are assigned to the same group. This is checked through a recursive call to ChangeGroupNum.

If the call to ChangeGroupNum at line 12 of Algorithm 10 returns false, then it implies that current successor, s, cannot be added to the same group as the previous
successors. It is removed as an immediate successor of the node \( v \) being analyzed, and added as an immediate successor to a copy of the node. This new graph can now be traversed for grouping the nodes. When no conflicts are found all array nodes have been grouped appropriately and the algorithm terminates. In the worst-case this algorithm will create a graph where all nodes have only one successor. Such a scenario would not be common in the kind of applications targeted here.

For the graph in Figure 5.2, arrays \( \texttt{ftoc} \) and \( \texttt{fine\_boxes} \) belong to the same group. The arrays \( \texttt{start\_x}, \texttt{start\_y}, \texttt{end\_y}, \texttt{end\_x}, \) and \( \texttt{phi} \) belong to the another group. Since the arrays of the second group are enforced to have same local data space on each process, a local indirection array \( \texttt{fine\_boxes\_l} \) can be used to access all of these arrays in the executor. This local indirection array mimics the behavior of the array \( \texttt{fine\_boxes} \) in Listing 5.1.

### 5.6.3 Modifying Index Expressions in the Executor Code

Once the grouping of nodes is complete, the index expressions in the executor code generated in Section 5.5.2 are modified as follows. For \( \langle \text{MapExpr} \rangle \)'s used in array index expressions which are of the form \( \langle \text{Array} \rangle[\langle \text{MapExpr} \rangle] \), the \( \langle \text{Array} \rangle \) is replaced to refer to one of the copies of the local indirection array, as decided by the grouping algorithm. The values in the indirection arrays are modified based on the order in which elements of the target arrays are arranged in local memory. \( \langle \text{MapExpr} \rangle \)'s of the same form occurring in loop-bound expressions and inner-index expressions of multi-dimensional arrays are just replaced with local array references without changing its values. For example, the arrays \( \texttt{start\_x}, \texttt{start\_y}, \texttt{end\_y} \) and
end_ x used in loop bounds and inner dimension of array access expressions can just be replaced with their local versions, without modifying their values.

For array index expressions of the form \langle \text{Iterator} \rangle, since iterator values of the generated executor code has the same value as the original computation, these index expressions have to be manipulated to refer to local elements of the array. Since the expression has a unit-stride with respect to the loop, local arrays and expressions similar to the ones described in Section 3.3.3 can be used to access the correct elements in the executor. The same expression can be used to access all arrays whose nodes are immediate successors of the iterator node and belong to the same group.

Finally, for array access expressions of the form \langle \text{Array} \rangle[\langle \text{Iterator} \rangle] in the original code, where the iterator is from the partitioned loop, the executor code generation replaced such expressions with a new expression of the form \langle \text{Array} \rangle_1[\langle \text{Array} \rangle_2[\langle \text{Iterator} \rangle]]. \langle \text{Array} \rangle_2 represents a temporary array that contained the original values of the iterator mapped to a process, i.e., belonging to set \( K^q \) of Eq (5.3). In the access graph, the node corresponding to the iterator, say \( p \), would have only one immediate successor, say \( l \), before the grouping algorithm (Algorithm 10) is employed. If, any successor of this node, say \( m \), has no other predecessors then it can be concluded that every iteration mapped to a process accesses only one element of the array corresponding to the node \( m \). Since the iterations are executed in order of their original index, and array elements are laid out in order of their original index too, these array elements can be accessed using the iterator itself. The node \( m \) is removed as a successor from node \( l \) and added as a successor to node \( p \). The array expression in the executor is changed back to be of the form \langle \text{Array} \rangle[\langle \text{Iterator} \rangle], where \langle \text{Array} \rangle refers to the corresponding local array. The temporary array added could be eliminated if the node \( l \)
has no successors after this modification. The above holds true for nodes representing ftoc and fine_boxes in Figure 5.2. Therefore, in the executor code, ftoc[k] and fine_boxes[k] are replaced by ftoc_1[k] and fine_boxes_1[k] respectively.

5.7 Case Studies

The performance of the generated distributed memory code was evaluated for computations that are 1) Irregular-outer Regular-Inner, 2) Completely Regular, i.e. code that have no irregular dimensions, and 3) Completely Irregular, i.e. codes that have no regular dimension. The code-generation scheme described in Section 5.5 was also implemented as a source-to-source transform within ROSE to target C-codes. The same experimental setup as Section 4.4 was used for evaluation.

All the applications/benchmarks used here also contained a sequence of parallel loops enclosed within one or more sequential time/convergence loops. The control flow and data access pattern for the parallel loops remain unchanged for every invocation within these outer sequential loops. Therefore, the inspector code could be hoisted out of the latter to amortize its overhead. For each of the benchmark/applications used, the performance of the generated executor, and the total execution time of the generated code (inspector + executor) are shown.

5.7.1 Poisson Equation Solver on Multigrids

This benchmark solves the Poisson Equation over a rectangular domain. An example of a parallel loop in this benchmark was shown in Listing 5.1. Similar to the abstractions used by AMR packages like Chombo, the physical space is divided into several boxes. The space enclosed within these boxes is further discretized using rectangular grids that are either coarse or fine. Each loop iterates over a subset of
boxes and updates values associated with grid-points within them. Loops that iterate over the boxes represent the irregular dimensions of the iteration space since the data associated with each box is accessed using indirection arrays. Loops that iterate over grid-points within a box have represent the regular dimensions, since value associated with them are updated through stencil operations. The stencil which might access values of grid-points within the same box, within neighboring boxes that use a similar discretization, or within boxes that use different discretization but are colocated in physical space. The benchmark used contained 4 parallel loops enclosed within an outer sequential loop. The inspector code for each of the loops could be hoisted outside of this outer loop. The partitioning of iterations and data was done using the approach similar to Section 3.4.

The problem size used for evaluation had 1024 coarse boxes and 2048 fine boxes. Each coarse box used a grid of size $128 \times 128$, while each fine box used a grid of size $126 \times 252$. Table 5.1 shows the running time for the executor and the total running time of the transformed code. The generated executor scales well upto 256 processes, with the inspector cost adding only a slight overhead. Since the inspector is parallel, the cost of the inspector reduces upto 128 processes. There is an abrupt increase in the inspector cost at 256 processes due to synchronization overhead while prefetching values of indirection arrays. The vectorization report generated by ICC

<table>
<thead>
<tr>
<th>Nprocs</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executor(s)</td>
<td>895.1</td>
<td>461.1</td>
<td>243.5</td>
<td>125.1</td>
<td>63.8</td>
<td>32.4</td>
<td>16.5</td>
<td>8.03</td>
</tr>
<tr>
<td>Total(s)</td>
<td>898.0</td>
<td>463.9</td>
<td>243.5</td>
<td>125.2</td>
<td>63.9</td>
<td>32.5</td>
<td>16.6</td>
<td>11.3</td>
</tr>
</tbody>
</table>

Table 5.1: Irregular-Outer Regular-Inner benchmark : Poisson equation solver
Figure 5.3: Comparison with Chapter 4

shows that loops that were vectorized previously are vectorized in the generated code too, indicating that the transformation scheme did not introduce artifacts that affect subsequent compiler optimizations. The parts of the generated code that perform the stencil operations have the same structure as the original code. Being bandwidth-bound the on-node performance of such operations can be further enhanced by using transformations like [99] that maximize reuse of data-elements read from memory.

The code generated by the approach in Chapter 4 fails to execute since the size of the traces generated exhausts the memory on a process. To get around this, the problem size used was reduced to $1/4^{th}$ of the above. While now this code is able to execute, the size of traces generated are still quite large. For 2 and 4 processes, the executor code from the trace approach is 4 times slower than the executor code from the current approach. Due to high inspector overheads, the total execution time is 8 times slower. As the number of processes increase the traces generated per process becomes smaller and fit in some level of cache, resulting in lesser stress on the bandwidth to main memory. This improves the executor time of the trace approach.
but the inspector costs still remain high. Consequently, the execution time is on average 3 times slower than the approach presented in this paper. These results show that the code generated by the current approach handles irregular-outer regular-inner more efficiently and is more scalable.

### 5.7.2 Affine computations

Next, the effectiveness of the approach presented in this chapter in parallelization of completely affine codes was evaluated. For bandwidth-bound computations like FDTD and 2D Jacobi stencil [100], time-tiling is an effective approach to increase the arithmetic intensity by increasing data-reuse across iterations of the outer time loop. Tiling for concurrent start [101], generates code such that the loop that iterates over tiles for a particular time-tile are parallel, while the loop that iterates over time-tiles is sequential. This approach to time-tiling eliminates the load imbalance created by traditional schemes that usually results in wavefront parallelism across tiles. This scheme has been implemented within Pluto [3], and the generated code was used as input to the transformation scheme described in this chapter. Being fully affine, the intra-timetile loop was block partitioned across processes. The local data space can be computed statically. The array elements owned by a process (\(O^p\)) was computed using a block distribution of the arrays used in the sequential code.

