RICE UNIVERSITY

Extending the Polyhedral Compilation Model for Debugging and Optimization of SPMD-style Explicitly-Parallel Programs

by

Prasanth Chatarasi

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Vivek Sarkar

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

Dr. John M. Mellor-Crummey
Professor of Computer Science and of
Electrical and Computer Engineering

Dr. Keith D. Cooper
Professor of Computer Science and
L. John and Ann H. Doerr Professor of
Computational Engineering

Dr. Jun Shirako
Research Scientist

Houston, Texas
24th April, 2017

ABSTRACT

Extending the Polyhedral Compilation Model for Debugging and Optimization of SPMD-style Explicitly-Parallel Programs

by

Prasanth Chatarasi

The SPMD (Single Program Multiple Data) parallelism continues to be one of the most popular parallel execution models in use today, as exemplified by OpenMP for multi-core systems, CUDA and OpenCL for accelerator systems, and MPI for distributed-memory systems. The basic idea behind the SPMD model, which differentiates it from task-parallel models, is that all logical processors (worker threads) execute the same program with sequential code executed redundantly and parallel code executed cooperatively.

This thesis extends the polyhedral compilation model to enable analysis of SPMD-style explicitly-parallel programs. This thesis demonstrates the value of this extended polyhedral model by describing its use for identification of data races, as well as identification and removal of redundant barriers. This thesis evaluates the effectiveness of these two applications using 34 OpenMP programs from the `OmpSCR` and the `PolyBench/ACC` OpenMP benchmark suites.

Dedicated to

Gautama Buddha (Siddhartha Gautama)

(c. 563 BCE/480 BCE c. 483 BCE/400 BCE)

&&

My grand parents

Late Ch.Rama Rao and T.V.Ravanamma

Acknowledgments

Taittiriya Upanishad, Shikshavalli I.20

मात्रुदेवो भव पित्रुदेवो भव आछार्यदेवो भव अतिथिदेवो भव

mātrudevo bhava pitrudevo bhava āchāryadevo bhava atithidevo bhava

"Respects to Mother, Father, Guru and Guest. They are all forms of God."

Foremost, I would like to express my gratitude to my mother *Ch. Anjanee Devi* and my father *Dr. C. V. Subbaiah* for always being there for me, and loving me unconditionally through out the situations of extreme happiness to depression. Without the two of you, I don't know where I would be. If I have learned anything while being away from you, it is that you are the most important people in my life, and I love you both more than anything.

I would like to express my sincere appreciation to my guru (advisor) *Prof. Vivek Sarkar*, who always has had time for me when I needed him – no matter whether the reason was a technical discussion, an administrative problem, academic development, or the planning of my next steps. There are no words that can express my gratitude towards your efforts. Thank you for believing in me and supporting. Without you, I would not be able to handle everything that graduate program and life throw at me. Having you backing me up 100% allows me to be at peace and do research. You're much more than an advisor, and thanks for helping me in my personal life too.

I am also very grateful to my co-advisor *Dr. Jun Shirako* with whom I had many very fruitful discussions on various topics of this thesis. Thank you for always

being supportive, even when I feel like I cant do it. Also, thank you for being a humble teacher in explaining answers to my questions. Thanks for respecting my opinions, and those lunch sessions not only are great to discuss my ideas but to build a relationship with you.

I would also like to thank rest of my committee *Prof. John Mellor-Crummey* and *Prof. Keith Cooper* for agreeing to be a part of my thesis committee. I am very fortunate to have taken John's course on Parallel Computing and Multi-core Computing, and Keith's course on Advanced Compiler Construction, which laid the foundations for my introduction to mainstream compiler optimizations and parallel programming. I would also like to thank you all for your time, feedback and suggestions for numerous valuable improvements in my thesis.

I am also very grateful to the members of Polyhedral Research Community – *Dr. Uday Bondhugula* for teaching me the foundations of polyhedral compilation techniques, *Dr. Albert Cohen* for broadening my knowledge on applying expansion techniques using polyhedral model, and *Dr. Martin Kong* for having collaboration on this thesis, and providing suggestions to improve my research. I would like to acknowledge the members of the Habanero Extreme Scale Software Research Project at Rice, PARKAS research team at INRIA Paris, and Multicore computing lab at IISc Bangalore for the research interactions.

I would also like to acknowledge my teacher *Prof. Kesav Nori* for piquing my curiosity about compilers during my undergraduate study at IIT Hyderabad. Without you, I would not be where I am today. Also, thanks are due to my bachelor thesis advisors *Dr. Aditya Nori, Dr. M. V. Panduranga Rao* and *Dr. Bheemarjuna Reddy* for giving me a research exposure in the undergraduate study itself. Also, thanks to all my professors who encouraged me to apply for graduate studies, and for quickly providing reference letters.

Furthermore, I also really appreciate my senior graduate students Dr. Shams

Imam, Dr. Milind Chabbi, Dr. Karthik Murthy, Dr. Deepak Majeti, and Dr. Rishi Surendran for all the helpful advice in both research and personal life. I cannot thank you enough for everything you taught me while your stay at Rice. I greatly value your kindness and the expertise you imparted to me as my mentors.

Thanks are due to my friends for all the support and encouragement during my stay at Rice. There are too many of you to mention, but I would especially like to thank Adithya, Ankush, Arghya, Arkabandu, Hamim, Kuldeep, Lechen, Mohit, Nishant, Priyanka, Quazi, Ramya, Rabimba, Rohan, Sharan, Sourav, Sriparna, Sriraj, Suguman, Vivek, and Yaswanth. Thanks for always being up for a good laugh over the years. I would also like to thank members of Rice Computer Science department staff especially Belle, Melissa, Sherry, Beth, Lena, Annepha and Carlyn for all the help I received during my stay at Rice.

Also, I would appreciate Office of International Students & Scholar (OISS) at Rice for organizing International Friends at Rice (IFR), through which I could meet a beautiful U.S. family Larry and Carole Huelbig. I express deeper gratitude to them for helping me to feel more at home, introducing me to various activities/shows in Houston, and sharing cross-cultural experiences.

The acknowledgments never end without mentioning siblings, i.e., my elder sister *Sree Pavani* and my elder brother *Sreenivas*. First and for most, life may not be that exciting without you. We may fight 50% of the time, but I love you a lot for being my best friends, toughening up during tough situations of my life, and celebrating with me during happy moments. As per a Vietnamese proverb, you both are as close as my hands and feet. Also, I would like to express my gratitude to my sister's family including my brother-in-law *Srinivasulu*, and my cute nieces *Teju* and *Sai*, for the homely support during my stay in the United States.

Contents

Abstract i

Acknowledgments iii

List of Illustrations ix

List of Tables xii

1 Introduction 1

1.1 Thesis Statement 4

1.2 Contributions 4

1.3 Outline 4

2 Background 6

2.1 Explicitly-Parallel Programs 6

2.1.1 SPMD-style Parallelism 7

2.1.2 Serial-elision Property 9

2.2 Mathematical Foundations for the Polyhedral Model 13

2.3 Polyhedral Model 17

2.3.1 Polyhedral Representation of Programs 18

2.3.2 Dependence Analysis 21

2.3.3 Affine Program Transformations 23

2.3.4 Code Generation 25

2.4 Limitations of the Polyhedral Model 25

3 Extensions to the Polyhedral Model for SPMD Programs 28

3.1 Important Concepts in an SPMD Execution 28

3.2 Space Mapping 30
3.3 Phase Mapping 33
3.4 May-Happen-in-Parallel (MHP) Analysis 40
3.5 Past Work in Extending Polyhedral Model for Explicitly-Parallel Programs 43

4 PolyOMP: A Polyhedral Framework for Debugging and Optimizations of SPMD Programs 46
4.1 Overall Workflow 47

5 Debugging Of SPMD Programs – Static Data Race Detection 50
5.1 Motivation 50
5.2 Our Approach 53
5.2.1 An Algorithm to Identify Data Races 53
5.3 Experimental Evaluation 55
5.3.1 Experimental Setup 55
5.3.2 OpenMP Source Code Repository 56
5.3.3 PolyBench/ACC OpenMP Suite 58
5.4 Strengths and Limitations of Our Approach 63
5.5 Past Work on Race Detection 64

6 Optimization Of SPMD Programs – Static Redundant Barrier Detection 67
6.1 Motivation 68
6.2 Our Approach 70
6.2.1 An Algorithm to Identify Redundant Barriers 70
6.2.2 A Greedy Approach to Compute a Set of Required Barriers . 74

6.3 Experimental Evaluation 76
6.3.1 Experimental Setup 76
6.3.2 OpenMP Source Code Repository 77
6.3.3 PolyBench/ACC OpenMP Suite 79
6.4 Strengths and Limitations of Our Approach 81
6.5 Past Work on Analysis of Barriers 82

7 Conclusions & Future Work **85**

Bibliography **88**

Illustrations

2.1 SPMD programs of the class C1 satisfy the serial-elision property. . . 11

2.2 SPMD programs (having barriers) of the class C2 don't satisfy the serial-elision property. 12

2.3 A two dimensional integer set $S = \{(i,j) \mid (2 \leq i \leq 8) \wedge (1 \leq j \leq i-1)\}$, with horizontal axis as dimension `i` and vertical axis as dimension `j`*(courtesy: `islplot` display tool [1]). 13

2.4 A map $M = \{(i,j) \rightarrow (i+j+3, j+1)\}$ with the input elements from the orange colored set $S_1 = \{(i,j) \mid 1 \leq i,j \leq 3\}$ and output elements from the blue colored set $S_2 = \{(i,j) \mid (j \geq -5+i) \wedge (2 \leq j \leq 4) \wedge (j \leq -3+i)\}$, with horizontal axis as dimension `i` and vertical axis as dimension `j`. . . . 14

2.5 Traditional workflow of polyhedral compilation frameworks 17

2.6 Working example: Smith-Waterman excerpt 18

2.7 Iteration domain of statement `S` in the Smith-Waterman kernel for the value of M = 5 and N = 9†(courtesy: `islplot` display tool [1]). . 19

2.8 Execution order of instances of statement `S` in the Smith-Waterman kernel (courtesy: `islplot` display tool [1]). 21

2.9 Dependence relations on statement `S` in the Smith-Waterman kernel (courtesy: `islplot` display tool [1]). 22

2.10 Iteration domain of statement S in the Smith-Waterman kernel after loop skewing to expose parallelism at loop-j (courtesy: `islplot` display tool [1]). 24

2.11 Transformed code of Smith-Waterman kernel with parallelism at innermost loop (j-loop) 25
2.12 An example to discuss limitations of the polyhedral model 26

3.1 An example to motivate important concepts in an SPMD execution . 29
3.2 Overall SPMD execution of the program in Figure 3.1 with two threads 29
3.3 An OpenMP SPMD-style program with various directives 32
3.4 An OpenMP SPMD program that includes barriers with depth > 0. . 35
3.5 Overall SPMD execution of the program in Figure 3.4 with two threads and value of N as 2 36

4.1 Summary of the `PolyOMP`, a polyhedral framework for debugging and optimizations of SPMD programs 46
4.2 Overview of the `PolyOMP` system built on top of the Polyhedral Extraction Tool (`PET`, version: pet-0.08-30-g77689da) [2]. 48

5.1 Data races in the Jacobi benchmark from `OmpSCR` benchmark suite . . 51
5.2 PolyBench/ACC OpenMP benchmark developer might have forgotten to mark certain variable as private variables (`x` in Cholesky, `nrm` in Gramschmidt), and there by resulting races on such variables. 61
5.3 PolyBench/ACC OpenMP benchmark developer have incorrectly parallelized the linear algebra kernels (some of them are notoriously hard to be parallelized because of complex dependence patterns), and there by resulting races on arrays `C` in `Symm` and `B` in `Trmm` benchmarks. 62

6.1 Redundant barrier (implicit) at line 11 in the `3mm` benchmark from `PolyBench/ACC` benchmark suite 68
6.2 Bipartite graph constructed by mapping each barrier in `3mm` benchmark to data races that can be avoided with the barrier 72

List of Algorithms

1 Building phase mappings of statements 38

2 Building May-Happen-in-Parallel (MHP) information between statements S and T. 43

3 An approach to compute a set of data races in an SPMD program . . . 54

4 An approach to compute a set of redundant barriers in an SPMD program 71

5 An approach to construct a bipartite graph from barriers to data races in an SPMD program 73

6 A greedy approach to compute a set of required barriers 75

Tables

5.1 Race detection analysis over the subset of `OmpSCR` benchmark suite. `PolyOMP` - Detection time / Reported / False +ves : Total time taken to detect races by PolyOMP, Number of reported races, Number of false positives among reported. ARCHER / Intel Inspector XE: Number of races reported. 57

5.2 Race detection analysis over the subset of PolyBench/ACC OpenMP benchmark suite. `PolyOMP` - Detection time / Reported / False +ves : Total time taken to detect races by PolyOMP, Number of reported races, Number of false positives among reported. Intel Inspector XE: Number of races reported, Hang up (H) and Application exception (A). 59

5.3 Closely related static approaches in race detection 65

6.1 Hardware specifications of the experimental setup for evaluating our approach to identify redundant barriers. 77

6.2 Redundant barrier detection analysis over the subset of `OmpSCR` benchmark suite. Benchmarks labeled with (*) have no true races but our race detection algorithm reported false positives, and benchmarks with (**) indeed have true races. Our tool ignored (I) the benchmarks with labels (*, **) because of the presence of races (including false positives). `size, k, error, numiter` are symbolic parameters in the corresponding benchmarks. Note that we also count implicit barriers after the `omp parallel` construct even though these implicit barriers cannot be removed from the source code. . . . 78

6.3 Redundant barrier detection analysis over the subset of PolyBench/ACC OpenMP benchmark suite. Benchmarks labelled with (*) doesn't have redundant barriers, and we didn't run (NR) the benchmarks for performance evaluation. Benchmarks labelled with (**) have true races, and our tool ignored (I) these benchmarks. A - Application exception, i.e., Segmentation fault in the original program itself. Note that we also count implicit barriers after the `omp parallel` construct even though these implicit barriers cannot be removed from the source code. 80

6.4 Closely related static approaches in barrier analysis 83

Chapter 1

Introduction

It is widely recognized that computer systems anticipated in the 2020 time frame will be qualitatively different from computer systems of previous decades. Specifically, they will be built using homogeneous and heterogeneous many-core processors with hundreds of cores per chip. Also, the performance of these processors will be driven by parallelism and constrained by energy and data movement [3]. This trend towards ubiquitous parallelism has forced the need for improved productivity and scalability in parallel programming models. Two classical programming models that were conceived to express parallelism are the Single-Program-Multiple-Data (SPMD) computational model [4] and the fork-join computational model [5].

In the SPMD computational model, all processes working together will execute the same program [4], i.e., all logical processors (worker threads) execute the same program with sequential code executed redundantly and parallel code executed cooperatively. In the last couple of decades, the SPMD computational model has been exemplified by OpenMP [6] for multi-core systems, CUDA [7] and OpenCL [8] for accelerator systems, as well as MPI [9] for distributed-memory systems.

In the fork-join computational model, a sequential thread spawns (fork) multiple threads to execute a portion of code concurrently and waits (join) for the spawned threads to finish their part of execution [5]. This computational model is used by task parallel programming models and libraries, such as OpenMP [6], Chapel [10], Cilk [11], and X10 [12]. There is a general agreement that the fork-join model is more productive than the SPMD model, i.e., programmers can express different styles of

parallelism and concurrency more conveniently in the fork-join model than in the SPMD model. However, the SPMD model has been developed as a straightforward method of low-overhead parallel execution compared to the fork-join model [4]. Hence, some compiler approaches [13, 14, 15, 16] transform a given region of fork-join code to the SPMD model for improved performance, a transformation technique that is referred to as "SPMDization".

Current production compilers enhance the performance of source programs by performing SSA-based optimizations, automatic vectorization, loop-level transformations based on data-dependence analysis, inspector-executor strategies, and profile guided optimizations [17]. In the coming decades, compiler research is expected to play a crucial role in addressing the multi-core programming challenge, i.e., how to exploit the parallelism in large-scale parallel hardware without undue programmer effort. Also, multi-core programming challenge is mentioned as one of the major challenges facing the overall computer field [18]. Traditionally, the two major approaches in handling the parallel programming challenge are automatic parallelization of sequential programs, and analysis and optimization of explicitly-parallel programs.

In the automatic parallelization approach [19, 20, 21, 22, 23, 24, 25] of sequential programs, the programmer provides either a portion of a sequential program or a high-level specication of a computation. Then, the compiler identifies parallelism available in the program and generates parallel code for a broad range of architectures. When successful, automatic parallelization removes the significant burden from a programmer to manually re-write sequential programs for parallel execution, which often requires the generation of different parallel code for different parallel platforms. The polyhedral model [26, 27, 28, 29], a mathematical algebraic framework, represents one of the major automatic parallelization approaches for a variety of architectures including multi-cores [23, 24], accelerators such as GPU's/ FPGA's [25, 30, 31], and distributed-memory systems [32]. The polyhedral model reasons about executions (instances) of statements, and can be used for accurate dependence analysis

over arrays, data flow analysis of arrays, applying loop transformations holistically, and generating transformed code in high-level languages. However, despite decades of research on automatic parallelization, fully automatic parallelization of sequential programs by compilers remains difficult due to its need for a complex program analysis and unknown input parameters (such as indirect array subscripts) during compilation [33, 34].

An alternate approach to address the multi-core programming challenge is through analysis and optimization of explicitly-parallel programs [35, 36, 37, 38, 39, 40, 15, 41, 42], in which the programmer species the logical parallelism and explicit synchronizations in the source program, and the compiler extracts the parallelism subset that is best suited for a given target platform. An interesting property of an explicitly-parallel program is that it specifies a partial execution order, unlike a sequential program, which specifies a total order. In this approach, the compiler is made aware of explicitly-parallel information (the partial execution order) such as Happens-Before (HB) relations, May-Happen-in-Parallel (MHP) relations and Never-Execute-in-Parallel (NEP) relations from the various constructs in the source program. Then the compiler leverages the parallel information to perform stronger and more focused debugging analyses with the goal of, for instance, detecting races, deadlocks or localizing the root cause of false sharing (an important source of "performance bugs"); on the other hand, the explicit parallelism can also be used to enable loop transformations after fusing SPMD regions, removing redundant barriers, code motion on a region of code surrounded by constructs that enforce mutual exclusion. The major challenge in the analysis of explicitly-parallel programs is the extraction and representation of parallel information (HB, MHP, NEP) from different parallel and concurrency constructs [43]. In this thesis, we propose a new system (`PolyOMP`) which captures the partial execution order present in an SPMD-style parallel program as May-Happen-in-Parallel (MHP) information in the polyhedral model, and utilizes the MHP information for debugging and optimizations of SPMD programs.

1.1 Thesis Statement

Though the polyhedral compilation model was designed for analysis and optimization of sequential programs, our thesis is that it can be extended to enable analysis of SPMD-style explicitly-parallel programs with benefits to debugging and optimization of such programs.

1.2 Contributions

This thesis makes the following contributions in defense of our thesis statement:

- It describes our extensions to the polyhedral compilation model to represent partial execution order present in SPMD-style parallel programs.
- It formalizes the partial order as May-Happen-in-Parallel (MHP) information using our extensions to the polyhedral model.
- It presents an approach for compile-time detection of data races in SPMD-style parallel programs [44].
- It also presents an approach for identification and removal of redundant barriers at compile-time in SPMD-style parallel programs.
- It demonstrates the effectiveness of our approaches on 34 OpenMP programs from the `OmpSCR` and the `PolyBench/ACC` OpenMP benchmark suites.

1.3 Outline

The rest of this thesis is organized as follows.

- Chapter 2 summarizes background on the SPMD parallel execution model and on the polyhedral model. Some of the fundamental concepts related to the

polyhedral model are taken from the PhD dissertations of Bondhugula, Grosser and Kong [45, 46, 47].

- Chapter 3 explains the lack of existing approaches in the polyhedral model to capture partial execution orders originating from barriers in SPMD programs. Then, we present our formal extensions (space and phase mappings) to the polyhedral representation, and also introduce an algorithm to formalize May-happen-in-parallel relations from the extensions as a way to capture the partial orders. Also, we summarize the past work in extending polyhedral model to enable analysis of explicitly-parallel programs.

- Chapter 4 provide an overview of our system (`PolyOMP`) including the description of all components involved in the system.

- Chapter 5 describes our approach to compile-time detection of data races in SPMD-style parallel programs. Then, we evaluate our technique for race detection on 34 OpenMP programs from the `OmpSCR` and the `PolyBench/ACC` OpenMP benchmark suites. Also, we summarize related work on approaches for compile-time detection of data races.

