Development of Solution-focused Automatic Parallelization Mechanism with Tools and Techniques

By

S. PREMA

(Enrolment No: ENGG 02 2011 04 039)

Indira Gandhi Centre for Atomic Research, Kalpakkam

A Thesis Submitted to the

Board of Studies in Engineering Sciences

In partial fulfillment of requirements

For the Degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

August, 2018

Development of Solution-focused Automatic Parallelization Mechanism with Tools and Techniques

By

S. PREMA

(Enrolment No: ENGG 02 2011 04 039)

Indira Gandhi Centre for Atomic Research, Kalpakkam

A Thesis Submitted to the

Board of Studies in Engineering Sciences

In partial fulfillment of requirements

For the Degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

August, 2018

Homi Bhabha National Institute[1]

Recommendations of the Viva Voce Committee

As members of the Viva Voce Committee, we certify that we have read the dissertation prepared by **Ms. S. Prema** entitled **"Development of Solution-focused Automatic Parallelization Mechanism with Tools and Techniques"** and recommend that it maybe accepted as fulfilling the thesis requirement for the award of Degree of Doctor of Philosophy.

Chairman - **Dr. K. Velusamy**		**Date:** 04/05/2019
Guide - **Dr. B. K. Panigrahi**		**Date:** 4/5/19
Co-Guide - **Prof. Rupesh Nasre**		**Date:** May 15, 2019.
Examiner - **Prof. Supratim Biswas**		**Date:** 04/05/2019
Member 1 - **Dr. B. P. C. Rao**		**Date:** 4/5/2019
Member 2 - **Dr. Sharat Chandra**		**Date:** 04/05/2019
Member 3 - **Dr. Anish Kumar**		**Date:** 4/5/2019
Technology Advisor - **Shri R. Jehadeesan**		**Date:** 04/05/2019

Final approval and acceptance of this thesis is contingent upon the candidate's submission of the final copies of the thesis to HBNI.

I/We hereby certify that I have read this dissertation prepared under my/our direction and recommend that it may be accepted as fulfilling the thesis requirement.

Date: 04-05-2019

Place: KALPAKKAM

Co-Guide May 15, 2019.

Guide 4/5/19

[1]This page is to be included only for final submission after successful completion of viva voce.

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at Homi Bhabha National Institute (HBNI) and is deposited in the Library to be made available to borrowers under rules of the HBNI.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the Competent Authority of HBNI when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

[S. Prema]

DECLARATION

I, hereby declare that the investigation presented in the thesis has been carried out by me. The work is original and has not been submitted earlier as a whole or in part for a degree / diploma at this or any other Institution / University.

[S. Prema]

List of Publications arising from the thesis

Journal

a. Published

1. Analysis of parallelization techniques and tools
   S Prema and R Jehadeesan
   Int'l J. Information Computation Tech (2013) 3(5):471-478

2. A Study on Popular Auto-Parallelization Frameworks
   S. Prema, Rupesh Nasre, R. Jehadeesan, and B. K. Panigrahi
   Concurrency and Computation Practice and Experience, 2019;e5168. https://doi.org/10.1002/cpe.5168)

b. Accepted

1. A comparative study on automatic parallelisation tools and methods to improve their usage
S. Prema, R. Jehadeesan, and B. K. Panigrahi
International Journal of High Performance Computing and Networking (In Press), (2018)
URL http://www.inderscience.com/info/ingeneral/forthcoming.php?jcode=ijhpcn

c. Communicated and under preparation

1. Solution-focused Mechanism for Automatic Parallelization of Sequential Codes
S. Prema, R. Jehadeesan, and B. K. Panigrahi
Journal of Supercomputing (Under Review, June 18, 2018)

2. Elimination of Non-affine Issues in Irregular and General-purpose Program
   S. Prema, Rupesh Nasre, R. Jehadeesan, and B. K. Panigrahi *(To be communicated)*

Conference proceedings

1. Dependency analysis and loop transformation characteristics of auto-parallelizers
   S. Prema, R. Jehadeesan, B. K. Panigrahi, and S. A. V. Satya Murty
   In National Conf. on Parallel Computing Technologies (PARCOMPTECH),IEEE (Feb 2015)
   doi: 10.1109/PARCOMPTECH.2015.7084524.

2. Development of solution based approach for auto-parallelization of serial codes
   S. Prema, R. Jehadeesan, and B. K. Panigrahi
   In 9th DAE-VIE 2016 Symposium on Emerging Trends in IC and Computer Systems,IGCAR, Tamil Nadu (June 2016)

3. Automatic parallelization tools: Potential programming pitfalls and approach to solution
   S. Prema, R. Jehadeesan, and B. K. Panigrahi
   In ICASC, 2016 International Conference on Advances in Scientific Computing, IITM, Chennai (November 2016)

4. Identifying pitfalls in automatic parallelization of NAS parallel benchmarks
   S. Prema, R. Jehadeesan, and B. K. Panigrahi
   In National Conf. on Parallel Computing Technologies (PARCOMPTECH),IEEE (Feb 2017)
   doi: 10.1109/PARCOMPTECH.2017.8068329

[S. Prema]

To my beloved Parents

Acknowledgements

I would like to thank the people who were directly involved in the process of making this thesis a reality.

I am profoundly indebted to my Guide, Dr. B. K. Panigrahi, for his advice and continuous support during my PhD and writing of this thesis. I oblige my deepest gratitude to him, for his constant motivation and helping me in all ways to carry out a collaborative work.

I am extremely thankful to my Co-guide, Prof. Rupesh Nasre, for being a true source of inspiration. I thank him for allowing me to carry out my research work at Indian Institute of Technology (IIT-M) and also for the guidance, constant support, motivation and the opportunities I have received from him during my PhD and writing of this thesis.

I am very thankful to my Technology Advisor, Shri R. Jehadeesan, for the continuous support he has delivered ever since the first day and all throughout my PhD and writing of this thesis. I thank him for his patience, and constant motivation during my PhD.

Besides my advisors, I would like to thank Dr. K. Velusamy, for his invaluable suggestions and helping me in all difficult situations. I would like to thank my Doctoral committee members, Dr. B. P. C. Rao, Dr. Sharat Chandra, and Dr. Anish kumar, for their insightful comments and encouragement.

I would like to thank Homi Bhabha National Institute (HBNI) and Department of Atomic Energy (DAE) for funding my research work. I would like to thank former Director of the Indira Gandhi Centre for Atomic Research (IGCAR) Shri. S. C. Chetal, for offering me the opportunity to carry out my research work at Electronics and Instrumentation Group, Computer Division, IGCAR. I am deeply grateful to the subsequent Directors, Dr. P. R. Vasudeva Rao and Dr. Satya Murty and also to the current Director, Dr. A. K. Bhaduri for allowing me to carry forward my research work in this esteemed institute (IGCAR). I thank former committee member Dr. M. SaiBaba and senior scientific officers Mrs. T. Jayanthi, Mrs. S. Rajeswari, Mrs. Jemimah Ebenezer, and Dr. M. L. Jayalal for their valuable suggestions and constant support.

I would like to thank my thesis reviewers, Prof. Supratim Biswas (IIT-M) and Prof. Rudolf Eigenmann (University of Delaware) for their detailed review which had improved the thesis considerably.

My special words of thanks to Dr. Sanjay Kumar D, who has been a pillar of strength. I thank him for his generous care, invaluable suggestions, and motivation during my PhD.

I express my gratitude to T. S. Shri Krishnan, Manas Ranjan Prusty, Deepika Vinod, and Vikas Kumar, for all their technical helps, suggestions, good company and constant support during the course of my PhD. I thank Dr. Subhra Rani Patra, Pradeep Ch., Prashant Sharma, Vinita Daiya, Molly Mehra, R. Bala Krishnan, M. Harish, and J. Harish for their good company and constant support. I would like to further thank all my colleagues from Computer Division for their support and motivation.

I would like to acknowledge my friends, Dr. Sravanthi, Sumathi V, Madhura, Mohasin, Dr. Greeshma, Roobala, Rajasekar, Dr. Anbu, Dr. Priya dharshini, Dr. Sumathi G, Santhosh Kumar, Naseema, Vijayalakshmi, Veena, Suma, Sumana, Prarthana, Sithara, Pavan and my batch mates for being with me in thicks and thins of life. During the course of my work, I had the pleasure to interact with BITS Pilani students Shikhar and Nikhil. I thank them for their valuable comments.

I thank Vamsi Krishna for his help during my course work at IIT-M. I would like to thank PACERS for their support and suggestions during my stay at IIT-M. I would like to acknowledge Raghesh Aloor for helping me in understanding Rose compiler framework and for all his suggestions. I thank Somesh, Jyothi V, Aman N, Manas, and Jyothi K for all their technical discussions, suggestions, and support. I thank my beloved friends from IIT-M, Nithila, Ramya, Nishanthi, and Sparsha for their constant support and good company during my course of PhD.

Infinite thanks to my wonderful family who mean world to me; Mother Mrs. S. Pandiammal, Father Mr. K. G. Soundrarajan, Brother Mr. S. Mohan Kumar, Sister-in-law Mrs. M. Revathi, Aunt Mrs. T. Mahalakshmi, and Uncle Mr. B. Tamilvanan for all the limitless love I received from them and for being so supportive all times and showing faith in me.

[S. Prema]

CONTENTS

Synopsis

Automatic parallelization is a user-friendly and an optimistic approach for exploiting the performance potential of today's multi-core processors. However, even today auto-parallelizers do not accomplish the consistent performance necessary to be considered true alternatives to manual parallelization. Recent research lacks in the examination of the present-day parallelizers. A complete understanding of such tools is vital for both researchers and developers.

The primary objective of the work is to evaluate the effectiveness of modern parallelizers viz. Cetus, Par4all, Rose, Intel C Compiler (ICC), and Pluto, their capabilities and limitations. In view of the above, a qualitative and quantitative analysis was carried out on the five frameworks under study. The qualitative investigation elucidates the combined effect of parallelization mechanisms, namely, dependence analysis and loop transformations. This study brings out the effect of tuning techniques on different dependence problems. Also, evaluation of the tool's behavior in parallelizing applications with complex coding styles (in total 76 different programming features) were tested to find the potential of auto-parallelizers. The study summarizes the limitations of parallelizing compilers in parallelizing the loops along with potential solutions to improve their effectiveness. The potential solutions increased the overall proficiency of individual frameworks by 10%.

This thesis explains the quantitative effect of five different frameworks on various PolyBench benchmarks (28 codes). This study investigates the reason for the non-parallelized PolyBench codes by each framework. The performance of the transformed code is evaluated by categorizing benchmarks based on different dependence types namely, (i) *loop-independent dependence* (ii) *loop-carried dependence due to scalar/vector*. It was observed that Pluto is a good framework that supports maximum number of codes and parallelizes codes with complex *loop-carried dependence due to vector* using loop peeling. The observations reveal that overall ICC (10.9×) and Pluto (9.4×) outperform all other frameworks.

In addition, the quantitative analysis is carried out on real-world codes: NAS parallel benchmarks (NPB) (10 codes). Pre- and post-transformation changes were applied to make the code amenable to parallelization. The results show that the performance delivered by the frameworks in parallelizing the NPB benchmarks was largely abysmal. Only ICC produced the best overall speedup 2.0× and is equal to baseline optimization (-O2). This work presents a thorough analysis to dig out the reasons for the behavior of the frameworks. The observation shows that all the frameworks fare inferior over NPB suite as the tools forbid parallelization due to many auto-parallelization issues.

In this work, an attempt has been made to address the shortcomings of Pluto, a polyhedral parallelizer, an open-source tool. The observations reveal that it is efficient in performing loop optimization and parallelizing loop with complex dependences. However, Pluto is unable to handle non-affine constructs (NAC), which limits its efficacy. Therefore, a method was introduced to eliminate the NAC in irregular and general-purpose programs to make the code amenable to parallelization. In total, 9 NACs were found by analyzing three distinct real-world benchmark suites namely, Green-Marl, Rodinia, and NPB. The elimination takes place a three-step process (Pre-, In-, and Post-elimination) to address the pitfalls. The correctness of the proposed method was validated, and further, the quantitative results are presented to bring out the usage of this method. Performance analysis shows that Green-Marl (6.6×) and Rodinia (19.9×) show considerable speedup using this method.

A solution-focused automatic parallelization mechanism was introduced to alleviate few of the parallelization pitfalls that occur specifically to Pluto. Overall, five pitfalls are explored and resolved in this work. The proposed method is a three-stage process consisting of *Profiling*, *Analysis and Code Transformation* (ACT), and *Parallelization*. The technique was validated using PolyBench codes. Performance improvement were observed on benchmarks namely `symm` 7.1× and `ludmcp` 4.6×. Overall, the thesis provides a detailed analysis of popular auto-parallelizers, their strengths and limitations, and proposes techniques to address a subset of the limitations.

LIST OF ABBREVIATIONS

ACT	Analysis and Code Transformation
ACT-1	Analysis and Code Transformation (Phase-1)
ACT-2	Analysis and Code Transformation (Phase-2)
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
AST	Abstract Syntax Tree
AU	Array Updation
CANDL	Chunky ANalyzer for Dependences in Loops
CC	Constructor Call
CFD	Computational Fluid Dynamics
CFG	Control Flow Graph
CLooG	Chunky Loop Generator
CUDA	Compute Unified Device Architecture
DECL	Declaration Statements
Dep	Number of Static Dependences
DMD	Dynamic Memory Disambiguation
EDG	Edison Design Group

FC	Function Call
FILE	File I/O operations
GCC	GNU C Compiler
Gprof	GNU profiler
GPU	Graphical Processing Unit
GUI	Graphical User Interface
HPC	High Performance Computing
ICC	Intel C Compiler
ICW	Insufficient Computational Work
ILP	Instruction Level Parallelism
IR	Intermediate Representation
ISL	Integer Set Library
LC	Loop-carried dependence
LLVM	Low Level Virtual Machine
LS	Left-shift operator
MI	Missed Inner-loop parallelism
MPI	Message Passing Interface
MO	Missed Outer-loop parallelism
MS	More number of Statements
NAC	Non-affine Constructs
NAL	Non-affine Loop bound
NAS	Non-affine Array Subscript

NIF	Non-affine IF constructs
NOP	No Optimization reported
NPB	NAS Parallel Benchmarks
NPC	Not a Parallelization Candidate
OpenACC	Open Accelerator
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PDEP	Existence of Parallel Dependence
PERFECT	PERFormance Evaluation for Cost-effective Transformations
PIPS	Parallelization Infrastructure for Parallel Systems
PGI	Portland Group Inc
POCC	Polyhedral Compiler Collection
RAW	Read-After-Write
RS	Right-shift operator
SCoP	Static Control Part
SGI	Silicon Graphics
SI	Scalar Increment
SL	Scalar variable in the ℓ-value of a statement
SMD	Static Memory Disambiguation
SMP	Symmetric Multiprocessing
SR	Scope Resolution Operator
Stmts	Number of Potentially Parallelizable Statements
SUIF	Stanford University Intermediate Format

UL	Upper to Lower bound
UPC	Unified Parallel C
VFC	Vienna Fortran Compiler
WAR	Write-After-Read
WAW	Write-After-Write
YACC	Yet Another Compiler-Compiler

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

This chapter presents a brief introduction to the importance of parallel programming and the need for a fully automated framework. It provides a comprehensive description of the existing auto-parallelizers and their difficulties. This chapter also describes the need to study the effectiveness of modern parallelizer and the motivation to develop a solution-focused automatic parallelization mechanism.

1.1 Automatic parallelization

Quest for performance has made multi-core processors mainstream [1]. It becomes vital for the programmers and algorithm developers to exploit this ubiquitous architecture with parallel programming. The importance and pervasiveness of multi-core processors and parallel programming have significantly increased in the last two decades. Utilizing the potential of multi-core processors through parallel programming is a significant challenge. Amid several approaches to tame this challenge, a very promising and programmer-friendly approach is *automatic parallelization* of sequential codes [2]. Parallel algorithms are developed in the field of numerical simulations, molecular interaction computation, machine learning, big data, and many more real-world problems [3]. Manual parallel programming requires programmer effort in analyzing the region of parallelism which is a time consuming process.

Figure 1.1 depicts a schematic approach for parallelization that elaborates the procedure to generate an efficient parallel code. Taking the general procedures for parallelization into consideration, it is clear that loop parallelization plays a major role since 95% of the execution time is because of loops. Hence, parallelizing the compute-intensive loop improve the speedup of the code. However, the speedup of the application may get affected by load balancing, synchronization, memory usage, dependences in code and many more [4]. One of the important downside is dependences across statement instances, which can be resolved using several tuning techniques.

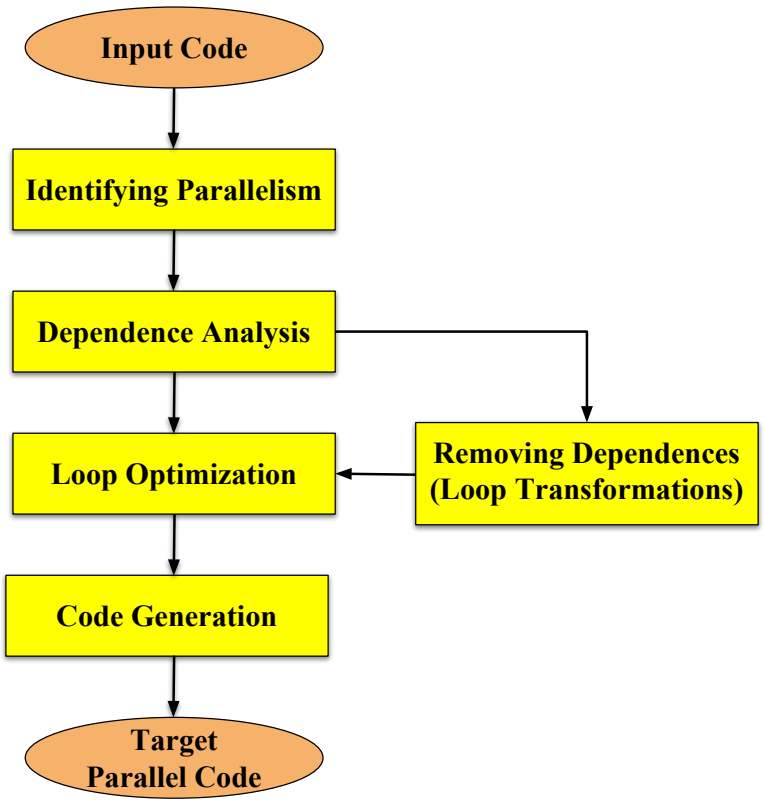


Figure 1.1: Schematic representation of parallelization procedure

A fully parallelizable loop comprising intra-iteration dependence is referred below:

```
1 for j = 1, N
2 A[j] = A[j] + B[j]
```

The code snippet shows that for any value j_1, j_2,...,j_N , we can compute the value of A[j_1]....A[j_N] simultaneously. For instance, the result of A[j_1] can be computed at the same time when A[j_2] is calculated without affecting the net result.