Table 5.2 shows the performance of the generated executor for time-tiled FDTD and Jacobi2D codes. All arrays used were of size 8192 × 8192. The original untiled code executed 1000 timesteps. A tile size of 32 was used for each loop. The performance was compared against the distributed memory code-generated by Pluto [102, 103]. Since the communication pattern that the generated code from both, the present
Table 5.2: Affine Computations: Time-tiled FDTD and Jacobi2D

approach and Pluto are similar, this behavior is as expected. While the set $O_{A}^{q,q'}$ could be evaluated statically, the current implementation still followed the approach described in Section 5.3.2. First the set $C_{A}^{q,q'}$ was evaluated on process $q'$ and transmitted to process $q$ to compute $O_{A}^{q,q'}$. This one-time cost adds a very slight overheads to the total execution time of the generated code.

Table 5.2 shows that the abstractions developed in this chapter are as effective as state of the art affine techniques developed for distributed memory parallelization of stencil-based applications.

5.7.3 Irregular Computations

Finally, we evaluate the performance of the distributed memory code generated for computations that contain no affine parts.

Boltzmann Transport Equation

The Boltzmann Transport Equation (BTE) for phonons [104] can be used to model heat conduction in semiconductor materials by using a 3D unstructured grid consisting of tetrahedral cells. Finite-Volume methods can be used to convert the
BTE into a linear equation at each cell-center. The computation proceeds by iterating over bands of phonon frequencies and discretized directions of the physical domain. A system of linear equations for the entire physical domain is solved for each band-direction pair. This is followed by an integration phase that combines data for a particular band. The loops over band-direction pairs and over bands are parallel and can be targeted for distributed memory execution. For transient problems, these steps are performed repeatedly within a time-loop. Since this application is written in Fortran, we implemented the transformation manually in keeping with the proposed techniques. The computation is an example where there are no loops with affine iterators.

For evaluation, we used a problem size containing 2491 cells, with 40 frequency bands and 40 discretized directions. Using the trace based approach of Chapter 4 for this application resulted in the inspector code exhausting the processes local memory while generating the trace of index expression values. The inspector code in the proposed approach was able to avoid this by creating local indirection arrays on each process that mimicked the behavior of indirection arrays of the original computation. Table 5.3 shows the execution times of the parallelized code for 10 time steps. The parallelized code shows linear scaling upto 256 processes. The approach described here is capable of parallelizing applications with deep loop-nests, where building the trace of expressions at the inner-most levels becomes infeasible.

**183.equake and P3-RTE**

Finally, the performance of the code-generated by the two approaches for two examples from Section 4.4, 183.equake and P3-RTE, were compared. ParMetis was
Table 5.3: Irregular Application : 3D BTE Solver

used for the partitioning since that gave the best performance for both these benchmarks (Section 4.4). Figure 5.4 shows that the speed-up obtained with the executor code for both approaches are comparable. Since these benchmarks do not have deep loop-nests the memory requirements to store the traces is not very high. Further, in the executor these traces are streamed through. Hardware prefetchers enable efficient accesses to these traces. The approach presented in Section 5.6 results in indirect accesses which cannot be optimized by the prefetcher. Figures 5.4c and 5.4d show that the inspector overheads are reduced using the approach presented in this chapter. This reduction improved the total execution times for equake (Figure 5.4a). For P3 (Figure 5.4b) the benefit of lesser inspector costs was limited since the inspector overheads are amortized effectively for this benchmark. Further, the reduction in inspector overheads was compensated by better executor running times for the approach described in Chapter 4.
Figure 5.4: Comparison between the two proposed approaches
CHAPTER 6

Forma : A DSL for Image Processing Pipelines

This chapter describes the syntax and semantics of Forma, the DSL developed for targeting image processing applications. The syntax of Forma allows easy specification of common operations like convolutions/stencil operations, interpolation, extrapolation, cropping and padding of images. These operations can be easily chained together to specify the producer-consumer relationships between the different stages. While the current focus of Forma has been in developing a convenient language syntax that results in compact specification of the image processing pipelines, prototype backends have been developed to target both multicore CPUs and NVIDIA GPUs. These back ends generate code that can utilize the underlying hardware effectively to obtain good performance. Further, the generated code can be easily incorporated into projects to handle computations that are outside the scope of the DSL.

6.1 Syntax of Forma Programs

The syntax of Forma programs was developed to make it easy to specify complex image processing pipelines. Convolutions, stencil operations, interpolations and extrapolation are some basic operations that are commonly used in image processing and can be conveniently specified. Variables in a Forma program are an orthogonal
domain of points with each point associated with some data. For example, a 2D gray
scale image can be viewed as a rectangular domain of points, with each point storing
a 32-bit floating point value. The size and type of domains associated with variables
is computed automatically by the compiler. Therefore, the programmer is freed from
the tedious and error-prone task of memory management. This feature is especially
beneficial in application which perform interpolations or extrapolations, where the
output image of different stages have different sizes.

Forma uses a single-assignment paradigm, i.e., once defined variables in Forma
cannot be modified. This allows the compiler to easily capture the dependences
between different stages of the pipeline and optimize across stages. Figures 6.1 and 6.2
describes the complete grammar of Forma program, with (Program) being the starting
non-terminal. Listing 6.1 shows a simple program in Forma.
Figure 6.1: Grammar of Forma program
(FnArg) ::= (VectorExpr) | (VectorExpr) `;` (BdyCondn)
(BdyCondn) ::= `constant` | `clamped` | `wrap` | `mirror`
(ComposeStmtList) ::= (DomainStmtList) | (DomainFnStmtList)
(DomainStmtList) ::= (DomainStmt) | (DomainStmtList) (DomainStmt)
(DomainStmt) ::= (Domain) `=` (VectorExpr) `;`
(DomainFnStmtList) ::= (DomainFnStmt) | (DomainFnStmtList) (DomainFnStmt)
(DomainFnStmt) ::= (DomainFn) `=` (VectorExpr) `;`
(VectorFn) ::= `vector` (ID) `('` (FnParamList) `)` `;` `;' (StencilStmt)
(StencilStmt) ::= `stencil` (ID) `('` (FnParamList) `)` `;` `;' (StencilBody)
(StencilBody) ::= (StencilStmtList) `return` (Expr) `;`
(StencilStmtList) ::= (StencilStmt) (StencilStmtList)
(StencilStmt) ::= (ID) `=` (Expr) `;`
(Expr) ::= ID | (Float) | (Double) | (Integer) | (Expr) (Op) (Expr) | `.' (Expr) `)'
| (ID) (DomainFn) | (ID) `('` (ExprList) `)` `;` `;' (Expr) `)'
| `struct` `('` (ExprList) `)` `;` | (ID) `.' | (ID)
(ExprList) ::= (Expr) | (ExprList) `,'` (Expr)
(Op) ::= `+' | `-` | `*` | `/` | `<` | `>` | `<=` | `>`=` | `==` | `!=`
(Domain) ::= `[` (RangeList) `]`
(RangeList) ::= (Range) | (RangeList) `,'` (Range)
(Range) ::= (ParametricExpr) `..` (ParametricExpr) | (ParametricExpr) `..`
(DomainFn) ::= `@[` (ScaleCoeffList) `]`
(ScaleCoeffList) ::= (ScaleCoeff) | (ScaleCoeffList) `,'` (ScaleCoeff)
(ScaleCoeff) ::= `('` (Integer) `,'` (Integer) `)` | (Integer)

Figure 6.2: Grammar of Forma programs - contd
The data associated with each point of the domain can be one of the following types: 8-bit integers (int8), 16-bit integer (int16), 32-bit integer (int), single-precision floating point number (float) or double-precision floating point number (double). It is also possible to associate each point with a user defined data-structures (TypeDefn). Lines 1-3 of Listing 6.1 show an example of the syntax used to define a data structure that holds the RGB information of a pixel. Fields of a user-defined structure cannot be structures themselves.

A Forma program takes an input a specification of the images input to the pipeline. The non-terminal ⟨DomainDefn⟩ defines the syntax to specify the size of these input images. An example of this is shown in Line 5 which defines a 2D domain of points, with #2 specifying the dimensionality of the domain input. The size of a domain along a dimension can be affine functions of variables defined as parameters (⟨ParametricExpr⟩). For example, the domain input is of size M x N, where M and N are defined as parameters at line 4 (⟨ParameterDefn⟩).

Each statement in a Forma program defines a stage in the pipeline. Every Forma program ends with a return statement that denotes the final image obtained after all the stages. The computation in Listing 6.1 is a simple pipeline where the output image is a copy of the input image.
Many filters in image processing can be expressed as stencil operations, where the value at a point in the output image is a linear combination of values at a corresponding point in the input image and its immediate neighbors. For example, in a 5-pt 2D Jacobi stencil, the value at a point \((i, j)\) of the output is a weighted average of values at points \((i, j), (i, j - 1), (i, j + 1), (i + 1, j)\) and \((i, j + 1)\) of the input domain. In Forma, stencils are specified as functions that are applied to domains. These function definitions \((\text{StencilFn})\) are prefixed with the keyword \texttt{stencil}. The parameters of the function \((\text{FnParamList})\) define the type and dimensionality of the domains used in the stencil body. Lines 1-4 of Listing 6.2 shows the specification of the 5-pt 2D jacobi stencil in Forma, which takes as input a 2D domain of integers. The stencil is applied to the domain \(x\) at line 7.