- Chapter 6 introduces an approach to compile-time detection and removal of redundant barriers in SPMD-style parallel programs by building on the race detection approach of Chapter 5. Then, we analyze performance of the 34 OpenMP programs evaluated in Chapter 5 after applying our technique to remove redundant barriers. Also, we summarize related work on approaches for compile-time analysis of barriers in the SPMD programs.

- Finally, Chapter 7 present our conclusions and directions for future research.

Chapter 2

Background

I believe in innovation and that the way you get innovation is you learn the basic facts.

Bill Gates

This chapter begins with a discussion of the motivation for explicitly-parallel programs, and briefly summarizes SPMD-style parallelism using OpenMP as an exemplar. Then, we summarize the mathematical foundations of the polyhedral model, which in turn provide the theoretical foundation for the contributions in this thesis. Then, we briefly summarize the polyhedral model including the polyhedral representation of programs, dependence analysis, loop transformations, and code generation. Finally, we conclude with limitations of the polyhedral model, which in turn provide the motivation for our research in this thesis.

2.1 Explicitly-Parallel Programs

Traditionally, there have been two different approaches in programming parallel architectures: the automatic parallelization and the explicitly-parallel programming approach. In the automatic parallelization approach, the programmer provides either a portion of a sequential program or a high-level specification of a computation. Then, the compiler identifies parallelism available in the program and generates parallel codes for a broad range of architectures. When successful, automatic parallelization

removes the significant burden from a programmer to manually re-write sequential programs for parallel execution, which often requires the generation of different parallel code for different parallel platforms. Also, there is a little effort required from the programmer, but there are many fundamental limitations that make it difficult for compilers to identify the parallelism from the input program.

The alternate approach is to write explicitly-parallel programs, in which the programmer specifies the logical parallelism and explicit synchronizations in the source program, and the compiler extracts the parallelism subset that is best suited for a given target platform. In this approach, the programmer takes care of providing the parallelism required for performance, and then the compiler takes care of generating low-level code for the architecture. Even though this approach can be tedious for programmers, it is the default approach used in practice because the programmers are more confident of a successful outcome with this approach than with automatic parallelization.

2.1.1 SPMD-style Parallelism

SPMD (Single Program Multiple Data) parallelism [13, 48] continues to be one of the most popular parallel execution models in use today, as exemplified by OpenMP [6] for multi-core systems, CUDA [7] and OpenCL [8] for accelerator systems, and MPI [9] for distributed-memory systems. The basic idea behind the SPMD model is that all logical processors (worker threads) execute the same program, with sequential code executed redundantly and parallel code (worksharing, barrier constructs, etc.) executed cooperatively. In this thesis, we focus on OpenMP [6] as an exemplar of SPMD parallelism. In the rest of this section, we explain the semantics of OpenMP constructs, which are considered in this thesis.

1. The OpenMP `parallel` construct creates a fixed number of parallel worker threads to execute an SPMD parallel region. The number of threads can be

specified in the code, or in an environment variable (`OMP_NUM_THREADS`), or via a runtime function, `omp_set_num_threads()` that is called before the `parallel` region starts execution.

2. The OpenMP `barrier` construct specifies a barrier operation among all threads in the current `parallel` region. In this thesis, we restrict our attention to textually aligned barriers [49], in which all threads reach the same textual sequence of barriers. Each dynamic instance of the same `barrier` operation must be encountered by all threads, e.g., it is not allowed for a barrier in a then-clause of an if statement executed by (say) thread 0 to be matched with a barrier in an else-clause of the same if statement executed by thread 1. We plan to address textually unaligned barriers as part of the future work. However, many software developers believe that textually aligned barriers are better from a software engineering perspective.

3. The OpenMP `for` construct indicates that the immediately following loop can be parallelized and executed in a work-sharing mode by all the threads in the parallel SPMD region. An implicit barrier is performed immediately after a `for` loop, while the `nowait` clause disables this implicit barrier. Further, a `barrier` is not allowed to be used inside a `for` loop. When the `schedule(`*`kind`*`,` *`chunk_size`*`)` clause is attached to a `for` construct, its parallel iterations are grouped into batches of *chunk_size* iterations, which are then scheduled on the worker threads according to the policy specified by *kind*.

4. The OpenMP `single` construct specifies that the enclosed code is to be executed by only one thread among all the threads in the parallel SPMD region. An implicit barrier is performed immediately after the enclosed code block by `single` construct, while the `nowait` clause disables this implicit barrier. A `single` OpenMP construct can be viewed as equivalent to a OpenMP `for` construct with a single-iteration loop.

5. The OpenMP `master` construct indicates that the immediately following region of code is to be executed only by the master thread of the parallel SPMD region. Note that, there is no implied barrier associated with this construct.

Since programmers can also leverage modern task-based parallel programming models such as Chapel [10] towards SPMD-style parallelism, we provide a brief background on required constructs in Chapel to express SPMD-style computation.

1. The chapel `coforall` loop construct creates a distinct task per loop iteration, each of which executes a copy of the loop body. The construct can be seen as a way to parallelize a loop where each iteration is independent of other iterations.

2. The chapel `barrier` construct can be used to prevent tasks from proceeding until all other related tasks have also reached the barrier. According to the Chapel documentation, it is legal to insert barriers in the parallel loops expressed by `coforall` construct unlike OpenMP which doesn't allow barriers inside the parallel loops.

2.1.2 Serial-elision Property

The *serial-elision* property is one of the interesting properties of explicitly-parallel programs. It is informally defined as the property that removal of all parallel constructs results in a sequential program that is a valid (albeit inefficient) implementation of the original parallel program semantics [50]. In the context of the Cilk programming language, serial-elision of a Cilk program is defined a Cilk program, when run on one processor, is semantically equivalent to the C program that results from the deletion of the Cilk keywords [51]. It has been shown in past work that restricting attention to parallel programs that satisfy the serial-elision property can simplify debugging and optimization of parallel programs [52, 41]. In contrast, this thesis focuses on debugging and optimization of SPMD programs, which in general

do not satisfy serial-elision property.

We classify SPMD programs into two classes, C1 – SPMD programs whose correctness doesn't depend on a fixed number of logical threads participating in the SPMD region, and C2 – SPMD programs which depend on a fixed number of logical threads for correct semantics. For SPMD programs in the class C1, one way to achieve a serial elision is to simply run the program with one thread. However, this approach cannot be applied to SPMD programs in the class C2, which include certain programs that conform with the OpenMP specification (see section 2.5.1 in [53]). Also, modern task-based parallel programming models such as Chapel [10] and X10 [12] allow barriers inside parallel loops unlike OpenMP programming model which doesn't allow barriers inside the parallel loops. To define the serial-elision version for generic SPMD programs with barriers, we consider SPMD programs (both in classes C1 and C2) written using OpenMP and Chapel programming models.

1) Class C1: In this class of SPMD programs, programmer guarantees the correctness independent of number of participating threads in the SPMD regions. Hence, serial-elision version of an SPMD program of class C1 is defined as a C program that results from the removal of `omp parallel`, `omp barrier` constructs in case of the OpenMP, and replacing `coforall` by a `for` loop, removal of all `barrier` constructs in case of the Chapel programming language. Since the semantics of SPMD programs of the class C1 is unchanged with the number of logical threads participating in the SPMD region, and execution of such programs with one thread is semantically equivalent to its serial-elision version, SPMD programs of the class C1 satisfy the serial-elision property. As can be observed from Figure 2.1, execution of both OpenMP and Chapel SPMD programs of the class C1 with one thread (T = 1) is semantically equivalent to its serial-elision version; hence SPMD programs of the class C1 satisfy the serial-elision property.

2) Class C2: In this class of SPMD programs, programmer guarantees the correct-

```
#pragma omp parallel \
    num_threads(T)
{
    S1;
#pragma omp barrier //B1
    S2;
    S3;
#pragma omp barrier //B2
}
```

(a) An OpenMP program of the class C1

⇓

```
{
    S1;
    S2;
    S3;
}
```

(c) Serial-elision version of the above program

```
var b = new Barrier(T);
coforall tid in 1...T do
{
    S1;
    b.barrier();
    S2;
    S3;
    b.barrier();
}
```

(b) A Chapel program of the class C1

⇓

```
{
    S1;
    S2;
    S3;
}
```

(d) Serial-elision version of the above program

Figure 2.1 : SPMD programs of the class C1 satisfy the serial-elision property.

ness only for a fixed number of participating threads in the SPMD regions. Hence, serial-elision version of an SPMD program of class C2 is defined as a C program that results from replacing each OpenMP parallel region containing a fixed number of threads, by a sequential loop that executes the body of the parallel region for num_threads() iterations, as well as removal of all OpenMP `barrier` constructs from the OpenMP program. In case of the Chapel, a serial-elision is obtained by replacing each `coforall` by a `for` loop, as well as of all `barrier` constructs from the program. Consider Figure 2.2 to illustrate the fact that eliding a barrier can alter the execution order of the original program, possibly leading to incorrect results and unexpected behavior. Eliding the barrier B1 can (for example) result in first logical thread executing statement S2 without waiting for third thread to complete executing statement S1. Likewise, eliding the barrier B2 can also lead to first thread not waiting for the

```
#pragma omp parallel \
    num_threads(3)
{
    S1;
#pragma omp barrier //B1
    S2;
    S3;
#pragma omp barrier //B2
}
```

(a) An OpenMP program of the class C2

```
for (tid = 0; tid < 3; \
    tid++)
{
    S1;
    S2;
    S3;
}
```

(c) Serial-elision version of the above program

```
var b = new Barrier(3);
coforall tid in 1...3 do
{
    S1;
    b.barrier();
    S2;
    S3;
    b.barrier();
}
```

(b) A Chapel program of the class C2

```
for tid in 1...3 do
{
    S1;
    S2;
    S3;
}
```

(d) Serial-elision version of the above program

Figure 2.2 : SPMD programs (having barriers) of the class C2 don't satisfy the serial-elision property.

remaining tasks to finish executing statements S2 and S3. Therefore, eliding a barrier from SPMD programs of the class C2 can alter the original program semantics, and violate the *serial-elision* property. However, SPMD programs of the class C2 without barrier satisfy the *serial-elision* property since executing all of thread related iterations in a serial fashion keeps semantics of original programs.

Our implementation (in `PolyOMP`) is specific to SPMD-style programs using the OpenMP programming model, and hence we don't encounter barriers in parallel loops; however, the ideas in this thesis can be applied to other languages (such as Chapel and CUDA) that support SPMD-style constructs with barriers in parallel loops, thereby violating the serial-elision property.

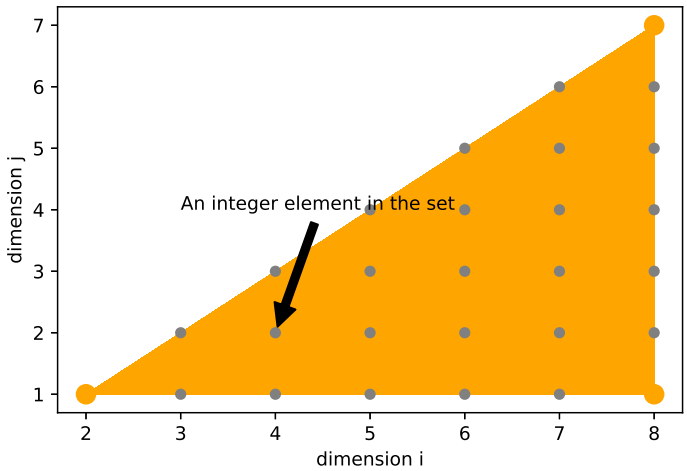


Figure 2.3 : A two dimensional integer set $S = \{(i,j) \mid (2 \leq i \leq 8) \wedge (1 \leq j \leq i-1)\}$, with horizontal axis as dimension `i` and vertical axis as dimension `j`*(courtesy: `islplot` display tool [1]).

2.2 Mathematical Foundations for the Polyhedral Model

In this section, we provide a brief overview of the polyhedral model [26, 27, 28, 29], a powerful framework that enables analyzing, reasoning, transforming and generating programs using mathematical representations. Some of the fundamental concepts related to the polyhedral model are taken from the PhD dissertations of Bondhugula, Grosser, and Kong [45, 46, 47].

Definition 2.2.1. (Integer set) A `d`-dimensional integer set is defined as a set of integer tuples from $\mathcal{Z}^d$ as described by Presburger formulas [54, 46]. Note that "*A Presburger formula is defined recursively as either a boolean constant, the result of a boolean operation such as negation, conjunction, and disjunction or implication, a quantified expression or a comparison between different quasi-affine expressions* [46]." For example, $S = \{(i,j) \mid (2 \leq i \leq N) \ \wedge \ (1 \leq j \leq i-1)\}$ is an example of a

*Note that the orange corner points also do belong to the triangle.

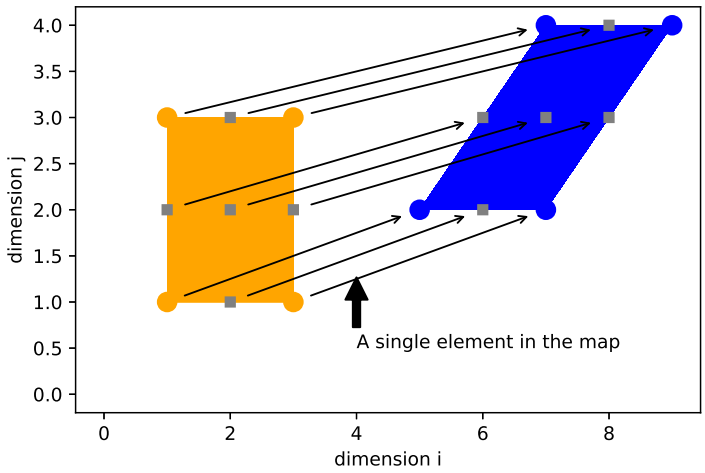


Figure 2.4 : A map $M = \{(i,j) \rightarrow (i+j+3, j+1)\}$ with the input elements from the orange colored set $S_1 = \{(i,j) \mid 1 \leq i,j \leq 3\}$ and output elements from the blue colored set $S_2 = \{(i,j) \mid (j \geq -5+i) \wedge (2 \leq j \leq 4) \wedge (j \leq -3+i)\}$, with horizontal axis as dimension `i` and vertical axis as dimension `j`.

two dimensional (dimensions i, j) integer set described in terms of a parameter `N`. Figure 2.3 illustrates the integer set S (for the value of N = 8), which consists of the brown integer points (including orange corner points) within the triangle. The shape is derived from the Presburger constraints $((2 \leq i \leq N) \ \wedge \ (1 \leq j \leq i-1))$ imposed on the set S and elements of the set are integral points in the shape. In general, integer sets are used in describing the iteration space of loops in a program.

Definition 2.2.2. (Integer Map) An integer map is a binary relation from an integer set of dimension d_1 to another integer set of dimension d_2. The first integer set in the relation is called the *domain* or the input set (according to `isl` [54]) and the second integer set is called the *range* or the output set. These integer maps are modeled as pairs of integer tuples from $\mathcal{Z}^{d_1} \times \mathcal{Z}^{d_2}$.

Figure 2.4 illustrates a map $M = \{(i,j) \rightarrow (i+j+3, j+1)\}$ with input elements from the orange colored set $S_1 = \{(i,j) \mid 1 \leq i,j \leq 3\}$ and output elements from the

blue colored set $S_2 = \{(i,j) \mid (j \geq -5+i) \wedge (2 \leq j \leq 4) \wedge (j \leq -3+i)\}$. Each black colored arrow represents an edge between an input tuple from the orange colored set S_1 to an output tuple from the blue colored set S_2. In general, integer maps are useful in transforming iteration space of loops in a program for optimizations. The map $M = \{(i,j) \rightarrow (i+j+3, j+1)\}$ (in Figure 2.4) can be viewed as loop skewing and shifting transformation on the S_1 set.

Definition 2.2.3. (Affine space/Affine set) A k-dimensional space `s` is called an affine space if the space is closed under affine combination i.e., if vectors $\vec{v_1}, \vec{v_2}$ are in the space `s`, then all the integer vectors lying on the line joining $\vec{v_1}$ and $\vec{v_2}$ should also belong to space `s`. In general, integer sets can be used to describe affine spaces, but the integer sets can also be used to describe quasi-affine spaces [54]. An example of a space that is quasi-affine but not affine is as follows:

$$S = \{j \mid (0 \leq i \leq 3) \ \wedge \ (j = 3 \times (i \ mod \ 3))\} = \{0, 3, 6\}$$

In the above quasi-affine space `S`, the element `2` on the line joining elements `0` and `3` doesn't belong to the space `S`.

Definition 2.2.4. (Affine function/Affine map) A function `f` is called an affine function [45] from a k-dimensional affine space to a d-dimensional affine space if it can be expressed of the form:

$$f(\vec{v}) = M_f \vec{v} + \vec{f_0} \tag{2.1}$$

where $\vec{v}$ and $\vec{f_0}$ are k-dimensional and d-dimensional vectors of integers, respectively. Also, M_f is an integer matrix with `d` rows and `k` columns. In general, integer maps can be used to describe affine maps, but integer maps can also describe quasi-affine maps [54].

Definition 2.2.5. (Affine hyperplane) An affine hyperplane can be viewed as a one-dimensional affine function that maps an n-dimensional space onto a one-dimensional space [45].

$$\phi(\vec{v}) = h.\vec{v} + c \tag{2.2}$$

Such an affine hyperplane can divide the input n-dimensional space into two half-spaces i.e., the positive half space ($\phi(\vec{v}) \geq 0$) and the negative half space ($\phi(\vec{v}) \leq 0$). The affine hyperplanes (a special case of affine functions) can also be described with integer maps.

Definition 2.2.6. (Polyhedron) A polyhedron can be formally defined as the set of solutions to a system of linear inequalities.

$$M\vec{x} \ \leq \ \vec{b} \tag{2.3}$$

where $\vec{x}, \vec{b}$ are `k` and `d` dimensional vectors respectively. A polyhedron can be viewed as an intersection of a finite number of half-spaces from affine hyperplanes. Also, M can be seen as set of `d` constraints on `k` dimensional vectors and expressed using an integer matrix with `d` rows and `k` columns. In general, an integer set cannot completely describe the set of points inside a polyhedron since the polyhedron can contain non-integer points as well. But, when the polyhedron is intersected with integer space, then the resulting polyhedron can be described using integer sets, and such resulting polyhedra are called Z-polyhedra.

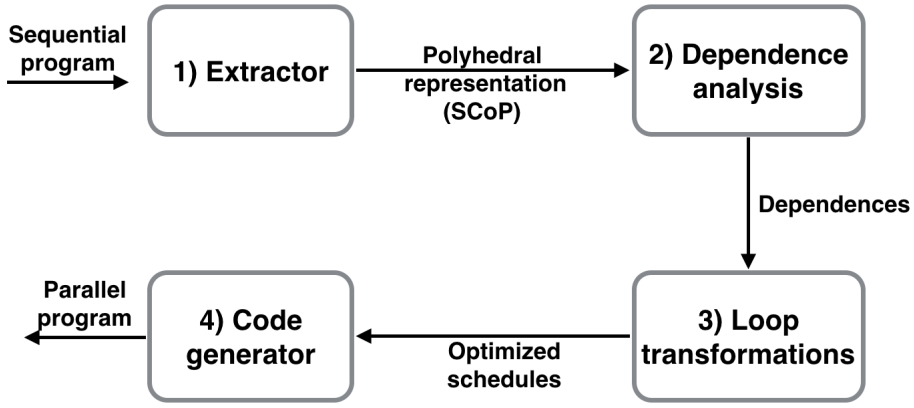


Figure 2.5 : Traditional workflow of polyhedral compilation frameworks

2.3 Polyhedral Model

Many scientific and engineering applications often spend most of their execution time in nested loops. Therefore, optimizing such nested loops can significantly improve the performance of the applications. The polyhedral model is a mathematical algebraic representation for such arbitrarily nested loops that enables systematic analysis and transformation at the granularity of array cells and statement instances [26, 27, 28, 29]. The polyhedral model has been shown to have significant advantages over AST-based (Abstract Syntax Tree) representations with respect to analysis and transformation of code regions consisting of loops and array accesses [55, 56]. One of its primary distinguishing factors is the use of powerful and robust code generation algorithms that can synthesize new code from algebraic specifications of transformations [57, 58, 59]. In the original formulation of the polyhedral model, all array subscripts, loop bounds, and branch conditions in *analyzable* programs were required to be affine functions of loop index variables and global parameters. However, decades of research since then have led to a significant expansion in the set of programs that can be considered analyzable by polyhedral compilation techniques [60, 61, 62].

```
for(int i = 1; i < M; i++) {
    for(int j = 1; j < N; j++) {
      A[i][j] = MAX(A[i-1][j], A[i-1][j-1], A[i][j-1]); //S
    }
}
```

Figure 2.6 : Working example: Smith-Waterman excerpt

Figure 2.5 presents a traditional workflow in restructuring input programs based on the polyhedral model. In the rest of this section, we describe each step of the workow using the Smith-Waterman kernel (shown in Figure 2.6) as a running example.