It is evident that all loops are not parallelizable, especially when there exists dependence between statement instances. For instance, the code referred below shows the loop-carried data dependence.

```
1 for j = 1, N
2 A[j] = A[j-1]
```

The resultant A[j], requires the written value from the previous iteration $j-1$. This loop can be parallelized after applying dependence removal techniques. Hence, for a complete

parallelization of the loop, dependence analysis and removal is necessary. This mechanism has become a significant concern for parallelization.

Loop transformation optimizes the code by increasing execution speed and reducing the overhead in loops. In case if the loop iterations are dependant, it is vital to eliminate those dependences by applying appropriate loop transformation techniques and finally generate the parallel code. Auto-parallelizers[i] eliminate the need for a programmer to transform a sequential code into a parallel code, which is quite attractive. However, parallelizers should be capable of performing these sequence of parallelization procedure automatically by retaining the semantic of the code. Therefore, researchers focus on developing a fully automated frameworks to complement manual parallelization [2, 5, 6]. Several of the existing auto-parallelizers such as Cetus [7], Par4all [8], and Rose [9] (parallelizers are listed in Table 2.1 in Chapter 2) perform source-to-source transformation with the insertion of parallel directives. These tools perform the parallelization procedure automatically and do source-to-source transformation in order to reduce the overall workload of programmers. However, there are several reasons these auto-parallelization fails, which this thesis discusses. For instance, complex coding patterns are not handled by existing auto-parallelizers.

1.2 Motivation

Although parallelizers find parallelism, they do not guarantee that the generated parallel code results in increased performance. Hence, automatic parallelization of sequential programs combined with performance tuning is an essential alternative to manual parallelization. Such a fused mechanism exploits the performance potential of today's multi-core processors. The studies related to evaluation of parallelizing compilers; tuning techniques and their importance; and also the investigations that showed irregular coding patterns affecting performance of parallelizers are reported in the literature.

Earlier studies of the parallelizing compilers using "PERFormance Evaluation for Cost-effective Transformations" (PERFECT [ii]) benchmarks performed a detailed analysis of the code restructuring techniques. The techniques include induction variable elimination,

[i] Auto-parallelizers is also referred as tools/frameworks/parallelizers.

[ii] The PERFECT benchmarks is a suite of 13 Fortran77 programs comprising 60000 lines of source code. They represent applications in a number of areas of engineering and scientific computing.

scalar expansion, forward substitution, strip-mining, and loop interchange. The studies found that usage of some of the techniques showed improvements, and that scalar expansion led to positive results [10]. An effectiveness study on Polaris compiler with open multi-processing (OpenMP) [11] parallel code using PERFECT benchmarks proclaimed performance lag in small parallel loops. The work has illustrated the importance of reduction operation, which resulted in a moderate (10%) performance improvement [12]. Another study have been carried out using an automatic parallelizing compiler to parallelize computational fluid dynamics (CFD) codes. It has revealed that automatic code conversion along with user-inserted directives and compiler analysis depicted 2.5 times better speedup than only the user-inserted directives [13]. A few works evaluated the overall performance of parallelizing compilers with improved loop timings [14, 15] due to restructuring techniques. In these works, the restructured loops have shown an execution time lesser than 70% of the vectorized loop.

Toru [16] claims that without any loop transformation a program does not have much coarse-grain parallelism between different loops. Recent research has therefore proposed automatic tuning techniques to be essential part of the auto-parallelizers with the aim to overcome the performance degradation [17, 18, 19, 20, 21]. **These studies do not evaluate the effect of loop transformation techniques such as tiling, fission, fusion, unrolling and peeling on parallelization.**

Apart from dependence issues, the primary reason for incomplete parallelization by auto-parallelizers is due to complex coding style. Earlier parallelizing compilers such as Paradigm [22], Polaris [23], Rice Fortran D [24], Stanford University Intermediate Format (SUIF) [25], and Vienna Fortran Compiler (VFC) [26] could easily parallelize Fortran codes. The reason is Fortran provides stronger guarantees about aliasing than languages such as C which support arbitrary usage of pointers. However, after the development of parallelizers based on C (discussed in Chapter 2), and due to the complex usage of the struct, pointers, and arrays, the codes have become unanalyzable by auto-parallelization frameworks. It sometimes stops the code from parallelization abruptly without spotting the source of error. Eventually, such a cause leads to performance degradation.

PETRA, a portable performance evaluation tool, found that parallelizers are reasonably successful in about half of the given science-engineering programs. It also illustrated that some of the algorithms are dependence-free, but the complex program constructs led to a conservative analysis [27].

Detailed literature survey on the existing parallelization techniques and parallelizing compilers concludes that it is necessary to investigate the ability of the tools in addressing loop parallelization issues [27, 28]. Presently, there are many open-source and commercial tools available that are widespread namely, Cetus [7], Par4all [8], Rose [9], Intel C compiler (ICC) [29], and Pluto [20] (discussed in Chapter 2). **A complete understanding of such tools is vital for both researchers and developers. So far no such examination has been made on present-day parallelizing compilers and its underlying techniques.** Detailed study on the support of different programming constructs by popular tools is unavailable. **Empirical evidence of the common loop transformation techniques and their effectiveness in parallelization of popular benchmarks is missing.** Hence, such a study would be extremely beneficial to identify the relative merits and demerits of various widely-used auto-parallelization frameworks. In addition, it would help the researchers or users in choosing the appropriate framework to improve the performance of an application.

In this thesis, an attempt has been made to address the shortcomings of the Pluto parallelizer. It is found that polyhedral parallelizers such as Pluto are powerful and perform various optimizations and transformations for improved performance. However, for the real-world problems arising due to several non-affine issues and general limitations, the code becomes unanalyzable by these frameworks. Hence, a method was introduced to eliminate the non-affine issues in irregular and general-purpose programs to make the code amenable to parallelization. Another solution-focused automatic parallelization mechanism was introduced to alleviate a few of the parallelization pitfalls that occur specifically for the polyhedral parallelizer. In this approach, we perform transformation by modifying the abstract syntax tree (AST) to elucidate the problems. The above two approaches are aimed at solving the problems pertaining to the polyhedral parallelizer, Pluto, which is an open-source tool.

1.3 Objectives

Automatic parallelization using OpenMP programming model is the focus of the present study, and the objective of the thesis are as follows:

1. To bring out the importance of parallelization mechanism (dependence analysis and loop transformation) in automatic parallelization.

2. To provide a comprehensive performance evaluation of modern parallelizing compilers namely, Cetus, Par4all, Rose, ICC, and Pluto and their underlying parallelization techniques.

3. Finding the capabilities and limitations of the above-mentioned auto-parallelization frameworks in order to bring out their ability in handling different dependence problems. Also to study the way in which the loop transformation techniques are applied, and complex coding styles are handled by auto-parallelizers.

4. To bring out the limitations of polyhedral parallelizer, Pluto and to eliminate non-affine problems in Pluto during parallelization of irregular and general-purpose programs.

5. To develop a new solution-focused auto-parallelization mechanism in addressing the parallelization limitations pertaining to the polyhedral parallelizer, (Pluto).

1.4 Organization of the thesis

Remaining part of the thesis is organized into seven chapters. The details of the content in each chapter are:

- Chapter 2 presents the literature review on the study of existing parallelizing compilers and their techniques. It also brings out the significance of loop transformation techniques. The auto-parallelizers from past two decades are listed, compared and

discussed in detail. The reason for choosing five auto-parallelizers namely Cetus, Par4all, Rose, ICC, and Pluto for the study are illustrated.

- Chapter 3 presents the importance of parallelization mechanisms i.e. dependence analysis and loop transformations. Chapter 3 also brings out the different dependence types (*loop-independent* and *loop-carried*) and the way in which tuning techniques are applied to handle such dependences. It also summarizes the limitations of parallelizing compilers in handling different complex programming constructs. Further, it discusses the potential solutions derived for improving the usage of tools based on synthetic codes.

- Chapter 4 emphasizes the quantitative effect on PolyBench suite (comprises 28 distinct applications) by five different auto-parallelizers. The support matrix of the frameworks in parallelization of PolyBench codes is explained in detail. The effectiveness of the transformation was examined and explained in this chapter. It discusses performance by categorizing benchmarks based on different dependence types namely, (i) *loop-independent dependence* (ii) *loop-carried dependence due to scalar/vector*.

- Chapter 5 emphasizes the quantitative effect on NAS parallel benchmarks (NPB) (comprises 10 distinct applications) by five different auto-parallelizers. It describes the support matrix of NPB over different frameworks. This chapter also describes the detailed analysis to elucidate the causes for the tools inability in parallelization. In-depth evaluation analysis of NPB results are brought out.

- In Chapter 6, the real-world codes namely Green-Marl, Rodinia, and NPB are examined and in total nine parallelization issues pertaining to Pluto tool are emphasized, viz. 1) scalar variables in ℓ-value of a statement, 2) constructor calls, 3) function calls, 4) declaration statements, 5) non-affine loop bound, 6) non-affine IF construct,

7) scope resolution operator, 8) non-affine array subscripts, and 9) file I/O operations. This chapter introduces a method to eliminate these issues to make the code amenable to parallelization using Pluto. It is a three-step process that involves pre-elimination, in-elimination, and post-elimination. The performance evaluation of the parallelized code is brought out that highlights the usage of this method.

- In Chapter 7, an automatic methodology has been developed to address the complications in the polyhedral parallelizer, Pluto. Overall, five pitfalls are explored and resolved in this chapter, namely, (i) while-loop (ii) scalar variables in the ℓ-value (iii) reverse for-loop (iv) for-loop updation by more than one-time step, and (v) irregular loop. It discusses three phases of the proposed method namely, *Profiling*, *Analysis and Code Transformation* (ACT), and *Parallelization*. Further, the performance analysis on the parallelized code are also studied in detail.

- Chapter 8 presents the highlights of this dissertation, conclusion derived, important findings and illustrates the scope for the future work.

CHAPTER 2

Survey on Existing Parallelizing Compilers and their Primary Techniques

This chapter discusses the literature review on the existing parallelizing compilers and their techniques. It also introduces five parallelization frameworks namely, Cetus, Par4all, Rose, Intel C compiler (ICC), and Pluto. It discusses in detail the underlying methodologies of these frameworks with a qualitative analysis.

2.1 Related works

Earlier and recent studies focus towards automatic parallelization. The goal of automatic parallelization is to complement hand-optimized code. Researchers have provided empirical evidence on the performance metrics of earlier parallelizing compilers, and also the importance of loop optimization techniques. Following sections briefly discusses the literature survey carried out.

2.1.1 Earlier studies on the performance evaluation of auto-parallelizers

Several studies had been conducted to evaluate the performance of auto-parallelizers. Nobayashi and Eoyang [14] performed comparative analysis on automatic vectorizing compilers. Their work illustrates that loops which use restructuring techniques enable the compiler to achieve greater than 70% speedup than the vectorized loops. Shen et al. [15] presented an evaluation of parallelizing compilers which focuses on data dependence analysis and a few parallel execution techniques. Several studies [10, 12, 30, 31] assessed the effectiveness of parallelizing compilers and their techniques using PERFECT benchmarks.[i] The

[i]The PERFECT benchmark is a suite of 13 Fortran77 programs consisting of 60000 LOC. They represent applications in many areas of engineering and scientific computing.

studies emphasized that scalar expansion and reduction replacement are critical in achieving significant performance gains in a large portion of the benchmark suite.

Hisley et al. [13] evaluated the effectiveness of silicon graphics (SGI) compiler in parallelizing CFD codes. The authors have reported that combining automatic parallelization capabilities of the compiler with user-inserted compiler directives showed nearly 2.5 times better speedup than only the user-inserted directives.

Eigenmann et al. [5] studied the Kap and Vast parallelizing compiler using PERFECT benchmarks. They assessed that parallelization of reduction operations and the substitution of generalized induction variables were the primary contributors to performance.

This thesis has similar goals for the modern set of parallelizing frameworks, and it assesses them by focusing primarily on the various kinds of loop dependences

2.1.2 Automatic parallelization techniques on irregular and general-purpose programs

Scientific applications for high performance computing (HPC) can be incredibly complex, in general, it contains irregular control constructs with complicated dependence structures. Therefore, it is difficult for the compilers to analyze and to perform optimizations [27, 32]. Such issues are especially applicable for source-to-source parallelizers.

Earlier works deal with improving the data dependence analysis techniques for program parallelization in irregular applications. Irregular codes comprising loop bounds or array subscript that are non-linear expressions were not handled by traditional dependence tests. Blume et al. [33, 34] proved such inequalities using range test, powerful symbolic analysis and constant propogation techniques. The effectiveness of range test were analyzed and the results had shown that optimized codes were close to the hand parallelized versions with maximum speeudp of $43\times$. Complementing to the work of [33] and [34], Kyriakopoulos et al. [35] and [36] had introduced set of polynomial-time techniques to identify and handle complex expressions such as nonlinear and symbolic expressions, complex loop bounds, arrays with coupled subscripts, and if-statement constraints. Evaluation results show that this technique had produced the highest degree of parallelization among all data dependence

tests. Banerjee et al. [37] presented an overview of automatic program parallelization techniques which covers dependence analysis techniques. The authors have surveyed several experimental studies on the effectiveness of parallelizing compilers. Automatic parallelization techniques that focus on irregular and general-purpose programs were developed and studied earlier [28, 38, 39, 40].

In this dissertation, modern parallelizers are examined using NAS parallel benchmarks (NPB), Green-Marl, and Rodinia, that comprises of several problems including irregular patterns, imperfectly nested loops, dependences, and other code complexity issues. The inability of the tools on parallelization of complex coding problems are investigated, and potential solutions were recommended. The situations where user-intervention was needed is highlighted.

2.1.3 Loop transformation and its significance

Earlier work on the tuning of automatically parallelized applications has described the significance of loop transformations [16, 18, 19] which helped in enhancing coarse-grain parallelism. The authors report that tuned application programs showed better results than only parallelize code. Rauchwerger [41] discussed the two essential and useful transformations namely, privatization and reduction parallelization applied to the loop. Liu et al. [42], Li [43], Tu and Padua [44] revealed that array privatization showed good performance improvement. Kim et al. [45] examined the performance analysis of Polaris compiler using PERFECT benchmarks. The authors found that reduction operation can make a significant improvement in speedup.

The evaluation results in this study also found that privatization and reduction variable recognition result in substantial performance improvement during auto-parallelization.

Bacon et al. [46] evaluated a large number of compiler transformations and high-level program restructuring techniques. The authors demonstrated that these transformations could yield high performance when applied appropriately. Studies on loop fusion [47, 48, 49] and loop fission [50] have shown improved parallelism due to data locality. Also, loop fusion reduced the use of temporary arrays. Loop unrolling is a well-known loop transformation used in optimizing compilers; it improves instruction level parallelism.

Davidson et al. [51] and Sarkar et al. [52] evaluated the importance of loop unrolling. In [51], dynamic memory disambiguation (DMD) in conjunction with loop unrolling, register renaming, and static memory disambiguation (SMD) showed an increase in the instruction level parallelism (ILP) of memory intensive benchmarks by as much as 300% over loops. In [52], the code generated by IBM XL Fortran compiler has shown a more substantial performance improvement of 2.2× speedup on matrix multiply, and an average 1.08× speedup on seven of the SPEC95fp benchmarks due to unrolling of nested loops. A set of former works [21, 53] shows the performance impact of loop tiling which provides load balance, locality, and parallelism. Another important loop transformation technique is loop peeling [54, 55] which relies on moving computations in early iterations out of the loop body such that the remaining iterations execute in parallel.

In this study, the polyhedral parallelizer, Pluto supports loop peeling. They are more powerful [56, 57] in performing automatic optimization, parallelization, and dependence analysis [58]. In this thesis, experimental evidence shows that the use of the model in parallelization has shown a significant gain in speedup.

The close to this thesis is a study by Mustafa and Eigenmann [27]. These authors evaluated the performance of five parallelizing compilers (Cetus, OpenUH, Rose, Portland Group Inc (PGI), and ICC). They studied the individual techniques of the parallelizers. Performance studies on Cetus revealed that the parallelized code along with tuning proves beneficial in achieving considerable performance [17, 27].

In addition to individual techniques of the parallelizers, the common loop transformation techniques were also examined namely loop tiling, loop fusion, loop unrolling, loop peeling, and loop fission. In this thesis, polyhedral parallelizer, Pluto, and parallelization infrastructure for parallel systems (PIPS) based parallelizer, Par4all are studied in detail.

Overall, this dissertation contributes in (i) performance evaluation of modern parallelizing compilers, (ii) investigating the ability of the tools in addressing the dependence structures, (iii) illustrating the role of loop transformation techniques in gaining speedup along with parallelization, (iv) studying the auto-parallelizer behavior in parallelizing regular as well as irregular programs, and (v) introducing methods to improve the usage of Pluto tool.

2.2 Auto-parallelizers from past two decades

Over the past two decades, researchers and developers have proposed different types of frameworks (automatic/semi-automatic/model) in order to reduce the manual effort.

Table 2.1 shows the list of some of the research and industrial compilers that were contributed to the parallel computing community. The table also illustrates the comparative study on the following tools, Cetus, Par4all, Pluto, Rose, ICC, Parallware, low level virtual machine (LLVM) Polly, ParaWise, ParaGraph, Stanford university intermediate format (SUIF), gnu C compiler (GCC) Graphite and Polaris. These frameworks support input languages, C/FORTRAN/C++ and generate target parallel code, OpenMP/OpenCL/multi-threaded code/open accelerator (OpenACC). Some tools generate compiler specific target code namely, Polaris, SUIF and LLVM Polly. Among these surveyed frameworks, the majority support symmetric multiprocessing (SMP), while few support graphical processing unit (GPU). Practical implementation of the message passing interface (MPI) [73] based tool is unavailable except ParaWise which is a semi-automatic and proprietary software. Proposed models on the development of OpenMP based tool, ParaGraph; and MPI based tool, SUIF is not implemented. Although these existing parallelizers offer considerable benefits, they still fall short of attaining manual transformation performance. In many programs, parallelizing compilers are only finding partial parallelism. Several parallelizers do not utilize all the static information available, while several others fall short of modeling precision. It leads to either missed parallelism opportunities or unwanted parallelization of sequential codes.

As a consequence, compared to the original sequential version, the auto-parallelized code may lead to parallelization overheads and may exhibit poorer performance. Hence, this thesis also focuses towards finding the capabilities and limitations of modern auto-parallelizers. The following section briefly describes the currently active OpenMP based parallelizers under study.