The body of a stencil function \((\text{StencilBody})\) evaluates the value at a single point in the output domain. It consists of a sequence of assignments statements \((\text{Stmt})\), where the right-hand side may contain mathematical operations, math functions or ternary expressions \((\text{Expr})\). The type of the variable on the left-hand side is automatically determined by the DSL compiler. Loops and conditionals
are not allowed within the stencil definition. The function definition ends with a `return` statement, whose expressions evaluates to the value at a point in the output domain.

Within the stencil specification, Forma allows the use of an offset operator, `[@,...]` (⟨ScaleCoeffList⟩), to access neighboring points within input domains. The list of integers specified within the square braces represent the offset of the point whose value is to be used. Therefore `X[@[-1,0]]` refers to the value at point \((i - 1,j)\) while computing the value at point \((i,j)\) of the output. Dropping the offset operator is a short-hand to refer to the value at point \((i,j)\) of the input domain.

To be able to express stencil operations like the one described in Listing 1.3, the offset operator can also contain a scaling factor for every dimension. Instead of a list of integers, one can specify a list of tuples of the form \((a,b)\) to refer to a point \(a + i \times b\) while computing the value at \(i^{th}\) point along that dimension of the output. Listing 6.3 shows the Forma specification of the filter in Listing 1.3. Using an integer instead of a tuple for a dimension in the offset operator, sets the scaling factor as 1 for that dimension. Using a scaling factor reduces the size of the output domain along that dimension by the scaling factor. Therefore, the size of `temp` is half the size of `image` along the x-dimension. The size of the output is half of the input `image` along both dimensions.

**Boundary Conditions**

Using the stencil specification as is to compute values along the edges of the output domain would be invalid. For example, while evaluating the value at point \((0,0)\) of the domain \(y\) while applying the stencil `jacobi2d` in Listing 6.2 would be invalid since the points \((-1,0)\) and \((0,-1)\) of the input domain, \(x\) are undefined. In many manual
implementations of image processing filters, and in scientific computing applications that use stencils, a significant amount of time and effort is spent on proper handling of boundary conditions.

To make it easier to handle boundaries, Forma provides syntax to specify certain common boundary conditions. A common method for handling computation along the edges of the output domain is to extend the input domain along each dimension to make it legal to apply the stencil along the edges. For the `jacobi2d` stencil in Listing 6.2 extending the input domain by 1 in both direction for all dimension would make it legal to apply the stencil along the edges of the output domain. Different boundary conditions set different values for the extended parts of the domain. Some of the common approaches used are

- **Constant**: The extended points are set to a constant value, say 0.

- **Clamped**: The extended points are set to the value at the corresponding edges of the input domain.
Wrap: The extended points assume values that give an impression of the input domain being wrapped around along that dimension.

Mirror: The extended points assume values such that the corresponding edge behaves like a mirror.

Forma provides keywords that can specified along with arguments to a stencil function application to automatically implement each of the above boundary conditions. The Forma compiler uses the stencil definition to decide the amount of extension required automatically. Different arguments to the stencil function can have different boundary conditions. If no boundary condition is specified, the value at only those points of the output domain are computed where it is valid to apply the stencil function. For example, in Listing 6.2 if the stencil were applied without a specified boundary condition, the points along the first and last, row and column of the output domain would not be evaluated. Section 6.1.3 describes how custom boundary conditions can be specified within Forma.

Extrapolation

To support the extrapolation operation (also referred to as upsample) shown in Listing 1.4, Forma allows the use of the offset operator outside of a stencil definition.
Listing 6.4 shows the syntax to specify the first step of the extrapolation operation. At line 7, use of the offset operator on the left-hand side of the assignment statement specifies that the value at a point \((i, j)\) of the domain on the right-hand side (which is the result of a stencil application) is assigned to the point \((2i, j)\) of the domain represented by \textit{temp}.

### 6.1.2 Cropping and Padding Operations

Forma allows the programmer to also extract an orthogonal sub-domain of points by using the \textit{domain operator}. The non-terminal \langle Domain \rangle in Figure 6.2 defines the syntax of the operator, and can be used to apply different filters on different parts of an image. The region to be extracted is specified within ‘[ ]’ as a list of lower and upper bounds for each dimension of the domain. An example of this operator is shown in line 2 of Listing 6.5, where a rectangular region formed between the points \((3, 5)\) and \((25, 29)\) of domain \(x\) is extracted. The size of the resulting domain, \(y\) is \(23 \times 25\).

The domain operator can also be used on the left-hand side of an assignment statement, with the upper-bound for each dimension unspecified. This represents an offset to be used while evaluating the left-hand side domain. For example, in line 3 of Listing 6.5, values from the domain represented by \(x\) are copied to the domain \(z\) after being padded by 3 along the first dimension and 5 along the second. The resulting

```c
vector#2 int x[M,N];
y = x[3..25,5..29];
z[3..,5..] = x;
w[1..,3..,5..] = x;
```

Listing 6.5: Crop and Padding Operations
domain is of size \((M+3)x(N+5)\). In conjunction with the \textit{compose operation} described in Section 6.1.3, this operator allows different parts of a domain to be computed using different expressions. When used to specify offsets in this manner, the size of the domain operator might be higher than the size of the domain on the right-hand side. The resulting image has a dimensionality equal to size of the domain operator on the left-hand side. For example, in line 4 of Listing 6.5, the domain \(w\) is 3-dimensional with its size being \(2x(M+3)x(N+5)\).

### 6.1.3 Compose Operation

While the operations described in Section 6.1.1 allows the specification of individual steps of the filters shown in Listing 1.3 and 1.4, the single assignment paradigm of Forma would not allow the programmer to assign to different points of a domain in successive statements. The \textit{compose operation} can be used to specify the operations to be used to compute different portions of the output domain simultaneously. The non-terminal \(\langle \text{ComposeStmtList} \rangle\) specifies the grammar of such expressions. Line 7 of Listing 6.6 shows an example of this operation used to specify the computation for the odd and even points along the x-dimension simultaneously.
The regions updated by individual statements of the compose operator maybe specified either through the offset operator \((\text{DomainFn})\) or the domain operator \((\text{Domain})\). The syntax of compose statements shown in Figure 6.2 does not allow using both kinds of operators simultaneously within a compose statement. The compiler assumes that each statement specifies the evaluation of disjoint portions of the output domain, allowing the computation of each statement independently and possibly in parallel. In cases where the regions are not disjoint, the behavior is unspecified and might result in race conditions.

Since the size of the left-hand side domain is computed automatically, each statement in the compose operations might result in a different size of the image. The size of the final image is the orthogonal hull of sizes computed from individual statements.

**Custom Boundary Conditions**

The compose operation can be used to specify custom boundary conditions. For example consider a stencil \texttt{jacobi1D} shown in Listing 6.7, that is to be applied at all points of an image except the first and last column, where the stencils \texttt{bdy0} and \texttt{bdyN} are to be applied instead. Lines 12 extracts the first and last column of the domain \texttt{a}, applies the respective stencils on these and places them along the first and last column of the output domain \texttt{b}. The interior points of \texttt{b} are computed by applying the stencil \texttt{jacobi1D} with default boundary conditions. The result of the stencil application is cropped at the edges and placed in the output with a padding of 1 along the x-dimension. The compose operator is used to specify these operations simultaneously in keeping with the single-assignment paradigm.
stencil jacobi1D(vector#2 int X) {
    return (X@[-1,0] + 2*X + X@[1,0])/4;
}

stencil bdy0(vector#2 int X) {
    return (3*X + X@[1,0])/4;
}

stencil bdyN(vector#2 int X) {
    return (3*X + X@[-1,0])/4;
}

parameter N;
vector#2 int a[N,N];
b = ([0..,0..] = bdy0(a[0..1,0..N-1]);
    [1..,0..] = jacobi1d(a[1..N-2,0..N]);
    [N-1..,0..] = bdyN(a[N-2..N-1,0..N-1]);
);
6.1.4 Vector Functions

It is a common occurrence that a sequence of filters are applied repeatedly over the course of the computation. As an example consider the pipeline shown in Figure 6.3, which is used to combine multiple RGB images at different exposures to get an HDR image [5]. The filters \texttt{exposure}, \texttt{make\_grey}, \texttt{laplacian}, \texttt{saturation} and \texttt{multiply} are applied in the same sequence to every input image. This sequence computes a weight to be used for each input image while combining them (\texttt{combine}) to get the final HDR image. Vector functions allow programmers to encapsulate a sequence of operations into a single function making the program more concise, less error-prone and easier to maintain. In Forma, vector functions are defined by adding the keyword \texttt{vector} to the function definition as shown in Listing 6.8.