2.3.1 Polyhedral Representation of Programs

Loop nests amenable to polyhedral (algebraic) representation are called *Static Control Parts* (SCoP's) and represented in the `SCoP` format, which includes three elements for each statement, namely, the iteration domain, access relations and program schedule [63].

Definition 2.3.1. (Iteration Vector) The iteration vector (denoted by $\vec{i_S}$) of a statement `S` is defined as a multi-dimensional vector in which each element corresponds to a loop that surrounds the statement, ordered from outermost to innermost. The length of the vector is the number of loops surrounding the statement. As an example, the iteration vector for the first instance of statement S in Figure 2.4 is

$$\vec{i_S} = (i = 1, j = 1)$$

Thus, an iteration vector represents a dynamic instance of a statement in a loop nest.

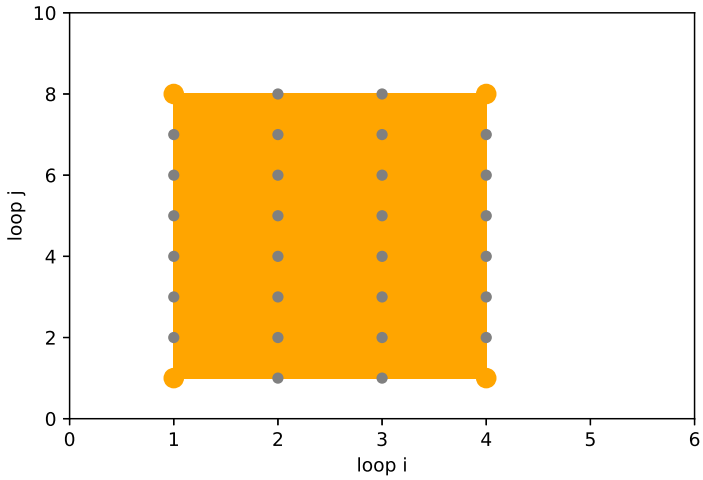


Figure 2.7 : Iteration domain of statement `S` in the Smith-Waterman kernel for the value of M = 5 and N = 9[†](courtesy: `islplot` display tool [1]).

Definition 2.3.2. (Statement Instance) Each dynamic instance of a statement `S` in a program is identified by its iteration vector ($\vec{i_S}$), which contains values for the iterators of the surrounding loops. For example, S(i = 1, j = 1) in the Smith-Waterman kernel refers to the statement `S` when the values of loop iterators `i,` and `j` are 1, i.e., the first instance that gets executed among all the statement instances of `S`.

Definition 2.3.3. (Iteration Domain) The iteration domain (denoted by $\mathcal{D}^S$) of a statement `S` is defined as the set of all iteration vectors, i.e., the set of all possible dynamic instances of the statement. The iteration domain of the statement `S` in the Smith-Waterman kernel as follows:

$$\mathcal{D}(S(\vec{i_S})) = \{(i,j) \mid (1 \leq i \leq M-1) \ \wedge \ (1 \leq j \leq N-1)\}$$

Figure 2.7 shows the iteration domain of statement `S` in the Smith-Waterman kernel for the value of M = 5 and N = 9. Each point in the iteration domain is an execution

[†]Note that the orange corner points also do belong to the iteration domain.

instance $\vec{i_S} \in \mathcal{D}$ of the statement. If the length of the iteration vector of the statement is `m`, then the iteration domain of the statement can be viewed as an m-dimensional bounded polyhedron(polytope) [64].

Definition 2.3.4. (Access Relation) Each memory reference (such as scalars, arrays, structures), denoted by $\mathcal{A}$, in a statement is expressed as an access relation, which maps a statement instance $\vec{i_S}$ to one or more memory locations to be read-/written [65]. The read memory reference `A[i-1][j-1]` in the statement `S` of the Smith-Waterman can be expressed as follows:

$$\mathcal{A}_1^{read}(S(\vec{i_S})) = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ M \\ N \\ 1 \end{bmatrix}$$

Definition 2.3.5. (Schedule) A schedule (denoted by $\Theta^S(S(\vec{i_S}))$) is an affine function that describes the order in which each dynamic instance of the statement will be executed in the original or transformed version of the program.

In other words, the execution order of a program is captured by the schedule, which maps a statement instance $S(\vec{i_S})$ to a logical timestamp. In general, a schedule is expressed as a multidimensional vector, and statement instances are executed according to the increasing lexicographic order of their timestamps. For example, the execution order of instances of statement S in the original version of the Smith-Waterman kernel is as follows:

$$\Theta^S(S(\vec{i_S})) = (i, j)$$

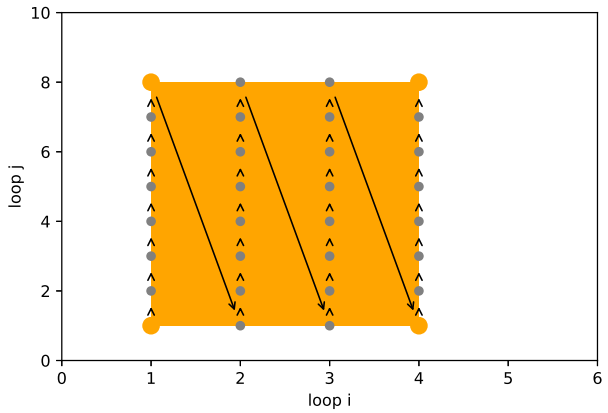
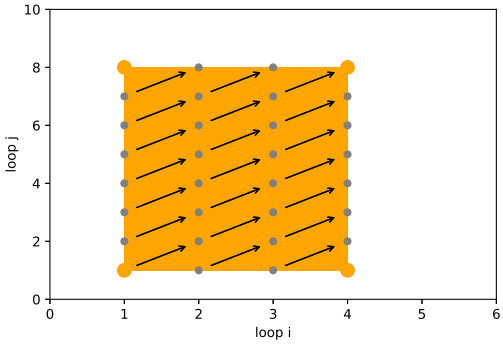


Figure 2.8 : Execution order of instances of statement `S` in the Smith-Waterman kernel (courtesy: `islplot` display tool [1]).

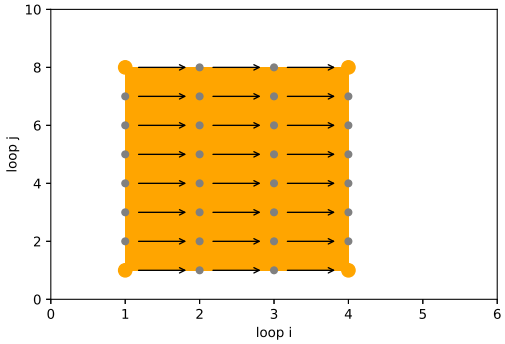
As can be seen from Figure 2.8, the points in the iteration domain of the statement `S` will be executed in the order specified by the schedule $\Theta^S(\vec{i_S})$.

2.3.2 Dependence Analysis

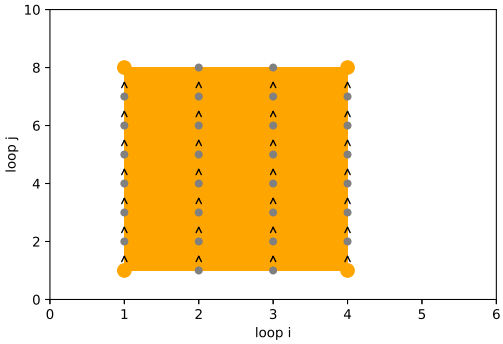
After extracting the polyhedral representation of an input program, dependence analysis phase computes memory based dependences which need to be preserved in transformations. A statement instance T is dependent on another statement instance S if they access the same memory location (at least one of the accesses is a write), and there exists a possible program execution path from S to T. Dependence relations denoted by $\mathcal{D}^{S \to T}$ capture the precise constraints under which the statement instances S and T are dependent on a memory location. For example, a dependence relation arising from the pair of memory accesses, i.e., `A[i][j]` and `A[i-1][j-1]` of statement `S` in the Smith-Waterman are as follows:



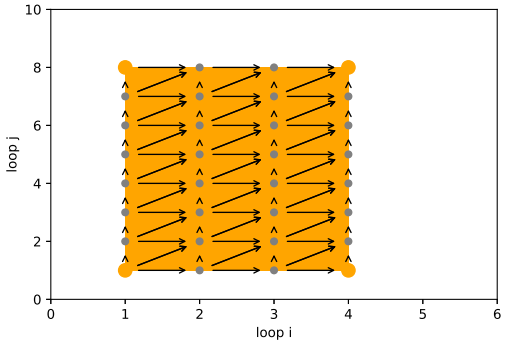
(a) Dependence between instances of statement S with read on A[i-1][j-1] and write on A[i][j]



(b) Dependence between instances of statement S with read on A[i-1][j] and write on A[i][j]



(c) Dependence between instances of statement S with read on A[i][j-1] and write on A[i][j]



(d) Union of dependences from parts (a), (b), (c)

Figure 2.9 : Dependence relations on statement `S` in the Smith-Waterman kernel (courtesy: `islplot` display tool [1]).

$$\mathcal{D}^{S\to S} = \{\, (\vec{i_S} = (i_s, j_s)) \to (\vec{i'_S} = (i'_s, j'_s)) \mid (\vec{i_S}, \vec{i'_S} \in \mathcal{D}(S(\vec{i_S})) \wedge (i_s = i'_s - 1) \wedge (j_s = j'_s - 1) \wedge \Theta^S(S(\vec{i_S})) \preceq \Theta^S(S(\vec{i'_S}))\,\}$$

$$where\ \Theta^S(S(\vec{i_S})) \preceq \Theta^S(S(\vec{i'_S})) = (i_s < i'_s) \vee (i_s = i'_s \wedge j_s < j'_s) \tag{2.4}$$

In the equation (2.4), $\vec{i_S}$, $\vec{i'_S}$ are iteration vectors of source and sink of the dependence relation respectively. The constraint that the iteration vectors of source and sink should belong to their corresponding iteration domain is captured by the condition, $(\vec{i_S}, \vec{i'_S} \in \mathcal{D}(S(\vec{i_S}))$. Also, the constraint that the source and sink of the dependence should access same memory location is captured by the condition, $((i_s = i'_s - 1) \wedge (j_s = j'_s - 1))$. Further more, the constraint that the source of the dependence should execute before the sink of the dependence is captured by the condition $\Theta^S(S(\vec{i_S})) \preceq \Theta^S(S(\vec{i'_S}))$.

Figure 2.9 shows all dependence relations among instances of S arising from all the read accesses (`A[i-1][j-1], A[i-1][j], A[i][j-1]`) and the write access (`A[i][j]`) in the Smith-Waterman kernel. Above dependence relations from the input program are then leveraged to compute a new program schedule that can expose parallelism in the Smith-Waterman kernel.

2.3.3 Affine Program Transformations

Polyhedral optimizers such as PLuTo [23], PolyAST [24], PPCG [25] take polyhedral representation of an input program along with memory based dependences as input. Then, the optimizers compute a best affine program transformation, i.e., a sequence of tens of textbook loop transformations, so as to enable parallelism, vectorization or improve data locality, while obeying the inherent dependences in the input program.

Definition 2.3.6. (Affine Transformation) "*Affine transformations (denoted by* $\mathcal{T}(S(\vec{i_S}))$*) are the class of transformations that preserve the collinearity and convexity of points in space, besides the ratio of distances*" [45].

Several loop transformations such as skewing, interchange, distribution, fusion, non-parametric tiling, and others can be represented using affine transformations.

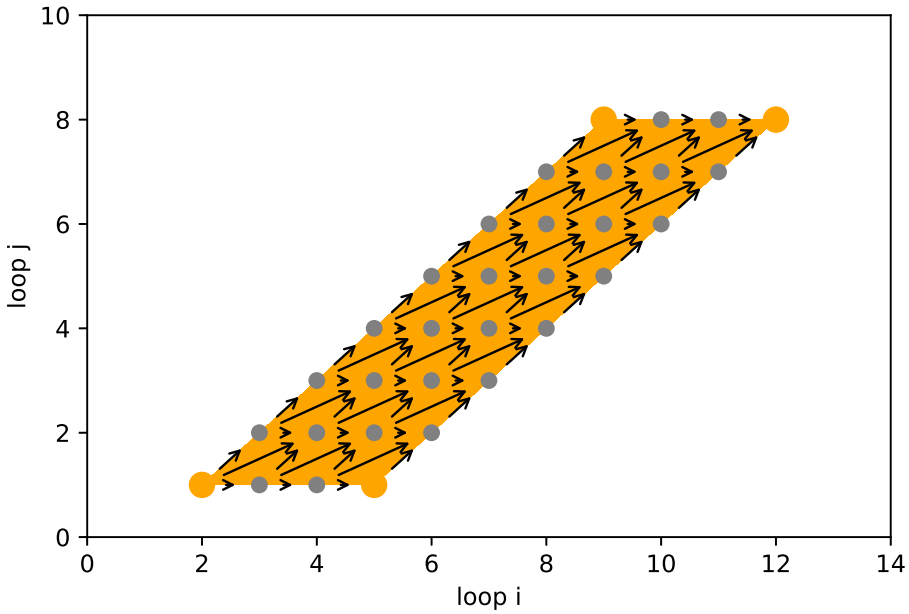


Figure 2.10 : Iteration domain of statement S in the Smith-Waterman kernel after loop skewing to expose parallelism at loop-j (courtesy: `islplot` display tool [1]).

An affine transformation can reorder the execution of statement instances to expose hidden parallelism while obeying the inherent dependences in the input program. For example, neither of the loops (`i,j loops`) in the Smith-Waterman kernel can be parallelized because of carried dependences on both `i, j` loops. However, loop skewing can be applied on the Smith-Waterman kernel to enable innermost loop (j-loop) to be executed in parallel. The affine program transformation (shown in Figure 2.10) corresponding to the loop skewing on Smith-Waterman kernel is as follows.

$$\mathcal{T}(S(\vec{i_S})) = \{(i,j) \rightarrow (i',j') \mid (i' = i+1) \wedge (j' = j)\} \tag{2.5}$$

In general, polyhedral optimizers rely on integer linear programming (ILP) based cost modeling to compute the best affine program transformation by respecting program dependences, and maximizing the parallelism and data locality [23, 24, 25].

```
for (int i = 2; i < M + N - 1; i += 1) {
    #pragma omp parallel for
    for(int j = max(1, -M + i + 1); j < min(N, i); j += 1){
        A[i-j][j] = MAX(A[i-j-1][j], A[i-j-1][j-1], \
                        A[i-j][j-1]);
    }
}
```

Figure 2.11 : Transformed code of Smith-Waterman kernel with parallelism at innermost loop (j-loop)

2.3.4 Code Generation

Generating code in the polyhedral model is finding a set of nested loops visiting each integral point of each polyhedra, once and only once, following the execution order from schedules. As explained, one of the key distinguishing factors of the polyhedral model is the use of powerful code generation algorithms (Quilleré [66] and Extended Quilleré [58]) that can synthesize new code from algebraic specifications of transformations. Figure 2.11 shows the transformed code of Smith-Waterman kernel after applying loop skewing transformation (show in equation 2.5).

2.4 Limitations of the Polyhedral Model

In the original formulation of the polyhedral model, all array subscripts, loop bounds, and branch conditions in *analyzable* programs were required to be affine functions of loop index variables and global parameters. As can be observed from an example program in Figure 2.12, the array subscript (`A[i*j]`) in the statement S1, the i-loop bound (`M*M`), and the branch condition (`C[j] > 0`) are non-affine functions of the loop index variables (`i, j`), and the global parameters (`M, N`); hence they are not analyzable with the polyhedral model precisely. However, decades of research since then have led to a significant expansion of programs that can be considered analyzable

```
1  int M;
2  int N = M/2;
3  int A[1000], B[1000], C[1000];

5  #pragma scop
6  int P = M/2;
7  for(int i = 0; i < M * M; i++) {
8      for(int j = i; j < N; j++) {
9          for(int k = j; k < P; k++) {
10             if(C[k] > 0)
11                 A[i*j] = B[j]; //S1
12         }
13     }
14 }
15 #pragma endscop
```

Figure 2.12 : An example to discuss limitations of the polyhedral model

by polyhedral frameworks using the approaches such as array region analysis [62], fuzzy array data flow analysis [67] and other variants to approximate the access relations for arrays having non-affine subscripts. Also, certain polyhedral extraction tools such as PET [2] can extract the relation between parameters (e.g., P = M/2 in line 6) in the program shown in Figure 2.12, and capture these relations into the polyhedral representation of the program. However, these extraction tools may not be able to capture some relations e.g., N = M/2 in line 2 of the above program since the extraction tools ignore regions of code not surrounded by pragma scop's. Finally, the remaining constraints in the polyhedral model stem from restrictions on various program constructs including pointer aliasing, unknown function calls, recursion, and unstructured control flow.

Previously [68, 41], we showed how explicit structured (loop and task level) parallelism can be harnessed to enhance dependence analysis, thereby enabling a larger set of transformations on the input program. The dependence analysis is improved by first enabling conservative dependence analysis of *serial-elision* version of the input program, i.e., the program without parallel constructs. (This past work assumed

that the input parallel program was constrained to satisfy the serial-elision property, which requires that the serial-elision version is a correct implementation of the parallel program.) Next, *happens-before* relations are identified from the explicitly-parallel constructs, such as tasks and parallel loops, and *intersected* with the conservative dependences. Finally, the resulting set of dependences is passed on to a polyhedral optimizer, such as PLuTo [22] or PolyAST [24], to enable transformations of explicitly-parallel programs with unanalyzable data accesses. The approaches in [68, 41] did not require alteration in the polyhedral intermediate representation since the serial-elision version of the input program always has a total program execution order.

In general, SPMD parallel programs have partial program execution order and don't satisfy the *serial-elision* property in general (See Section 2.1.2). Also, there are no existing approaches to extract the partial execution order originating from barriers in the SPMD programs, and represent in the polyhedral model. Hence, we propose extensions to the polyhedral model in the next chapter to capture such partial execution orders and enable debugging and optimizations of SPMD programs.

Chapter 3

Extensions to the Polyhedral Model for SPMD Programs

Research is creating new knowledge.

Neil Armstrong

In the polyhedral model, schedules (defined in Section 2.3) are introduced to capture total execution orders present in the sequential programs. These total execution orders are extracted and expressed in a variety of ways including 2d+1-schedules [56], Schedule trees [69, 59] among others in the existing polyhedral frameworks. These schedules can also express some partial execution orders by assigning the same logical timestamp to multiple statement instances, thereby indicating that they can execute at the same time [63]. However, there are no existing approaches to extract the partial execution order originating from barriers in the SPMD programs, and express them onto schedules. In this thesis, we present novel extensions (i.e., space and phase mappings) to the polyhedral model to express such partial execution orders from SPMD programs. Also, these extensions are subsequently used (explained in further chapters) to enable both debugging and optimizations of SPMD programs.

3.1 Important Concepts in an SPMD Execution

There are two important concepts in an SPMD execution with barriers to extract partial order in the execution.

```
1 #pragma omp parallel
2 {
3         {S1;}

5     #pragma omp barrier // B1

7         {S2;}

9     #pragma omp barrier // B2

11    #pragma omp master
12        {S3;}
13 } // B3
```

Figure 3.1 : An example to motivate important concepts in an SPMD execution

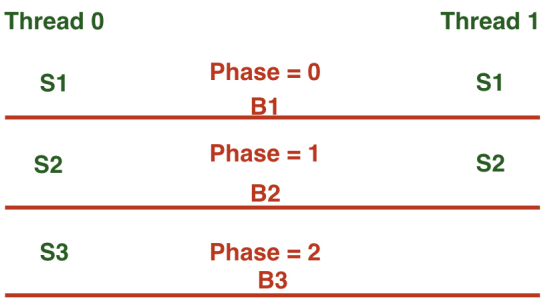


Figure 3.2 : Overall SPMD execution of the program in Figure 3.1 with two threads

The overall execution of the SPMD program in Figure 3.1 with two threads is shown in Figure 3.2. As can be seen from it, both threads (threads with id's 0 and 1) execute the same program with sequential code (S1, S2) redundantly, and parallel code (B1, B2, S3) cooperatively. Since the programmer annotated the statement S3 (line 12) with `omp master`, only master thread, i.e., thread with id 0, can execute the statement. Hence, the *thread mapping* information, i.e., which thread executes which

statements, is one of the important mappings required to capture the partial order in an SPMD program.