Table 2.1: Comparisons of auto-parallelizers from past two decades reported in the literature

Tools	L.D.	Input	Target	Availability	Reference
Pluto	2018	C	OpenMP, CUDA	Open-source	[20, 21]
LLVM Polly	2018	C	LLVM IR code	Open-source	[59]
ICC	2018	C & FORTRAN	Multithreaded code	Proprietary	[29]
Rose	2018	C & C++	OpenMP	Open-source	[9, 60]
Parallware	2017	C	OpenMP, OpenACC	Proprietary	[61, 62]
Cetus	2017	C	OpenMP	Open-source	[7, 63, 64, 65]
ParaWise*	2016	C & FORTRAN	OpenMP, MPI	Proprietary	[66]
Par4all	2015	C & FORTRAN	OpenMP, CUDA, OpenCL	Open-source	[8, 67]
GCC Graphite	2012	C	Multithreaded code	Open-source	[68]
ParaGraph	2010	C	OpenMP	Model	[69]
Polaris	2002	FORTRAN	KAP	Proprietary	[23, 70]
SUIF	1999	C & FORTRAN	SUIF	Open-source	[25, 71]
SUIF	1999	C & FORTRAN	MPI	Model	[72]

L.D. = Last Development, Input = Input language supported, Target = Generated target parallel code, Model = Model proposed and no implementation, *Semi-automatic parallelizer

Table 2.2: Popular auto-parallelizers and their methodologies reported in the literature

Tools	Dependence Analysis	Framework(s) Used
Cetus	Banerjee [7], Range [7]	ANTLR [74], YACC [75], Bison [76]
Par4all	Dependence Graph [67]	PIPS Framework [77]
Rose	Dependence Graph [60]	EDG Front-end [60]
ICC	Data Flow Analysis [78]	Proprietary Parallelization Framework
Pluto	ISL [79], CANDL [80]	Cloog [81], Clang [82], Openscop [83]

2.3 Overview of the modern auto-parallelizers and their characteristics

Five parallelization frameworks viz. Cetus, Par4all, Rose, ICC, and Pluto are introduced along with their methodologies. The evaluation study in the upcoming chapters uses these five parallelizers. The selection criteria for these tools are (i) OpenMP based (ii) widely used and, (iii) non-compiler specific. Typically OpenMP supports for-loop parallelization.

Table 2.2 presents an overview of auto-parallelizers and its underlying methodologies viz. dependence techniques and the frameworks adopted by distinct tools. Note: The methodologies described in the table are obtained from the respective papers pertaining to the tools. While all others are source-to-source transformers, ICC works at the intermediate representation (IR) level and does not reveal the transformed code. It prints explicitly a summary of parallelized transformations performed.

2.3.1 Cetus

Cetus is a source-to-source built-in C parser written in another tool for language recognition (ANTLR) with intermediate representation (IR) classes and optimization passes. Cetus has a more memory-efficient IR, and it parses the source code quickly [65]. It has an array data-dependence testing framework that includes loop-based affine array-subscript

testing. Cetus uses Banerjee-Wolfe inequalities [84, 85] for dependence testing; utilizes yet another compiler-compiler (YACC) and Bison for parsing.

Cetus [7, 63, 65] performs analysis and transformation passes which include a set of general passes and parallelization passes. It performs the following checks to identify if a loop is parallelizable: (i) whether the loop is a canonical loop, (ii) whether function calls within the loop are without side-effects, (iii) whether control-flow modifiers are not present within the loop body, and (iv) whether the loop increment is an integer constant. Cetus includes the following general passes and parallelization passes:

- The symbolic analysis that manipulates symbolic expressions, which is essential while dealing with real programs
- Points-to alias and use-def chain analyses [7]
- Function inlining in place of complete inter-procedural analysis and transformation [7]. However, the parallel coverage tends to be less when using selective inlining.
- Several parallelization passes such as induction variable recognition and substitution, reduction recognition and transformation, scalar and array privatization [42, 43, 86], data dependence analysis [86], loop parallelizer, and code generator [63] are available in Cetus.

Not every loop should be parallelized. Cetus performs profitability tests for eliminating the smaller loops that are likely to cause overheads. The framework does not parallelize loops with a workload below a certain threshold. If the workload expression is not evaluated at compile time, the technique uses a runtime OpenMP IF construct technique, called *model-based profitability test*. Cetus does not support nested parallelism and parallelizes only the outer loop. In addition, Cetus has an easy-to-use graphical user interface (GUI) which makes the parallelization process user-friendly.

2.3.2 Par4all

Par4all [8, 67] is a source-to-source compiler which supports C and FORTRAN as the input language and can generate OpenMP or compute unified device architecture (CUDA) [87] or open computing language (OpenCL) [88] code. It follows the parallelization infrastructure for parallel systems framework (PIPS) [77]. PIPS and programmable pass manager together perform various program analyses, locate parallel loops and generate parallel codes [8]. It delivers advantage from its inter-procedural capabilities such as memory effects [89, 90], reduction detection, parallelism detection, and also from polyhedral analyses such as convex array regions [91] and preconditions. Par4all exploits polyhedral compiler collection (POCC) [92] for optimizing loops. It supports memory access transformations to improve locality [8].

In order to obtain a good balance between portability and performance, Par4all relies on an intermediate representation of the input program. It allows increasing code portability also enabling intermediate code inspection and editing to simplify both debugging and low-level code optimization. Par4all is very efficient for low-level optimization and code generation purposes, primarily when it collects coding rules (e.g., no pointers) and hints (e.g., allocation of functions onto resources) that simplify its work to apply parallelization operations. Par4all implements fine-grained parallelism [93]. Similar to Cetus, Par4all does not support nested parallelism. However, unlike Cetus, Par4all may choose to parallelize an inner loop depending upon an inbuilt heuristic cost function.

2.3.3 Rose

Rose [9, 60] is an open-source compiler framework to build source-to-source code transformations for C, C++, FORTRAN, OpenMP, and unified parallel C (UPC) applications. Rose comprises multiple front-ends, implemented using Edison Design Group (EDG), a mid-end operating on its internal IR, and backends regenerating (unparsed) source code from IR. The IR encompasses abstract syntax tree (AST), control flow graphs (CFG), and symbol table with various interfaces for quickly building source-to-source translators. Rose also includes several program analyses such as control-flow and data-flow analyses, data dependence, and system dependence analyses. Rose has developed a wide range

of transformations and optimizations via manipulating AST, including partial redundancy elimination, constant folding, inlining, outlining, automatic parallelization, and various loop transformations. Rose comprises the autoPar tool, which is an implementation of automatic parallelization using OpenMP. It can automatically insert OpenMP 3.0 directives into serial C/C++ codes. In addition, Rose can also perform several loop transformations such as fusion, fission, interchange, unrolling, blocking, privatization and reduction.

2.3.4 ICC

ICC, a compiler from Intel [29], is capable of performing powerful code transformations, including auto-parallelization. The code transformations and optimizations in ICC mainly include code restructuring and inter-procedural optimizations, automatic parallelization, and vectorization. It comprises high-level and scalar optimizations such as loop control and data transformations. Unlike other tools, ICC is not a source-to-source parallelizer. It examines the data-flow of the code fragments and generates multi-threaded code. ICC relies on an analytic engine for deciding when it is profitable to parallelize a loop. Through a user-defined threshold, ICC tries to distribute the overhead of creating multiple threads versus the amount of work at hand to be shared among the threads. ICC requires the number of iterations to be known before entry into a loop; if not, the compiler treats the loop to be computationally less intensive for parallelization. Also, the compiler performs data-flow analysis to ensure correct parallel execution. ICC performs loop blocking, vectorization, data prefetch, scalar replacement, data alignment, optimization level (O_n), and subroutine inlining [29].

2.3.5 Pluto

The polyhedral parallelizer, Pluto [20, 21] is designed for programs that deal with linear algebra, linear programming, and high-level transformations. When the data access functions and loop bounds are affine combinations of the enclosing loop variables and parameters, then the polyhedral model is relevant for loop analysis and transformations. Pluto uses the scanner, parser, and dependence tester from the LooPo infrastructure, which is a polyhedral source-to-source transformer. The portion of code to be parallelized is called the static control part (SCoP) [94] and it contains essential information to build a complete

Table 2.3: Compiler options for auto-parallelization of sequential code

Parallelizer	Compiler command to generate parallel code
Cetus	via GUI*
Par4all	p4a <input>
Rose	autoPar -c <input>
ICC	icc -parallel -par-report:3 <input>
Pluto	polycc –parallel <input>

* No command line interface

source-to-source framework in a polyhedral model [83]. The Clan tool [94, 95] acts as a front end and automatically translates a SCoP to an OpenScop [83], which is a polyhedral, matrix-based representation. 'Chunky loop generator' (CLooG) [81] works as the back-end compiler which improves code locality in the generated parallel code with several program transformation facilities and scheduling [96]. It creates the loop by scanning the Z-polyhedra output from Clan. Pluto uses integer set library (ISL) [79] by default to compute dependences. Dependence analysis can be performed by using 'chunky analyzer for dependences in loops' (CANDL) [56]. Pluto performs affine transformation analysis for efficient loop tiling, loop fusion, and loop unrolling. It implements multipipe which extracts multiple degrees of parallelism [97]

2.4 Qualitative study of auto-parallelization frameworks

Investigating the differences in parallelization of auto-parallelizers under study is necessary. Hence, this section provides a qualitative comparison of various features supported by the frameworks. First, a simple example parallelized by all the frameworks is presented, and then the variations in their support functionality are highlighted (Figure 2.1).

Figure 2.1(a) shows a sequential for-loop with loop-independent (intra-iteration) dependence. The qualitative comparison illustrates how this small example is transformed

by each of the frameworks, namely, Cetus (version 1.3.1), Par4all (1.4.6), Rose (0.9.9.47), ICC (15.0.0) and Pluto (0.11.4). Note: ICC does not output the transformed source code, but works at the IR level. For each of the tools' transformed OpenMP codes, a snippet of the corresponding Intel assembly code generated by gcc is also presented. Table 2.3 lists the compiler options that were used to obtain the parallelized code with the aid of various parallelizers.

Figure 2.1 shows that all tools can parallelize this simple loop (Figure 2.1(a)), and insert the OpenMP pragma for the for-loop. However, there are variations in the transformations even for this simple example. These variations stem from the techniques used by the tools, which are discussed next.

Cetus: The transformed code from Cetus shown in Figure 2.1(b) contains an if condition, which is an artifact of a cost model which Cetus uses to identify if a loop is parallelizable (section 2.3.1). This technique forbids parallelization of codes with very few iterations and accesses and compensates for the overhead of thread-creation, etc. The assembly code shows the comparison with the constant value 10000 which is the default threshold cost.

Par4all: Unlike others, Par4all does not explicitly mark the loop-index i to be a private variable. While this does not change the behavior, as, by default, the loop indices are considered private, this shows that Par4all does not look for opportunities for privatization. It forces the runtime to assume variables to be shared, necessitating synchronization conservatively. Besides, it might lead to an overall reduction in their performance. The assembly code listed in Figure 2.1(c) shows that the OpenMP compiler inserts a call to GOMP_parallel function[ii]

Rose: Rose performs transformation of the code similar to other frameworks as depicted in Figure 2.1(d), except for the usage of firstprivate. Usage of firstprivate marks variables n, C, and E as private, but copies their initial values from the outer scope (from

[ii]This is invoked when gcc is linked with -fopenmp. Call to this function generates efficient OpenMP applications.

(a) Original code

```
for(i = 0; i < n; i++)
{
    A[i] = B[i] + C;
    D[i] = A[i] + E;
}
```

(b) Cetus

```
#pragma omp parallel for \\
if((10000<(1L+(4L*n)))) \\
private(i)
for (i=0; i<n; i++ )
{
A[i]=(B[i]+C);
D[i]=(A[i]+E);
}
```

```
...
cmpq    $10000, %rax
setg    %dl
movq    -144(%rbp), %rax
movq    %rax, -64(%rbp)
movq    -128(%rbp), %rax
movq    %rax, -56(%rbp)
movl    %edx, %eax
xorl    $1, %eax
movzbl  %al, %edx
leaq    -64(%rbp), %rax
movl    $0, %ecx
movq    %rax, %rsi
movl    $main._omp_fn.0, %edi
call    GOMP_parallel
...
```

(c) Par4all

```
#pragma omp parallel for
  for(i=0; i <= n-1; i+=1)
  {
    A[i] = B[i]+C;
    D[i] = A[i]+E;
  }
```

```
...
movq   %rax, -104(%rbp)
movq   -144(%rbp), %rax
movq   %rax, -64(%rbp)
movq   -128(%rbp), %rax
movq   %rax, -56(%rbp)
leaq   -64(%rbp), %rax
movl   $0, %ecx
movl   $0, %edx
movq   %rax, %rsi
movl   $main._omp_fn.0, %edi
call   GOMP_parallel
...
```

First column denotes the transformed code and **second column** denotes the assembly code

Figure 2.1: Parallelization of ((a) Original code) by various frameworks (b) Cetus (c) Par4all, (continued in next page)

(d) Rose

```
#pragma omp parallel for \\
private(i) firstprivate(n,C,E)
 for (i=0; i <= n-1; i+=1) {
    A[i] = B[i] + C;
    D[i] = A[i] + E;
  }
```

```
...
call    GOMP_parallel
movq    -64(%rbp), %rax
movq    %rax, -136(%rbp)
movq    -56(%rbp), %rax
movq    %rax, -128(%rbp)
...
```

(e) ICC

```
LOOP BEGIN at sample.c(34,3)
<Peeled>
LOOP END
LOOP BEGIN at sample.c(34,3)
   remark #17109 : LOOP WAS AUTO-PARALLELIZED
   remark #17101 : parallel loop shared={ } private={ }
   firstprivate={ B C A E D i } lastprivate={ } \\
   firstlastprivate={ } reduction={ }
LOOP END
```

```
...
call    __kmpc_ok_to_fork
testl   %eax, %eax
je      ..B1.15         # Prob 50%
movl    $L_main_27__hpo_threaded_loop0_2.26, %edx
lea     (%rsp), %rcx
movl    $.2.3_2_hpo_loc_struct_pack.28, %edi
...
movq    $0, 8(%rsp)
call    __kmpc_fork_call
...
```

(f) Pluto

```
if (n >= 1) {
  lbp=0;
  ubp=n-1;
#pragma omp parallel for \\
private(lbv, ubv, t2)
  for (t1=lbp;t1<=ubp;t1++)
  {
    A[t1] = B[t1] + C;;
    D[t1] = A[t1] + E;;
  }
}
```

```
...
movl  %eax, -16(%rbp)
movq  -120(%rbp), %rax
movq  %rax, -192(%rbp)
movq  -112(%rbp), %rax
...
leaq  -192(%rbp), %rax
movl  $0, %edx
movq  %rax, %rsi
movl $main.omp_fn.0, %edi
call GOMP_parallel_start
...
```

First column denotes the transformed code and **second column** denotes the assembly code

Figure 2.1: Parallelization of ((a) Original code) by various frameworks (d) Rose (e) ICC (f) Pluto (Contd.)

outside the parallel scope). The assembly listing shows that there are several movq instructions *after* the call to GOMP_parallel which is a deviation compared to other parallelizers.

ICC: ICC reports only the diagnostic information. However, this feature can be controlled by using `-par-report:n`. When `n = 3`, it informs the auto-parallelizer to report the diagnostic messages for loops that were successfully and unsuccessfully auto-parallelized. The report provides additional information about any proven or assumed dependences inhibiting auto-parallelization. Similar to Rose, ICC also uses `firstprivate` clause. However, the assembly code for ICC looks quite different compared to other frameworks; this is primarily due to ICC working at a lower level code form (IR) compared to other tools (source). It uses KMPC library routines to create threads (Figure 2.1(e)). Interestingly, it also inserts a comment (`# Prob 50%`) about branch prediction.

Pluto: Pluto does not use a cost model, and in fact, inserts a condition $n >= 1$ as shown in Figure 2.1(f). Thus, it does not care for the thread creation overhead. However, the number of private variables used by Pluto is relatively large. The assembly code reveals that there are several movq instructions added for the additional private variables. Efficient execution of such a transformed code relies heavily on compiler optimizations for removing the unnecessary temporaries.

Thus, it could be concluded that different frameworks apply different transformations to the same code. However, the transformation differs in their abilities to use the cost model, or to identify the thread-private variables, or to work at a lower level of the program representation.

2.5 Summary

In this chapter, existing parallelization methodologies and their limitations when applied by different tools were discussed in detail. In total five parallelization frameworks viz. Cetus, Par4all, Rose, ICC, and Pluto were introduced. Additionally, the usage of individual methodologies of tools was also discussed in detail. Further, the underlying methodologies of an individual framework are also explained. A qualitative analysis was carried out to uncover their support functionality.

It was observed that all frameworks primarily deal with loop parallelization but differ in the techniques in which they transform the code within the parallelizable region. From this chapter, we can conclude that the empirical analysis on modern parallelizing frameworks and their underlying techniques are sorely missing. Therefore, it is important to investigate the current auto-parallelizing compilers and also to understand their behavior.

CHAPTER 3

Qualitative Capabilities of Auto-parallelizers

This chapter presents the two important parallelization mechanisms namely, dependence analysis and loop transformation techniques. The common loop transformation techniques and their support in different parallelizers are well illustrated. This chapter describes the effect of different dependence problems and the techniques adopted by auto-parallelizers in handling these issues. It presents the capabilities and limitations of auto-parallelizers in handling loops with complex coding style.

3.1 Parallelization mechanisms

This section discusses the combined effect of two vital parallelization mechanisms, (i) loop transformation techniques and, (ii) dependence analysis using the five auto-parallelizers namely, Cetus, Par4all, Rose, Intel C compiler (ICC), and Pluto. In general, optimizing compilers have two main phases, program analysis and program transformation. There are different program analysis techniques such as data dependence analysis, flow analysis, alias analysis, and symbolic analysis. Our focus is toward showing how loop transformation techniques effect on dependence problems. Hence we restrict our study to dependence analysis.

3.1.1 Loop transformation techniques

Loop transformation and its significance have been reported in the literature (discussed in Chapter 2). This section illustrates the behavior of modern auto-parallelization in applying common loop transformation techniques. In general, loop optimization is the process of increasing execution speed and reducing the overhead associated with the loops. This can be performed by using sequence of specific loop transformation techniques. These techniques play a vital role in making effective use of parallel processing capabilities. However,

Table 3.1: Tools support matrix for loop transformation techniques

	Priv.	Fission	Peeling	Fusion	Red.Var.	Tiling	Unrolling	Ind.Var.
Tool	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**
Cetus	✓	✗	✗	✗	✓	✗	✗	✓
ICC	✓	✓	✓	☑	✓	☑	☑	✗
Par4all	✓	✗	✗	✓	✓	☑	☑	✗
Pluto	✓	✗	✓	✓	✗	☑	☑	✗
Rose	✓	☑	✗	☑	✓	☑	☑	✗

✓ is Parallelized (default), ✗ is Not Parallelized, Priv. is Privatization, Red.Var. is Reduction Variable Recognition, Ind.Var. is Induction Variable Substitution, ☑ is Parallelized after getting enabled via the command line

such a transformation generally must preserve the semantics of the original code (be a legal transformation) [46]. Table 3.1 illustrates the common loop transformation techniques and their support using the parallelizers under study.