6.1.5 Loops and Qualified Variables

In some image processing pipelines, loops provide a natural way of expressing the computation. A prime example of this is the Laplacian Pyramid [4] computation. Here, two image pyramids are constructed from the input image, namely the \textit{Gaussian}...
The Laplacian Pyramid. The input image forms the base of the Gaussian pyramid. Successive level are constructed by sampling from the images at a lower level. The Laplacian pyramid is constructed by extrapolating the images at a level of the Gaussian pyramid and subtracting the image at the immediately lower level. In Forma, the loop construct can be used to specify the construction of these pyramids.

Listing 6.9 shows the specification of the Laplacian pyramid computation in Forma, with the use of loops at line 17. Loops within Forma can only have numeric bounds and increments of +1 or −1. The variables gaussian and laplacian are tagged by an additional parameter specified within ‘< >’. Such variables are referred to as qualified variables and represent a list of domains, each of which are of the same type but can have different sizes. To maintain the single-assignment property, all statements within a loop body can only assign to variables that are qualified. The compiler also ensures that a particular domain within a qualified variable has been defined before it is used. For example, at line 18 in Listing 6.9 the compiler checks that \((i - 1)\)th domain of qualified variable, gaussian has been defined before its use. As a result, the 0\(^{th}\) domain of gaussian has to be defined outside the loop-body at line 16 for the program to be correct.

6.2 Optimization Across Stages of Image Processing Pipelines

The previous sections outlined the syntax and semantics of various operations that allow the programmer to express image processing pipelines in a compact manner. The Forma compiler uses this high-level specification to build an internal representation of the dependence between stages of a pipeline, allowing optimization across
vector sample(vector#2 float Z){
    return sample_y(sample_x(Z));
}

vector upsample_x(vector#2 float X){
    return (@[(0,2),(0,1)] = upsample_x_even(X);
    @[(1,2),(0,1)] = upsample_x_odd(X); );
}

vector upsample_y(vector#2 float Y){
    return (@[(0,1),(0,2)] = upsample_y_even(Y);
    @[(0,1),(1,2)] = upsample_y_odd(Y); );
}

vector upsample(vector#2 float Z){
    return upsample_y(upsample_x(Z));
}

parameter M,N;
vector#2 float image[M,N];
gaussian<0> = image;
for i = 1..3
    gaussian<i> = sample(gaussian<i-1>);
    laplacian<i> = subtract(gaussian<i>, upsample(gaussian<i-1>));
endfor

Listing 6.9: Laplacian Pyramid Computation

them. In this section we discuss this internal representation and one such inter-stage optimization.

Directed Acyclic Graphs (DAGs) are a convenient abstraction to represent this dependences between stages. Every stencil function application, vector function application and compose expression is represented by a node in this DAG. An edge is added from one node to another when the result of a function application represented by the former is used as an argument to function represented by the latter. The single-assignment paradigm used in Forma allows the compiler to add edges from nodes representing the right-hand side of assignment statements to all uses of the defined variable. The edges also stores information of the sub-domain of the source node used in the target node, or if an extrapolation function or padding operator is
used to compute the target domain. A separate DAG is built for each vector function defined within the Forma program. Before building a DAG, each loop is completely unrolled. Figure 6.3 shows the DAG built by the compiler to capture the dependence between the different stages involved in merging images to form an HDR image.

6.2.1 Fusing Stages of a Pipeline

This DAG representation built for a Forma program can be used to implement optimizations across multiple stages of an image processing pipeline. In this section, one such optimization currently implemented in the Forma compiler is described. This optimization fuses stages of the pipeline when it doesn’t result in redundant computation.

Since the compiler is tasked with the memory management for a Forma program, during the code-generation process, memory is allocated to store the domain represented by every node in the computation DAG. Fusing nodes in the DAG amounts
to fusing stages of the pipeline. This not only reduces the number of intermediate buffers (and hence the footprint) needed for the computation, but also reduces the total memory bandwidth requirements. Within Forma, fusion is performed when both the source and target nodes are stencil functions and only when it doesn’t result in redundant computation. For example, consider the DAG shown in Figure 6.4b. The filters `laplacian` and `make_grey` are shown in Listing 6.10. If these two filters were combined, before using the value at the neighboring points to compute the laplacian, the grey value at all the neighboring points would have to be computed redundantly. To avoid this, stages are only combined when the function represented by the target node doesn’t refer to neighbors of the domain represented by the source node. Further, nodes are merged only if the source node has only one outgoing edge, i.e., the result at the source node is not used in multiple stages of the pipeline. Under these two constraints, the nodes `saturation`, `exposure` and `laplacian` can be merged with the node `multiple`, but `make_grey` cannot be merged with `laplacian`.

To be able to fuse as many stages as possible, Forma inlines all vector functions before fusing stages. Therefore, the nodes representing the vector function `compute_weights` in Figure 6.4a is replaced with the the DAG in Figure 6.4b, with the input and output nodes in the latter (X and Y respectively) replaced with the predecessor and successors of the substituted node in the former.

```
stencil make_grey(vector#2 rgb X){
    return (0.02126f * X.r + 0.7152f * X.g + 0.0722f * X.b) / 255;
}
stencil laplacian(vector#2 float X){
    return 4 * X - ( X@[-1,0] + X@[0,-1] + X@[1,0] + X@[0,1] );
}
```

Listing 6.10: Laplacian Filter
Fusing two nodes might unearth further fusion opportunities. Consider a computation DAG shown in Figure 6.5a where none of the stages refer to neighboring points of input domains. Under the constraints described earlier, only the stages $h_1$ and $h_2$ can be fused with their successor, resulting in a computation DAG shown in Figure 6.5b. This results in the nodes $f$ and $g$ now having only one successor and can be fused with it. To fuse as many stages as possible, the Forma compiler does multiple passes over the computation DAG, terminating only when no candidates for fusion are found. Figure 6.6 shows the computation DAG obtained as a result of the fusion optimization starting with the DAG in Figure 6.4. The number of stages in the computation reduces from 11 to 3, reducing the number of intermediate buffers needed from 10 to 2.
6.3 Code Generation

Forma compiler generates C code with OpenMP pragmas to target multicore CPUs and CUDA code to target NVIDIA GPUs. Here we discuss architecture specific optimizations that are implemented within Forma to enhance the performance of the generated code.

6.3.1 Targeting CPUs

For multicore CPUs, Forma generates loop-nests to apply the filters represented by the stages of the image processing pipeline (after stages have been fused according to the discussion in Section 6.2.1). The size of all intermediate buffers are automatically computed by the compiler from the program specification, freeing the programmer from explicit memory management. Using the computation DAG (after performing fusion as described in Section 6.2), the compiler back-end allocates memory to store the data associated with a domain represented by a node in the DAG.
Since every point in the domain can be computed independently, there is no loop carried dependence at any level of the generated loop nest. This allows the use of OpenMP pragmas to parallelize the outermost loop, at the same time enabling vectorization of the innermost loop for all the generated loop nests. Therefore the generated C code is able to exploit both coarse-gained parallelism across cores, and fine-grained parallelism through use of vector functional units on each core. Both of these are critical to obtain good performance on modern CPU architectures.

To handle boundary conditions, as described in Section 6.1.1, additional loop-nests are generated to initialize the extended portions of domains that are inputs to stencil functions. This approach avoids branching within the inner-most loop on the CPU code which would hamper vectorization. While this would require additional memory and would incur an overhead of copying data from the original domain to the new extended domain, the benefit of vectorization of the branch-free code makes up for this overhead.

### 6.3.2 Targeting GPUs

For NVIDIA GPUs, Forma generates separate kernels for each node in the computation DAG that represents the image processing pipeline (after stages have been fused). Each thread on the device is responsible for computation of a single point of the output domain. Apart from handling memory allocation and deallocation on the device, Forma also generates code to move data to and from the device. The kernel code is generated such that the access to global memory are unit-stride, as far as possible, to enable coalesced accesses across threads. Similar to the CPU code-generation, to handle boundary conditions, additional kernels are generated to
initialize the extended portions of domains that are inputs to stencils. This method eliminates the need to check for boundaries within a kernel, reducing the amount of branch divergence amongst threads. The benefit of this approach is more on GPUs than on CPUs due to the high penalty incurred for divergence between threads of a warp.

Additionally, a compile time option can be specified to utilize the texture units on the device. Buffers that are read-only within a kernel, and are pitch-linear in global memory can be mapped to texture fetch units of the device. These units exploit spatial locality of accesses from multiple threads to reduce the cost of global memory reads on a device. In addition, the hardware provides different addressing modes that can be used to efficiently handle boundary conditions supported by Forma, i.e., constant, clamped, wrap and mirror. Further details about the texture units and the various addressing modes can be found in [105]. The current implementation of the Forma GPU back-end uses only texture reference API which is less efficient than the texture object API. Future work will aim to use the latter.