A key property of the SPMD programs is that their execution can be partitioned into a sequence of phases separated by the textually aligned barriers. The overall SPMD execution in Figure 3.2 can be seen as a sequence of three different partitions (*execution phases*) separated by the barriers B1, B2 and B3. It has been observed in past work that statements from different execution phases cannot execute concurrently [70], i.e., the statement S1 can never run in parallel with the statement S3. So, the *phase mapping* information, i.e., which statement is in which phase, is another important mapping, and together with *thread mapping* can help in capturing the partial orders in the SPMD programs.

3.2 Space Mapping

A space mapping (denoted by $\Theta^A(S(\vec{i_S}))$) is an affine function which assigns a logical processor ID to a statement instance ($\mathrm{S}(\vec{i_S})$) on which the instance has to be executed [71]. The space mapping can be viewed as a one-dimensional affine function that maps a statement instance onto a one-dimensional space of logical processors. For example, the space mappings of the statements in Figure 3.1 are as follows (Note that we replaced the parallel region with a logical loop that iterates over threads, and the loop induction variable is `tid`):

$$\Theta^A(S1(\vec{i_{S1}})) = tid\ ,\ where\ \vec{i_{S1}} = (tid)$$
$$\Theta^A(S1(\vec{i_{S2}})) = tid\ ,\ where\ \vec{i_{S2}} = (tid)$$
$$\Theta^A(S1(\vec{i_{S3}})) = 0\ ,\ where\ \vec{i_{S3}} = (tid)$$

In our tool `PolyOMP`, we recognize the SPMD-style parallelism using OpenMP constructs with the support for `omp parallel, omp for, omp parallel for, omp`

`barrier, omp single, omp master` directives and the nested parallel regions. To compute the space mappings in case of the above OpenMP constructs, we

1. Replace the `omp parallel` region header by a logical parallel loop that iterates over threads to model the entire SPMD execution with all threads,
2. Enclose the body of a statically scheduled worksharing loop in an `if` block with the condition on the thread iterator to be a function of lower, upper loop bounds, the loop chunk size if specified and total number of threads participating in the worksharing loop (the last two are treated as fixed but unknown program parameters),
3. Enclose the body of a non-statically scheduled worksharing loop or body of a `omp single` region in an `if` block with the condition on the thread iterator to be a function,
4. Enclose the body of a `omp master` region in an `if` block with the condition on the thread iterator to be zero,
5. Insert an explicit `barrier` immediately after a parallel region (or) a worksharing loop (or) a `single` region if a `nowait` clause is not specified.

Note that the transformations are only performed for the purpose of computing space mappings in an easier and cleaner fashion, and don't change the semantics of the original program. Also, note that the temporary function name (introduced as part of above transformations) for all the static scheduled worksharing loops is the same.

The SPMD program (shown in Figure 3.3) contain several OpenMP constructs of type worksharing (`omp parallel for, omp for, omp single`), and synchronization (`omp master, omp barrier`). This input SPMD program is transformed at AST level by applying the above transformations so that space mappings can be easily computed. For example, the `omp single` construct enclosing statement S3 (line 10) is

```
1 #pragma omp parallel num_threads(T1)
2 {
3     {S1;}

5     #pragma omp for
6     for(int i = 0; i < N; i++)
7         {S2;}

9     #pragma omp single
10        {S3;}

12    #pragma omp master
13        {S4;}

15    #pragma omp parallel for num_threads(T2)
16    for(int j = 0; j < N; j++)
17        {S5;}
18 }
```

Figure 3.3 : An OpenMP SPMD-style program with various directives

replaced with an `if` block with a condition on the thread iterator to be equal to a function f_2. An explicit `omp barrier` is also inserted after the `if` block since the `omp single` has a default enclosing barrier according to OpenMP specifications [53]. After the transformations, space mappings of the statements in Figure 3.3 are as follows:

$$\Theta^A(S1(\vec{i_{S1}})) = tid_1 \ , \ where \ \vec{i_{S1}} = (tid_1)$$
$$\Theta^A(S2(\vec{i_{S2}})) = f_1(i, 0, N, T_1) \ , \ where \ \vec{i_{S2}} = (tid_1, i)$$
$$\Theta^A(S3(\vec{i_{S3}})) = f_2(T_1) \ , \ where \ \vec{i_{S3}} = (tid_1)$$
$$\Theta^A(S4(\vec{i_{S4}})) = 0 \ , \ where \ \vec{i_{S4}} = (tid_1)$$
$$\Theta^A(S5(\vec{i_{S5}})) = f_1(j, 0, N, T_2) \ , \ where \ \vec{i_{S5}} = (tid_1, tid_2, j)$$

Consider space mapping $(\Theta^A(S2(\vec{i_{S2}})))$ of the statement instance S2$(\vec{i_{S2}})$. The

logical processor id that executes the statement instance (S2($\vec{i_{S2}}$)) is a function f_1's value over the variable i, and parameters N, T_1. In general, space mappings can consist of non-affine and unknown functions (in the case of S2, S3, and S5) particularly when the statement instance is surrounded by an OpenMP `single` construct or an OpenMP worksharing loop. When comparing space mappings of two statements, the non-affine mappings can create hard challenges for program analysis. Therefore, we conservatively compare only the name and arguments of these mappings to distinguish the non-affine space mappings of the statements [55].

3.3 Phase Mapping

In this thesis, we assume the barriers in SPMD programs to be textually aligned, which statically ensures that all threads of the SPMD region reach the same textual sequence of barriers inside the SPMD region [70]. A fundamental property of the SPMD programs with barriers is that their execution can be partitioned into a sequence of phases separated by the textually aligned barriers.

A phase mapping (denoted by $\Theta^P(S(\vec{i_S}))$) is a multi-dimensional affine function that assigns a logical identifier, which we refer to as a *phasestamp*, to each statement instance. The statement instances are executed according to the increasing lexicographic order of their phase timestamps, and similarly, the instances follow increasing lexicographic order of the schedule (defined in Section 2.3) within a given phasestamp. For example, phase mappings of the statements in Figure 3.1 are as follows (Note that we replaced the parallel region with a logical parallel loop that iterates over threads,

and the loop induction variable is `tid`):

$$\Theta^P(S1(\vec{i_{S1}})) = 0\ ,\ where\ \vec{i_{S1}} = (tid)$$
$$\Theta^P(S1(\vec{i_{S2}})) = 1\ ,\ where\ \vec{i_{S2}} = (tid)$$
$$\Theta^P(S1(\vec{i_{S3}})) = 2\ ,\ where\ \vec{i_{S3}} = (tid)$$

Also, phase mappings of the statements in Figure 3.3 are as follows:

$$\Theta^P(S1(\vec{i_{S1}})) = (0)\ ,\ where\ \vec{i_{S1}} = (tid_1)$$
$$\Theta^P(S2(\vec{i_{S2}})) = (0)\ ,\ where\ \vec{i_{S2}} = (tid_1, i)$$
$$\Theta^P(S3(\vec{i_{S3}})) = (1)\ ,\ where\ \vec{i_{S3}} = (tid_1)$$
$$\Theta^P(S4(\vec{i_{S4}})) = (2)\ ,\ where\ \vec{i_{S4}} = (tid_1)$$
$$\Theta^P(S5(\vec{i_{S5}})) = (2)\ ,\ where\ \vec{i_{S5}} = (tid_1, tid_2, j)$$

Definition 3.3.1. (Depth of a barrier) The depth of a barrier is defined as the number of sequential loops surrounding the barrier from the immediately enclosing SPMD region. For example, all the barriers present in Figures 3.1 and 3.3 have zero depth, i.e., these barriers are not surrounded by any sequential loop from its immediately enclosing SPMD region. Most relevant past work [72] in computing *phasestamps* were limited to barriers having zero depth.

The example SPMD program shown in Figure 3.4 contains two explicit barriers, i.e., one barrier (at line 10) with depth 2 and another barrier (at line 16) with depth 1. This pattern of barrier usage is common in accelerator programming where each thread proceeds in a lock step fashion [73]. The overall execution of the SPMD program in Figure 3.4 with two threads is shown in Figure 3.5. As can be seen from it, the statement instances S3(i = 0) and S1(i = 1, j = 0) are in same execution phase of computation, i.e., they are not separated by any barrier during the program

```
1 #pragma omp parallel
2 {
3     for(int i = 0; i < N; i++)
4     {
5         for(int j = 0; j < N; j++)
6         {
7             {S1;} //S1(i, j)

9             // Barrier B1(i, j) with depth 2
10            #pragma omp barrier

12            {S2;} //S2(i, j)
13        }

15        // Barrier B2(i) with depth 1
16        #pragma omp barrier

18        #pragma omp master
19            {S3;} // S3(i)
20    }
21 } // Implicit Barrier3 with depth 0
```

Figure 3.4 : An OpenMP SPMD program that includes barriers with depth > 0.

execution.

$$\Theta^P(S1(\vec{i_{S1}})) = \Theta^P(S3(\vec{i_{S3}})) \text{ for } \vec{i_{S1}} = (1,0) \; and \; \vec{i_{S3}} = (0)$$

In the rest of section, we propose a novel approach to compute phase mappings of the statements in SPMD programs including barriers with depth > 0.

Definition 3.3.2. (Barrier instance): Similar to the definition of a statement instance (in Section 2.3), each dynamic instance of a barrier `B` in an SPMD region is identified by its iteration vector ($\vec{i_B}$). For example, B1(i = 1, j = 1) in Figure 3.4 refers to the barrier `B1` (at line 10) when the values of loop iterators `i, j` are 1.

Figure 3.5 : Overall SPMD execution of the program in Figure 3.4 with two threads and value of N as 2

Definition 3.3.3. (Reachable barriers / Immediately succeeding barriers): Reachable barriers (or) Immediately succeeding barriers of a statement instance (denoted by $\mathcal{RB}(S(\vec{i_S}))$) is defined as the set of barrier instances that can be executed after the statement instance $(S(\vec{i_S}))$ without an intervening barrier instance. For example, a reachable barrier/ immediately succeeding barrier for the statement instance (S1(i = 1, j = 0)) from Figure 3.5 is B1(i = 1, j= 0). Symbolically, reachable barriers for a statement instance $(S2(\vec{i_{S2}} = (i, j)))$ in Figure 3.4 include the barrier instance $(B1(\vec{i_{B1}}))$ in the next iteration of `j-loop`, and another barrier instance $(B2(\vec{i_{B2}}))$ in the same iteration of `i-loop`. These reachable barriers are shown below:

$$\mathcal{RB}(S2(\vec{i_{S2}})) = \{\ B1(\vec{i_{B1}})\ \mid\ i = i'\ \wedge\ j = j' - 1\ ,$$
$$B2(\vec{i_{B2}})\ \mid\ i = i''\ \wedge\ j = N - 1\ \}$$
$$where\ \vec{i_{S2}} = (i, j),\ \vec{i_{B1}} = (i', j'),\ and\ \vec{i_{B2}} = (i'')$$

During the execution, there exists only one reachable barrier for a given dynamic statement instance under the assumption of textually aligned barriers, and it would

be one (based on the program parameters, for example, N in Figure 3.4) from the statically determined set of reachable barriers.

Observation: Two statement instances are in same execution phase if and only if they have same set of reachable barrier instances. For example, the statement instances S3(i = 0) and S1(i = 1, j = 0) are in same execution phase of computation since they have same reachable barrier/ immediately succeeding barrier instance, i.e., B1(i = 1, j = 0).

Definition 3.3.4. (Phase mapping): The phase mapping of a statement instance (denoted by $\Theta^P(S(\vec{i_S}))$) is computed as the *union* (collection) of the schedules (timestamps, defined in Section 2.3) of reachable barriers of the statement instances.

$$\Theta^P(S(\vec{i_S})) = \Theta^S(\mathcal{RB}(S(\vec{i_S})))$$

Algorithm 1 summarizes the overall approach to compute the phase mappings of the statements by taking regular statements and barriers schedules as an input (at lines 2-3). For example, the 2d+1-schedules of regular statements and barriers in Figure 3.4 are as follows:

$$\Theta^S(S1(\vec{i_{S1}}) = (0,i,0,j,0),\ \Theta^S(S2(\vec{i_{S2}})) = (0,i,0,j,2),\ \Theta^S(S3(\vec{i_{S3}})) = (0,i,1,0,0)$$
$$\Theta^S(B1(\vec{i_{B1}})) = (0,i,0,j,1),\ \Theta^S(B2(\vec{i_{B2}})) = (0,i,1,0,0),\ \Theta^S(B3) = (1,0,0,0,0)$$

Then, reachable barriers are computed by identifying the lexicographically closest barrier instances to each regular statement instance (at lines 4-8).

$$\mathcal{RB}(S1(\vec{i_{S1}})) = \{\ B1(\vec{i_{B1}})\ \mid\ i = i'\ \wedge\ j = j'\ \}$$
$$where\ \vec{i_{S1}} = (i,j),\ and\ \vec{i_{B1}} = (i',j')$$

Algorithm 1: Building phase mappings of statements

Input : Regular statements (S) and barriers (B)

1 **begin**

/* Extract original program schedules (time stamps, defined in Section 2.3) */

2 $\Theta^S(S)$:= Schedules of the regular statements

3 $\Theta^S(B)$:= Schedules of the barriers

/* Build a map from the regular statements to the barriers such that the statements are lexicographically strictly smaller than those of barriers */

4 $\Delta^{S \to B} := \{\vec{x} \to \vec{y} : \Theta^S(\vec{x}) \prec \Theta^S(\vec{y}), \vec{x} \in S, \vec{y} \in B\}$

/* Build another map from the time stamps of the regular statements to the time stamps of the barriers that must precede it */

5 $\Delta^{\Theta^S(S) \to \Theta^S(B)} := (\Theta^S(S))^{-1} \circ \Delta^{S \to B} \circ \Theta^S(B)$

/* Extract a map from pairs of statement and barrier timestamps to their time difference */

6 $\Delta^{(S,B) \to (\Theta^S(B) - \Theta^S(S))} :=$
$\{(\Theta^S(\vec{x}) \to \Theta^S(\vec{y})) \to (\Theta^S(\vec{y}) - \Theta^S(\vec{x})) : \vec{x} \in S, \vec{y} \in B\}$

/* Build another map from each statement time stamp to the time stamp of the immediately succeeding barriers i.e., reachable barriers */

7 $\beta^{\Theta^S(S) \to \Theta^S(B)} := dom(lexmin(\Delta^{(S,B) \to (\Theta^S(B) - \Theta^S(S))}))$

/* Build a map from each statement instance to the immediately succeeding barriers, i.e., reachable barriers */

8 $\beta^{S \to B} := lexmin(\Theta^S(S) \circ \beta^{\Theta^S(S) \to \Theta^S(B)}) \circ (\Theta^S(B))^{-1}$

/* Compute phase mappings of a statement instance by *union* of timestamps of the reachable barriers of the statement instance */

9 Phase mappings, $\Theta^P := \beta^{S \to B} \circ \theta^B$

10 **end**

$$\begin{aligned}\mathcal{RB}(S2(\vec{i_{S2}})) = \{\ & B1(\vec{i_{B1}}) \ \mid\ i = i' \ \wedge\ j = j' - 1\ , \\ & B2(\vec{i_{B2}}) \ \mid\ i = i'' \ \wedge\ j = N - 1\ \} \\ & where\ \vec{i_{S2}} = (i, j),\ \vec{i_{B1}} = (i', j'),\ and\ \vec{i_{B2}} = (i'')\end{aligned}$$

$$\begin{aligned}\mathcal{RB}(S3(\vec{i_{S3}})) = \{\ & B1(\vec{i_{B1}}) \ \mid\ i = i' - 1 \ \wedge\ j' = 0\ , \\ & B3 \ \mid\ i = N - 1\ \} \\ & where\ \vec{i_{S3}} = (i, j),\ and\ \vec{i_{B1}} = (i', j')\end{aligned}$$

Finally, phase mappings of each statement instances are obtained by `union` of the schedules (timestamps) of reachable barriers of the statement instance (at line 9).

$$\begin{aligned}\Theta^P(S1(\vec{i_{S1}})) &= \Theta^S(\mathcal{RB}(S1(\vec{i_{S1}}))) \\ &= \{\Theta^S(B1(\vec{i_{B1}})) \ \mid\ i = i' \ \wedge\ j = j'\ where\ \vec{i_{S1}} = (i, j)\ and\ \vec{i_{B1}} = (i', j')\} \\ &= \{(0, i', 0, j', 1) \ \mid\ i = i' \ \wedge\ j = j'\ \} \\ &= \{(0, i, 0, j, 1)\ \}\end{aligned}$$

$$\begin{aligned}\Theta^P(S2(\vec{i_{S2}})) &= \Theta^S(\mathcal{RB}(S2(\vec{i_{S2}}))) \\ &= \{\Theta^S(B1(\vec{i_{B1}})) \ \mid\ i = i' \ \wedge\ j = j' - 1\ where\ \vec{i_{S2}} = (i, j)\ and\ \vec{i_{B1}} = (i', j'), \\ &\qquad \Theta^S(B2(\vec{i_{B2}})) \ \mid\ i = i'' \ \wedge\ j = N - 1\ where\ \vec{i_{S2}} = (i, j)\ and\ \vec{i_{B2}} = (i'')\} \\ &= \{(0, i', 0, j', 1) \ \mid\ i = i' \ \wedge\ j = j' - 1\ , \\ &\qquad (0, i'', 1) \ \mid\ i = i'' \ \wedge\ j = N - 1\ \} \\ &= \{(0, i, 0, j + 1, 1) \ \mid\ j + 1 < N\ , \\ &\qquad (0, i, 1) \ \mid\ j = N - 1\}\end{aligned}$$

$$
\begin{aligned}
\Theta^P(S3(\vec{i_{S3}})) &= \Theta^S(\mathcal{RB}(S3(\vec{i_{S3}}))) \\
&= \{\Theta^S(B1(\vec{i_{B1}})) \mid i+1 = i' \wedge j' = 0 \; where \; \vec{i_{S3}} = (i) \; and \; \vec{i_{B1}} = (i', j'), \\
&\quad \Theta^S(B3) \mid i = N-1 \; where \; \vec{i_{S3}} = (i) \; \} \\
&= \{(0, i', 0, j', 1) \mid i+1 = i' \wedge j' = 0, \\
&\quad (1,0,0,0,0) \mid i = N-1\} \\
&= \{(0, i+1, 0, 0, 1) \mid i+1 < N \; , \\
&\quad (1,0,0,0,0) \mid i = N-1\}
\end{aligned}
$$

In general, the partial execution order of parallel programs are expressed either through Happens-before (HB) relations or May-happen-in-parallel (MHP) relations. In this thesis, after computing both space and phase mappings of all statement instances in an SPMD program, we construct partial execution order of the SPMD program in the form of MHP relations.

3.4 May-Happen-in-Parallel (MHP) Analysis

Parallel programming languages offer many high-level parallel constructs for parallelism and synchronization. All these parallel constructs indicate the relative progress and interactions of logical threads during execution. Furthermore, these interactions among threads can impact the possible execution order of statements. For example, statements before and after a `barrier` are ordered within an SPMD region, as they cannot execute simultaneously. Knowledge of these possible orderings can be very helpful when debugging parallel programs. "*May-Happen-in-Parallel (MHP) analysis determines if it is possible for execution instances of two statements (or the same statement) to run in parallel*" [43]. The MHP can be reformulated as follows with our extensions to the polyhedral representation.

Definition 3.4.1. May-Happen-in-Parallel : Two statement instances S($\vec{i_S}$) and T($\vec{i_T}$) can run in parallel if and only if both the instances are in the same execution phase (based on `barriers`) and are executed by two different logical threads of the SPMD region.

$$\begin{aligned} \text{MHP}(S(\vec{i_S}), T(\vec{i_T})) \ = True \ \Leftrightarrow (\ \Theta^P(S(\vec{i_S})) = \Theta^P(T(\vec{i_T}))) \\ \wedge \ (\Theta^A(S(\vec{i_S})) \neq \Theta^A(T(\vec{i_T})) \ ) \end{aligned} \tag{3.1}$$

For example, any instance of the statement S2 would never execute in parallel with any instance of the statement S3 in Figure 3.4 because they are always either separated by `barrier` B2 (at line 14) or `barrier` B1 (at line 7).