Privatization

Privatization creates a private copy of a shared variable, thereby removing data races [98]. Privatization reduces thread interference but needs to be applied carefully to ensure correctness. All frameworks inherently perform privatization. Figure 3.1 illustrates an example of privatization performed in Rose.

Loop Peeling

Loop peeling is a transformation technique which helps in removing *loop-independent* dependences, by splitting the dependent portion of the loop outside the parallel scope [46] [85] [99]. Peeling can help to improve the overall parallelization of the loop

```
for(i = 0; i < n; i++)
{
S1: A[i] = i+1;
S2: B[i] = A[i]+i;
}
```

Original Code

```
#pragma omp parallel for private(i) \\
firstprivate(n)
  for (i = 0; i <= n - 1; i += 1) {
    S1: A[i] = i + 1;
    S2: B[i] = A[i] + i;
  }
```

Transformed Code

Figure 3.1: Privatization applied by Rose

```
for(i = 1; i < n; i++)
{
S1: A[i] = i;
S2: B[i] = A[i-1]+X;
}
```

Original Code

```
if (n >= 2) {
  B[1] = A[1 -1]+X;;
  lbp=2;
  ubp=n-1;
#pragma omp parallel for \\
private(lbv,ubv,t2)
  for (t1=lbp;t1<=ubp;t1++) {
    A[(t1-1)] = (t1-1);;
    B[t1] = A[t1-1]+X;;
  }
  A[(n-1)] = (n-1);;
}
```

Transformed Code

Figure 3.2: Loop Peeling applied by Pluto

by separating its parallelizable and dependent portions. Only ICC and Pluto support loop peeling. Figure 3.2 illustrates Pluto performing loop peeling.

Loop Fusion

Loop fusion [84] reduces the index variable maintenance overhead by merging two identical number of loop iterations to a single loop which improves locality. This is inherently performed by Par4all and Pluto. Figure 3.3 exemplifies the way in which loop fusion occur in Par4all.

Loop Fission

Loop fission helps in attaining partial parallelization of the loop by splitting it into multiple loops over the same index range with each taking only a part of the original loop's

```
for(i = 0; i < n; i++)
 A[i] = i+1;
for(i = 0; i < n; i++)
 B[i] = A[i]+5;
```

Original Code

```
#pragma omp parallel for
for(i = 0; i <= n-1; i += 1)
{
 A[i] = i+1;
 B[i] = A[i]+5;
}
```

Transformed Code

Figure 3.3: Loop Fusion applied by Par4all

```
for(i = 0; i < n; i++)
{
S1: A[i] = i+1;
S2: B[i] = B[i-1]+A[i];
}
```

Original Code

```
LOOP BEGIN at fission.c(10,1)
<Distributed chunk1>
   remark #17100: DISTRIBUTED LOOP WAS \\
   AUTO-PARALLELIZED
   remark #17101: parallel loop shared={ }
       private={ } firstprivate={ i A } last
       private={ } firstlastprivate={ }
       reduction={ }
LOOP END

LOOP BEGIN at fission.c(10,1)
<Distributed chunk2>
   remark #17104: loop was not parallelized
       : existence of parallel dependence
LOOP END

LOOP BEGIN at fission.c(10,1)
<Peeled, Distributed chunk1>
LOOP END
```

Transformed Code

Figure 3.4: Loop Fission applied by ICC

body [84]. The auto-parallelizing tools need to ensure that the original data dependences within and across iterations are preserved after fission. Loop fission/distribution can improve both spatial and temporal localities and is uniquely supported by ICC. Figure 3.4 depicts ICC's output where loop distribution is applied (Note: ICC is not a source-to-source translator and it provides only diagnostic information), where S_1 and S_2 are non-dependant and, data dependant instruction respectively.

```
for(i = 0; i < n; i++)
{
S1: A[i] = i+1;
S2: sum += A[i];
}
```

Original Code

```
#pragma omp parallel for private (i)\\
reduction (+:sum) firstprivate (n)
  for (i = 0; i <= n - 1; i += 1) {
    A[i] = i + 1;
    sum += A[i];
  }
```

Transformed Code

Figure 3.5: Reduction applied by Rose

Reduction variable recognition

In reduction variable recognition, each thread is provided with a copy of the read/write conflicting variable to operate on and to produce a partial result, which combines with other threads' copy to produce a global result [98]. Threads need to cooperate with each other to enable parallel computation, if failed will lead to incorrect output. Reduction is applicable only when the underlying computation follows certain algebraic properties such as associativity (in practice, tools also expect the computation to follow commutativity). Except Pluto, all other frameworks by default perform reduction technique. Pluto does not parallelize loop when the ℓ-value of a statement is a scalar. Figure 3.5 depicts reduction applied by Rose.

Loop Tiling

Loop tiling is another technique which can be used to improve temporal locality. It partitions a loop's iteration space into smaller chunks or blocks, to ensure the data used in a loop (tile) stays within cache [100]. Figure 3.6 shows the tiling performed by Pluto when opted. Tiling has been widely used by auto-parallelizres for optimizing matrix-based codes.

Loop Unrolling

Loop unrolling is the combination of two or more loop iterations together with a corresponding reduction of trip count. It is used to expose more instruction level parallelism (ILP). Loop Unrolling is depicted in Figure 3.7 (with unroll factor=8).

```
for(i = 1; i < n; i++)
{
S1: A[i] = i;
S2: B[i] = A[i-1]+X;
}
```

Original Code

```
if (n >= 2) {
  lbp=0;
  ubp=floord(n,32);
#pragma omp parallel for \\
private(lbv,ubv,t2,t3)
  for (t1=lbp;t1<=ubp;t1++) {
    if (t1 == 0) {
      B[1] = A[1-1]+X;;
    }
    lbv=max(2,32*t1);
    ubv=min(n-1,32*t1+31);
#pragma ivdep
#pragma vector always
    for (t2=lbv;t2<=ubv;t2++) {
      A[(t2-1)] = (t2-1);;
      B[t2] = A[t2-1]+X;;
    }
    if (t1 >= ceild(n-31,32)) {
      A[(n-1)] = (n-1);;
    }
  }
}
```

Transformed Code

Figure 3.6: Loop Tiling applied by Pluto

```
for(i = 0; i < n; i++)
{
S1: A[i] = i+1;
S2: sum += A[i];
}
```

Original Code

```
if (n >= 1) {
  for (t1=0; t1<=n-8; t1=t1+8) {
    A[t1] = t1+1;;
    sum += A[t1];;
    A[(t1+1)] = (t1+1)+1;;
    sum += A[(t1+1)];;
    .
    .
    .
    A[(t1+7)] = (t1+7)+1;;
    sum += A[(t1+7)];;
  }
  for (; t1<=n-1; t1=t1+1) {
    A[t1] = t1+1;;
    sum += A[t1];;
  }
}
```

Transformed Code

Figure 3.7: Loop Unrolling applied by Pluto

```
1  j=0
2  for(i = 1; i < n; i++)
3  {
4  S1: j = j +2;
5  S2: A[j]=B[j]+1;
6  }
```

Original Code

```
1  #pragma omp parallel for \\
2  if((10000<(-2L+(3L*n)))) lastprivate(i)
3  for (i=1; i<n; i ++ )
4  {
5  A[2*i]=(B[2*i]+1);
6  }
7  if ((-2+n)>=0)
8  {
9  j+=(-2+(2*n));
10 }
```

Transformed Code

Figure 3.8: Induction Variable Substitution applied by Cetus

Induction Variable Substitution

Induction Variable Substitution [46] in a loop uses the previous value of the induction variable to compute a new value, usually by adding or multiplying a scalar expression. If the induction variable derived does not refer to its previous value, then dependence is removed. Cetus uniquely supports induction variable substitution (Figure 3.8).

Overall,Table 3.1 summarizes that all the frameworks in this study support privatization of variables by default (T1). All the frameworks except Pluto inherently recognize reduction variables (T5). However, certain transformations (T8, T2) are uniquely supported i.e. Cetus performs induction variable substitution (T8) and ICC performs loop fission (T2) by default. Some of the loop transformations are supported when enabled via command-line i.e. tiling (T6) and unrolling (T7). In total, Pluto and Par4all support five of the eight transformations; Rose supports six (except loop peeling); while ICC supports all of them. Section 3.2 explains the effect of loop transformation techniques on dependence problems.

3.1.2 Dependence analysis

Amdahl's law [1] reveals that the sequential part of the program limits the amount of achievable parallelism. Typically, sequential execution is an artifact of data dependence, which forces an execution order [101]. A loop that contains millions of operations which are non-dependant can perform parallel, however a single dependant statement may force the code to run in serial. In order to effectively parallelize an application, dependence

analysis must be carefully managed, and then measures should be taken to remove those dependences. Further, removing the dependences should not affect the original flow of the program. In general, dependences can be categorized as *loop-independent* [98] [102] and *loop-carried* [98] [102] dependences. Each of these dependences could be scalar or vector (array-based).

Loop-independent dependences

Data dependences between statement instances S_1 and S_2 which belong to same loop iterations is termed as *loop-independent* dependences.

- *Data dependence* between statements $S_1 \delta^f S_2$ symbolizes that an instruction S_2 depends on the result of a previous instruction S_1
- *Anti-dependence* symbolically represented as $S_1 \delta^a S_2$ occurs when an instruction S_1 requires a value from S_2 that is later updated
- *Output dependence* $S_1 \delta^o S_2$ shows that S_2 modifies the value which S_1 has written, where the change in the order of instructions may affect the final output

Loop-independent dependence can be handled using privatization [86] by replicating variables across loop iterations. OpenMP automatically handles static dependence in loop with the use of private clause [11].

Loop-carried dependences

Data dependences between statement instances that belong to different loop iterations is termed as *loop-carried* dependences.

- *Loop-carried data dependence due to vector* occurs when the variables written in one iteration are then read in a different iteration. For better illustration consider the following example:

```
1 for i = 1, N
2   A[i] = A[i-1]
```

- *Loop-carried data dependence due to scalar* occurs due to reduction operation. For example:

```
1 for i = 1, N
2   sum += A[i]
```

- *Loop-carried anti-dependence* that is specified under is encountered when variables read in one iteration and then written in a different iterations. A typical example for *loop-carried anti-dependence* is shown below:

```
1 for i = 1, N
2   A[i] = A[i+1]
```

- *Loop-carried output dependence* normally take place when variable written in one iteration and then rewritten again in a different iterations. For example:

```
1 for i = 1, N
2   A[i] = B[i];
3   A[i+1] = C[i];
```

Loop-carried dependence is handled by using loop transformation techniques such as loop peeling, loop fission, or by manual intervention. Reduction variable recognition is used in parallelizing code which comprises *loop-carried data dependence due to a scalar*.

3.2 Effect of loop transformations on dependences

This section discusses the way in which the frameworks handle various kinds of dependences. It presents the effect of loop transformation techniques on different dependences problems and the study is categorized as:

1. Effect of loop transformations on *loop-independent* (scalar and vector) and loop-carried scalar dependence.
2. Effect of loop transformations on *loop-carried vector dependence*.

Table 3.2: Effect of loop transformation techniques on loop-independent (scalar and vector) and loop-carried scalar dependence

Dependence Problem	Cetus	Par4all	Pluto	Rose	ICC
Loop-independent (scalar)	✓T1	✓T1	✗	✓T1	✓T1
Loop-independent (vector)	✓T1	✓T1 T4	✓T1 T4	✓T1	✓T1
Loop-carried (scalar)	✓T1 T5	✓T1 T5	✗	✓T1 T5	✓T1 T5

✓ is parallelized, ✗ is not parallelized, T1 Privatization, T4 Fusion, T5 Reduction

3.2.1 Effect of loop transformations on loop-independent (scalar and vector) and loop-carried scalar dependence

Table 3.2 shows the behavior of frameworks under study in parallelizing simple for-loops in the presence of intra-iteration and *loop-carried scalar dependences*. Dependence problems are addressed through several of the loop transformation techniques.

For each dependence-framework pair, it is mentioned if the pair is supported (✓), and if so, what transformation technique (T1...T7) the framework uses for enabling the support. It was observed that the tools primarily rely on privatization (T1), fusion (T4), and reduction variable recognition (T5). Privatization helps in handling all the dependence problems. Loop fusion [84] reduces the index variable maintenance overhead by merging two identical number of loop iterations to a single loop which improves locality, and is used by Par4all and Pluto. Further, all frameworks (except Pluto) utilize reduction variable recognition and it removes the loop-carried dependences with scalar variables. The reason for Pluto's poor support for scalar variables is because it is primarily targeted towards polyhedral transformations of array accesses, rather than individual shared items.

3.2.2 Effect of loop transformations on loop-carried vector dependence

This section brings out the support of *loop-carried* dependence problems (data, anti, and output dependences, along with their combinations) by the frameworks.

Loop-carried dependence problems in single loop

Figure 3.9 depicts the different scenarios, which can be used to test various frameworks by changing the distance vector ($\alpha = 1, 2, 3$). The two cases (Case 1 and Case 2) mentioned in the figure depict the forward and the backward dependences, and accordingly, have different loop-index initialization and guard. Since programs with loop-carried dependences are not uncommon (extensive work on real-world benchmarks is elaborated in Chapter 4 and 5), the way in which various frameworks react to the presence of these dependences are studied.

Table 3.3 summarizes our observations. Overall, Cetus, Par4all, and Rose did not parallelize codes with *loop-carried* dependence in single loop. On the other hand, Pluto and ICC support all the loop-carried backward dependence problems (Case 1). However, ICC did not parallelize *loop-carried* output forward dependence (Case 2), while Pluto did not parallelize loop-carried data+anti+output forward dependence. Both these frameworks utilize privatization (T1) and loop peeling (T3) for handling the dependence problems. ICC additionally makes use of loop fission (T2) along with other optimizations (Table 3.1).

Loop-carried dependence problems in nested loop

This section explains the behavior of the frameworks on programs having nested loops with *loop-carried* dependence. Figure 3.10 depicts six such cases: backward i, backward j, backward i and j, and their forward counterparts. These simple templates suffice to assess the versatility of each framework.

Table 3.4 summarizes our findings, with distance vector ($\alpha = 1, 2, 3$). It is observed that Pluto supports all the six cases of *loop-carried* dependence problems. This is an artifact of the powerful polyhedral model underlying Pluto's transformations. On the other hand, ICC supports all the cases of *loop-carried* dependence only when α is 1. When α is 2 or 3,

(a) Loop-carried data dependence

```
for(i=α; i<n; j++)
{
  A[i] = i;
  B[i] = A[i−α]+X;
}
```
Case 1

```
for(i=0; i<n−α; i++)
{
  A[i] = i;
  B[i] = A[i+α]+X;
}
```
Case 2

(b) Loop-carried anti dependence

```
for(i=α; i<n; i++)
{
  A[i−α] = B[i];
  C[i] = A[i];
}
```
Case 1

```
for(i=0; i<n−α; i++)
{
  A[i+α] = B[i];
  C[i] = A[i];
}
```
Case 2

(c) Loop-carried output dependence

```
for(i=α; i<n; i++)
{
  A[i] = B[i];
  A[i−α] = C[i];
}
```
Case 1

```
for(i=0; i<n−α; i++)
{
  A[i] = B[i];
  A[i+α] = C[i];
}
```
Case 2

(d) Loop-carried data+anti+output dependence

```
for(i=α; i<n; i++)
{
  A[i] = B[i]*X;
  C[i] = A[i−α]+X;
  C[i−α] = A[i]+X;
  D[i−α] = C[i]+X;
  F[i] = D[i];
}
```
Case 1

```
for(i=0; i<n−α; i++)
{
  A[i] = B[i]*X;
  C[i] = A[i+α]+X;
  C[i+α] = A[i]+X;
  D[i+α] = C[i]+X;
  F[i] = D[i];
}
```
Case 2

(Case 1 is backward dependence, Case 2 is forward dependence)

Figure 3.9: Templates of tested loop-carried dependence problems (a) Loop-carried data dependence (b) Loop-carried anti dependence (c) Loop-carried output dependence (d) Loop-carried data+anti+output dependence

Table 3.3: Support of auto-parallelizers for loop-carried dependence problems in single loop using synthetic programs

	Tools	**Cetus**			**Par4all**			**Rose**			**ICC**			**Pluto**		
Dependence Problems	**Distance Vectors**	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Loop-carried Data	Case 1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
	Case 2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
Loop-carried Anti	Case 1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
	Case 2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
Loop-carried Output	Case 1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
	Case 2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T3	✓ T1 T3	✓ T1 T3
Loop-carried Data+Anti +Output	Case 1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
	Case 2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✗	✗	✗

✓ is Parallelized, ✗ is Not Parallelized, T1 = Privatization, T2 = Loop Fission, T3 = Loop Peeling, Distance Vector α = 1, 2, and 3, Case 1 is backward dependence, Case 2 is forward dependence

(a) Loop-carried data dependence problems

```
for(i=α; i<n; i++)
for(j=0; j<n; j++)
{
 A[i][j]=j;
 B[i][j]=A[i−α][j]+X;
}
```
Case 1

```
for(i=0; i<n; i++)
for(j=α; j<n; j++)
{
 A[i][j]=j;
 B[i][j]=A[i][j−α]+X;
}
```
Case 2

```
for(i=α; i<n; i++)
for(j=α; j<n; j++)
{
 A[i][j]=j;
 B[i][j]=A[i−α][j−α]+X;
}
```
Case 3

```
for(i=0; i<n−α; i++)
for(j=0; j<n; j++)
{
 A[i][j]=j;
 B[i][j]=A[i+α][j]+X;
}
```
Case 4

```
for(i=0; i<n; i++)
for(j=0; j<n−α; j++)
{
 A[i][j]=j;
 B[i][j]=A[i][j+α]+X;
}
```
Case 5

```
for(i=0; i<n−α; i++)
for(j=0; j<n−α; j++)
{
 A[i][j]=j;
 B[i][j]=A[i+α][j+α]+X;
}
```
Case 6

(b) Loop-carried anti dependence problems

```
for(i=α; i<n; i++)
for(j=0; j<n; j++)
{
 A[i−α][j] = B[i][j];
 C[i][j] = A[i][j]+X;
}
```
Case 1

```
for(i=0; i<n; i++)
for(j=α; j<n; j++)
{
 A[i][j−α] = B[i][j];
 C[i][j] = A[i][j]+X;
}
```
Case 2

```
for(i=α; i<n; i++)
for(j=α; j<n; j++)
{
 A[i−α][j−α] = B[i][j];
 C[i][j] = A[i][j]+X;
}
```
Case 3

```
for(i=0; i<n−α; i++)
for(j=0; j<n; j++)
{
 A[i+α][j] = B[i][j];
 C[i][j] = A[i][j]+X;
}
```
Case 4

```
for(i=0; i<n; i++)
for(j=0; j<n−α; j++)
{
 A[i][j+α] = B[i][j];
 C[i][j] = A[i][j]+X;
}
```
Case 5

```
for(i=0; i<n−α; i++)
for(j=0; j<n−α; j++)
{
 A[i+α][j+α] = B[i][j];
 C[i][j] = A[i][j]+X;
}
```
Case 6