6.3.3 Interface to the generated code

The generated Forma code (for GPU or CPU) is encapsulated within a C-function with the compiler also generating a header file with the signature of this function. The arguments to this functions are

- Variables that are defined as input within the Forma program,
- Parameters used within the program, and
- Pointers to memory location which should contain the result of the pipeline (specified along with the return expression in the program).
The header file also has a description of the size of the output buffer needed, which is computed by the Forma compiler but has to be allocated by the calling function. On returning from the Forma generated function, the output buffer contains the result of the pipeline.

6.4 Case Studies and Performance Evaluation

In this section we demonstrate the use of Forma both in terms of ease of specifying simple to complex image processing pipelines and in terms of performance achieved on CPUs and GPUs. For each of the pipelines discussed here the performance of the generated code is compared against Halide [6], another DSL that targets image processing applications.

The experimental evaluation of the CPU back-end was performed on quad-core 2-way hyper-threaded Intel Core i7-3820. PGCC 13.10 was used to compile the C code generated by Forma. The reported execution times are in milliseconds and are averaged over 5 runs. The scalability of the generated code across multiple cores is also evaluated.

The GPUs performance was evaluated on a NVIDIA GTX 680 GPU which is based on the Fermi architecture and an NVIDIA K20Xm which is based on the Kepler architecture. For each of the applications below we show the performance with and without utilizing the texture fetch units. The reported execution times are the total kernel execution times averaged across 5 runs.

6.4.1 Simple Stencils

We start with two simple stencil applications, emboss and blur. The former is just simple 5-pt stencil. The latter contains two stages, the first stage applies a
3-pt stencil along the x-direction, with the second stage applying a 3-pt stencil in y-direction. The implementation of blur within the Halide package was chosen for comparison. The emboss kernel was implemented in Halide by the authors. An image of size $512 \times 512$ using floats was used as input for emboss. For blur, we used an image of size $1992 \times 1998$.

Table 6.1a shows the performance of both the Forma generated C/OpenMP code, and the Halide code. The Forma generated code achieves good scaling across 8 threads of the CPU and performs better than the Halide generated code on this benchmark. The same scaling is not observed with blur due to memory bandwidth limitations on the CPU for the given input size. The best performance obtained for the Forma generated code is comparable to the best performance achieved by Halide. Table 6.1b compares the GPU performance of Halide and Forma. Once again the Forma generated GPU code performs better than the Halide generated code. The use of textures results in further reduction of the execution time.

### 6.4.2 Camera Pipe

Camera pipeline is a benchmark from [6], and is used in many cameras to convert raw data from the sensors into an image. An implementation from Halide [34] was used as a reference to develop the Forma implementation. The original Halide implementation used 32 functions and 22 different stencils. The Forma implementation used 13 stencil functions (it was easier to express the computation by combining some of the stencils) and 1 vector function (to define the demosaic stack used in the computation). To get good performance with Halide it is necessary for the application
Table 6.1: Performance of Simple Stencils

<table>
<thead>
<tr>
<th>Target</th>
<th>Hardware</th>
<th>GTX 680</th>
<th>Tesla K20Xm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Global Memory</td>
<td>Texture</td>
</tr>
<tr>
<td>emboss</td>
<td>Forma</td>
<td>0.030</td>
<td>0.022</td>
</tr>
<tr>
<td></td>
<td>Halide</td>
<td>0.036</td>
<td>-</td>
</tr>
<tr>
<td>blur</td>
<td>Forma</td>
<td>0.656</td>
<td>0.535</td>
</tr>
<tr>
<td></td>
<td>Halide</td>
<td>0.576</td>
<td>-</td>
</tr>
</tbody>
</table>

developer to specify a schedule for every target architecture. While the implementation of this benchmark that is shipped with Halide contains a schedule optimized for x86 architecture, the schedule for GPUs had to be developed in-house. This schedule inlined as many stages of the pipeline as possible, similar to the schedule that would be used by the Forma generated code.

Table 6.2 compares the performance of Forma generated code and Halide code for both CPU and GPU. Both codes scale well across CPU cores, with the absolute performance obtained from Forma code being better than that of the Halide code. On GPUs the Forma generated code shows a 30% improvement over the Halide generated code.
### Table 6.2: Performance of Camera Pipe (ms)

<table>
<thead>
<tr>
<th>Target Hardware</th>
<th>GTX 680</th>
<th>Tesla K20Xm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Global Memory</td>
<td>Texture</td>
</tr>
<tr>
<td>Forma</td>
<td>0.453</td>
<td>0.482</td>
</tr>
<tr>
<td>Halide</td>
<td>0.650</td>
<td>–</td>
</tr>
</tbody>
</table>

#### 6.4.3 Laplacian Pyramid

Laplacian Pyramids were briefly introduced in Section 6.1.5. Implementation of pyramids in a low-level language, like C++/CUDA, is challenging since images at different levels of the pyramid have different sizes, and are dependant on the filter used for sampling and extrapolation. The application developer would have to compute these sizes manually and allocate memory for these intermediate levels. Changing the filters used would require recomputing these sizes.

In Forma, the specification of the pyramid computation used 26 lines of code, making it easy to debug and maintain. Since the compiler automatically deduces the sizes for each level of the pyramid from the program specification, the filters used could be easily modified by the programmer, allowing for easier evaluation of the best filter to use. Handling boundary conditions adds an additional challenge while using low-level languages, while Forma provides support for standard boundary conditions.

The Halide implementation of the computation used for comparison was adapted from examples shipped with Halide, which were optimized for x86 architectures and
for NVIDIA GPUs. Table 6.3a shows that the Forma generated CPU code performs better than the Halide implementation, while achieving good scaling across CPU cores for this pipeline as well. Table 6.3b shows that a similar trend is observed on GPUs as well.

Table 6.3 also shows the execution times for the computation while using mirror boundary conditions. On CPUs, the additional loop nests generated to handle boundaries add some overhead to the total execution times. Similarly, on GPUs, when texture units are not used, there is an overhead due to the additional kernels generated to handle boundary conditions. On the other hand, enabling the use of texture fetch units on the GPU handles boundary conditions with negligible performance impact.

Table 6.3 also shows the execution times for the pyramid computation on RGB images. Here, first the individual channels are separated out. Gaussian and Laplacian pyramids are computed for each channel and are recombined to get the channels of the output image. The performance while using mirror boundary condition is also shown. Here too, the use of textures on GPU handles the boundary conditions for free.

6.4.4 Exposure Fusion

Section 6.2.1 introduced a basic method for computing HDR images. In some cases using this scheme results in seam effects. Exposure Fusion [5] is another method used to compute HDR images which starts by building 3 Laplacian pyramids and 3 Gaussian pyramids for each input image, one for each color. The Laplacian pyramid for
(a) CPU Performance

<table>
<thead>
<tr>
<th>NThreads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pyramid</td>
<td>Forma</td>
<td>3.042</td>
<td>1.334</td>
<td>1.113</td>
</tr>
<tr>
<td></td>
<td>Halide</td>
<td>5.317</td>
<td>2.787</td>
<td>1.636</td>
</tr>
<tr>
<td>Pyramid-Mirror</td>
<td>4.422</td>
<td>3.011</td>
<td>2.463</td>
<td>2.061</td>
</tr>
<tr>
<td>Pyramid-RGB-Mirror</td>
<td>10.542</td>
<td>5.928</td>
<td>4.594</td>
<td>3.569</td>
</tr>
</tbody>
</table>

(b) GPU Performance

<table>
<thead>
<tr>
<th>Target Hardware</th>
<th>GTX 680</th>
<th>Tesla K20Xm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Global Memory</td>
<td>Texture</td>
</tr>
<tr>
<td>Pyramid</td>
<td>Forma</td>
<td>0.208</td>
</tr>
<tr>
<td></td>
<td>Halide</td>
<td>0.231</td>
</tr>
<tr>
<td>Pyramid-Mirror</td>
<td></td>
<td>0.539</td>
</tr>
<tr>
<td>Pyramid-RGB</td>
<td></td>
<td>0.533</td>
</tr>
<tr>
<td>Pyramid-RGB Mirror</td>
<td></td>
<td>1.376</td>
</tr>
</tbody>
</table>

Table 6.3: Performance of Laplacian Pyramid on CPU/GPU (ms)

each color is combined using the method described in Section 6.2.1 to get the Laplacian pyramid for that color of the output image. Finally the Gaussian pyramid of the output image is computed to give the final combined HDR image. Our implementation of this method takes the 4 images shown in Figure 6.7 as inputs (taken from [5]), resulting in the construction of 30 pyramid objects. Improper handling of boundary conditions results in edge effects as shown in Figure 6.8a. Figure 6.8b shows the final HDR images constructed when using mirror boundary conditions for all stencil applications. This complex image processing pipeline can be expressed in Forma with just 184 lines of code, using 25 stencil functions, 9 vector functions and 2 loops.