The MHP condition in (3.1) appears quite simple because MHP contains less information than the happens-before (HB) information. If MHP(S($\vec{i_S}$), T($\vec{i_T}$)) is true, then we know that HB(S($\vec{i_S}$), T($\vec{i_T}$)) and HB(T($\vec{i_T}$), S($\vec{i_S}$) must both be false.

$$\begin{aligned} MHP(S(\vec{i_S}), T(\vec{i_T})) \ = True \implies (HB(S(\vec{i_S}), T(\vec{i_T})) = False \ ) \\ \wedge \ (HB(T(\vec{i_T}), S(\vec{i_S})) = False \ ) \end{aligned} \tag{3.2}$$

However, if MHP(S($\vec{i_S}$), T($\vec{i_T}$)) is false, then we know either of HB(S($\vec{i_S}$), T($\vec{i_T}$)) or HB(T($\vec{i_T}$), S($\vec{i_S}$)) must be true and the other false, but there is insufficient information in MHP(S($\vec{i_S}$), T($\vec{i_T}$)) to indicate which of the two disjuncts evaluates to true and

which to false.

$$
\begin{aligned}
MHP(S(\vec{i_S}), T(\vec{i_T})) \; = False \implies & (\, (HB(S(\vec{i_S}), T(\vec{i_T})) = True \,) \\
& \wedge (HB(T(\vec{i_T}), S(\vec{i_S})) = False \,)) \\
\cup \; & ((HB(S(\vec{i_S}), T(\vec{i_T})) = False \,) \\
& \wedge (HB(T(\vec{i_T}), S(\vec{i_S})) = True \,) \,) \qquad (3.3)
\end{aligned}
$$

Algorithm 2 summarizes the overall steps to build the MHP information on a given pair of statements S and T. Lines 2-3 of the algorithm extract the space and phase mappings of both statements S and T. Line 4 builds a map from each point in the iteration domain of S to each point in the iteration domain of T such that their phase mappings are the same. Likewise, line 5 computes a map such their space mappings are same. To compute a map (line 7) such that thread mapping of S and T are different, we subtract the cross product of thread maps (line 5) from the map having same thread mappings (line 6)(e.g. the identity mappings). Lastly, MHP information (line 8) between a pair of statements S and T are obtained by intersecting the maps with the same phase from line 4 and the maps with different space from line 7.

Finally, the program execution order in an SPMD program is captured through MHP relations from the combination of Space mappings (θ^A), Phase mappings (θ^P), and Schedule (time) mappings (θ) in the polyhedral model.

Algorithm 2: Building May-Happen-in-Parallel (MHP) information between statements S and T.

Input : Regular statements S and T

begin

```
/* Extract space and phase mappings of statements S and T    */
```

$\Theta^A(S(\vec{i_S})), \Theta^A(T(\vec{i_T}))$:= Space mappings of S and T

$\Theta^P(S(\vec{i_S})), \Theta^P(T(\vec{i_T}))$:= Phase mappings of S and T

```
/* Build a map from S to T such that an element from S is
   mapped to another element in T with same phase mappings   */
```

$\Delta^{S \to T}_{SamePhase} := \Theta^P(S(\vec{i_S})) \circ (\Theta^P(T(\vec{i_T})))^{-1}$

```
/* Build a map from S to T such that an element from S is
   mapped to another element in T with same space mappings   */
```

$\Delta^{S \to T}_{SameSpace} := \Theta^A(S(\vec{i_S})) \circ (\Theta^A(T(\vec{i_T})))^{-1}$

```
/* Compute cross product of S and T                          */
```

$\Delta^{S \to T}_{CrossProduct} := dom(\Theta^A(S(\vec{i_S}))) \times dom(\Theta^A(T(\vec{i_T})))$

```
/* Build another map from S to T such that an element from S
   is mapped to another element in T with different space
   mappings                                                  */
```

$\Delta^{S \to T}_{NotSameSpace} := \Delta^{S \to T}_{CrossProduct}$ - $\Delta^{S \to T}_{SameSpace}$

```
/* Build MHP information by intersecting the map with same
   phase mappings and another map with different space
   mappings                                                  */
```

$\Delta^{S \to T}_{MHP} := \Delta^{S \to T}_{NotSameSpace} \cap \delta^{S \to T}_{SamePhase}$

end

3.5 Past Work in Extending Polyhedral Model for Explicitly-Parallel Programs

In the last few years, a significant interest ([41, 74, 75, 76, 77, 78]) from the polyhedral research community has started to address the challenges in using the polyhedral

model to analyze and optimize explicitly-parallel programs.

Firstly, Yuki et al. [75] started with addressing the problem of data-flow analysis of explicitly-parallel programs using the polyhedral model. It included an adaptation of array data-flow analysis to X10 programs with finish/async parallelism [75] and extended some support to clocks [77]. In this approach, the happens-before (HB) relations are first analyzed, and then the data-flow is computed based on the partial order imposed by happen-before relations. Their work [77] also extended happens-before relations to X10 clocks and proved that comparing two statement instances with the extended happens-before relations to be undecidable. But, our extensions to the polyhedral model focus on the partial orders arising from the textually aligned barriers (subset of X10 clocks) in an SPMD program, and comparing two statement instances with the MHP relations from the barriers turned out to be decidable.

Secondly, our prior works [41, 68], addressed the problem of analyzing and transforming programs with explicit parallelism (doall, task parallelism in OpenMP 4.0, and doacross parallelism in > OpenMP 5.0) that satisfy the *serial-elision* property. The work starts by enabling a conservative dependence analysis of a given region of code, which may contain non-affine constructs. Next, it identifies happens-before relations from the explicitly-parallel constructs, such as tasks and parallel loops, and intersects them with the conservative dependences. Finally, the resulting set of dependences is passed on to a polyhedral optimizer, such as PLuTo [22, 23] Or PolyAST [24], to enable the transformation of explicitly-parallel programs with unanalyzable data accesses. However, the approach in [41, 68] does not apply to general SPMD parallel programs with barriers because they don't satisfy *serial-elision* property in general and the approach doesn't consider barriers in the analysis.

Thirdly, PENCIL [79], a platform-neutral compute intermediate language, aimed at facilitating automatic parallelization and optimization on multi-threaded SIMD hardware for domain specific languages. The language allows users to supply informa-

tion about dependences and memory access patterns to enable better optimizations. PENCIL provides directives such as *independent, reductions* to remove data dependences on the loop, but doesn't have support to enable analysis for *barriers*.

Lastly, Pop and Cohen have presented a preliminary approach to increase optimization opportunities for parallel programs by extracting the semantics of the parallel annotations [80]. This extracted information is brought into compiler's intermediate representation and leverage existing polyhedral frameworks for optimizations. They also planned to consider streaming OpenMP extensions carrying explicit dependence information, to enhance the accuracy of data dependence analyses.

Chapter 4

PolyOMP: A Polyhedral Framework for Debugging and Optimizations of SPMD Programs

High achievement always takes place in the framework of high expectation.

Charles Kettering

In this chapter, we introduce `PolyOMP`, a framework extending the polyhedral model to enable analysis for debugging and optimization of SPMD programs which are expressed through OpenMP. The summary of the `PolyOMP` framework is shown in Figure 4.1.

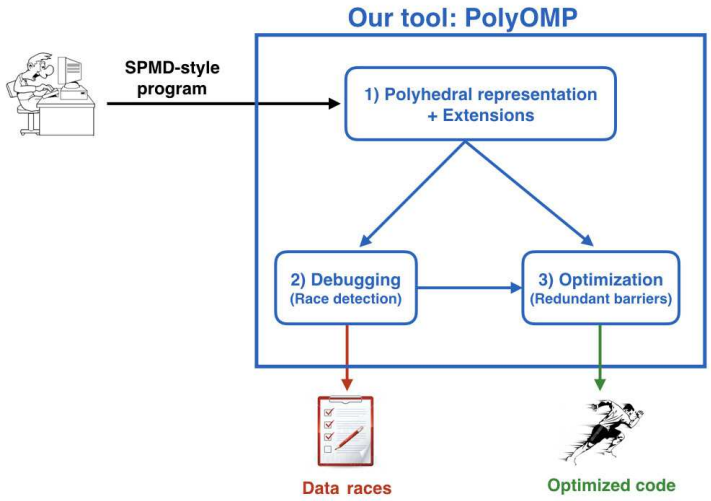


Figure 4.1 : Summary of the `PolyOMP`, a polyhedral framework for debugging and optimizations of SPMD programs

On a high-level, `PolyOMP` extracts partial execution orders from SPMD programs, and represents them in the polyhedral model as MHP relations (explained in Chapter 3). Then, `PolyOMP` uses these MHP relations to enable debugging (i.e., static data race detection in Chapter 5) and optimizations (i.e., static redundant barrier removal in Chapter 6) of the SPMD programs.

4.1 Overall Workflow

In this section, we briefly explain overall workflow of the `PolyOMP`, which is implemented as an extension to the Polyhedral Extraction Tool (PET, version: pet-0.08-30-g77689da) [2], and consists of the following components (See Figure 4.2):

1. `Clang OMP Parser` – Conversion from input OpenMP-C program (with support for `omp parallel, for, parallel for, barrier, single, master` directives and nested parallel regions) to Clang AST with the help of Clang-omp (version: 3.5) [81] and LLVM [82](version: 3.5.svn)

2. `PET AST Builder` – Conversion from Clang AST to PET AST (defined in [2])

3. `Polyhedral SCoP Extractor` – Extract components of the polyhedral representation (See Section 2.3) such as iteration domain, access relations, and schedules of the statements from the PET AST.

   The first three components, i.e., `Clang OMP Parser, PET AST Builder,` and `Polyhedral SCoP Extractor` are part of the Polyhedral Extraction Tool (PET, version: pet-0.08-30-g77689da).

4. `Space Mapping Builder` – Build space mappings of the statements from the PET AST (See Section 3.2).

5. `Phase Mapping Builder` – Build phase mappings of the statements from the components of polyhedral representation especially schedules of both regular

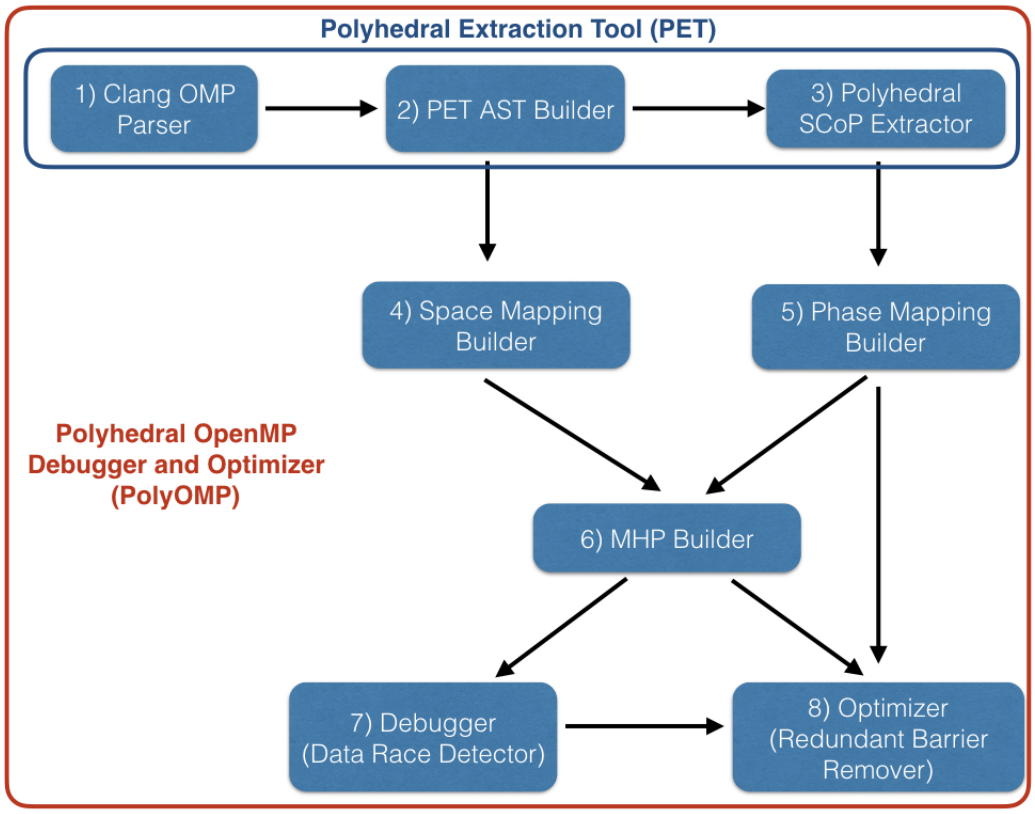


Figure 4.2 : Overview of the `PolyOMP` system built on top of the Polyhedral Extraction Tool (`PET`, version: pet-0.08-30-g77689da) [2].

statements and barriers (See Section 3.3).

6. `MHP Builder` – Build May-happen-in-parallel (MHP) relations from the space and phase mappings of the statements (See Section 3.4).

7. `Debugger - Data Race Detector` – Identify data races present among the statements at compile-time with the help of the MHP relations (See Chapter 5).

8. `Optimizer - Redundant Barrier Remover` – Identify and remove redundant barriers in the input program at compile-time by using the race detector and the phase mapping builder (See Chapter 6).

In the rest of this thesis, we discuss how the proposed extensions and the MHP relations can be used to enable debugging (i.e., static data race detection in Chapter 5) and optimizations (i.e., static redundant barrier removal in Chapter 6) of the SPMD programs.

Chapter 5

Debugging Of SPMD Programs – Static Data Race Detection

Debugging is twice as hard as writing the code in the first place.

Brian Kernighan

Data races are the dominant cause of semantic errors in multi-threaded programs. A data race happens when two or more logically parallel threads perform conflicting accesses (such that at least one access is a write) to a shared memory location without any synchronization. Complicating the matter, data races may occur for a certain input or may happen only in certain executions of a parallel program, thereby making the races notoriously hard to detect and reproduce. Hence, data race detection remains a challenging and hard problem, nevertheless the significant progress on the restricted subsets of fork-join and SPMD programs [83, 84, 72], as well as for higher-level programming models [85, 75, 86, 87]. In this chapter, we propose and evaluate an approach to identify data races in SPMD programs at compile-time with our extensions (introduced in Chapter 3) to the polyhedral intermediate representation.

5.1 Motivation

To motivate our approach for detection of data races at compile-time, we use an SPMD program (`Jacobi03` benchmark from the `OmpSCR` benchmark suite [88]) as

an illustrative example. The excerpt shown in Figure 5.1 is a 2-dimensional `Jacobi`

```
1 #pragma omp parallel private(resid, i)
2 {
3     while (k <= maxit && error > tol) { //S1
4     /* copy new solution into old */
5 #pragma omp for
6       for (j=0; j<m; j++)
7         for (i=0; i<n; i++)
8           uold[i + m*j] = u[i + m*j];
9
10        /* compute stencil, residual and update */
11 #pragma omp for reduction(+:error)
12      for (j=1; j<m-1; j++) {
13        for (i=1; i<n-1; i++) {
14          resid =(ax*(uold[i-1+m*j] + uold[i+1+m*j])
15                  + ay*(uold[i+m*(j-1)] + uold[i+m*(j+1)])
16                  + b*uold[i+m*j] - f[i+m*j]) / b;
17
18          /* update solution */
19          u[i + m*j] = uold[i + m*j] - omega * resid;
20
21          /* accumulate residual error */
22          error =error + resid*resid;
23        }
24      }
25
26      /* error check */
27 #pragma omp master
28      {
29        k++; //S2
30        error = sqrt(error) /(n*m); //S3
31      }
32    } /* while */
33 } /* end parallel */
```

Figure 5.1 : Data races in the Jacobi benchmark from `OmpSCR` benchmark suite

stencil computation from the `OmpSCR` benchmark suite [88]. The computation is parallelized using OpenMP `parallel` construct with worksharing directives (at lines 5, 11) and synchronization directives (implicit barriers from worksharing loops at lines

5, 11). The first for-loop is parallelized (at line 5) to produce values of the array `uold`. Likewise, the second for-loop is parallelized (at line 11) to consume values of the array `uold`. The `reduced error` (from the reduction clause at line 11) is updated by only the `master` thread in the region (lines 28-31). Finally, the entire computation in lines 5–31 is repeated until it reaches the maximum number of iterations (or) the error is less than a threshold value. This pattern is very common in many stencil programs, often with multidimensional loops and multidimensional arrays [89]. Although the worksharing parallel loops have implicit barriers, the programmer who contributed this code to the `OmpSCR` suite likely overlooked the fact that a `master` region does not include a barrier. As a result, data races are possible in this example since statement S1's (at line 3) read access of variables `k, error` by a non-master thread can execute in parallel with an update of the same variables performed in statements S2 (at line 29) and S3 (at line 30) by the master thread. These races can be fixed by inserting another barrier immediately after the `master` region or converting it to a single region.

We observe that existing static race detection tools (e.g., [84, 85]) are unable to identify such races since they don't model barriers inside of imperfectly nested sequential loops in the SPMD regions. We also observe that existing dynamic race detection tools such as Intel Inspector XE (2015 Update 1) in its `default` mode miss this true race [90] and hybrid race detection tools such as ARCHER incurred significant runtime overhead to detect this true race [87]. Furthermore, these techniques are also known to be input dependent and only guaranteed for a given input. In contrast, our proposed approach using the extended polyhedral model can identify such races at compile-time by effectively capturing execution phases from `barrier` directives via static analysis of SPMD regions.

5.2 Our Approach

In this section, we begin by computing May-Happen-in-Parallel (MHP) relations with our extensions (introduced in Chapter 3) to the polyhedral model. Subsequently, we explain our approach to identify data races using the MHP relations.

5.2.1 An Algorithm to Identify Data Races

Detecting read-write and write-write data races become straightforward with the availability of MHP information. A data race happens when two or more logically parallel threads perform conflicting accesses (such that at least one access is a write) to a shared memory location without any synchronization. The race condition can be formulated as follows with the MHP information and access relations (defined in Section 2.3).

Definition 5.2.1. Race condition: A race exists between statement instances $\mathrm{S}(\vec{i_S})$ and $\mathrm{T}(\vec{i_T})$ on a memory location if and only if $\mathrm{MHP}(\mathrm{S}(\vec{i_S}), \mathrm{T}(\vec{i_T}))$ is true, and access relations of $\mathrm{S}(\vec{i_S})$ and $\mathrm{T}(\vec{i_T})$ have the same memory location in common and at-least one of them is a write relation.

$$\begin{aligned} Race(S(\vec{i_S}), T(\vec{i_T})) \; = True \; \Leftrightarrow (\; & (\mathrm{MHP}(S(\vec{i_S}), T(\vec{i_T})) \; = True) \\ & \wedge \; (\mathcal{A}(S(\vec{i_S})) = \mathcal{A}(T(\vec{i_T})) \;) \;) \end{aligned} \tag{5.1}$$

For example, data races are possible in Figure 5.1 since statement S1's (at line 3) read access of variables `k, error` by a non-master thread can execute in parallel with an update of the same variables performed in statements S2 (at line 29) and S3 (at line 30) by the master thread.

The major idea in our approach (shown in Algorithm 3) is to construct race constraints for all possible pairs of the regular statements (excluding barriers) and

solve for the existence of solutions. Algorithm 3 identifies read-write and write-write

Algorithm 3: An approach to compute a set of data races in an SPMD program

begin

/* Extract all regular statements (excluding barriers) */

S := Set of all regular statements

/* Build MHP information for every pair of statements */

$\Delta_{MHP} := \bigcup_{S_i \in S} \bigcup_{S_j \in S} \text{MHP}(S_i, S_j)$

/* Union of read access relations from all statements */

$\Delta_R := \bigcup_{S_i \in S} \bigcup_{j=1}^{|Reads| \text{ in } S_i} \mathcal{A}_j^{Read}(S_i)$

/* Union of write (including may writes) access relations from all statements */

$\Delta_W := \bigcup_{S_i \in S} \bigcup_{j=1}^{|Writes| \text{ in } S_i} \mathcal{A}_j^{Write}(S_i)$

/* Build a map from read access relations to write access relations such that they access same memory location */

$\Delta_{SameLocation}^{R \to W} := \Delta_R \circ (\Delta_W)^{-1}$

/* Build a map from write access relations to write access relations such that they access same memory location */

$\Delta_{SameLocation}^{W \to W} := \Delta_W \circ (\Delta_W)^{-1}$

/* Build Read-Write races by intersecting the MHP relations and RW maps */

$\Delta_{RWRaces} := \Delta_{MHP} \cap \Delta_{SameLocation}^{R \to W}$

/* Build Write-Write races by intersecting the MHP relations and WW maps */

$\Delta_{WWRaces} := \Delta_{MHP} \cap \Delta_{SameLocation}^{W \to W}$

end

data races in the SPMD programs by beginning with computing MHP information (at line 3) for every pair of regular statements (excluding barriers) with Algorithm 2. Then, the algorithm aggregates all possible reads, writes (including may-writes arising from `unanalyzable` data accesses and control flow) present in the regular statements (lines 4-5). Thanks to the PET framework [2] for handling non-affine constructs (in both data subscripts and control flow) elegantly in the form of may-write access relations. Next, the algorithm identifies pairs of read and write access relations that touch the same memory location (at line 6), and likewise, it also computes pairs of write and write access relations that access same memory cell (at line 7). Finally, the MHP relations and read-write relations are intersected to compute read-write races (at line 8). Similarly, the MHP relations are also intersected with write-write relations to compute write-write races (at line 9).