The six cases are: backward i, backward j, backward i and j, forward i, forward j, forward i and j. The variable i denotes dependences in outer loop, The variable j denoted dependences in inner loop

Figure 3.10: Templates of loop-carried dependence problems in nested loops (a) Loop-carried data dependence problems (b) Loop-carried anti dependence problems, *(continued in next page)*

(c) Loop-carried output dependence problems

```
for(i=α; i<n; i++)
for(j=1; j<n; j++)
{
 A[i][j]=B[i][j];
 A[i−α][j]=C[i][j]+X;
}
```
Case 1

```
for(i=1; i<n; i++)
for(j=α; j<n; j++)
{
 A[i][j]=B[i][j];
 A[i][j−α]=C[i][j]+X;
}
```
Case 2

```
for(i=α; i<n; i++)
for(j=α; j<n; j++)
{
 A[i][j]=B[i][j];
 A[i−α][j−α]=C[i][j]+X;
}
```
Case 3

```
for(i=1; i<n−α; i++)
for(j=1; j<n; j++)
{
 A[i][j]=B[i][j];
 A[i+α][j]=C[i][j]+X;
}
```
Case 4

```
for(i=1; i<n; i++)
for(j=1; j<n−α; j++)
{
 A[i][j]=B[i][j];
 A[i][j+α]=C[i][j]+X;
}
```
Case 5

```
for(i=1; i<n−α; i++)
for(j=1; j<n−α; j++)
{
 A[i][j]=B[i][j];
 A[i+α][j+α]=C[i][j]+X;
}
```
Case 6

(d) Loop-carried data+anti+output dependence problems

```
for(i=α; i<n; i++)
for(j=1; j<n; j++)
{
 A[i][j]=B[i][j]*X;
 C[i][j]=A[i−α][j]+X;
 C[i−α][j]=A[i][j]+X;
 D[i−α][j]=C[i][j]+X;
 F[i][j]=D[i][j];
}
```
Case 1

```
for(i=1; i<n; i++)
for(j=α; j<n; j++)
{
 A[i][j]=B[i][j]*X;
 C[i][j]=A[i][j−α]+X;
 C[i][j−α]=A[i][j]+X;
 D[i][j−α]=C[i][j]+X;
 F[i][j]=D[i][j];
}
```
Case 2

```
for(i=α; i<n; i++)
for(j=α; j<n; j++)
{
 A[i][j]=B[i][j]*X;
 C[i][j]=A[i−α][j−α]+X;
 C[i−α][j−α]=A[i][j]+X;
 D[i−α][j−α]=C[i][j]+X;
 F[i][j]=D[i][j];
}
```
Case 3

```
for(i=1; i<n−α; i++)
for(j=1 ; j<n; j++)
{
 A[i][j]=B[i][j]*X;
 C[i][j]=A[i+α][j]+X;
 C[i+α][j]=A[i][j]+X;
 D[i+α][j]=C[i][j]+X;
 F[i][j]=D[i][j];
}
```
Case 4

```
for(i=1; i<n; i++)
for(j=1 ; j<n−α; j++)
{
 A[i][j]=B[i][j]*X;
 C[i][j]=A[i][j+α]+X;
 C[i][j+α]=A[i][j]+X;
 D[i][j+α]=C[i][j]+X;
 F[i][j]=D[i][j];
}
```
Case 5

```
for(i=1; i<n−α; i++)
for(j=1 ; j<n−α; j++)
{
 A[i][j]=B[i][j]*X;
 C[i][j]=A[i+α][j+α]+X;
 C[i+α][j+α]=A[i][j]+X;
 D[i+α][j+α]=C[i][j]+X;
 F[i][j]=D[i][j];
}
```
Case 6

The six cases are: backward i, backward j, backward i and j, forward i, forward j, forward i and j. The variable i denotes dependences in outer loop, The variable j denoted dependences in inner loop

Figure 3.10: Templates of loop-carried dependence problems in nested loops (c) Loop-carried output dependence problems (d) Loop-carried data+anti+output dependence problems (Contd.)

it could handle Cases 2 and 5 only (backward and forward dependences in the *inner* loop). Thus, when the dependence occurs due to the *outer* loop of the nested for when α is 2 or 3, ICC failed to parallelize. This is interesting because it indicates that the parallelization was tuned for distance vector of unity. We believe that ICC supports unit length distance vector alone as that case was found to be common in practice. In the transformation, Pluto and ICC use privatization (T1) and loop peeling (T3) for handling the dependence. ICC additionally makes use of loop fission (T2) along with other optimizations for Cases 2 and 5, that is, when the dependence occurs due to the inner loop of the nested for.

Overall conclusion is that Cetus, Par4all, and Rose could not parallelize codes with *loop-carried* dependence in single loop. However, it is observed that these three frameworks could support *loop-carried dependence due to vector* occurring either in the inner loop or the outer loop of a nested for (but not both the loops). Further, they parallelize the dependence-free loop (inner/outer). Hence concluded the result as non-parallelized for the nested loops.

Table 3.4: Support of auto-parallelizers for loop-carried dependence problems in nested loop using synthetic programs

Tools	**Cetus**			**Par4all**			**Rose**			**ICC**			**Pluto**		
Distance Vector	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Case 1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T3	✗	✗	✓ T1 T3	✓ T1 T3	✓ T1 T3
Case 2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
Case 3	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T3	✗	✗	✓ T1 T3	✓ T1 T3	✓ T1 T3
Case 4	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T3	✗	✗	✓ T1 T3	✓ T1 T3	✓ T1 T3
Case 5	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
Case 6	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T3	✗	✗	✓ T1 T3	✓ T1 T3	✓ T1 T3

The six cases are: backward i, backward j, backward i and j, forward i, forward j, forward i and j. The variable i denotes dependences in outer loop, The variable j denoted dependences in inner loop. (✓ Parallelized, ✗ Not Parallelized, T1 = Privatization, T2 = Loop Fission, T3 = Loop Peeling)

3.3 Limitations faced by auto-parallelizers

This section illustrates the limitations of auto-parallelizers in handling complex coding style. In general, identifying the code sections that have parallelism possibility is a cumbersome task. Auto-parallelizer may not parallelize all the loops. Under certain conditions, these tools fail to insert parallel directives. Auto-parallelizer sometimes may or may not parallelize the loop. The plausible reasons could be due to: (i) dependence within the code (ii) failure to support specific programming constructs. The different dependences and their support by auto-parallelizer frameworks are studied and reported in the previous section (Section 3.2). Identifying and addressing the problematic code section is a fundamental aspect of this thesis. At several instances, the parallelization performed using auto-parallelizer results in an erroneous transformation without offering any hints towards the source of the error. There is a potential for improvement in finding the origin of the error. This study helps the programmers to understand the problems in code section which prevents transformation.

The primary programming features (in total 76) are categorized and tested within the parallelizable region (loops) in the subsequent section (Section 3.4). OpenMP supports for-loop parallelization. Hence, support of different programming features is studied only within for-loop. Testing of these programming features helps us to find the capabilities and limitations of the auto-parallelizers in parallelizing the loops. This study confers the results and provides recommendations for compiler researchers, tool developers, and application scientists.

3.4 Auto-parallelizer behavior on complex coding style

This section reveals some of the problems which prevent parallelization by different frameworks. Further, it discusses the method to solve the issues prohibiting better utilization of parallelizers. Input serial codes are parallelized comprising distinct programming features via the five frameworks under study. In addition, the testing is performed on an industrial compiler, Parallware [61] (an industrial framework).

The programming constructs tested (in total 76) are categorized into three groups, viz., (i) Loops and Loop Conditions (ii) Statements, Data Types and Storage Classes (iii) Functions. In case if the parallelizers fail to handle the code due to few non-supported programming constructs, then solutions are exposed to uncover the problems and make them amenable for parallelization. The parallelization pitfalls along with the potential solutions are discussed in the subsequent sections (Section 3.4.1 to 3.4.4).

3.4.1 Support for loops and loop conditions

The effectiveness of auto-parallelizers in parallelizing the loops and loop conditions were examined. Figure 3.11 illustrates the parallelization of all for-loops. Three types of for-loop parallelization, viz. *simple*, *concatenated*, and *nested* are tested. It is observed that all the tools support parallelization of all these three types of for-loops. Here, *concatenated* loop resembles the loops iterating with same problem size. The remaining loops (while, do-while) and loop conditions are discussed later in Section 3.4.4.

Figure 3.11(b) shows the case where concatenated loops are parallelized using loop fusion transformation. It was observed that Par4all and Pluto merged the loops with the same number of iterations into a single loop during parallelization. Figure 3.11(c) shows the parallelization of nested for-loop case. From the qualitative analysis discussed in Chapter 2, the following conclusion are made. Among the six frameworks, Pluto, Par4all, Parallware, ICC, and Rose add parallel directives only to the outermost loop. Cetus enables nested parallelism by default, hence it inserts parallel directives to all the inner and outer loops. This eventually leads to execution overhead problem when the number of threads exceeds the physical core size.

3.4.2 Support of auto-parallelizers for statements, data types and storage classes

The codes comprising programming constructs, namely statements, data types and storage classes are tested (in total 14). Table 3.5 shows the various features tested and also lists which frameworks were able to parallelize the constructs successfully. The obtained results pertaining to the observations in Table 3.5 are discussed in detail below.

(a) Parallelization of simple for-loop

```
for(j = 0; j < n; j++)
  c[j] = a[j] + b[j];
```

Original Code

```
#pragma omp parallel for private(j)
for(j = 0; j < n; j++)
  c[j] = a[j] + b[j];
```

Transformed Code

(b) Parallelization of concatenated for-loop

```
for(r = 0; r < n; r++)
{
  a[r] = r;
  b[r] = r;
}
for(i = 0; i < n; i++)
  c[i] = a[i]*(a[i] + b[i]) - b[i];
```

Original Code

```
#pragma omp parallel for
for(r = 0; r <= n-1; r +=1)
{
  a[r] = r;
  b[r] = r;
  c[r] = a[r]*(a[r]+ b[r])-b[r];
}
```

Transformed Code

(c) Parallelization of nested for-loop(s)

```
for(l = 0; l < n; l++)
for(m = 0; m < n; m++)
for(p = 0; p < n; p++)
 c[l][m] += a[l][p] * b[p][m];
```

Original Code

```
#pragma omp parallel for \\
private(m,p)
for(l = 0; l <= n-1; l +=1)
for(m = 0; m <= n-1; m +=1)
for(p = 0; p <= n-1; p +=1)
 c[l][m]+=a[l][p]*b[p][m];
```

Transformed Code

Figure 3.11: Parallelization of for-loop by auto-parallelizers (a) Parallelization of simple for-loop (b) Parallelization of concatenated for-loop (c) Parallelization of nested for-loop(s)

if statements All six frameworks can apply parallelization when an if statement is used within for-loop. Figure 3.12(a) (second and third column) shows the parallelization results of one the auto-parallelizer (Par4all).

switch-case statements Another well-known construct used within the parallelizable region is switch-case statement. Figure 3.12(a) depicts the way in which switch-case is handled by Par4all. It was observed that Par4all, Rose, and ICC successfully parallelize the code by replacing the switch-case with multiple if-else statements, without affecting the functionality of the program. Similarly, switch-case can be transformed to if-else to make the code amenable to parallelization.

Data types and storage classes Table 3.5 shows the parallelization support of data types and storage classes when used within for. There are no data types and storage classes that could not be parallelized using any of the tools. Par4all supports the maximum features, while Cetus supports the least. There are no generic solutions for solving the issue. However, it was observed that defining the unsupported constructs outside the parallelizable region could alleviate the pitfall.

Table 3.5: Support of auto-parallelizers for statements, data types and storage classes

Tested Programming Features		Cetus	Par4all	Pluto	Parallware	Rose	ICC	T_{prog}	Suggested Solutions to Auto-parallelization
Statements	if	✓	✓	✓	✓	✓	✓	2	-
	switch-case	✗	✓	✗	✗	✓	✓		switch-case may be replaced by multiple if and else if statements
Data Types	struct	✗	✓	✗	✓	✓	✗	8	Unsupported Data Types may be defined outside the parallelizable region
	Array of structs	✗	✓	✓	✗	✗	✓		
	union	✗	✗	✗	✓	✓	✗		
	typedef	✓	✓	✓	✓	✓	✓		
	enum	✓	✓	✓	✓	✓	✓		
	pointers	✗	✓	✗	✗	✓	✗		
	Array of pointers	✗	✓	✓	✗	✗	✓		
	Global Variables	✓	✓	✓	✗	✓	✓		
Storage Classes	auto, register	✓	✓	✓	✓	✓	✓	4	Unsupported storage classes may be defined outside the parallelizable region
	static, extern	✓	✓	✓	✗	✓	✓		
-	-	8	13	10	7	12	11	14	-

✗ Not Parallelized, ✓ Parallelized, T_{prog} = Total number of programming features tested under each category

3.4.3 Support of auto-parallelizers for functions

The tools' ability is investigated for handling different functions, such as library, user-defined, and recursive functions within the parallelizable region (in total 46). Table 3.6 lists the parallelization results of functions by various frameworks. The results derived upon parallelizing different function using various frameworks are elaborated in detail below.

Library functions

Six categories of library functions are examined, namely, standard I/O, string, memory, mathematical, standard library, file I/O. Table 3.6 shows the result of frameworks which were able to handle certain library functions when used within the for-loop. Interestingly, all frameworks except Rose successfully parallelize loops comprising mathematical functions. However, the performance of Rose upon usage of library functions was found to be inferior compared to all other frameworks. Parallware supports more number of programming features. In the standard I/O group, except scanf and getc, all other constructs are supported by at least minimum of the frameworks. In addition, it is found that parallelizers support the majority of the library functions when used outside the parallelizable region. Therefore, it could be concluded that the usage of these functions inside the for-loop needs to be avoided.

User-defined functions

In order to assess the support of user-defined functions in parallelizers, two cases are taken up viz. (i) loops within function call (ii) function call inside the parallelizable region. Table 3.6 shows the parallelization results of both cases. Loops within function call is a primitive scenario which is handled by all the frameworks. Although, the second case is complicated, it is supported by all frameworks except Rose. To make the latter case support by the tools, inlining of function can be applied. Figure 3.12(b) depicts the method to address the pitfall.

Table 3.6: Support of auto-parallelizers for functions

Tested Programming Features		Cetus	Par4all	Pluto	Parallware	Rose	ICC	T_{prog}	Suggested Solutions to Auto-parallelization
Standard I/O[a]	scanf	✗	✗	✗	✗	✗	✗	4	Avoid using unsupported functions within the parallelizable region
	printf	✗	✗	✓	✓	✗	✗		
	getc	✗	✗	✗	✗	✗	✗		
	putc	✗	✗	✓	✗	✗	✗		
String[a]	strstr,strchr	✗	✗	✗	✓	✗	✓	7	
	strlen	✓	✗	✗	✓	✗	✓		
	strcmp	✓	✗	✗	✗	✗	✗		
	strcat	✗	✗	✓	✗	✗	✗		
	strcpy	✗	✗	✓	✓	✗	✗		
	strtok	✗	✗	✗	✓	✗	✗		
Memory[a]	memcmp	✓	✓	✗	✓	✗	✓	3	
	memcpy, memmove	✗	✗	✓	✓	✗	✗		
Mathematical[a]	floor, round, ceil, sin, cos, tan, sinh, cosh, tanh, log, log10, exp, sqrt, pow, trunc, abs	✓	✓	✓	✓[b]	✗	✓	16	
Std library[a]	system	✗	✓	✓	✓	✗	✗	5	
	atoi, atof	✓	✓	✗	✓	✗	✓		
	rand	✓	✗	✓	✓	✗	✓		
	abs	✓	✓	✓	✓	✗	✓		
File I/O[a]	fputc, fputs, fscanf, fprintf, fwrite	✓	✓	✓	✓	✗	✓	8	
	fgets, fgetc, feof	✓	✓	✗	✓	✗	✓		
User-defined	Loops within function call	✓	✓	✓	✓	✓	✓	3	Inlining of functions
	Function call inside parallelizable region	✓	✓	✓	✓	✗	✓		
	Recursive Functions	✗	✗	✓	✓	✗	✓		Recursive function may be replaced by for-loop
-	-	33	31	33	40	1	35	46	-

✗ Not Parallelized, ✓ Parallelized, T_{prog} = Total number of programming features tested under each category,
a - Predefined C Procedures, b - Except trunc all mathematical functions are parallelized)

Table 3.7: Solution for OpenMP limitations

Tested Programming Features	Tools	Suggested Solutions to Auto-parallelization
while loop	✓[a]	Use for-loop instead of while-loop
do-while loop	✗	Use for-loop instead of do-while loop
Multiple index variable in for-loop condition	✗	Reform the code structure by removing multiple initializations and increments in for-loop condition
Logical Operators (&&, ||, !)	✗	The for-loop condition with logical operators may be replaced as if statements within the body
Relational Operators (!=, ==)	✗	The for-loop condition with relational operators may be replaced as if statements within the body
exit, return, and goto	✗	Avoid using exit, return, and goto
9	1	-

✗Not Parallelized, ✓Parallelized, a - Parallware and ICC supports while-loop

Recursive functions are difficult to parallelize due to uncertain recursive depth. Therefore it is unknown how many times this function will call itself, which eventually complicates parallelization across a fixed number of threads. When tested recursive function using auto-parallelizers the following results were observed. Pluto, Parallware, and ICC could support these features and are listed in Table 3.6. In order to solve this problem, the recursive loops are replaced with for-loop(s) without altering the functionality of the code. Figure 3.12(c) shows the method to resolve the recursive functions.

3.4.4 Limitations due to OpenMP programming model

Besides parallelization and implementation issues, auto-parallelizer also face few confines due to OpenMP model. Table 3.7 lists the limitations due to OpenMP and the plausible solution to overcome these issues.

Parallelization of while and do-while loops The ability of tools in handling while and do-while loops are discussed here. Among all six frameworks only Parallware and ICC support parallelization of while-loop. Parallware could transform while-loop to for-loop,

and apply parallelization. Figure 3.12(d) illustrates the transformation and parallelization process. It is worth noting that no tools parallelized do-while loop.

Multiple index variables in for-loop condition OpenMP does not support multiple initialization and multiple increment/decrement in for-loop condition, and no tools parallelize such scenario. The explication to overcoming this issue is by reforming the code such that loop splitting eliminates the second index variable. Figure 3.12(e) shows the manual transformation and parallelization of input for-loop with multiple initialization/updation.

Logical operators in loop condition Predominantly, OpenMP directives do not support the use of logical operators (&&, ||, !) in for-loop condition. Hence, no tools aid parallelization of such cases. Figure 3.12(f) illustrates the use of logical operation in for-loop condition. The conditions used in for-loop were converted into if statements inside the for. After applying these modifications, all the tools could successfully parallelize the code.