Table 6.4 shows the performance of the generated code from Forma for both the simple version described in Section 6.2.1 and the version which uses pyramids...
to combine the images. The Halide implementation was developed in-house. The schedule used was determined using a combination of the schedule used for stencils and pyramids within Halide. The results again shows that the performance of the code generated by Forma for both CPU and GPU is better than the Halide generated code.
Figure 6.8: Effect of improper boundary conditions

<table>
<thead>
<tr>
<th>NThreads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without Pyramids</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forma</td>
<td>26.375</td>
<td>13.319</td>
<td>7.206</td>
<td>7.773</td>
</tr>
<tr>
<td>Halide</td>
<td>31.681</td>
<td>16.056</td>
<td>8.358</td>
<td>7.574</td>
</tr>
<tr>
<td>With Pyramids</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forma</td>
<td>85.063</td>
<td>46.953</td>
<td>29.079</td>
<td>29.130</td>
</tr>
<tr>
<td>Halide</td>
<td>122.498</td>
<td>64.157</td>
<td>34.938</td>
<td>31.291</td>
</tr>
</tbody>
</table>

(a) CPU Performance

<table>
<thead>
<tr>
<th>Target Hardware</th>
<th>GTX 680</th>
<th>Tesla K20Xm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Global Mem</td>
<td>Texture</td>
</tr>
<tr>
<td>Without Pyramids</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forma</td>
<td>0.305</td>
<td>0.317</td>
</tr>
<tr>
<td>Halide</td>
<td>0.369</td>
<td>–</td>
</tr>
<tr>
<td>With Pyramids</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forma</td>
<td>3.435</td>
<td>3.255</td>
</tr>
<tr>
<td>Halide</td>
<td>4.577</td>
<td>–</td>
</tr>
</tbody>
</table>

(b) GPU Performance

Table 6.4: Performance of Exposure Fusion on CPU/GPU (ms)
CHAPTER 7

Future Work

7.1 Distributed Memory Parallelization of Irregular Computations

Chapters 3, 4 and 5 developed a framework for distributed memory parallelization of class of irregular loop computations. While the developed abstractions in Chapter 5 allows the compiler to split the computation into regular and irregular parts, the benefit of recognizing the regular parts have to be fully explored. The current framework used the regular parts mainly to reduce the cost of inspection by computing data space slices at compile time. The structure of regular parts of the code was maintained in the executor. Interfacing with polyhedral compilers like PolyOpt [12] would be necessary to enable optimizing transformations for these parts of the computation for improved on-node performance. Many polyhedral compilers can also generate GPU code from the affine specifications. This could be used to target heterogeneous clusters (nodes with an attached GPU) by executing the regular parts of the code on the GPU.

The model of reasoning about iterations space slices can also be integrated into the Sparse Polyhedral Framework [66] (SPF). This framework allows representing
iterations space and data accesses in computation while representing indirection arrays as Uninterpreted Function Symbol (UFS). Applying loop transformations and computing data spaces in presence of UFS requires reasoning about properties of the UFS. For example, consider the iteration space of loop $i$ and access to array $x$ in Listing 1.1 expressed in SPF,

$$I := \{(i, j) \mid 0 \leq i < N \land ia(i) \leq j < ia(i+1)\}$$ \hspace{1cm} (7.1)

$$D_x := \{(i, j) \rightarrow k \mid k = col(j)\}$$ \hspace{1cm} (7.2)

Here $ia$ and $col$ are UFS to represent the indirection arrays $ia$ and $col$, respectively. Using the approach in [66], the footprint of $x$ can be computed by applying $D_x$ to $I$

$$F_x := \{k \mid 0 \leq i < N \land ia(i) \leq col^{-1}(k) < ia(i+1)\}$$ \hspace{1cm} (7.3)

This expression uses $col^{-1}$ which in general is not a valid since $col$ is not usually bijective. The need to make assumptions about UFS makes it hard to apply meaningful transformations. On the other hand, from description of the computation within SPF, generating slices of the iteration space that allow use of affine transformations can be readily incorporated within it.

In adaptive computations, like particle in cell codes or adaptive mesh refinement codes, the values in indirection arrays change during the course of the computation, but do so infrequently. This would require a re-execution of the inspector to partition the computation for load-balancing and communication minimization for this modified control flow and data access pattern. While the approach to inspector/executor code-generation described in Chapter 5 reduced the inspector costs, further work is
necessary to be able to use softwares like Zoltan [106] that allow light-weight reparation of computations. This would increase the scope of applications that could be targeted for distributed memory parallelization.

7.2 Improvements to Forma

While the current state of the Forma DSL (Chapter 6) allows the specification of many stencil based pipelines, support for other commonly used operations like histogram and reductions is lacking. Addressing this issue is important to increase the scope of applications targeted by Forma. Vector functions allow the specification of such image-level operations. A simple solution would be to interface with external libraries that implement them, but this would restrict the compiler’s ability to optimize across stages involving such operations. A more promising approach would be to incorporate knowledge of these operations within the compiler, and future work will focus on this aspect.

Another drawback within Forma is the way the code-generation is done for loop constructs. Currently, loops are unrolled completely before code-generation. As a result, the generated code contains thousands of lines for some of the benchmarks, like Exposure Fusion with pyramids used for evaluation. The main reason behind using this approach was enable static evaluation of the size of buffers needed to store intermediate results of operations within the loop body. However, this could be evaluated at run time too. Developing such a run time system would result in more compact code. While presently this code-explosion doesn’t seem to have too much impact on the performance of the implemented pipelines, this could become an issue for larger pipelines.
Finally, the code-generation strategy within Forma is relatively straightforward, but still achieves performance comparable to the strategy in Halide that was used for comparison. For other applications, Halide might be able to find a schedule that is significantly better than the one implemented in Forma. We are interested in exploring the different scheduling strategies within Forma. One way to do this is to interface with polyhedral compilers like PoCC [107] or Polly [108]. Some initial exploration suggests that it might not be very helpful to subject the entire pipeline to polyhedral optimizations. The space of possible optimizations might be too large for these tools to be effective. Instead, the Forma compiler can be enhanced to subject different portions of the pipeline to optimization by polyhedral tools. Such an approach would allow the Forma compiler to leverage state-of-the-art optimizations that target stencil computations.
CHAPTER 8

Conclusion

This thesis developed a framework that allows distributed memory parallelization of a class of loops that contain data dependant control flow and array access patterns. Compiler algorithms were developed to detect program regions that are amenable to transformations and to generate code that uses the inspector/executor approach to parallelize these program regions. The generated parallel inspector partitioned the computation in a load balanced fashion while minimizing the communication using the iteration-to-data affinity of the partitioned loops. The approach was enhanced by using the abstraction of iteration space slices that allow the compiler to reason about affine parts of the computation statically, reducing the inspector costs while maintaining program properties of the affine parts. The effectiveness of the approach was demonstrated by parallelization of several benchmarks and real-world applications.

This thesis also described a Domain Specific Language, Forma, that provide syntax that makes it easy to specify complex image processing pipelines, focusing mainly on stencil/convolution operations. The tedious parts of programming like memory-management and handling boundary conditions was left to the compiler. At the same time, this high-level specification allowed the compiler to fuse different stages of the pipeline to reduce memory-bandwidth requirements. Backends that target multicore
CPUs and Nvidia GPUs were also developed. Several complex stencil based pipelines were ported to Forma, and the performance obtained on both CPUs and GPUs were on-par/better than state-of-the-art DSLs that target similar applications.
APPENDIX A

Code Generation for an Example AMR Code

Here we describe a sample AMR computation along with the inspector/executor code generated using methods described in Chapter 5.

A.1 Original Code

Figure A.1 represents the abstraction used in AMR applications. The solid line represents a box which uses a coarse grid, while the dashed lines represent boxes that use a finer grid and are colocated in physical space with boxes using a coarse grid. In the example code shown in Listing A.1, the fine grid resolution is twice the coarse grid resolution along both directions, i.e., each coarse grid-point is associated with 4 fine grid-points. The resolution of the coarse grid is assumed to be same as the resolution of integer points of the underlying physical domain. The total number of boxes (fine and coarse) used in the computation is \( n_{\text{boxes}} \), with each box having an index between 0 and \( n_{\text{boxes}} - 1 \), both included. The arrays \texttt{fine_boxes} is of size equal to the number of fine boxes used in the computation and stores their indices. Similarly, array \texttt{coarse_boxes} is of size equal to the number of coarse boxes used in the computation and stores their indices.
The array `start_x` and `start_y` is of size equal to the number of boxes used in the computations, both coarse and fine. They store the integer co-ordinates of the lexicographically smallest point of each box, i.e., the co-ordinates of point E for the box shown in Figure A.1. The array `end_x`, `end_y` are of the same size as `start_x`, `start_y` and store the co-ordinates of the lexicographically largest point of a box, i.e., point D. The values of $\phi$ and $\phi_{old}$ at grid points within a box are stored in arrays `phi` and `phi_old` respectively. These are 3D arrays, such that `phi[k][..][..]` holds the values of $\phi$ for the $k$-th box.

In the first annotated parallel loop at line 3, the values of `phi` at the coarse grid-points are updated using stencil operations with values of `phi_old` at the coarse grid-points used as input. The dotted line represent regions in surrounding boxes that are accessed during this operation. The array `phi_c_t` is used as a scratch pad to store values of `phi_old` from current and neighboring boxes used for the stencil operation. For each box, the information about patches of neighboring boxes accessed is maintained by storing the integer co-ordinates of the lexicographic minimum and maximum values of the patch in the arrays, `patch_start_y`, `patch_start_x`, `patch_end_y`, `patch_end_x`. 