5.3 Experimental Evaluation

In this section, we evaluate our approach for race detection at compile-time using the extensions to the polyhedral model. Firstly, we briefly describe our experimental setup and benchmark suites used for the evaluation. Then, we present our discussion on the obtained results for each of the benchmark suites.

5.3.1 Experimental Setup

Since the race detection part of our tool `PolyOMP` is developed to help OpenMP programmers in debugging, the experiments have been performed on a local development machine having quad cores (each core is a core-i7 with 2.2GHz clock frequency) with 16 GB of main memory. In the evaluation, we compare the following three race detection tools: 1) ARCHER*, a recently developed race detection tool employing both

* We had challenges in installing ARCHER on our local machine. Hence, we compared with ARCHER only on the `OmpSCR` suite, for which ARCHER published races in [87].

static and dynamic analysis [87], 2) Intel Inspector XE, a dynamic memory and threading error checking tool from Intel [90], 3) Our tool PolyOMP using the proposed race detection approach in Algorithm 3.

5.3.2 OpenMP Source Code Repository

`OmpSCR`, an OpenMP Source Code Repository [88], consists of OpenMP applications written in C, C++ and Fortran. This repository includes a wide spectrum of applications including stencils, LU decomposition, molecular dynamics, FFT, pi computation, quick sort among others. There are 18 OpenMP-C benchmarks in this repository, 6 of which use `C structs` and pointer arithmetic. Since we defer support for `C structs` and pointer arithmetic in our current toolchain for future work, our results focus on the remaining 12 OpenMP-C benchmarks in `OmpSCR`, which are listed in Table 5.1.

Discussion. This benchmark suite contains known races, as reported in prior work on hybrid data race detection in the ARCHER tool [87]. Our evaluation shows that `PolyOMP` is able to detect all of the documented races in the following applications: `Jacobi03, LoopA.bad, LoopB.bad1, LoopB.bad2`. All reported races (column `Reported`) were manually verified. (Note: each reported data race corresponds to a static pair of conflicting accesses). The `False +ves` column shows the number of reported races that actually are false positives. In addition, we compared our reported races with those reported by the ARCHER[†]. As mentioned in [87], the data race in `Jacobi03` benchmark highly influenced the execution time of the benchmark, varying it by a factor of 1000 from run to run. In contrast, our tool `PolyOMP` is able to detect the races present in `Jacobi03` benchmark in less than two seconds during the compilation time.

[†]ARCHER is known to not have any false positives or false negatives for a given input, but may have false negatives for inputs that it has not seen.

Benchmark	ARCHER (Static + Dynamic) Reported races	Intel Inspector XE (Dynamic) Reported races	PolyOMP (Static)		
			Reported races	False +ve races	Detection time (seconds)
Jacobi01	0	0	2	2	1.38
Jacobi02	0	0	2	2	3.91
Jacobi03	2	0	4	2	1.54
Lud	0	1	0	0	0.30
LoopA.bad	1	2	1	0	0.20
LoopA.sol1	0	2	0	0	0.44
LoopA.sol2	0	0	7	7	1.21
LoopA.sol3	0	0	7	7	1.19
LoopB.bad1	1	2	1	0	0.20
LoopB.bad2	1	2	1	0	0.21
LoopB.pipe	0	0	7	7	2.40
C_pi	0	0	0	0	0.05
Total (12)	**5**	**9**	**30**	**25**	**13.03 (seconds)**

Table 5.1 : Race detection analysis over the subset of `OmpSCR` benchmark suite. `PolyOMP` - Detection time / Reported / False +ves : Total time taken to detect races by PolyOMP, Number of reported races, Number of false positives among reported. ARCHER / Intel Inspector XE: Number of races reported.

Even though Intel Inspector XE (2015 update 1 with `default` mode) was able to identify the true races in `LoopA.bad, LoopB.bad1` and `LoopB.bad2`, it failed to detect the races in `Jacobi03` (explained in Section 5.1) even after multiple runs. Furthermore, it reported additional false races (according to OpenMP specifications) on the iterators of parallel loops for benchmarks `Lud, LoopA.bad, LoopA.sol1, LoopB.bad1, LoopB.bad2` and `C_pi`.

Our tool `PolyOMP` computes races conservatively when unanalyzable control flow or data accesses are present and result in false positive races.

This is evident in benchmarks `Jacobi01, Jacobi02, Jacobi03, LoopA.sol2, LoopA.sol3` and `LoopB.pipe` since they contain linearized array subscripts, thereby yielding 27 false positives which could have been avoided with a delinearization pass before detecting races. However, when the parallel region fully satisfies all the assumptions of standard polyhedral frameworks (e.g., all array accesses and branch conditions must be affine functions of the loop variables, and as well as no known relations between parameters) then all reported races are true races.

5.3.3 PolyBench/ACC OpenMP Suite

We also use PolyBench/ACC OpenMP suite [91], another benchmark suite partially derived from the standard PolyBench benchmark suite [89]. This suite consists of benchmark codes for linear algebra, linear algebra solvers, data-mining, and stencils, all with static control parts. There are 32 OpenMP-C benchmarks in this suite, for which we were unable to compile ten benchmarks due to incorrect usage of OpenMP directives in those codes. This benchmark suite is relatively new and is perhaps still in development compared to other benchmark suites. Thus, our results focus on the remaining 22 OpenMP-C benchmarks in PolyBench/ACC. We had challenges in installing ARCHER on our local machine. Hence, we compared results of our tool `PolyOMP` only with Intel Inspector XE for this benchmark suite.

Discussion. All of the benchmarks in this suite have statically analyzable control flow, affine subscripts and completely fit the assumptions of the polyhedral model without any conservative estimates. We manually verified the reported races and found the races to be real. Moreover, our static analysis does not need to resort to conservative estimations for these benchmarks, as they meet all the standard affine requirements. It also verifies our claim that our approach is guaranteed to be exact (with neither false positives nor false negatives) if the input program satisfies all the standard preconditions of the polyhedral model (without any non-affine constructs,

Benchmark	Intel Inspector XE (Dynamic) Reported races	PolyOMP (Static)		
		Reported races	False +ve races	Detection time (seconds)
Correlation	H	0	0	2.30
Covariance	H	0	0	1.04
2mm	0	0	0	0.64
3mm	0	0	0	1.13
Atax	2	2	0	0.37
Bicg	2	2	0	0.43
Cholesky	8	28	0	0.49
Doitgen	0	0	0	0.54
Gemm	0	0	0	0.34
Gemver	0	0	0	0.75
Gesummv	0	0	0	0.52
Mvt	0	0	0	0.32
Symm	5	5	0	0.64
Syrk	0	0	0	0.39
Syr2k	0	0	0	0.52
Trmm	1	1	0	0.28
Durbin	0	6	0	0.73
Gramschmidt	8	12	0	0.36
Lu	5	5	0	0.33
Convolution-2	0	0	0	0.25
Convolution-3	A	0	0	0.42
Fdtd-ampl	0	0	0	1.62
Total (22)	**31**	**61**	**0**	**14.41 (seconds)**

Table 5.2 : Race detection analysis over the subset of PolyBench/ACC OpenMP benchmark suite. `PolyOMP` - Detection time / Reported / False +ves : Total time taken to detect races by PolyOMP, Number of reported races, Number of false positives among reported. Intel Inspector XE: Number of races reported, Hang up (H) and Application exception (A).

and aware of any known relations between parameters).

Currently, we are not aware of any prior work reporting data races in this benchmark suite. Hence, we compared our reported races with those reported by the Intel Inspector XE tool (2015 update 1 with `default` mode), which (unlike ARCHER) is known to have false negatives even for a given input. Overall, our tool reported a total of 61 races whereas Intel Inspector XE could only find 31 races. The details are presented in Table 5.2. A table entry marked with the letter "H" indicates that the Intel Inspector XE tool would get into a hang mode for that benchmark, while a table entry marked with the letter "A" indicates that the Intel Inspector XE tool encountered an Application exception for that benchmark.

Majority of the data races in the PolyBench/ACC OpenMP suite arises from:

- The PolyBench/ACC OpenMP suite developer might have simply forgotten to declare certain variables as private, although they were used in this way. The default sharing attribute rules of OpenMP specification will make the variable `shared` in this case, and resulting in data races on those variables. Also, this particular mistake is mentioned as one of the important source of errors in OpenMP programming [92]. As can be seen from Figure 5.2, the variable `x` in `Cholesky` benchmark and the variable `nrm` in `Gramschmidt` of PolyBench/ACC suite can be privatized to avoid races on those variables. In such scenarios, privatization can be realized either by moving the declaration of those variables into the parallel region or inserting the variables into `private` data sharing attribute list or adding `default(none)` to the OpenMP directive to get compiler errors on these variables.

- The benchmark developer might have incorrectly parallelized some of the linear algebra kernels (e.g., Symm and Trmm in Figure 5.3). These kernels are extremely hard to be parallelized by novice OpenMP programmers since these kernels have complex dependence patterns and requires much knowledge to ex-

```
1 DATA_TYPE x;
2 #pragma omp parallel for private (j,k)
3     for (i = 0; i < _PB_N; ++i)
4     {
5         x = A[i][i];
6         for (j = 0; j <= i - 1; ++j)
7             x = x - A[i][j] * A[i][j];
8         p[i] = 1.0 / sqrt(x);
9         for (j = i + 1; j < _PB_N; ++j)
10        {
11            x = A[i][j];
12            for (k = 0; k <= i - 1; ++k)
13                x = x - A[j][k] * A[i][k];
14            A[j][i] = x * p[i];
15        }
16    }
```

(a) Data races on the variable `x` in the Cholesky benchmark

```
1 DATA_TYPE nrm;
2 #pragma omp parallel for private (i, j)
3     for (k = 0; k < _PB_NJ; k++)
4     {
5         nrm = 0;
6         for (i = 0; i < _PB_NI; i++)
7             nrm += A[i][k] * A[i][k];
8         R[k][k] = sqrt(nrm);
9         for (i = 0; i < _PB_NI; i++)
10            Q[i][k] = A[i][k] / R[k][k];
11        for (j = k + 1; j < _PB_NJ; j++)
12        {
13            R[k][j] = 0;
14            for (i = 0; i < _PB_NI; i++)
15                R[k][j] += Q[i][k] * A[i][j];
16            for (i = 0; i < _PB_NI; i++)
17                A[i][j] = A[i][j] - Q[i][k] * R[k][j];
18        }
19    }
```

(b) Data races on the variable `nrm` in the Gramschmidt benchmark

Figure 5.2 : PolyBench/ACC OpenMP benchmark developer might have forgotten to mark certain variable as private variables (`x` in Cholesky, `nrm` in Gramschmidt), and there by resulting races on such variables.

```
#pragma omp parallel
{
/*  C := alpha*A*B + beta*C, A is symetric */
#pragma omp for private(j,acc,k)
    for (i = 0; i < _PB_NI; i++)
        for (j = 0; j < _PB_NJ; j++)
        {
            acc = 0;
            for (k = 0; k < j - 1; k++) {
                C[k][j] += alpha * A[k][i] * B[i][j];
                acc += B[k][j] * A[k][i];
            }
            C[i][j] = beta * C[i][j] + alpha * A[i][i]
                        * B[i][j] + alpha * acc;
        }
}
```

(a) Data races on the array `C` in the Symm benchmark

```
/*  B := alpha*A*B, A triangular */
#pragma omp parallel for private (j, k)
    for (i = 1; i < _PB_NI; i++)
        for (j = 0; j < _PB_NI; j++)
            for (k = 0; k < i; k++)
                B[i][j] += alpha * A[i][k] * B[j][k];
```

(b) Data races on the array `B` in the Trmm benchmark

Figure 5.3 : PolyBench/ACC OpenMP benchmark developer have incorrectly parallelized the linear algebra kernels (some of them are notoriously hard to be parallelized because of complex dependence patterns), and there by resulting races on arrays `C` in `Symm` and `B` in `Trmm` benchmarks.

pose hidden parallelism in the benchmarks. In such scenarios, our tool `PolyOMP` can be of a great help in aiding the programmers while debugging, because our tool can provide precise information (including precise iteration values) about the races.

5.4 Strengths and Limitations of Our Approach

In this section, we present strengths and limitations of our race detection approach using our extensions to the polyhedral model.

Strengths:

- The current implementation of race detection approach in `PolyOMP` supports OpenMP constructs such as `omp parallel for`, `parallel for`, `barrier`, `single`, `master` directives and nested parallel regions.
- Our approach reports races in a program independent of inputs to the program, unlike approaches based on dynamic analysis (e.g., Intel Inspector XE) which report races guaranteed on a given input.
- Our approach allows number of threads to be an unknown symbolic parameter unlike other approaches [84, 72] which are applicable only to a fixed number of threads in a given program.
- Our approach is guaranteed to be exact (with neither false positives nor false negatives) if the input program satisfies all the standard preconditions of the polyhedral model (without any non-affine constructs, and aware of any known relations between parameters). This has been evident in case of the evaluation on PolyBench/ACC OpenMP suite.
- Our approach can identify challenging data races (e.g., data race on variable k in jacobi03 benchmark in Figure 5.1) which can influence the program execution

overhead in dynamic analysis techniques. Hence, we believe that coupling our static approach with dynamic analysis techniques (e.g., Intel Inspector XE, ARCHER) can reduce overall program execution overhead in detecting races in larger OpenMP programs.

Limitations:

- Our tool currently does not perform any pointer based analysis. However, previous works on pointer analysis can be added as a pre-pass to our race detection stage to enhance the race detection.
- In our approach, we restrict our attention to textually aligned barriers, in which all threads encounter the same textual sequence of barriers. Each dynamic instance of the same `barrier` operation must be encountered by all threads. We plan to address textually unaligned barriers as part of the future work. However, many software developers believe that textually aligned barriers are better from a software engineering perspective.
- The support for analyzing SPMD programs with constructs that enforce mutual exclusion and task-based parallel constructs are part of future work.

5.5 Past Work on Race Detection

There is an extensive body of literature on identifying races in explicitly-parallel programs (at compile-time [83, 84, 72, 85, 75, 86], run-time [93], and hybrid combinations [87]). We focus our discussion on past work that is most closely related to static analysis techniques for identifying data races in SPMD-style parallel programs. Table 5.3 lists the details of related static analysis tools for race detection and their limitations with respect to `PolyOMP` .

Among the static analysis techniques, symbolic approaches have received a lot of attention in analyzing parallel programs, especially in the context of OpenMP. Yu et

	Supported Constructs	Approach	Guarantees
Pathg (Yu et al.)	OpenMP worksharing loops, barriers with depth 0, Atomic	Thread automata	Per no. of threads
OAT (Ma et al.)	OpenMP worksharing loops, Barriers, locks, Atomic, single, master	Symbolic execution	Per no. of threads
ompVerify (Basupalli et al.)	OpenMP 'parallel for'	Dependence analysis using Polyhedral model	Per worksharing loop
ARCHER (static) (Atzeni et al.)	OpenMP 'parallel for'	Dependence analysis using Polyhedral model	Per worksharing loop
PolyOMP Our Approach	OpenMP worksharing loops, Barriers in arbitrary nested loops, Single, master	MHP relations computed from the extensions to the polyhedral model	Per program

Table 5.3 : Closely related static approaches in race detection

al. [84] presented a symbolic approach for checking the consistency of multi-threaded programs with OpenMP directives using extended thread automata (with a tool called Pathg). However, their race detection is only guaranteed for a fixed number of worker threads. Ma et al. [72] also use a symbolic execution-based approach (running the program on symbolic inputs and fixed number of threads) to detect data races in OpenMP codes, based on constraint solving using an SMT solver. The data races reported from this toolkit (called OAT) are applicable only to a fixed number of input threads, unlike our approach which takes the number of threads as variable.

As part of static analysis techniques, polyhedral based approaches have also gained significant interest in analyzing parallel programs because these approaches perform exact analysis if the input program fits into the polyhedral model (without any non-affine constructs). Basupalli et al. [85] presented an approach (ompVerify) to detect data races inside a given worksharing loop using polyhedral dependence analysis. However, this approach handled only affine constructs and limited to work-

sharing loops. Yuki et al. [75] presented an adaptation of array data-flow analysis to X10 programs with finish/async parallelism. In this approach, the happens-before relations are first analyzed, and the data-flow is computed based on the partial order imposed by happen-before relations. This extended array dataflow analysis is used to certify determinacy in X10 finish/ async parallel programs by identifying the possibility of multiple sources of write for a given read. Their extended work [77] formulated the happens-before relations with X10 clocks in a polyhedral context. This approach provides the race-free guarantee of clocked X10 programs by disproving all possible races. But, it doesn't provide races present in the input program since computing happens-before relations involves polynomials in a general case.

Atzeni et al. [87] introduced a hybrid approach (ARCHER) to achieve high accuracy, low overheads on large OpenMP applications to detect data races. The static part of ARCHER tool still leverages the existing polyhedral dependence analyzer to identify races in a given worksharing loop. Our static approach can be complemented with the dynamic analysis of ARCHER tool to further reduce overheads as observed for the benchmark in Figure 5.1.

Chapter 6

Optimization Of SPMD Programs – Static Redundant Barrier Detection

Optimization is detrimental to future success.

Erik Naggum

As we are evolving towards homogeneous and heterogeneous many-core processors, and relying on SPMD model for the homogeneous and SIMT model for heterogeneous cores, it is likely that redundant synchronization will become more prevalent. The performance of a parallel program is often determined by its synchronization behavior. Barriers are one of the popularly used synchronization construct in SPMD-style parallel programs particularly with OpenMP and MPI, but barriers introduce execution overheads along with influencing scalability of parallel programs. Technically, a barrier is a redundant barrier if no data races emanate after removing the barrier. The goal of redundant barrier detection in an input SPMD-style program is to identify a set of barriers that can be eliminated without affecting the semantics of the program. Complicating the matter, barriers may be enclosed in imperfectly nested sequential loops of an SPMD region, thereby making static analysis harder to reason. Henceforth, detecting redundant barriers has been receiving a fair attention [94, 95, 96, 49] in the parallel programming. In this chapter, we propose and evaluate our approach to identify redundant barriers in SPMD-style programs at compile-time with our extensions (introduced in Chapter 3) to the polyhedral intermediate representation.

6.1 Motivation

To motivate our approach for detection of redundant barriers at compile-time, we consider a SPMD-style program as an illustrative example. The excerpt shown in Fig-

```
1 #pragma omp parallel private (j, k)
2     {
3         /* E := A*B */
4 #pragma omp for
5         for (i = 0; i < _PB_NI; i++) {
6             for (j = 0; j < _PB_NJ; j++) {
7                 E[i][j] = 0;
8                 for (k = 0; k < _PB_NK; ++k)
9                     E[i][j] += A[i][k] * B[k][j];
10            }
11        }

13        /* F := C*D */
14 #pragma omp for
15        for (i = 0; i < _PB_NJ; i++) {
16            for (j = 0; j < _PB_NL; j++) {
17                F[i][j] = 0;
18                for (k = 0; k < _PB_NM; ++k)
19                    F[i][j] += C[i][k] * D[k][j];
20            }
21        }

23        /* G := E*F */
24 #pragma omp for
25        for (i = 0; i < _PB_NI; i++) {
26            for (j = 0; j < _PB_NL; j++) {
27                G[i][j] = 0;
28                for (k = 0; k < _PB_NJ; ++k)
29                    G[i][j] += E[i][k] * F[k][j];
30            }
31        }
32    }
```

Figure 6.1 : Redundant barrier (implicit) at line 11 in the `3mm` benchmark from `PolyBench/ACC` benchmark suite

ure 6.1 is a part of the `3mm` benchmark from the `PolyBench/ACC` OpenMP suite [91], which computes a sequence of three matrix multiplications E = A.B; F = C.D; G = E.F. The excerpt contains a parallel region (lines 1-32) spanning three worksharing loops having implicit barriers. As no dependences flow between the first two worksharing loops, the implicit barrier between them is redundant and conservative. This pattern of over-conservative synchronization is quite common especially while programmer trying to parallelize loops with worksharing directives which have implicit barriers by default. Such redundant barriers not only introduce execution overheads but also affect the scalability of applications since they involve system-wide communication and coordination.

Our approach identifies such redundant barriers by building on our work on data race detection as follows. First, our static analysis temporarily elides all barriers in the program and computes the resulting data races. Next, it maps each barrier to a set of data races which can potentially be fixed with that barrier and eventually it builds a bipartite graph from the barriers to the data races. This mapping information is then used to compute sets of required barriers in the program that can completely fix all the data races. Then, a set of redundant barriers is computed by subtracting the set of required barriers from all the barriers in the program. To illustrate the potential performance impact of the optimization, we have performed experiments on 57-cores Intel Knights Corner (Xeon Phi) system with four threads per each core. As reported in Table 6.3, there is 2.5% improvement by removing the redundant barrier between the first two worksharing loops. Also, we have observed a performance improvement from this optimization as high as 9% for the `2mm` benchmark on Intel Xeon Phi from `PolyBench ACC` OpenMP suite [91].