Solution for relational operators in for-loop condition Parallelization of loops with relation operators in for-loop condition are tested. All tools parallelize loops with relational operators (<, >, <=, >=). However, OpenMP does not support relational operators (!= and ==) when utilized in the for-loop condition. Therefore, in such scenario, loop transformation techniques can be applied in order to replace these unsupported operators into other supported relational operators within the if clause. Figure 3.12(g) shows the solution for parallelization of loops with unsupported relational operators.

Statements: exit, return, and goto OpenMP does not support loop containing exit statements mainly break, return and goto. Hence, it is recommended to avoid the usage of these statements within the parallelizable region.

(a) Parallelization of switch-case

```
for(i = 0; i < n;i++)
{
  switch(s[i])
  {
  case '+':
  sum1 = num[i];
  break;
  case '*':
  sum = num[i];
  break;
  }
}
```

```
for(i = 0; i < n; i++)
{
   if(s[i] == '+')
   sum1 = num[i];
   if(s[i] == '*')
   sum = num[i];
}
```

```
#pragma omp parallel\\
for reduction(*:sum) \\
reduction(+:sum1)
for(i=0; i<=n-1; i+=1)
{
  if(s[i] == '+')
  {
    sum1 += num[i];
  }
  else if(s[i] == '*')
  {
    sum *= num[i];
  }
}
```

(b) Parallelization of function call inside parallelizable region

```
main() {
  for(k = 0; k<n; k++)
    Add(a,b,c,k);
}
int Add(int a[100],\\
int b[100],\\
int c[100],int k)
{
  c[k] = a[k] + b[k];
}
```

```
for(k = 0; k < n; k++)
  c[k] = a[k] + b[k];
```

```
#pragma omp parallel for
for(k = 0; k < n; k++)
  c[k] = a[k] + b[k];
```

(c) Parallelization of recursive function

```
main()
{
for(i = 0; i < n; i++)
  fact[i]=fac(i);
}
int fac(unsigned\\
int i)
{
  if(i <= 1)
    return 1;
  else
    return i*fac(i-1);
}
```

```
for(i=0; i<=n; i++)
{
  fact = 1;
  for(j=1; j<=i; j++)
    fact = fact * j;
}
```

```
#pragma omp parallel for
for(i=0; i<=n; i++)
{
  fact = 1;
  for(j=1; j<=i; j++)
    fact = fact * j;
}
```

First column refers to original code, **second column** refers to reformed code, and **third column** refers to reformed+parallelized code

Figure 3.12: Solutions for the non-supported constructs (a) Parallelization of switch-case (b) Parallelization of function call inside parallelizable region (c) Parallelization of recursive function, (continued in next page)

(d) Parallelization of while*-loop*

```
i=0;
while(i < n)
{
  c[i] = a[i] + b[i];
  i++;
}
```

```
for (i=0; i<n; i++)
  c[i]=a[i]+b[i];
```

```
#pragma omp parallel\\
shared(a,b,c) private(i)
{
  #pragma omp for \\
  schedule (static)
  for (i=0; i<n; i=i+1)
    c[i] = a[i] + b[i];
}
```

(e) Parallelization of codes with multiple index variables in for*-loop condition*

```
for (j=0,k=n-1; j<n;\\
j++,k--)
{
  c[j] = a[j] + b[j];
  d[k] = a[k] - b[k]
}
```

```
for(j=0 ;j<n; j++)
  c[j] = a[j] + b[j];
for(k=n-1; k>=0; k--)
  d[k] = a[k] - b[k];
```

```
#pragma omp parallel for
for(j= 0;j <= n-1; j+=1)
  c[j] = a[j]+b[j];
#pragma omp parallel for
for(k=n-1; k>=0; k+=-1)
  d[k] = a[k]-b[k];
```

(f) Parallelization of codes with logical operators in for*-loop condition*

```
for(j=0;(j<s1&&(j-k)\\
<s2); j++)
  c[j] = a[j] + b[j];
```

```
for(j=0; (j<s1); j++)
  if((j-k)<s2)
    c[j]=a[j] + b[j];
```

```
#pragma omp parallel\\
for private(j)
for (j = 0; j < s1; j++)
  if ((j-k) < s2))
   c[j]=a[j] + b[j];
```

(g) Parallelization of codes with relational operators in for*-loop condition*

```
for(i=0;((a[i]%2)==0\\
&& i < n); i++)
  b[i] = i ;
```

```
for(i = 0;i < n;i++)
  if((a[i]%2) == 0)
    b[i] = i ;
```

```
#pragma omp parallel\\
for private(i)
for(i = 0;i < n;i++)
  if((a[i]%2) == 0)
    b[i] = i ;
```

First column refers to original code, **second column** refers to reformed code, and **third column** refers to reformed+parallelized code

Figure 3.12: Solutions for the non-supported constructs (d) Parallelization of while*-loop (e) Parallelization of codes with multiple index variables in* for*-loop condition (f) Parallelization of codes with logical operators in* for*-loop condition (g) Parallelization of codes with relational operators in* for*-loop condition (Contd.)*

3.5 Evaluation of auto-parallelizers

The tested programming features (in total 76) are categorized into three groups for comparative analysis, viz., (i) loops and loop conditions (ii) statements, data types, and storage classes (iii) functions. This section discusses the parallelization of programming constructs by auto-parallelizers *before* (pre-evaluation) and *after* (post-evaluation) applying our code modifications.

3.5.1 Pre-evaluation

Figure 3.13(a) shows the support matrix (programming features supported) of auto-parallelizers, without manual intervention, in parallelizing loops and loop conditions. Here, T_{prog} is the total number of programming features tested under each category, which is 8. Parallware and ICC parallelize all 8 constructs, and remaining frameworks (Cetus, Par4all, Rose, and Pluto) could parallelize 7 of them. This shows that the commercial tools (Parallware and ICC) have better support for various constructs.

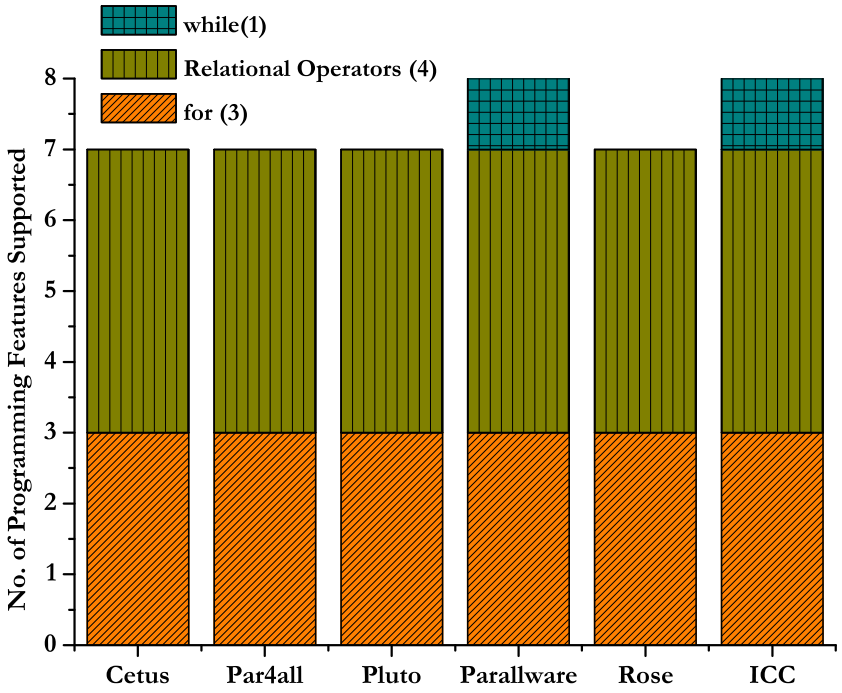
Figure 3.13(b) depicts the tools support for statements, data types and storage classes, here T_{prog} is 14. Table 3.5 shows the support of individual features under this category. The number of constructs parallelized by each tool is: Cetus 8, Par4all 13, Pluto 10, Parallware 7, Rose 12, ICC 11. This shows that Par4all, Rose and ICC have better support for statements, data types and storage classes.

Figure 3.13(c) describes the parallelizers' support for distinct functions, here T_{prog} is 46. Table 3.6 shows the number of programming features tested under each function category. Among the 46 constructs, Parallware could parallelize 40, ICC is a close second parallelizing 35 of them, while Cetus and Pluto could support 33, and Par4all supports 31. Rose supports only one construct and is inferior to other tools. These statistics indicate that the commercial compilers, Parallware and ICC supports maximum features.

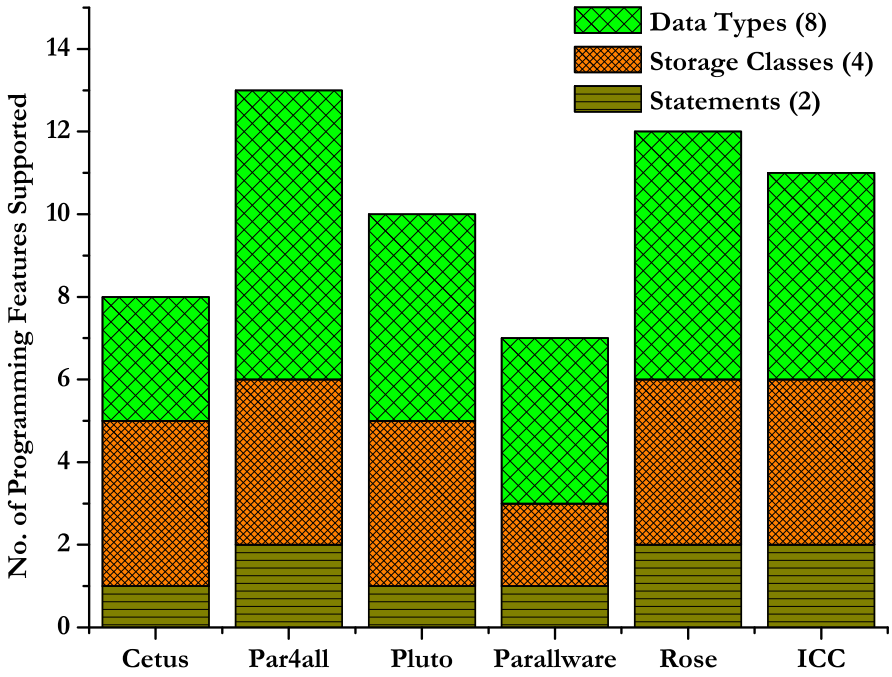
In Table 3.8, the second and third columns illustrates the number of programming features supported before applying solutions (T_1) and its acceptance ratio (A_α) (Equation 3.1) for each framework.

$$A_\alpha = \frac{T_1}{T_P} \tag{3.1}$$

(a) Loops and loop conditions (T_{prog}=8; for - simple, concatenated and nested; Relational Operators - <, >, <=, >=)



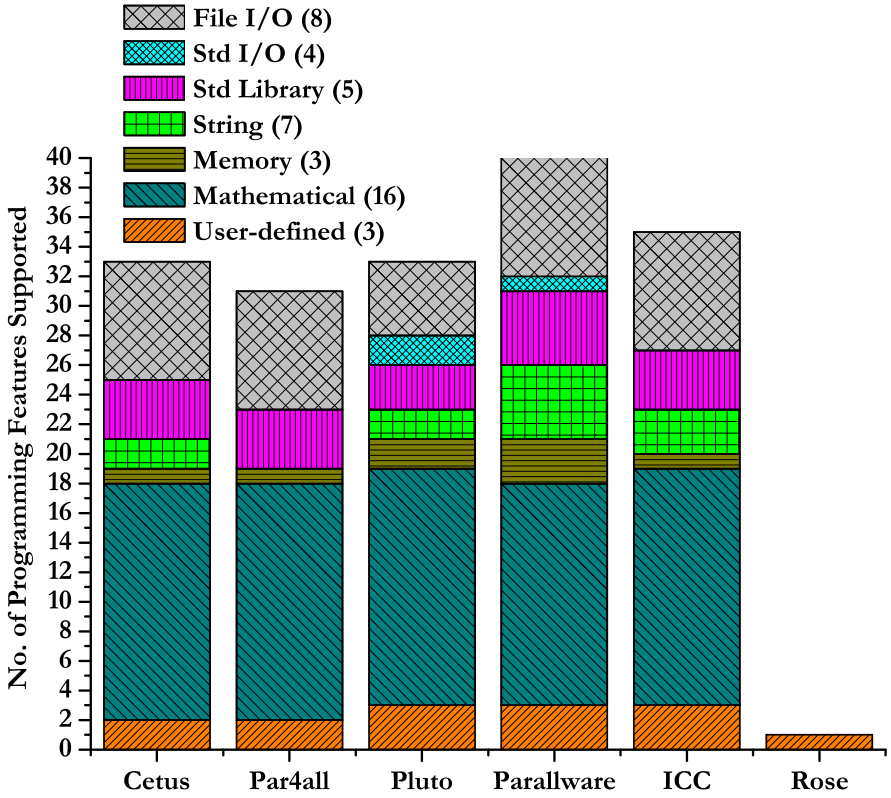
(b) Statements, data types and storage classes (T_{prog}=14)



The value within the parentheses is the total number of programming features tested under each subcategory. T_{prog}-Total number of programming features tested under each category

Figure 3.13: Parallelizers support for different programming features without manual intervention (pre-evaluation) (a) Loops and loop conditions (b) Statements, data types and storage classes, (continued in next page)

(c) Functions (T_{prog}=46)



The value within the parentheses is the total number of programming features tested under each subcategory. T_{prog}-Total number of programming features tested under each category

Figure 3.13: Parallelizers support for different programming features without manual intervention (c) Functions (Contd.)

Here, T_p is the number of programming features tested (76). The results show that Parallware performs better among all frameworks. Except Rose all other frameworks show good support close to Parallware.

3.5.2 Post-evaluation

Post-evaluation of the results reveal that there is a significant improvement in the performance of auto-parallelizers upon applying of transformations to the unsupported programming constructs. Table 3.8 shows performance summary of auto-parallelizers where in T_2 indicates the total number of programming features which were supported and their corresponding acceptance ratio A_β after applying coding changes (Equation 3.2).

Table 3.8: Performance summary of auto-parallelizers on synthetic codes

Tools	$\mathbf{T_1}$	$\mathbf{A}_\alpha$	$\mathbf{T_2}$	$\mathbf{A}_\beta$
Cetus	48	63.2	58	76.3
Par4all	51	67.1	60	78.9
Pluto	50	65.8	59	77.6
Parallware	55	72.4	63	82.9
Rose	20	26.3	30	39.5
ICC	54	71.1	61	80.3

T_1 = Programming Features Supported Before Applying Solutions, T_2 = Programming Features Supported After Applying Solutions, A_α = Acceptance Ratio Before Applying Solutions (%), (A_β) = Acceptance Ratio After Applying Solutions (%)

$$A_\beta = \frac{T_2}{T_p} \tag{3.2}$$

Here, T_p is the total number of features that were tested (76). It was observed that a similar trend in the support matrix and acceptance ratio of tools as the previous section. However, Rose shows better performance after applying solutions. Further, it was noted that there is a considerable increase in the acceptance ratio in all auto-parallelizers is observed upon applying of solutions to the identified pitfalls.

3.6 Summary

This chapter emphasized the combined effect of two important parallelization mechanisms namely, dependence analysis and loop transformation techniques. It was examined that few of the loop transformation techniques are supported only by some of the frameworks. Specific techniques are exclusively applied, for instance, ICC supported loop fission, and Cetus performed induction variable substitution. All frameworks except Pluto had inherently support for privatization and reduction.

In addition, the frameworks exhibited varying capabilities in handling the *loop-carried dependences*. Cetus, Par4all, and Rose were confined to support codes with *loop-independent* dependence using privatization. Pluto and ICC parallelized loops with *loop-carried dependence due to vector*. Under these circumstances, Pluto applied loop peeling techniques, while ICC performed both loop peeling and loop fission besides privatization.

This chapter also highlighted the limitations which are faced by auto-parallelizers in handling complex coding structures. It was observed that the parallelizers had restrictions in handling and supporting several programming constructs. This study helped us in understanding the behavior and sophistication of various frameworks. The suggested manual transformations to the problematic constructs assisted in increasing the percentage of acceptances of code for parallelization. We note that our conclusions are drawn based only on the experiments performed as part of this thesis. While we believe that the results generalize to other programs, we cannot make such a claim for arbitrary programs.

CHAPTER 4

Quantitative Analysis of Parallelization Frameworks using PolyBench Benchmarks

This chapter explains in detail about the quantitative effect of auto-parallelization frameworks on various PolyBench benchmarks. It investigates the support matrix of the frameworks in parallelization of PolyBench codes. This chapter also discusses performance by categorizing benchmarks based on different dependence types.

4.1 Background

In the previous chapters, using the synthetic programs, the various scenarios under which the frameworks are able to auto-parallelize were identified. In this chapter, empirical analyses are carried out on the frameworks for the performance of their transformed codes. PolyBench codes [103, 104] are used for such evaluation. In this section, a detailed description on PolyBench and experimental setup is illustrated.

4.1.1 PolyBench

PolyBench [103] is a suite of linear algebra kernels with static control parts that can be represented in the polyhedral model. Hence, these benchmarks have parameteric loop bound with affine arrary accesses, and are suitable for Polyhedral parallelizers. PolyBench is a suite of thirty programs implemented in C comprising numerical computations from different application domains such as linear algebra, image processing, physics simulation, dynamic programming, and statistics. The PolyBench suite contains several nested loop structures with complex dependences, hence makes challenging set of codes to be auto-analyzed by the tools. The benchmarks are based on statistical computation, matrix/matrix-vector/triangular-matrix multiplication, and stencil computations.

4.1.2 Experimental configuration

In this chapter, the empirical analysis are carried out on an Intel Xeon E5-2650 v2 machine with 32 cores clocked at 2.6 GHz with 100 GB RAM, 32KB of L1 data cache, 256KB of L2 cache and 20MB of L3 cache. The machine runs CentOS 6.5 and 2.6.32-431 kernel, with GCC version 4.4.7, ICC version 15.0.0, and OpenMP version 4.0.

For empirical analysis, the number of threads are varied as 16, 32 and 64, and compared the speedup against that of the sequential version. The achieved speedup depends upon several factors such as the amount of total work, load imbalance across threads, overhead of thread creation, and synchronization. It is often smaller than the ideal speedup (K$\times$ with K threads). To have uniformity in the configuration across other frameworks, the default behavior of all the frameworks was retained during evaluation.

4.2 Auto-parallelization of PolyBench

The PolyBench/C 4.2 benchmarks are executed using the five frameworks namely, Cetus, Par4all, Rose, ICC, and Pluto. Various command-line parameters used to invoke the parallelization frameworks are shown in Table 2.3 referred in Chapter 2. It is observed that out of the 30 benchmarks, the sequential execution times of two benchmarks (`gesummv` and `deriche`) are negligible; hence, the remaining 28 compute-intensive programs is considered for this experimentation.