Figure A.1: Coarse and Fine Boxes used in AMR
patch_end_y and patch_end_x. For example, region within the north-neighboring box accessed is captured by storing the co-ordinates of point A in patch_start_x and patch_start_y, and the co-ordinates of point C are stored in patch_end_x and patch_end_y. The array neighb holds the index of the neighboring coarse boxes that contains this patch. Loops i and j at lines 12 and 13, iterate through the points within the patches. The position in phi_old[neighb[p]][..][..] of the required value from the neighboring box is obtained by subtracting the lexicographic minimum point of that box, i.e., point A, from the value of the loop iterators.

The second annotated loop at Line 22, updates values at fine-grid points with the values at coarse-grid points. Each fine box is assumed to be colocated in physical space with only one coarse box. The array ftoc stores the index of this coarse box for every fine box. The loops at line 24 and 25 iterate over the integer points bounded by the fine box. Since there are 4 fine grid-points for every coarse grid-points, statements from line 26 to 29 updates the value for each of them. In general, the value at a fine grid-point is updated using a stencil operation involving values at the coarse grid-point, with a different co-efficients used for each statement from line 26 to 29. Here, a simple copy is used. The points in the array phi_old[fine_boxes[k]][..][..] to be updated is indexed by subtracting the lexicographic minimum point of the fine box from the iterator values and multiplying it by the resolution of the finer grid along that dimension, i.e. 2 for both x and y. The position of grid points within phi[ftoc[k]][..][..] used on the right-hand side of these statements are obtained by subtracting from the iterator values, the lexicographic minimum of the coarse box.
#pragma parallel
#pragma temporary_array phi_c_t
for ( k = 0; k < nc; k++ ) {  // Non-affine Iterator
    /// Affine Loops
    for ( i = 0; i < size_y[coarse_boxes[k]]; i++ )
        for ( j = 0; j < size_x[coarse_boxes[k]]; j++ )
            phi_c_t[i+1][j+1] = phi_old[coarse_boxes[k]][i][j];
    
    /// Non-affine Iterator
    for ( p = patches[coarse_boxes[k]]; p < patches[coarse_boxes[k]+1]; p++)
        /// Affine Loops
        for ( i = patch_start_y[p]; i < patch_end_y[p]; i++ )
            for ( j = patch_start_x[p]; j < patch_end_x[p]; j++ )
                phi_c_t[i1+start_y[coarse_boxes[k]]]
                    [j1+start_x[coarse_boxes[k]]]
                    = phi_old[neighb[p]][i - start_y[neighb[p]]]
                    [j - start_x[neighb[p]]];
        
        /// Affine Loops
        for ( i = 0; i < size_y[coarse_boxes[k]]; i++ )
            for ( j = 0; j < size_x[coarse_boxes[k]]; j++ )
                phi[coarse_boxes[k]][i][j] +=
                    ( phi_c_t[i][j+1] +
                    phi_c_t[i+1][j] + phi_c_t[i+1][j+2] +
                    phi_c_t[i+2][j+1] - 4 * phi_c_t[i+1][j+1]
                    - rho[coarse_boxes[k]][i][j] ) / 8.0;
}

#pragma parallel
for ( k = 0; k < nf; k++) {  // Non-affine Iterator
    /// Affine Loops
    for ( i = start_y[fine_boxes[k]]; i < end_y[fine_boxes[k]]; i++ )
        for ( j = start_x[fine_boxes[k]]; j < end_x[fine_boxes[k]]; j++ )
            phi_old[fine_boxes[k]][(i-start_y[fine_boxes[k]])*2]
                [(j-start_x[fine_boxes[k]])*2]
                = phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]];
    phi_old[fine_boxes[k]][(i-start_y[fine_boxes[k]])*2+1]
        [(j-start_x[fine_boxes[k]])*2]
        = phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]];
    phi_old[fine_boxes[k]][(i-start_y[fine_boxes[k]])*2]
        [(j-start_x[fine_boxes[k]])*2+1]
        = phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]];
    phi_old[fine_boxes[k]][(i-start_y[fine_boxes[k]])*2+1]
        [(j-start_x[fine_boxes[k]])*2+1]
        = phi[ftoc[k]][i-start_y[ftoc[k]]][j-start_x[ftoc[k]]];
}

Listing A.1: Original loop-nests of AMR
A.2 Iteration Space Description

The first step in the code-generation scheme is to replace indirection array accesses within regular parts of the code with temporary variables. For each statement in Listing A.1, the expressions replaced, the iteration space and data access map for arrays phi_old, phi and rho are described below.

Statement at Line 7

Indirection array accesses expressions replaced by temporary variables for the statement:

- \( p1 := \text{coarse\_boxes}[k] \)
- \( p2 := \text{size\_y}[\text{coarse\_boxes}[k]] \)
- \( p3 := \text{size\_x}[\text{coarse\_boxes}[k]] \)

Non Affine Iterators : \( k \)

Iteration Space

\[
I_1 := \{(k,i,j) \mid k = kc \land 0 \leq i < p2 \land 0 \leq j < p3\} \tag{A.1}
\]

Access to array phi_old

\[
D_1 := \{(k,i,j) \rightarrow (l,a,b) \mid l = p1 \land a = i \land b = j\} \tag{A.2}
\]

Statement at Line 14

Indirection array accesses expressions replaced by temporary variables for the statement:

- \( p4 := \text{patches}[\text{coarse\_boxes}[k]] \)
- \( p5 := \text{patches}[\text{coarse\_boxes}[k]+1] \)
- \( p6 := \text{patch\_start\_y}[p] \)
• $p_7 := \text{patch\_end\_y}[p]$
• $p_8 := \text{patch\_start\_x}[p]$
• $p_9 := \text{patch\_end\_x}[p]$
• $p_{10} := \text{start\_y}[\text{coarse\_boxes}[k]]$
• $p_{11} := \text{start\_x}[\text{coarse\_boxes}[k]]$
• $p_{12} := \text{start\_y}[\text{neighb}[p]]$
• $p_{13} := \text{start\_x}[\text{neighb}[p]]$
• $p_{14} := \text{neigh}[p]$

Non Affine Iterators : $k, p$

Iteration Space

$$I_2 := \{(k, p, i, j) \mid k = k_c \land p = p_c \land p_6 \leq i < p_7 \land p_8 \leq j < p_9\} \quad (A.3)$$

Access to array $\phi_{\text{old}}$

$$D_2 := \{(k, p, i, j) \rightarrow (l, a, b) \mid l = p_{14} \land a = i - p_{12} \land b = j - p_{13}\} \quad (A.4)$$

Statement at Line 18

Indirection array accesses expressions replaced by temporary variables for the statement:

• $p_1 := \text{coarse\_boxes}[k]$
• $p_2 := \text{size\_y}[\text{coarse\_boxes}[k]]$
• $p_3 := \text{size\_x}[\text{coarse\_boxes}[k]]$

Non Affine Iterators : $k$

Iteration Space

$$I_3 := \{(k, i, j) \mid k = k_c \land 0 \leq i < p_2 \land 0 \leq j < p_3\} \quad (A.5)$$
Access to array $\phi$

$$D_3 := \{(k, i, j) \rightarrow (l, a, b) \mid l = p1 \land a = i \land b = j\} \quad (A.6)$$

Access to array $\rho$

$$D_4 := \{(k, i, j) \rightarrow (l, a, b) \mid l = p1 \land a = i \land b = j\} \quad (A.7)$$

Statement at Line 26-29

Indirection array accesses expressions replaced by temporary variables for the statement:

- $p_{15} := \text{fine\_boxes}[k]$  
- $p_{16} := \text{ftoc}[k]$  
- $p_{17} := \text{start\_y}[\text{fine\_boxes}[k]]$  
- $p_{18} := \text{start\_x}[\text{fine\_boxes}[k]]$  
- $p_{19} := \text{end\_y}[\text{fine\_boxes}[k]]$  
- $p_{20} := \text{end\_x}[\text{fine\_boxes}[k]]$  
- $p_{21} := \text{start\_y}[\text{ftoc}[k]]$  
- $p_{22} := \text{start\_x}[\text{ftoc}[k]]$

Non Affine Iterators: $k$

Iteration Space

$$I_4 := \{(k, i, j) \mid k = kf \land p17 \leq i < p19 \land p18 \leq j < p20\} \quad (A.8)$$
Access to array \texttt{phi\_old}

\[ D_5 := \{ (k, i, j) \rightarrow (l, a, b) \mid l = \text{p15} \land a = 2 \ast (i - \text{p17}) \land b = 2 \ast (j - \text{p18}) \} \]

\[ \cup \{ (k, i, j) \rightarrow (l, a, b) \mid l = \text{p15} \land a = 2 \ast (i - \text{p17}) + 1 \land b = 2 \ast (j - \text{p18}) \} \]

\[ \cup \{ (k, i, j) \rightarrow (l, a, b) \mid l = \text{p15} \land a = 2 \ast (i - \text{p17}) \land b = 2 \ast (j - \text{p18}) + 1 \} \]

\[ \cup \{ (k, i, j) \rightarrow (l, a, b) \mid l = \text{p15} \land a = 2 \ast (i - \text{p17}) + 1 \land b = 2 \ast (j - \text{p18}) + 1 \} \]