6.2 Our Approach

In this section, we present an approach (See Algorithm 4) to identify redundant barriers in SPMD programs. Removing a barrier from an SPMD program is valid (keeping semantics preserved) as long as removing the barrier doesn't introduce data races in the input program. Henceforth, our approach ignores input programs which have data races.

6.2.1 An Algorithm to Identify Redundant Barriers

Algorithm 4 summarizes overall steps in identifying redundant barriers at compile-time in an SPMD program and reports a warning (at lines 4-5) if the input SPMD program has data races. The following are the major steps involved in Algorithm 4.

1) Firstly, our approach temporarily elides all barriers in the input program (at lines 6-7) and computes data races* with our race detection approach in Algorithm 3. For example, temporarily eliding all barriers in the `3mm` benchmark (shown in Figure 6.1) results in two races, i.e., race r1 between statements on line 9, 29 on the array `E`, and another race r2 between statements on line 19, 29 on the array `F`.

2) Then, our approach constructs a bipartite graph with one set being barriers and another set being the data races from the previous step (line 10). For the `3mm` benchmark, the barriers and the data races in the bipartite graph are implicit barriers b1, b2, b3 at lines 11, 21 31 in Figure 6.1, and races r1, r2 respectively.

3) Next, our approach maps each barrier in the bipartite graph to a set of data races which can be avoided with that barrier. As can be seen from Figure 6.2, the implicit barrier b1 can potentially fix the race r1, and the barrier b2 can fix both races r1 and r2, whereas the barrier b3 fixes neither of races. The mappings

*Note that any race detection tool can be used in place of our approach to recognize races.

Algorithm 4: An approach to compute a set of redundant barriers in an SPMD program

Input : An SPMD program, $\mathcal{P}$
Output: A set of redundant barriers (REDBARR) in the SPMD program

```
1  begin
2      B ← Set of barriers in the input SPMD program P

       /* Identify data races in the input SPMD program with our race
          detection approach in Algorithm 3                          */
3      R ← Dataraces(P)

4      if R ≠ φ then
5          Report a warning that the input SPMD program P has races, and our
           approach ignores racy input SPMD programs; Return

       /* Temporarily elide all barriers in the input program and
          computes data races                                        */
6      P^1 ← Elide the set B from the input SPMD program P
7      R^1 ← Dataraces(P^1)

       /* Return all barriers in original program as redundant
          barriers if there are no races originating after eliding
          the barriers                                               */
8      if R^1 = φ then
9          Output B

       /* If there are data races after eliding barriers, then
          construct a bipartite graph (shown in Algorithm 5), from
          the barriers to the data races, to identify a set of
          redundant barriers                                         */
10     G ← Build a bipartite graph (P, R^1)

       /* After building bipartite graph, compute a set of required
          barriers, using a greedy approach (shown in Algorithm 6) to
          cover all data races in the bipartite graph                */
11     REQBARR ← Compute required barriers(G)

       /* Compute redundant barriers by subtracting required barriers
          from all barriers in the SPMD program                      */
12     REDBARR ← B - REQBARR
13     Output REDBARR
14 end
```

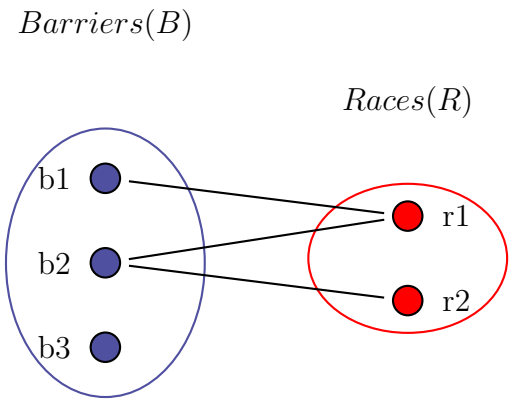


Figure 6.2 : Bipartite graph constructed by mapping each barrier in `3mm` benchmark to data races that can be avoided with the barrier

between the barriers and the data races are constructed using Algorithm 5 and the approach is summarized below.

For each barrier,

- Our approach recomputes phase mappings of all statements in the SPMD program with only that barrier (line 4 in Algorithm 5).
- Then for each identified data race in the bipartite graph, if the source and target of the data race have different recomputed phase mappings, then our approach adds an edge between the barrier and the data race, i.e., the barrier can potentially avoid the race, to the bipartite graph (at lines 5-9 in Algorithm 5).

4) After constructing the bipartite graph, our approach uses a greedy strategy (at line 11) to compute a set of minimum number of required barriers to keep the original semantics of the program, i.e., all data races in the bipartite graph are covered by the set of required barriers. For the `3mm` benchmark, the barrier b2 is sufficient enough to cover both of the races r1 and r2.

Algorithm 5: An approach to construct a bipartite graph from barriers to data races in an SPMD program

```
   Input  : An SPMD program (P), and a set of races (R)
   Output: A bipartite graph (G) from barriers in P to race in R
 1 begin
 2     B ← Set of barriers in the input SPMD program P
 3     for barrier b in B do
           /* Recompute phase mappings based on the barrier b for all
              the statements in the SPMD program with our approach
              in Algorithm 1                                          */
 4         P^1 ← P ∪ {b}
 5         Θ^P ← Phases(P^1)
           /* Loop through each data race to verify whether the data
              race can be avoided by the barrier b                    */
 6         for race r in R do
 7             S, T := Source and target of the data race r
               /* If the phase mappings of S and T are different, then
                  add an edge between the barrier b and the data race r
                  to the bipartite graph                              */
 8             if Θ^P(S) ≠ Θ^P(T) then
 9                 G ← G ∪ {(b ⟶ r)}
10     Ouput G
11 end
```

5) Finally, our approach computes set of redundant barriers by eliminating the set of required barriers from the barriers in the input SPMD program (at line 12). The set of redundant barriers in case of the `3mm` benchmark are the implicit barriers b1 and b3, and their removal doesn't influence semantics of the original benchmark.

6.2.2 A Greedy Approach to Compute a Set of Required Barriers

In this subsection, we present a greedy approach to compute a minimum set of required barriers in an SPMD program, which needs to be retained in the original program to avoid data races. The greedy approach (shown in Algorithm 6) takes a bipartite graph from barriers to data races as an input and outputs a minimum set of barriers, that can cover all the races in the bipartite graph[†]. The entire approach is summarized below.

1) Firstly, our greedy approach includes any barriers that are the only barriers that address particular races into a set of required barriers. Then, our approach removes all the races that these barriers address, and also these barriers from the bipartite graph (lines 3-8). For the bipartite graph in Figure 6.2 of the `3mm` benchmark, our approach would first consider the barrier b2 since it is the only barrier that can address the race r2. Then, our approach would add the barrier b2 to a set of required barriers, and then remove the barrier b2 from the bipartite graph. Also, it removes both of the races r1 and r2 from the bipartite graph since the barrier b2 can address both of them.

2) Then, our approach works by considering a barrier that can address more races from the remaining bipartite graph, i.e., keeping that barrier can fix more races. Then, the approach adds that barrier to the set of required barriers. Similarly to the first step, our approach then removes the barrier and all the races that it can address from the bipartite graph (lines 9-13). After removing the barrier b2, and races r1, r2 in the previous step on the bipartite graph of the `3mm` benchmark, there will be no races left in the bipartite graph that our approach would need to address.

[†]Note that this problem can be modeled as an instance of minimum set cover problem, and the proposed greedy approach (excluding the first step) is equivalent to the $\log n$ based approximation algorithm for minimum set cover problem.

Algorithm 6: A greedy approach to compute a set of required barriers

Input : Bipartite graph G from barriers (B) to races (R)
Output : A set of required barriers (RB) in B such that all races in R are covered

```
1  begin
       /* RB indicates required barriers to retain semantics        */
2      RB ← φ
       /* Include any barriers that are the only barriers that
          address particular races, and remove all the races these
          barriers address                                          */
3      for race r in R do
4          if In-degree(r) in G = 1 then
5              b ← Source of the edge to r
6              RB ← RB ∪ {b}
7              R ← R - ⋃_{r∈R} { race r s.t there is an edge from barrier b to it }
8              B ← B - { b}
       /* Repeat until all races are covered without exhausting
          barriers                                                  */
9      while R is not empty and B is not empty do
           /* Greedy choice                                        */
10         b ← Pick up a barrier from G with highest outgoing degree
           /* Add barrier b to required barriers                   */
11         RB ← RB ∪ {b}
           /* Remove races from R that are connected to barrier b  */
12         R ← R - ⋃_{r∈R} { race r s.t there is an edge from barrier b to it }
           /* Remove barrier b from B                              */
13         B ← B - { b}
14 end
```

3) Next, the second step is repeated until all the races are covered or no more barriers left in the remaining bipartite graph. In case of the `3mm` benchmark, our greedy approach would finally return the barrier b2 as the required barrier that can address all the races in the input bipartite graph.

Since the proposed greedy approach may not find an optimal solution in certain scenarios, we defer the efficient approaches to find the optimal solution to future work. Also, our greedy approach has the following assumptions: 1) All barriers (regardless of its depth) are treated uniformly in finding minimum number of required barriers, and 2) all edges in the bipartite graph have equal weights which may be not true in certain SPMD programs having barriers with higher depths.

6.3 Experimental Evaluation

In this section, we evaluate our approach by measuring performance improvement of SPMD programs after removing redundant barriers identified from the approach. Firstly, we briefly describe our experimental setup and benchmark suites used for the evaluation. Then, we present our discussion on the obtained results for each of the benchmark suites.

6.3.1 Experimental Setup

Our evaluation uses two different multi-way SMP multicore setups: an Intel Xeon Phi and a IBM Power8 system. Table 6.1 lists their hardware specifications. In the evaluation, we compare two experimental variants: 1) OpenMP to show the original OpenMP parallel version running with all threads - i.e., 228 on Intel KNC and 192 on Power8, and 2) PolyOMP to show the transformed version by our framework running with all threads. The improvement factor is defined as the execution time of the original version of the parallel program divided by the execution time of the optimized

	Intel Knights Corner (KNC)	**IBM Power 8E (Power 8)**
Micro architecture	Xeon Phi	Power PC
Clock speed	1.10GHz	3.02GHz
Cores/socket	57	12
Total cores	57	24
Threads per core	4	8
Total threads	228	192
Compiler	Intel ICC v15.0.0	IBM XLC v13.1.2
Compiler flags	-O3 (highest)	-O5 (highest)

Table 6.1 : Hardware specifications of the experimental setup for evaluating our approach to identify redundant barriers.

parallel version of the program by removing redundant barriers.

6.3.2 OpenMP Source Code Repository

OpenMP Source Code Repository (OmpSCR) [88] consists of 18 OpenMP-C benchmarks, in which 6 use `C structs` and pointer arithmetic. We defer support for `C structs` and pointer arithmetic in our current toolchain for future work, and hence we ignore these six benchmarks. Hence, our results focus on the remaining 12 OpenMP-C benchmarks, which are listed in Table 6.2.

Discussion. Our tool `PolyOMP` identified absence of races (including false positives) in the three benchmarks `Lud, LoopA.sol1, C_pi` and applied the Algorithm 4 to detect redundant barriers. But, the tool recognized that all barriers are necessary to respect program semantics and hence no elimination applied to these benchmarks. Since our tool `PolyOMP` enables redundant barrier optimization only for race-free input programs, the tool refuses to optimize the remaining nine benchmarks having

Benchmark	#Barriers in the input program	#Barrier instances (dynamic count) during the program execution	#Eliminated in the input program
Lud	1	`size` - 1	0
LoopA.sol1*	2	2×`numiter`	0
C_pi	1	1	0
Jacobi01*	2	2 × f_1(`k`, `error`)	I
Jacobi02*	3	2 × f_2(`k`, `error`) + 1	I
LoopA.sol2*	3	2×`numiter` + 1	I
LoopA.sol3*	2	2×`numiter`	I
Jacobi03**	3	2 × f_3(`k`, `error`) + 1	I
LoopA.bad**	1	`numiter`	I
LoopB.bad1**	1	`numiter`	I
LoopB.bad2**	1	`numiter`	I
LoopB.pipe**	3	2×`numiter` + 1	I

Table 6.2 : Redundant barrier detection analysis over the subset of `OmpSCR` benchmark suite. Benchmarks labeled with (*) have no true races but our race detection algorithm reported false positives, and benchmarks with (**) indeed have true races. Our tool ignored (I) the benchmarks with labels (*, **) because of the presence of races (including false positives). `size, k, error, numiter` are symbolic parameters in the corresponding benchmarks. Note that we also count implicit barriers after the `omp parallel` construct even though these implicit barriers cannot be removed from the source code.

data races (including false positives). But the benchmarks `Jacobi01, Jacobi02, LoopA.sol2, LoopA.sol3` don't have true races, and still, our tool ignored them because of false positive races arising from the unanalyzed array subscripts in those benchmarks. However, the tool can be improved to enable redundant barrier optimization for input programs, which have no true races but our race detection algorithm reported false positives, with programmer's support.

6.3.3 PolyBench/ACC OpenMP Suite

The PolyBench/ACC OpenMP Suite [91] consists of OpenMP implementations of the original PolyBench suite [89] to run on GPU's and accelerators. The suite contains 32 benchmarks, and for which our tool was unable to compile ten benchmarks due to the incorrect usage of OpenMP directives in those benchmarks according to OpenMP specifications. Hence, our results focus on the remaining 22 benchmarks. For each benchmark among those 22 benchmarks, Table 6.3 shows the number of barriers in the original benchmark, how many times barriers executed, how many barriers were removed by our optimization, and the performance improvement factor by the barrier elimination on Intel KNC and Power 8. We have used `large` dataset as an input to measure the performance improvement since the evaluation on `large` dataset has less standard deviation compared to other datasets.

Discussion. Our tool `PolyOMP` identified a total of 19 redundant barriers from 11 benchmarks (`Correlation, Covariance, 2mm, 3mm, Doitgen, Gemm, Gemver, Mvt, Syrk, Syr2k, Convolution-3d, fdtd-apml`) among 22 benchmarks considered for the evaluation. The geometric mean of improvement factors after removing these 19 redundant barriers are 1.032x on Intel KNC and 1.007x on Power8. Among these 19 redundant barriers, 12 redundant barriers (from `Correlation, 3mm, Gemver, Mvt, Syrk, Syr2k`) are the implicit barriers between worksharing loops which don't have data dependences flowing between them, and our tool removed these redundant barriers by adding `nowait` clause to the worksharing loop. Since these 12 redundant barriers are between worksharing loops and these worksharing loops in the benchmarks have better load balance, the improvement factors are not significant after removing these redundant barriers. However, we believe that the improvement may be significant in the case of benchmarks having 1) more dynamic instances of redundant barriers during the program execution, 2) redundant barriers between unbalanced worksharing loops. As shown in Table 6.3, eliminating redun-

Benchmark	#Barriers in the input program	#Barrier instances (dynamic count) during execution	#Eliminated in the input program	Mean improvement (Standard deviation)	
				Intel KNC	Power 8
Correlation	5	5	2	1.008 (± 0.019)	1.049 (± 0.146)
3mm	4	4	2	1.024 (± 0.018)	1.006 (± 0.041)
Gemver	5	5	2	1.006 (± 0.010)	1.004 (± 0.041)
Mvt	3	3	2	1.025 (± 0.012)	0.971 (± 0.038)
Syrk	3	3	2	1.003 (± 0.030)	0.999 (± 0.012)
Syr2k	3	3	2	0.994 (± 0.023)	1.000 (± 0.010)
Covariance	4	4	1	1.008 (± 0.032)	0.993 (± 0.020)
2mm	3	3	1	1.090 (± 0.022)	0.985 (± 0.084)
Doitgen	2	2	1	1.091 (± 0.610)	0.997 (± 0.010)
Gemm	2	2	1	1.011 (± 0.040)	1.016 (± 0.059)
Gesummv	2	2	1	0.991 (± 0.034)	0.998 (± 0.043)
Fdtd-apml	2	2	1	1.149 (± 0.510)	1.068 (± 0.205)
Convolution-3	2	2	1	A	A
Convolution-2*	1	1	0	NR	NR
Atax**	3	3	I	NR	NR
Bicg**	3	3	I	NR	NR
Cholesky**	2	2	I	NR	NR
Symm**	2	2	I	NR	NR
Trmm**	2	2	I	NR	NR
Durbin**	3	3	I	NR	NR
Gramschmidt**	1	1	I	NR	NR
Lu**	2	2	I	NR	NR

Table 6.3 : Redundant barrier detection analysis over the subset of PolyBench/ACC OpenMP benchmark suite. Benchmarks labelled with (*) doesn't have redundant barriers, and we didn't run (NR) the benchmarks for performance evaluation. Benchmarks labelled with (**) have true races, and our tool ignored (I) these benchmarks. A - Application exception, i.e., Segmentation fault in the original program itself. Note that we also count implicit barriers after the `omp parallel` construct even though these implicit barriers cannot be removed from the source code.

dant barriers generally contributes to overall performance while small slowdown was observed in some benchmarks (e.g., `Gesummv, Syr2k` benchmarks). On Power8, the IBM XL compiler supports efficient runtime barriers, which reduce the effect of barrier eliminations on the application performance, compared to the ICC compiler on Intel KNC. Remaining seven redundant barriers (from `Covariance, 2mm, Doitgen, Gemm, Gesummv, Convolution-3, fdtd-apml`) out of 19 are the implicit barriers from the worksharing loops which are immediately succeeded by the end of the omp parallel region construct. We believed that the existing compilers (Intel ICC, IBM XLC) could identify and eliminate these seven redundant barriers as part of their optimizations, but we could still observe these redundant barriers in the assembly codes generated by these compilers. Hence, we believe that adding this redundant barrier optimization can help the existing compilers to improve the performance and even enable more opportunities for further optimizations.

`PolyOMP` also recognized the absence of redundant barriers in 1 benchmark, i.e., `Convolution-2` out of the 22 benchmarks. Hence, we didn't evaluate this benchmark for performance improvement. Also, `PolyOMP` identified true races in the remaining eight benchmarks (`Atax, Bicg, Cholesky, Symm, Trmm, Durbin, Gramschmidt, Lu`). Since our tool checks for absence of data races in the input program before identifying redundant barriers, these eight benchmarks were ignored by our tool and no evaluation was performed on these benchmarks.

6.4 Strengths and Limitations of Our Approach

In this section, we present strengths and limitations of our approach to identify a minimum set of required barriers in an SPMD program to preserve its semantics.

Strengths:

- Our approach can model barriers with higher depths, i.e., deeply enclosed in

arbitrarily nested sequential loops in a SPMD program.

- Also, our approach can compute phase mappings very precisely unlike other approaches [95, 97] which computes phase information conservatively in case of barriers with higher depths. Such precise phase mappings may help in removing more redundant barriers compared to [95, 97], and enabling other transformations such as fusion of SPMD regions to enable more loop transformations across SPMD regions.

Limitations:

- Since the problem of finding the minimum set of required barries can be modeled as an instance of minimum set cover problem, our algorithm in Algorithm 4 can be strengthened by replacing the greedy approach with an ILP formulation.

6.5 Past Work on Analysis of Barriers

There is an extensive research ([94, 95, 96, 49]) done towards analyzing barriers present in SPMD programs. In this section, we focus on closely related compile-time approaches for analysis of the barriers and the summary is presented in Table 6.4.

In the beginning, Aiken et al. have developed an inference system that detects the SPMD structure and verifies the correctness of global barrier synchronization [94]. In this work, `single-valued` expressions are introduced to evaluate to the same value in all the processes, to ensure that all processes execute the same number of barriers. Kamil et al. [95] extended the work in [94] by constructing a concurrency graph from a program written in the context of Titanium parallel programming language [49]. Then, MHP relations are computed by performing depth-first traversals on the concurrency graph. However, this approach results in conservative MHP relations in case of programs with barriers enclosed in nested sequential loops, unlike our approach

	Style	Key idea	Limitations
Kamil et al LCPC'05	SPMD	Tree traversal on concurrency graph	Conservative MHP in case of barriers enclosed in loops
Tseng et al PPoPP'95	SPMD + fork-join	Communication analysis b/w computation partitions	Structure of loops enclosing barriers
Zhao et al PACT'10	fork-join	SPMDization by loop transformations	Join (barrier) synchronization from only for-all loops
Surendran et al PLDI'14	fork-join	Dynamic programming on scoped dynamic structure trees	Limited to *finish* construct but the finish placement algorithm is optimal
Our approach	SPMD	Precise MHP analysis with extensions to Polyhedral model	Can support barriers in arbitrarily nested loops

Table 6.4 : Closely related static approaches in barrier analysis

which computes MHP relations precisely in such scenarios.