Table 4.1 shows static characteristics of the benchmarks, such as the application domain, the number of potentially parallelizable statements (**Stmts**), and also the number of static dependences (**Dep**) of the type Read-After-Write (**RAW**), Write-After-Write (**WAW**), and Write-After-Read (**WAR**). The chunky analyzer for dependences in loops (CANDL) tool [56], is used to obtain these characteristics. The eighth and the ninth columns of the table also list the problem size used for executing the benchmark and the sequential execution time in seconds for each benchmark. The benchmarks were grouped based on *loop-independent* and *loop-carried* dependences. It can be seen that 12 out of 28 benchmarks have *loop-independent* (□) computations and are likely to scale well with the number of threads . On the other hand, the remaining 16 benchmarks exhibit *loop-carried* (■) dependence which affects their scalability.

Table 4.1: Characteristics of PolyBench benchmarks and the parallelization results by various frameworks

Benchmarks	Domain	Stmts	Dep	RAW	WAW	WAR	Problem Size	Seq. Exec. Time T_S (s)	Cetus	Par4all	Rose	ICC	Pluto
correlation	DM	22	77	42	18	17	1E4	4219.65	✗[b]	✓	✓	✓	✓
covariance	DM	15	34	16	11	7	1E4	4220.66	✗[b]	✓	✓	✓	✓
gemm	LA	5	6	2	2	2	1E4	3620.39	✓	✓	✓	✓	✓
gemver	LA	7	13	7	3	3	1E5	403.94	✗[b]	✓	✓	✓	✓
syr2k	LA	5	6	2	2	2	1E4	6482.79	✓	✓	✓	✓	✓
syrk	LA	5	6	2	2	2	1E4	4060.29	✓	✓	✓	✓	✓
2mm	LA	10	13	6	4	3	1E4	20081.44	✗[b]	✓	✓	✓	✓
3mm	LA	15	19	10	6	3	1E4	24170.55	✗[b]	✓	✓	✓	✓
atax	LA	6	12	6	4	2	1E5	64.49	✓	✓	✓	✓	✓
bicg	LA	6	10	4	4	2	1E5	57.93	✓	✓	✓	✗[c d]	✓
doitgen	LA	10	30	10	10	10	1E3	4204.52	✗[b]	✓	✓	✗[c d]	✓
mvt	LA	4	6	2	2	2	1E5	307.31	✗[b]	✓	✓	✓	✓
symm	LA	10	33	11	11	11	1E4	6338.77	✗[b]	✓	✓	✓	✗[e]
durbin	LA	10	55	27	13	15	1E6	4205.76	✓	✓	✓	✗[c d]	✗[e]
gramschmidt	LA	14	34	17	8	9	1E4	7476.37	✓	✓	✓	✓	✓
ludcmp	LA	22	181	66	56	59	1E4	1839.32	✓	✓	✓	✗[c d]	✗[e]
trmm	LA	5	8	2	2	4	1E4	4194.12	✗[b]	✓	✓	✓	✓
adi	Stencils	34	140	68	22	50	1E4	43285.76	✗[b]	✓	✓	✓	✓
fdtd-2d	Stencils	11	28	12	4	12	1E4	14255.44	✗[b]	✓	✓	✓	✓
heat-3d	Stencils	8	42	20	2	20	1E4	27887.57	✗[b]	✓	✓	✓	✓
jacobi-1d	Stencils	4	14	6	2	6	1E6	7052.42	✗[b]	✓	✓	✓	✓
jacobi-2d	Stencils	6	22	10	2	10	1E4	14871.71	✗[b]	✓	✓	✓	✓
cholesky	LA	8	22	14	4	4	1E4	571.44	✗[a]	✗[a]	✗[a]	✗[c d]	✓
lu	LA	8	16	10	3	3	1E4	2603.65	✗[a]	✗[a]	✗[a]	✗[c d]	✓
trisolv	LA	4	13	7	4	2	1E5	15.61	✗[a b]	✗[a]	✗[a]	✗[c d]	✓
floyd-warshall	Medley	3	21	10	1	10	1E4	3484.39	✗[a]	✗[a]	✗[a]	✗[d]	✓
nussinov	Medley	11	52	19	9	24	1E4	1157.45	✗[a]	✗[a]	✗[a]	✗[d]	✓
seidel-2d	Stencils	3	27	13	1	13	1E4	13019.95	✗[a]	✗[a]	✗[a]	✗[d]	✓
28 codes	–	271	940	421	212	307	–	224153.69	8	22	22	18	25

□ *Loop-independent* dependence problems
□ *Loop-carried* dependence problems
[a] *Loop-carried* dependence problems
[b] If applying inlining+parallelization, these benchmarks can be parallelized (lack of inter-procedural optimization).
[c] Insufficient computational work
[d] Existence of *parallel dependence*
[e] Pluto fails in parallelizing loops comprising of scalar variables.

(Stmts = The number of potentially parallelizable statements, Dep = The number of static dependences, RAW = Read-After-Write, WAW = Write-After-Write, WAR = Write-After-Read, DM = Data Mining, LA = Linear Algebra, ✓Parallelized, ✗Not Parallelized)

4.2.1 Differences in parallelization

The support matrix indicating which benchmark is successfully parallelized by each framework is presented. Table 4.1 lists the successful and unsuccessful transformation of individual benchmark. Overall, it is observed that none of the frameworks parallelized all 28 programs. Pluto could parallelize the largest number (25) of benchmarks. Par4all and Rose could successfully parallelize the same set of 22 benchmarks each, whereas ICC parallelized 18 of them.

The scenarios which forbid the frameworks from parallelizing certain benchmarks were carefully analyzed. In order to elucidate the difference between the frameworks, the benchmarks were categorized into two groups namely, (i) Benchmarks with *loop-independent* problem (□) (ii) Benchmarks with *loop-carried* problems (□) which is highlighted in Table 4.1.

Pluto could parallelize all the benchmarks with *loop-carried dependence due to vector*. Other frameworks could support a subset of loop dependent problems, but they do not work when complex dependences are present. Cetus, Par4all, and Rose failed to parallelize some benchmarks with *loop-carried* dependences. Cetus relies on function inlining for improving the scope for intra-procedural optimizations [7]. However, Cetus does not inline functions by default. Hence, it failed in parallelizing some of the *loop-independent* dependence problems. Clearly, this is a serious limitation of the frameworks (with default behavior). ICC reported existence of parallel dependence but insufficient computational work.

Pluto worked with vectors but did not parallelize for-loops when the ℓ-value of the statement is a scalar variable. It failed to parallelize the benchmarks with *loop-carried dependence due to scalar*. This indicates that the constructs and the dependences appearing in PolyBench benchmarks are well-covered across the auto-parallelization frameworks.[i]

[i]Some implementation issues was encountered, which can be fixed by the framework developers. For instance, Cetus does not support multiple '\' (line continuation character) in a single preprocessor macro definition. As a fix, such definitions were split into multiple macro definitions with single '\'. Similarly, in Rose, *conditional* preprocessor macros are not correctly interpreted.

4.3 Result analysis

This section discusses the quantitative effect of individual parallelizer on various PolyBench benchmarks. Three benchmarks results are plotted to illustrate the scalability factor, `gemm`, `syr2k`, and `syrk` . The criteria for choosing these applications for scalability study is because these benchmarks have (i) *loop-independent* dependence and (ii) best speedup plot. Figure 4.1 shows the effect of number of threads on speedup. The results shows that the speedup scale well with increase in thread size (2, 4, 8, 16, 32, and 64). Hence for evaluation purpose, we examine the results across 16, 32, and 64 threads.

The parallelization results are discussed in more detail subsequently. Figures 4.2 and 4.3 shows speedup obtained by the frameworks on each benchmark. The performance is discussed by categorizing benchmarks based on the absence or presence of *loop-carried* dependence, which is discussed next.

4.3.1 Benchmarks with loop-independent dependence

Table 4.1 shows the benchmarks grouped under *loop-independent* dependence problems. It is observed from the support matrix that all frameworks support benchmarks with *loop-independent* dependence using privatization. At the same time, Cetus and ICC fail on several of these benchmarks. This happens because Cetus requires function inlining for improving the scope for intra-procedural optimizations (Section 2.3.1 in Chapter 2), while ICC reports the existence of parallel dependence. ICC also reports insufficient computational work for two of the benchmarks (`bicg` and `doitgen`).

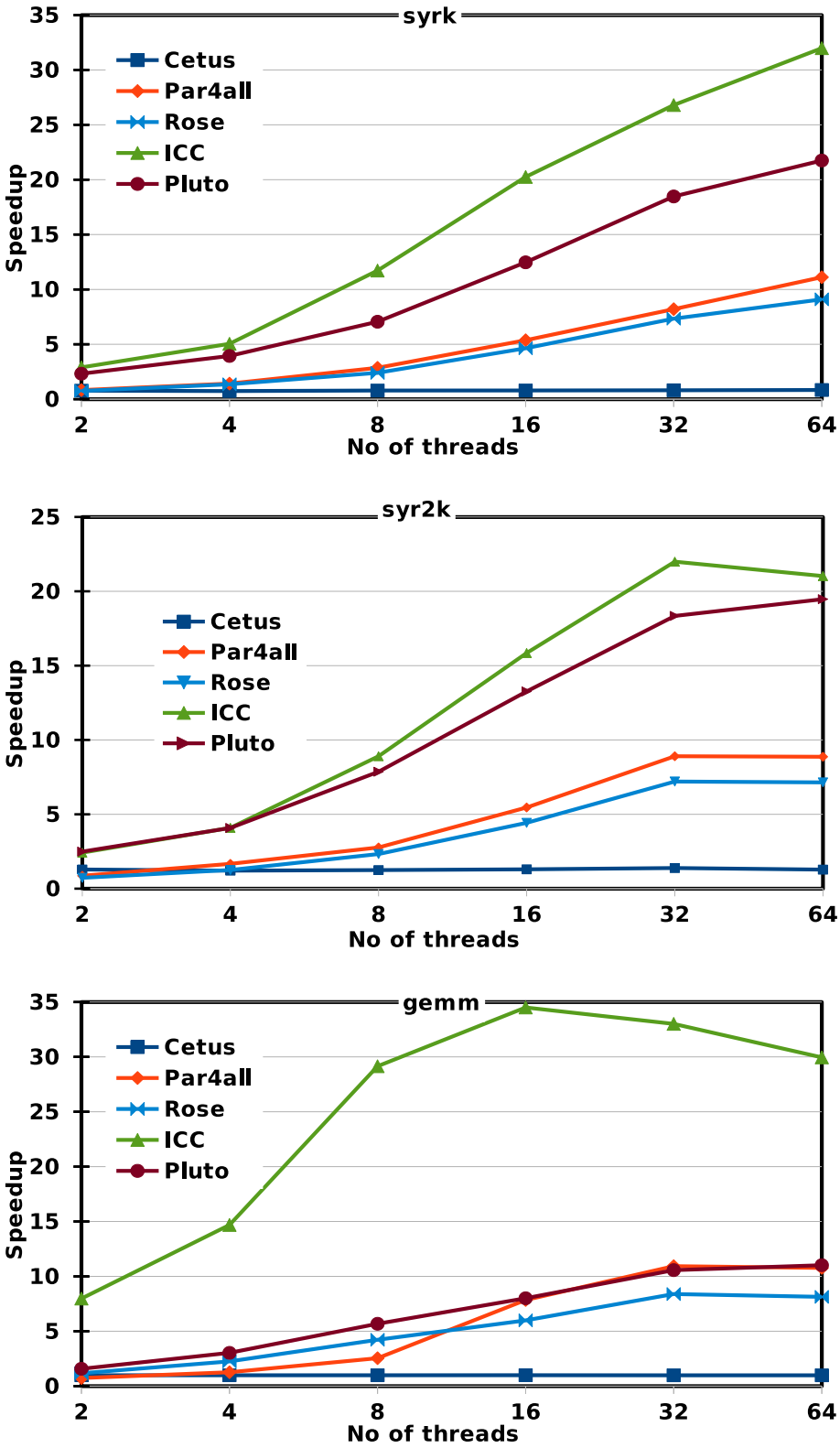


Figure 4.1: Effect of number of threads on speedup for benchmarks; `syrk`, `syr2k`, *and* `gemm`

This section also discusses the performance impact of frameworks in parallelizing the *loop-independent* dependence problems shown in Figure 4.2.[ii]

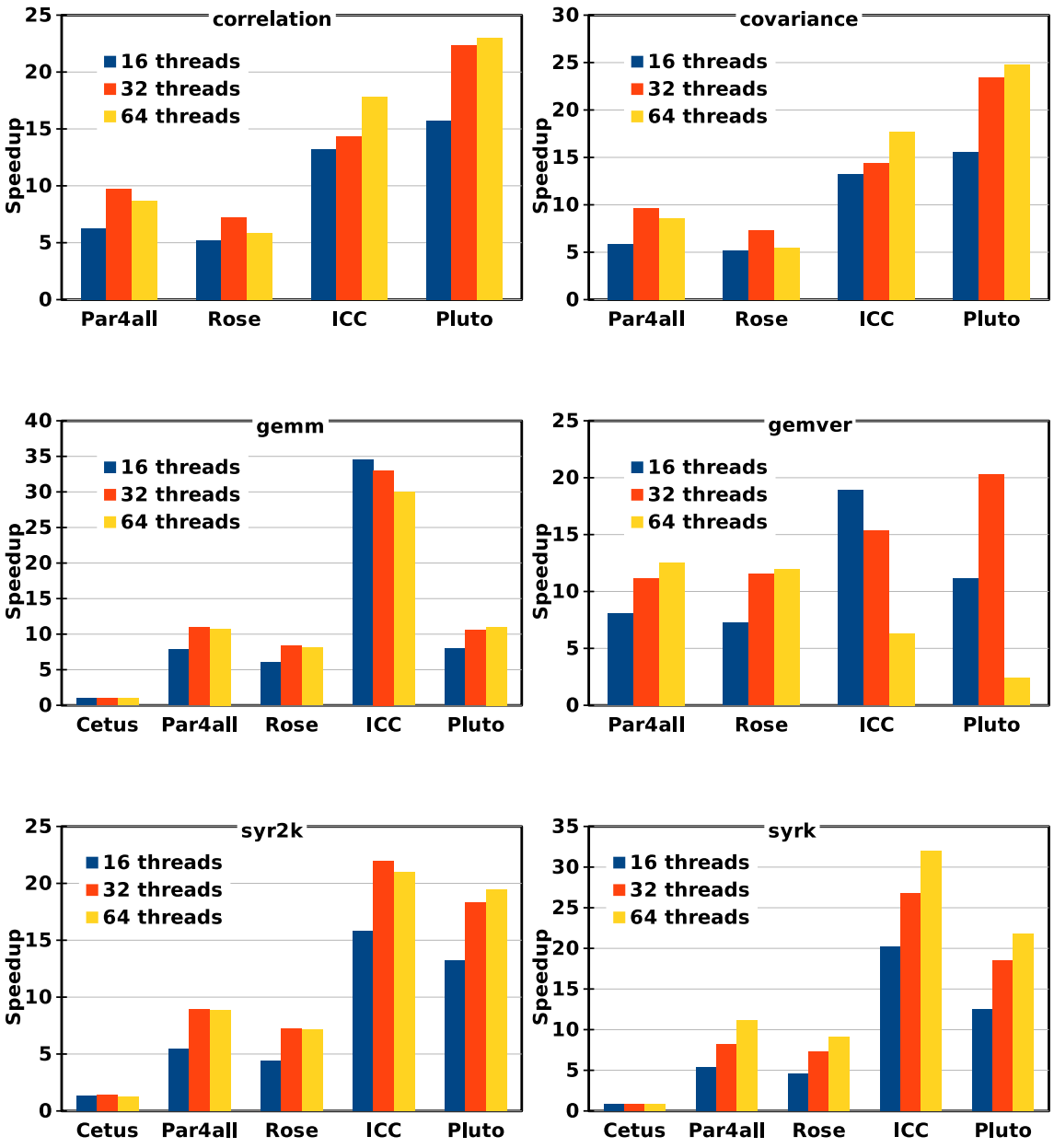
Overall, Pluto and ICC exhibit relatively better speedups compared to others. ICC achieves the best overall speedup (with 32 threads: `gemm` 33.0×, `syr2k` 22.0×, and `syrk` 26.8×; with 1 thread[iii]: `gemm` 5.1×, `syr2k` 2.3×, and `syrk` 2.8×). This is primarily due to vectorization and privatization. Pluto performs quite well (with 32 threads: `correlation` 22.4×, `covariance` 23.5×, `gemver` 20.2×, `2mm` 19.4×, `3mm` 13.1×, `atax` 4.6×, `bicg` 2.4×). The best speedup is achieved for Pluto due to the effect of polyhedral optimization and privatization. However, `atax` and `bicg` exhibit limited speedup as the parallel coverage is less compute-intensive. Also, Pluto achieves lesser speedup for 4 benchmarks (`gemm`, `syr2k`, `syrk`, `mvt`).

Among Cetus, Par4all and Rose, it is observed that Par4all performs better than Rose (with 32 threads: `correlation` and `covariance` 9.7×, `gemm` 10.9×, `syr2k` 8.9×, `syrk` 8.2×, `2mm` 14.6×, `3mm` 10.6×). Besides, Rose exhibits good results (with 32 threads: `gemver` 11.5×, `doitgen` 10.3×, `mvt` 10.7×). Cetus fares overall inferior to other frameworks obtaining average performance equivalent to the sequential run.

The effectiveness of the frameworks depends upon their techniques and methodologies. For instance, all the five frameworks apply array privatization. However, Cetus, Par4all and Rose do not perform any additional optimization. On the other hand, ICC relies heavily on vectorization while Pluto exploits polyhedral optimization. Although ICC and Pluto perform overall better than others, the performance improvements by Par4all and Rose suggest that array privatization provides considerable performance benefits.