Access to array \texttt{phi}

\[ D_6 := \{ (k, i, j) \rightarrow (l, a, b) \mid l = \text{p16} \land a = i - \text{p21} \land b = j - \text{p22} \} \]

(A.10)

A.3 Inspector Code to compute the footprint

The inspector code generated using Algorithm 8 is shown in Listings A.2 and A.3.
for (k = 0; k < nc ; k++ )
if( get_home(loop_0,k) == myid ){
    p1 = get_elem(arr_coarse_box,k);
    p2 = get_elem(arr_size_y, get_elem(arr_coarse_boxes,k));
    p3 = get_elem(arr_size_x, get_elem(arr_coarse_boxes,k));
    if( p2 >= 1 && p3 >= 1 ){
        // Record access to phi_old[coarse_boxes[k]]
        p_outer = p1; update_access(arr_phi_old,p_outer);
        // Record access lexmin/max for all the inner dimension
        // lexmin(D1(I))
        lexmin_dim1 = 0; lexmin_dim2 = 0;
        update_lexmin(arr_phi_old,p Outer,lexmin_dim1,lexmin_dim2);
        // lexmax(D1(I))
        lexmax_dim1 = p2-1; lexmax_dim2 = p3-1;
        update_lexmax(arr_phi_old,p_outer,lexmax_dim1,lexmax_dim2);
    }
    for( p = get_elem(arr_patches, get_elem(arr_coarse_boxes,k));
p< get_elem(arr_patches, get_elem(arr_coarse_boxes,k)+1); p++ ){
        p4 = get_elem(arr_patches, get_elem(arr_coarse_boxes,k));
        p5 = get_elem(arr_patches, get_elem(arr_coarse_boxes,k)+1);
        p6 = get_elem(arr_patch_start_y,p);
        p7 = get_elem(arr_patch_end_y,p);
        p8 = get_elem(arr_patch_start_x,p);
        p9 = get_elem(arr_patch_end_x,p);
        p10 = get_elem(arr_start_y, get_elem(arr_coarse_boxes,k));
        p11 = get_elem(arr_start_x, get_elem(arr_coarse_boxes,k));
        p12 = get_elem(arr_start_y, get_elem(arr_neighb,p));
        p13 = get_elem(arr_start_x, get_elem(arr_neighb,p));
        p14 = get_elem(arr_neighb,p);
        if( p7 >= p6 + 1 && p9 >= p8 + 1 ){
            // Record access to phi_old[neighb[p]]
            p_outer = p14; update_access(arr_phi_old,p_outer);
            // Record access lexmin/max for all the inner dimension
            // lexmin(D2(I2))
            lexmin_dim1 = p6-p12; lexmin_dim2 = p8-p13;
            update_lexmin(arr_phi_old,p_outer,lexmin_dim1,lexmin_dim2);
            // lexmax(D2(I2))
            lexmax_dim1 = p7-p12-1; lexmax_dim2 = p9-p13-1;
            update_lexmax(arr_phi_old,p_outer,lexmax_dim1,lexmax_dim2);
        }
    }
    if( p2 >= 1 && p3 >= 1 ){
        // Record access to phi[coarse_boxes[k]]
        p_outer = p1; update_access(arr_phi,p_outer);
        // Record access lexmin/max for all the inner dimension
        // lexmin(D3(I3))
        lexmin_dim1 = 0; lexmin_dim2 = 0;
        update_lexmin(arr_phi,p_outer,lexmin_dim1,lexmin_dim2);
        // lexmax(D3(I3))
        lexmax_dim1 = p2-1; lexmax_dim2 = p3-1;
        update_lexmax(arr_phi,p_outer,lexmax_dim1,lexmax_dim2);

        // Record access to rho[coarse_boxes[k]]
        p_outer = p1; update_access(arr_rho,p_outer);
        // Record access lexmin/max for all the inner dimension
        // lexmin(D4(I4))
        lexmin_dim1 = 0; lexmin_dim2 = 0;
        update_lexmin(arr_rho,p_outer,lexmin_dim1,lexmin_dim2);
        // lexmax(D4(I4))
        lexmax_dim1 = p2-1; lexmax_dim2 = p3-1;
        update_lexmax(arr_rho,p_outer,lexmax_dim1,lexmax_dim2);
    }
}
}
for (k = 0; k < nf ; k++)
if( get_home(loop_1,k) == myid ){
    p15 = get_elem(arr_fine_boxes,k);
    p16 = get_elem(arr_ftoc,k);
    p17 = get_elem(arr_start_y,get_elem(arr_fine_boxes,k));
    p18 = get_elem(arr_start_x,get_elem(arr_fine_boxes,k));
    p19 = get_elem(arr_end_y,get_elem(arr_fine_boxes,k));
    p20 = get_elem(arr_end_x,get_elem(arr_fine_boxes,k));
    p21 = get_elem(arr_start_y,get_elem(arr_ftoc,k));
    p22 = get_elem(arr_start_x,get_elem(arr_ftoc,k));

    if( p19 >= p17 + 1 && p20 >= p18 + 1 ){
        // Record access to phi_old
        p_outer = p15;
        // Record access lexmin/lexmax for all the inner dimension
        lexmin_dim1 = 0; lexmin_dim2 = 0;
        update_lexmin(arr_phi_old,lexmin_dim1,lexmin_dim2);
        lexmax_dim1 = 2 * p19 - 2 * p17 - 1;
        lexmax_dim2 = 2 * p20 - 2 * p18 - 1;
        update_lexmax(arr_phi_old,lexmax_dim1,lexmax_dim2);
    }
    // Record access to phi
    p_outer = p16;
    // Record access lexmin/lexmax for all the inner dimension
    lexmin_dim1 = p17 - p21; lexmin_dim2 = p18 - p22;
    update_lexmin(arr_phi,lexmin_dim1,lexmin_dim2);
    lexmax_dim1 = p19 - p21 - 1;
    lexmax_dim2 = p20 - p21 - 1;
    update_lexmax(arr_phi,lexmax_dim1,lexmax_dim2);
}
}

Listing A.3: Inspector Code to compute footprint - Loop 2
A.4 Executor Code

The executor code generated using Algorithm 9 is shown in Listings A.4 and A.5.

```c
// Update ghosts in array phi_old with values at owners
communicate_reads(loop_0);
// Initialize ghosts in array phi to 0.0;
init_write_ghosts(loop_0);
loop_0=0;
for (k = 0; k < nlocal_2; ++k) {
  p1 = coarse_boxes_1[k];
  p2 = size_y_l[coarse_boxes_1[k]];
  p3 = size_x_l[coarse_boxes_1[k]];
  for (i = 0; i < p2; ++i)
    for (j = 0; j < p3; ++j)
      phi_c_t[i+1][j+1] = phi_old_l[p1][i-lexmin_phi_old_dim1[p1]]
                       [j-lexmin_phi_old_dim2[p1]];
  access_offset = access_phi_old[loop_0]
               - patches_l[coarse_boxes_1[k]];
  for (p = patches_l[coarse_boxes_1[k]]; p < ub[k]; ++p) {
    p4 = patches_l[coarse_boxes_1[k]];
    p5 = ub[k];
    p6 = patch_start_y_l[p+access_offset];
    p7 = patch_end_y_l[p+access_offset];
    p8 = patch_start_x_l[p+access_offset];
    p9 = patch_end_x_l[p+access_offset];
    p10 = start_y_l[coarse_boxes_1[k]];
    p11 = start_x_l[coarse_boxes_1[k]];
    p12 = start_y_l[neighb_l[p+access_offset]];
    p13 = start_x_l[neighb_l[p+access_offset]];
    for (i = p6 ; i < p7; ++i)
      for (j = p8 ; j < p9; ++j)
        phi_c_t[i+1-p10][j+1-p11] = phi_old_l[p14][i-p12-lexmin_phi_old_dim1[p14]]
                                    [j-p13-lexmin_phi_old_dim2[p14]];
  }
  loop_0 ++;
}
for (i = 0; i < p2; ++i)
  for (j = 0; j < p3; ++j)
    phi_l[p1][i-lexmin_phi_dim1[p1]][j-lexmin_phi_dim2[p1]] +=
      (phi_c_t[i][j+1] + phi_c_t[i+1][j] - rho_l[p1][i-lexmin_rho_dim1[p1]]
       [j-lexmin_rho_dim2[p1]])/8;
// Transfer partial contributions in ghosts of array phi to the owners
communicate_writes(loop_0);
```

Listing A.4: Executor Code - Loop 1
Listing A.5: Executor Code - Loop 2
BIBLIOGRAPHY