Tseng [97, 98] proposed a greedy approach that combines array data dependence analysis and communication analysis over threads for redundant barrier elimination in a hybrid programming model employing fork-join and SPMD techniques. In particular, the communication analysis is performed by constructing a system of inequalities and solving it with the Fourier-Motzkin elimination process, an early polyhedral technique. However, if a loop contains one or more required barriers, this approach is not always able to detect the barrier at the end of the loop body, which may be redundant and can be eliminated. But, our approach can identify such redundant barriers if they exist in the input program.

Zhao et al.[14] addressed the problem of barrier elimination of explicitly-parallel programs by SPMDization of region code in the fork-join model. They proposed a compiler based approach to SPDMize the code so as to reduce the number of spawned

tasks, thereby reducing the number of required synchronizations. Their work leverages typical transformations such as loop interchange and introduces novel ones such as redundant next-single elimination. However, this approach is limited to barriers (as part of join synchronization) arising from only `forall` construct. But, our approach can handle not only barriers from `forall` construct but also barriers enclosed in arbitrarily nested sequential loops.

Surendran et al. [99] addressed the problem of inserting `finish` synchronization construct in X10 parallel programs, where parallelism is expressed using `async` construct. Their approach starts with a program having data races and then determines where additional synchronization constructs should be inserted to guarantee correctness (absence of data races), with the goal of maximizing parallelism. But, our approach ignores an input program if it has data races, and also our approach doesn't insert any additional synchronization constructs (barriers) to eliminate data races. Instead, our approach checks for the redundant synchronization already present in the input program.

Chapter 7

Conclusions & Future Work

A story really isn't truly a story until it reaches its climax and conclusion.

Ted Naifeh

This work is motivated by the observation that software with explicit parallelism is on the rise, and that SPMD parallelism is a common model for explicit parallelism as evidenced by the popularity of OpenMP, OpenCL, and CUDA. As with other imperative parallel programming models, data races are a pernicious source of bugs in the SPMD model and may occur only in few of the possible schedules of a parallel program, thereby making them extremely hard to detect dynamically. However, effective approaches to static data race detection remains an open problem, despite significant progress in recent years. Further, in addition to debugging parallel programs, it is important to extend classical code optimization techniques (such as partial/total redundancy elimination) to operations such as synchronization barriers that incur large overheads in current parallel programming models.

In this work, we formalized May-happen-in-parallel (MHP) relations to capture partial execution orders in SPMD program by extending the polyhedral model with "space" and "phase" mappings. We demonstrate the value of these extensions and formalized MHP relations by its use in two applications to help developers of SPMD programs — identification of data races, as well as identification and removal of redundant barriers. We evaluate our approaches on the 34 OpenMP programs from

the `OmpSCR` and `PolyBench/ACC` benchmark suites.

In summary, the contributions of this thesis include the following: 1) It describes our extensions to the polyhedral compilation model to represent partial execution order present in SPMD programs. 2) It formalizes the partial order as May-Happen-in-Parallel (MHP) information using our extensions to the polyhedral model. 3) It presents an approach for compile-time detection of data races in SPMD programs [44]. 4) It presents an approach for identification and removal of redundant barriers at compile-time in SPMD programs. 5) It demonstrates the effectiveness of the approaches on 34 OpenMP programs from the `OmpSCR` and the `PolyBench/ACC` OpenMP benchmark suites.

As part of future work, 1) We plan to use the proposed extensions for further static applications such as detecting false sharing issues, and identifying deadlocks in input parallel programs. 2) We also plan to enhance existing hybrid race detection tools [87, 100], by either adding our race detection approach in the static analysis of the hybrid approaches or help the dynamic analysis (e.g., in [101]) with the MHP information from our extensions. 3) We also plan to extend the work on race detection to enable program repair by automatically inserting barrier synchronization to eliminate the data races that were detected, as has been done with `finish` synchronization for Habanero-Java [99]. 4) Also, we are interested in extending our work on redundant barrier detection to replace non-redundant barriers with fine-grained synchronization constructs [97, 98], in both user-written code and in the output of automatic program repair. 5) Finally, we plan to enable classic scalar optimizations (code motion) on concurrency constructs in SPMD programs with our proposed extensions to the polyhedral model, as has been done in optimizing remote access on distributed memory machines using the Split-C language as a global address layer [36].

Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience.

Roger Bacon

Bibliography

[1] T. Grosser, "islplot." https://github.com/tobig/islplot, 2014.

[2] S. Verdoolaege and T. Grosser, "Polyhedral Extraction Tool," in *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, (Paris, France), Jan. 2012.

[3] V. Sarkar, W. Harrod, and A. E. Snavely, "Software Challenges in Extreme Scale Systems," *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012045, 2009.

[4] F. Darema, D. George, V. Norton, and G. Pfister, "A single-program-multiple-data computational model for EPEX/FORTRAN," *Parallel Computing*, vol. 7, no. 1, pp. 11 – 24, 1988.

[5] L. Nyman and M. Laakso, "Notes on the History of Fork and Join," *IEEE Annals of the History of Computing*, vol. 38, no. 3, pp. 84–87, 2016.

[6] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Comput. Sci. Eng.*, vol. 5, pp. 46–55, Jan. 1998.

[7] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, pp. 40–53, Mar. 2008.

[8] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *IEEE Des. Test*, vol. 12, pp. 66–73, May 2010.

[9] Anon, "MPI: A Message Passing Interface," in *Proceedings of the Supercomput-*

ing Conference, pp. 878–883, 1993.

[10] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, (New York, NY, USA), pp. 207–216, ACM, 1995.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, (New York, NY, USA), pp. 519–538, ACM, 2005.

[13] R. Cytron, J. Lipkis, and E. Schonberg, "A Compiler-assisted Approach to SPMD Execution," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, (Los Alamitos, CA, USA), pp. 398–406, IEEE Computer Society Press, 1990.

[14] J. Zhao, J. Shirako, V. K. Nandivada, and V. Sarkar, "Reducing Task Creation and Termination Overhead in Explicitly Parallel Programs," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 169–180, ACM, 2010.

[15] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar, "A Transformation Framework for Optimizing Task-Parallel Programs," *ACM Trans. Program. Lang. Syst.*, vol. 35, pp. 3:1–3:48, Apr. 2013.

[16] M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K. Y. Wang, and M. Burke, "PTRAN II - A Compiler for High Performance Fortran," in *4th International*

Workshop on Compilers for Parallel Computers, 1993.

[17] "The Next Generation of Compilers," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, (Washington, DC, USA), IEEE Computer Society, 2009.

[18] M. Hall, D. Padua, and K. Pingali, "Compiler Research: The Next 50 Years," *Commun. ACM*, vol. 52, pp. 60–67, Feb. 2009.

[19] V. Sarkar, "Parallel Functional Languages and Compilers," ch. PTRAN–the IBM Parallel Translation System, pp. 309–391, New York, NY, USA: ACM, 1991.

[20] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy, "The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler," tech. rep., Stanford, CA, USA, 1994.

[21] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford, "Polaris: The Next Generation in Parallelizing Compilers," in *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, Springer-Verlag, Berlin/Heidelberg, 1994.

[22] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic Transformations for Communication-minimized Parallelization and Locality Optimization in the Polyhedral Model," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, (Berlin, Heidelberg), pp. 132–146, Springer-Verlag, 2008.

[23] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," in *Proceedings of*

the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, (New York, NY, USA), pp. 101–113, ACM, 2008.

[24] J. Shirako, L.-N. Pouchet, and V. Sarkar, "Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 287–298, IEEE Press, 2014.

[25] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral Parallel Code Generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, pp. 54:1–54:23, Jan. 2013.

[26] P. Feautrier, "Parametric integer programming," *RAIRO - Operations Research - Recherche Oprationnelle*, vol. 22, no. 3, pp. 243–268, 1988.

[27] P. Feautrier, "Dataflow Analysis of Array and Scalar References," *International Journal of Parallel Programming*, vol. 20, 1991.

[28] P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem: I. One-dimensional Time," *Int. J. Parallel Program.*, vol. 21, pp. 313–348, Oct. 1992.

[29] P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time," *International journal of parallel programming*, vol. 21, no. 6, pp. 389–420, 1992.

[30] J. Shirako, A. Hayashi, and V. Sarkar, "Optimized Two-level Parallelization for GPU Accelerators Using the Polyhedral Model," in *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, (New York, NY, USA), pp. 22–33, ACM, 2017.

[31] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based Data Reuse Optimization for Configurable Computing," in *Proceedings of the*

ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, (New York, NY, USA), pp. 29–38, ACM, 2013.

[32] U. Bondhugula, "Compiling Affine Loop Nests for Distributed-memory Parallel Architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 33:1–33:12, ACM, 2013.

[33] G. Martinovic, Z. Krpic, and S. Rimac-drlje, "Parallelization Programming Techniques: Benefits and Drawbacks," in *Proceedings of the First International Conference on Cloud Computing, GRIDs, and Virtualization*, 2010.

[34] M. Frumkin, M. Hribar, H. Jin, A. Waheed, and J. Yan, "A Comparison of Automatic Parallelization Tools/Compilers on the SGI Origin 2000," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, (Washington, DC, USA), pp. 1–22, IEEE Computer Society, 1998.

[35] V. Sarkar, "Analysis and Optimization of Explicitly Parallel Programs Using the Parallel Program Graph Representation," in *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '97, (London, UK, UK), pp. 94–113, Springer-Verlag, 1998.

[36] A. Krishnamurthy and K. Yelick, "Optimizing Parallel Programs with Explicit Synchronization," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, (New York, NY, USA), pp. 196–204, ACM, 1995.

[37] D. Novillo, *Compiler Analysis and Optimization Techniques for Explicitly Parallel Programs*. PhD thesis, University of Alberta, 2000.

[38] J. Ferrante, D. Grunwald, and H. Srinivasan, "Compile-time Analysis and Optimization of Explicitly Parallel Programs," *Parallel Algorithms and Applications*, vol. 12, no. 1-3, pp. 21–56, 1997.

[39] J. Collard, "Array SSA for Explicitly Parallel Programs," in *Euro-Par '99 Parallel Processing, 5th International Euro-Par Conference, Toulouse, France, August 31 - September 3, 1999, Proceedings*, pp. 383–390, 1999.

[40] J. Collard and M. Griebl, "Array Dataflow Analysis for Explicitly Parallel Programs," *Parallel Processing Letters*, vol. 7, no. 2, pp. 117–131, 1997.

[41] P. Chatarasi, J. Shirako, and V. Sarkar, "Polyhedral Optimizations of Explicitly Parallel Programs," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, (Washington, DC, USA), pp. 213–226, IEEE Computer Society, 2015.

[42] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, (New York, NY, USA), pp. 249–265, ACM, 2017.

[43] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar, "May-happen-in-parallel Analysis of X10 Programs," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, (New York, NY, USA), pp. 183–193, ACM, 2007.

[44] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar, "An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection," in *Languages and Compilers for Parallel Computing - 29th International Workshop, LCPC 2016, Rochester, NY, USA, September 28-30, 2016, Revised Papers*, pp. 106–120, 2016.

[45] U. K. R. Bondhugula, *Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model*. PhD thesis, Department of Computer Science and Engineering at Ohio State University, OH, USA, 2008.

[46] T. Grosser, *A Decoupled Approach to High-level Loop Optimization : Tile*

Shapes, Polyhedral Building Blocks and Low-level Compilers. Theses, Université Pierre et Marie Curie - Paris VI, Oct. 2014.

[47] M. R. Kong, *Enabling Task Parallelism on Hardware/Software Layers using the Polyhedral Model*. PhD thesis, Department of Computer Science and Engineering at Ohio State University, OH, USA, 2016.

[48] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for EPEX/FORTRAN.," *Parallel Computing*, vol. 7, no. 1, pp. 11–24, 1988.

[49] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A High-Performance Java Dialect," *Concurrency Practice and Experience*, vol. 10, pp. 825–836, 9 1998.

[50] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, (New York, NY, USA), pp. 212–223, ACM, 1998.

[51] A. Mani, "OSS-Based Grid Computing," *CoRR*, vol. abs/cs/0608122, 2006.

[52] M. Feng and C. E. Leiserson, "Efficient Detection of Determinacy Races in Cilk Programs," in *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, (New York, NY, USA), pp. 1–11, ACM, 1997.

[53] "OpenMP Application Program Interface, Version 4.0." http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf, July 2013.

[54] S. Verdoolaege, *isl: An Integer Set Library for the Polyhedral Model*, pp. 299–302. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[55] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The Polyhedral Model is More Widely Applicable Than You Think," in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, (Berlin, Heidelberg), pp. 283–303, Springer-Verlag, 2010.

[56] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies," *Int. J. Parallel Program.*, vol. 34, pp. 261–317, June 2006.

[57] M. Griebl, C. Lengauer, and S. Wetzel, "Code Generation in the Polytope Model," in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, (Washington, DC, USA), IEEE Computer Society, 1998.

[58] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, vol. 0, pp. 7–16, 2004.

[59] T. Grosser, S. Verdoolaege, and A. Cohen, "Polyhedral AST Generation Is More Than Scanning Polyhedra," *ACM Trans. Program. Lang. Syst.*, vol. 37, pp. 12:1–12:50, July 2015.

[60] J.-F. Collard, D. Barthou, and P. Feautrier, "Fuzzy Array Dataflow Analysis," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, (New York, NY, USA), pp. 92–101, ACM, 1995.

[61] B. Creusillet and F. Irigoin, "Interprocedural Array Region Analyses," *Int. J. Parallel Program.*, vol. 24, pp. 513–546, Dec. 1996.

[62] B. Creusillet and F. Irigoin, "Exact Versus Approximate Array Region Anal-

yses," in *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '96, (London, UK, UK), pp. 86–100, Springer-Verlag, 1997.

[63] C. Bastoul, "A Specification and a Library for Data Exchange in Polyhedral Compilation Tools Edition 1.0, for Openscop Specification 1.0 and Openscop Library 0.8.4," 2012.

[64] P. Feautrier and C. Lengauer, "Polyhedron Model," in *Encyclopedia of Parallel Computing* (D. A. Padua, ed.), pp. 1581–1592, Springer, 2011.

[65] D. G. Wonnacott, *Constraint-based Array Dependence Analysis*. PhD thesis, University of Maryland at College Park, MD, USA, 1995.

[66] F. Quilleré, S. Rajopadhye, and D. Wilde, "Generation of Efficient Nested Loops from Polyhedra," *Int. J. Parallel Program.*, vol. 28, pp. 469–498, Oct. 2000.

[67] D. Barthou *et al.*, "Fuzzy Array Dataflow Analysis," *J. Parallel Distrib. Comput.*, vol. 40, no. 2, pp. 210–226, 1997.

[68] P. Chatarasi, J. Shirako, and V. Sarkar, "Polyhedral Transformations of Explicitly Parallel Programs," in *5th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, (Amsterdam, Netherlands), Jan. 2015.

[69] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen, "Schedule Trees," in *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques* (S. Rajopadhye and S. Verdoolaege, eds.), (Vienna, Austria), January 2014.

[70] Y. Zhang, E. Duesterwald, and G. Gao, "Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers," in *Languages and Compilers for Parallel Computing* (V. Adve, M. Garzarn, and P. Petersen, eds.), vol. 5234 of *Lecture Notes in Computer Science*, pp. 95–109, Springer Berlin Heidelberg, 2008.

[71] M. Griebl, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. Habilitation thesis.

[72] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang, "Symbolic Analysis of Concurrency Errors in OpenMP Programs," in *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, (Washington, DC, USA), pp. 510–516, IEEE Computer Society, 2013.

[73] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, (Washington, DC, USA), pp. 44–54, IEEE Computer Society, 2009.

[74] A. Darte, A. Isoard, and T. Yuki, "Liveness Analysis in Explicitly-Parallel Programs," Research Report RR-8839, CNRS ; Inria ; ENS Lyon, Jan. 2016. Corresponding publication at IMPACT'16 (http://impact.gforge.inria.fr/impact2016).

[75] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat, "Array Dataflow Analysis for Polyhedral X10 Programs," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, 2013.

[76] A. Cohen, A. Darte, and P. Feautrier, "Static Analysis of Open-Stream Programs," Research Report RR-8764, CNRS ; Inria ; ENS Lyon, Jan. 2016. Corresponding publication at IMPACT'16 (http://impact.gforge.inria.fr/impact2016).

[77] T. Yuki, P. Feautrier, S. V. Rajopadhye, and V. Saraswat, "Checking Race Freedom of Clocked X10 Programs," *CoRR*, vol. abs/1311.4305, 2013.

[78] J.-F. Collard and M. Griebl, "Array Dataflow Analysis for Explicitly Parallel

Programs," in *Proceedings of the Second International Euro-Par Conference on Parallel Processing*, Euro-Par '96, 1996.

[79] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, "PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, (Washington, DC, USA), pp. 138–149, IEEE Computer Society, 2015.

[80] A. Pop and A. Cohen, "Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers," in *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, 2010.

[81] "CLANG OMP: CLANG Support for OpenMP 3.1." https://clang-omp.github.io.

[82] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, (Washington, DC, USA), IEEE Computer Society, 2004.

[83] J. Mellor-Crummey, "Compile-time Support for Efficient Data Race Detection in Shared-memory Parallel Programs," in *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, PADD '93, (New York, NY, USA), pp. 129–139, ACM, 1993.

[84] F. Yu, S.-C. Yang, F. Wang, G.-C. Chen, and C.-C. Chan, "Symbolic Consistency Checking of OpenMp Parallel Programs," in *Proceedings of the 13th*

ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems, LCTES '12, (New York, NY, USA), pp. 139–148, ACM, 2012.

[85] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott, "ompVerify: Polyhedral Analysis for the OpenMP Programmer," in *Proceedings of the 7th International Conference on OpenMP in the Petascale Era*, IWOMP'11, (Berlin, Heidelberg), pp. 37–53, Springer-Verlag, 2011.

[86] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: A Verifier for GPU Kernels," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, (New York, NY, USA), pp. 113–132, ACM, 2012.

[87] S. Atzeni, G. Gopalakrishnan, Z. Rakamarić, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "Archer: Effectively Spotting Data Races in Large OpenMP Applications," in *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Chicago*, 2016.

[88] A. J. Dorta, C. Rodriguez, F. d. Sande, and A. Gonzalez-Escribano, "The OpenMP Source Code Repository," in *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP '05, (Washington, DC, USA), pp. 244–250, IEEE Computer Society, 2005.

[89] L.-N. Pouchet and T. Yuki, "PolyBench/C 3.2," 2012.

[90] "Intel Inspector XE." http://software.intel.com/en-us/intel-inspector-xe, 2015.

[91] S. Grauer-gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a High-Level Language Targeted to GPU Codes," in *In Innovative Parallel Computing Conference. IEEE*, 2012.

[92] M. Süßand C. Leopold, "Common Mistakes in OpenMP and How to Avoid

Them: A Collection of Best Practices," in *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming*, IWOMP'05/IWOMP'06, (Berlin, Heidelberg), pp. 312–323, Springer-Verlag, 2008.

[93] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Scalable and Precise Dynamic Data Race Detection for Structured Parallelism," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 531–542, ACM, 2012.

[94] A. Aiken and D. Gay, "Barrier Inference," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, (New York, NY, USA), pp. 342–354, ACM, 1998.

[95] A. Kamil and K. Yelick, "Concurrency Analysis for Parallel Programs with Textually Aligned Barriers," in *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, LCPC'05, (Berlin, Heidelberg), pp. 185–199, Springer-Verlag, 2006.

[96] Y. Zhang and E. Duesterwald, "Barrier Matching for Programs with Textually Unaligned Barriers," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, (New York, NY, USA), pp. 194–204, ACM, 2007.

[97] C.-W. Tseng, "Compiler Optimizations for Eliminating Barrier Synchronization," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, (New York, NY, USA), pp. 144–155, ACM, 1995.

[98] H. Han, C.-W. Tseng, and P. Keleher, "Eliminating Barrier Synchronization for Compiler-Parallelized Codes on Software DSMs," *Int. J. Parallel Program.*, vol. 26, pp. 591–612, Oct. 1998.

[99] R. Surendran, R. Raman, S. Chaudhuri, J. Mellor-Crummey, and V. Sarkar, "Test-driven Repair of Data Races in Structured Parallel Programs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 15–25, ACM, 2014.

[100] R. O'Callahan and J.-D. Choi, "Hybrid Dynamic Data Race Detection," in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, (New York, NY, USA), pp. 167–178, ACM, 2003.

[101] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu, "Efficient Data Race Detection for Distributed Memory Parallel Programs," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 51:1–51:12, ACM, 2011.