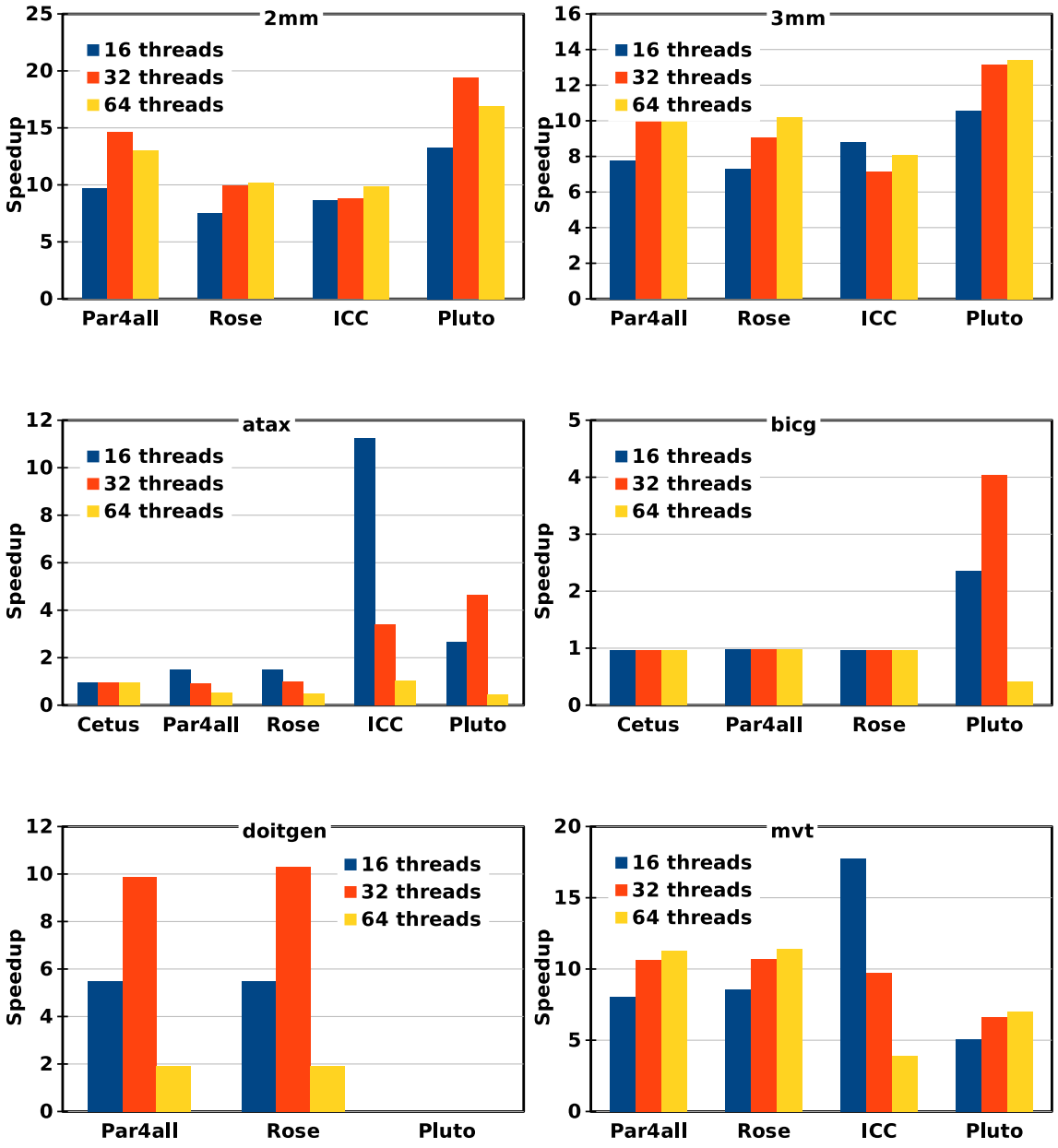
[ii] Benchmarks with 0 speedup indicate a timeout of three hour.

[iii] Since ICC's baseline involves default optimization (-O2), we also provide single-thread results. This is to show that ICC exploits better overall parallelism.



The speedup plots show the performance impact of the auto-parallelizers in parallelizing the *loop-independent* dependence problems.

Figure 4.2: Quantitative analysis of PolyBench benchmark with loop-independent dependences (`correlation`,`covariance`, `gemm`, `gemver`, `syr2k`, and `syrk`), (continued in next page)



The speedup plots show the performance impact of the auto-parallelizers in parallelizing the *loop-independent* dependence problems. Benchmarks with 0 speedup indicate a timeout of three hours.

Figure 4.2: Quantitative analysis of PolyBench benchmark with loop-independent dependences (`2mm`, `3mm`, `atax`, `bicg`, `doitgen`, and `mvt`) (Contd.)

4.3.2 Benchmarks with loop-carried dependences

The benchmarks shown in Table 4.1 exhibit three types of *loop-carried* dependence:

- Case 1: *Loop-carried dependence due to scalar variable(s)*. Such benchmarks are potentially amenable to hierarchical reduction. This is exhibited by four benchmarks – `symm, durbin, gramschmidt, ludcmp`.
- Case 2: *Loop-carried dependence due to vector variable(s)* in the inner/outer loop of a nested for. Such benchmarks are potentially amenable to simple loop transformations and array privatization. This is exhibited by nine benchmarks – `symm, durbin, ludcmp, trmm, adi, fdtd-2d, heat-3d, jacobi-1d, jacobi-2d`. Note that a benchmark may have multiple kinds of *loop-carried* dependence, based on different variables. Out of these nine, only `adi` contains dependence in the inner loop; all others contain dependence in the outer loop.
- Case 3: Complex *loop-carried* dependence. Such benchmarks are potentially amenable to sophisticated loop analysis such as polyhedral loop transformations. Due to complex dependence pattern, such benchmarks require considerable loop transformations or manual code changes for achieving parallelization. This is exhibited by totally six benchmarks, namely, `cholesky, trisolv, floyd-warshall, nussinov, lu, and seidel-2d`.

Table 4.1 shows the support matrix of benchmarks comprising *loop-carried* dependence problems for the above three cases. Investigation of the benchmarks in Case 1 reveals that all the frameworks except Pluto apply reduction variable recognition optimization [81] besides parallelization and array privatization. Since Pluto is a polyhedral tool and performs the affine transformation based on array-index expressions [21], it does not parallelize loops when the ℓ-value of a statement is a scalar.

For Case 2, the frameworks parallelize the dependence-free loop (inner/outer) of the nested for-loop as described in Section 3.2.2 in Chapter 3 via the micro-benchmarks, the following observations are made. Par4all and Rose parallelize all the benchmarks. However, ICC reports the existence of parallel dependence and also insufficient computational

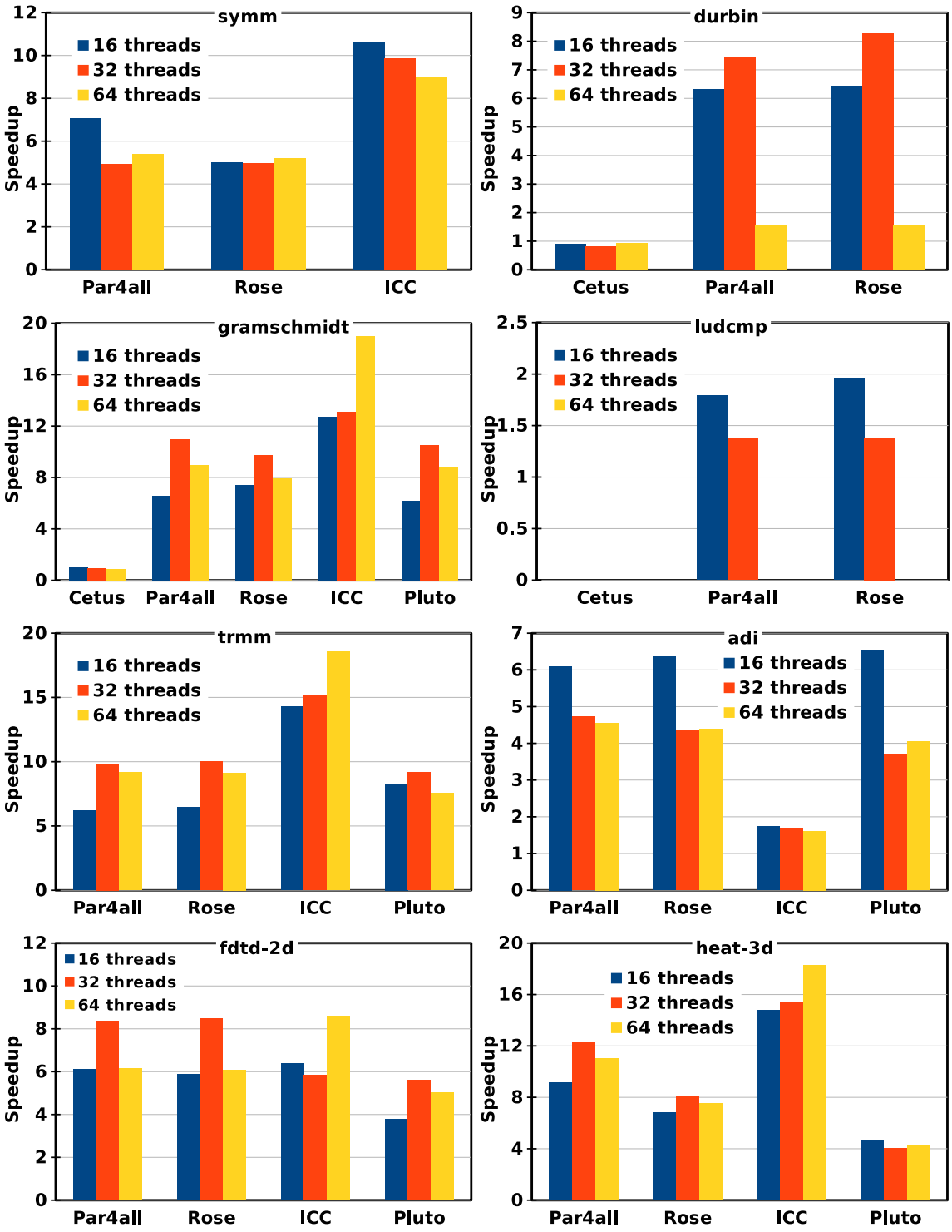
work for two of the benchmarks (`durbin` and `ludcmp`). Cetus supported the least number of benchmarks in our setup (`durbin`, `gramschmidt`, and `ludcmp`) as it does not, by default, support inter-procedural optimization[iv].

For Case 3, i.e., the benchmarks with complex *loop-carried* dependence are parallelized only by Pluto. These benchmarks do not have dependence-free loops; hence the scope for parallelization is less. It requires additional loop analysis and transformation to remove the dependences. Pluto applies loop peeling in addition to parallelization, privatization, and polyhedral optimization to achieve this. ICC reports existence of parallel dependence and also insufficient computational work for all the Case 3 benchmarks; and therefore, does not parallelize these loops.

The performance of the five frameworks in parallelizing the *loop-carried* dependence problems depicted in Figure 4.3 is discussed now. Overall, for Case 2 benchmarks, ICC performs better than all the other frameworks (with 32 threads: `symm` 9.8×, `gramschmidt` 13.1×, `trmm` 15.1×, `heat-3d` 15.4×, and `jacobi-1d` 18.4×; with 1 thread: `symm` 2.0×, `gramschmidt` 1.8×, `trmm` 2.0×, `heat-3d` 5.5×, and `jacobi-1d` 3.9×). This performance stems from ICC applying additional transformations such as loop peeling along with array privatization, parallelization, and vectorization. Note that the single-threaded performance of ICC also achieves good speedup over the sequential baseline as ICC enables -O2 optimizations by default. Among Cetus, Par4all, Rose, and Pluto frameworks, Par4all and Rose provide better benefits (Par4all with 32 threads: `gramschmidt` 11.0×, `adi` 4.7×, `heat-3d` 12.3×, `jacobi-1d` 11.3×; Rose with 32 threads: `symm` 5.0×, `durbin` 8.3×, `trmm` 10.0×, `fdtd-2d` 8.5×; Par4all and Rose with 32 threads: `jacobi-2d` 10.7×). This indicates that compared to polyhedral transformations, array privatization provides better performance improvement on PolyBench. Parallelization of `adi` shows that additional optimization (loop fission) by ICC affects the scalability.

For Case 3 benchmarks with complex dependences, only Pluto could parallelize the codes but the performance is relatively lower (with 32 threads: `cholesky` 5.6×, `lu` 8.8×, `trisolv` 2.3×, `floyd-warshall` 1.8×, `nussinov` 8.6×, and `seidel-2d` 9.1×).

[iv] Using function inlining, Cetus can improve its parallelization effect (Section 2.3.1). However, to have uniformity in the configuration concerning other frameworks, the default behavior of Cetus is retained and did not explicitly enable function inlining.



The speedup plots depict the performance of the auto-parallelizers in parallelizing the *loop-carried* dependence problems

Figure 4.3: Quantitative analysis of PolyBench benchmark with loop-carried dependences due to scalar/vector, (`symm`, `durbin`, `gramschmidt`, `ludcmp`, `trmm`, `adi`, `fdtd-2d`, `heat-3d`) (continued in next page)

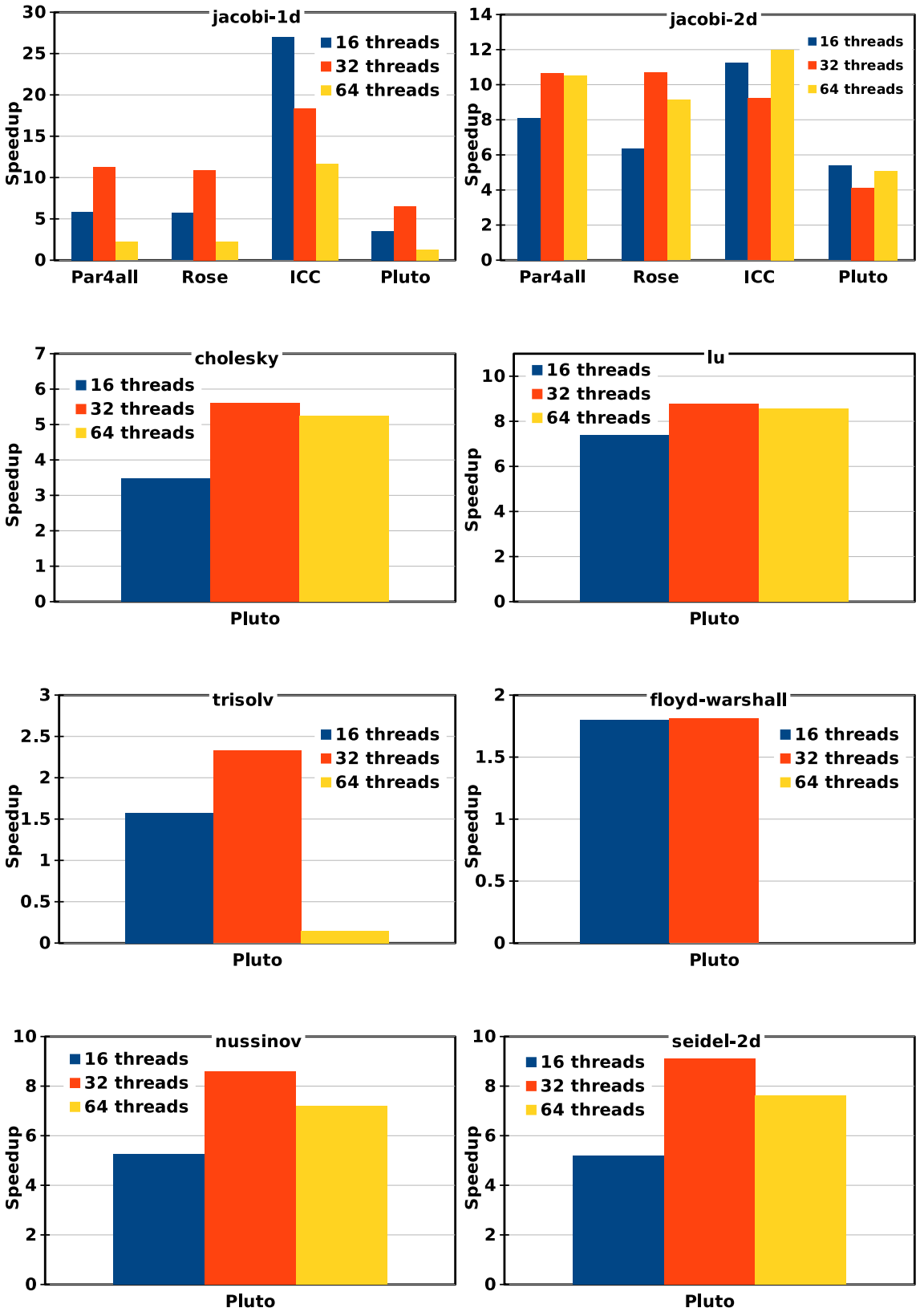


Figure 4.3: Quantitative analysis of PolyBench benchmark with loop-carried dependences due to scalar/vector (`jacobi-1d`, `jacobi-2d`, `cholesky`, `lu`, `trisolv`, `floyd-warshall`, `nussinov`, `seidel-2d`) (Contd.)

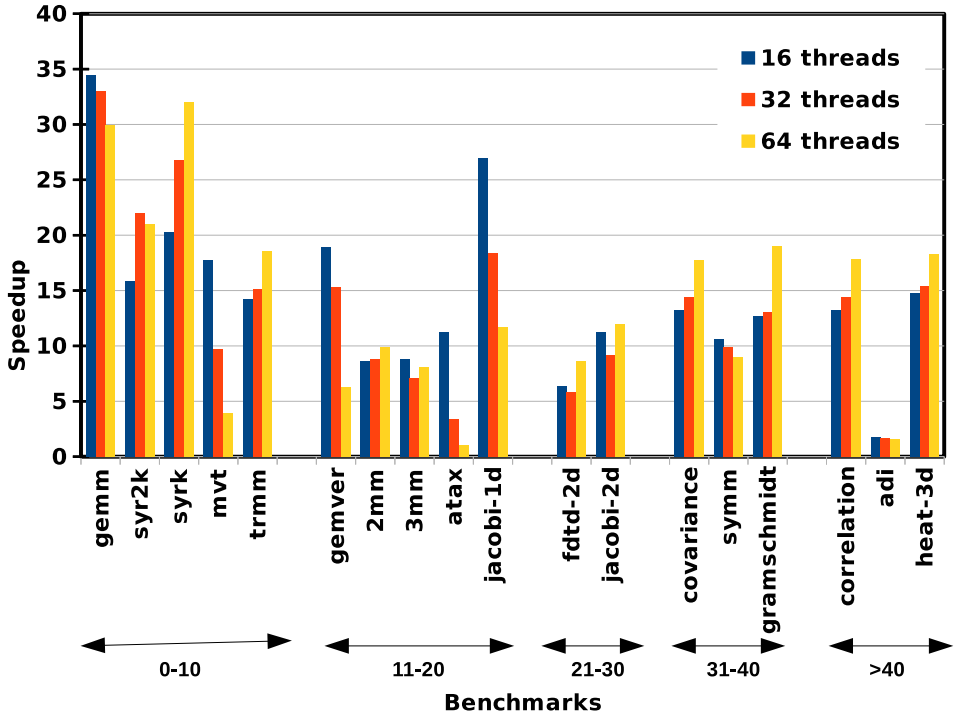
4.4 Effect of static dependences

The fourth column of Table 4.1 (named **Dep**) shows the number of static dependences present in each benchmark. To assess how it affects performance, we bucketize the range of **Dep** values: 0-10, 11-20, 21-30, 31-40 and >40, and study the performance of benchmarks falling into a range. The high-level expectation is that more the dependence, smaller is the parallelization benefit (however, note that the overall speedup gets affected by other factors also). Columns Stmts and RAW in Table 4.1 are also indicative. However, Stmts and RAW follow the same trend as **Dep**, and hence the below discussion applies also to these two parameters.

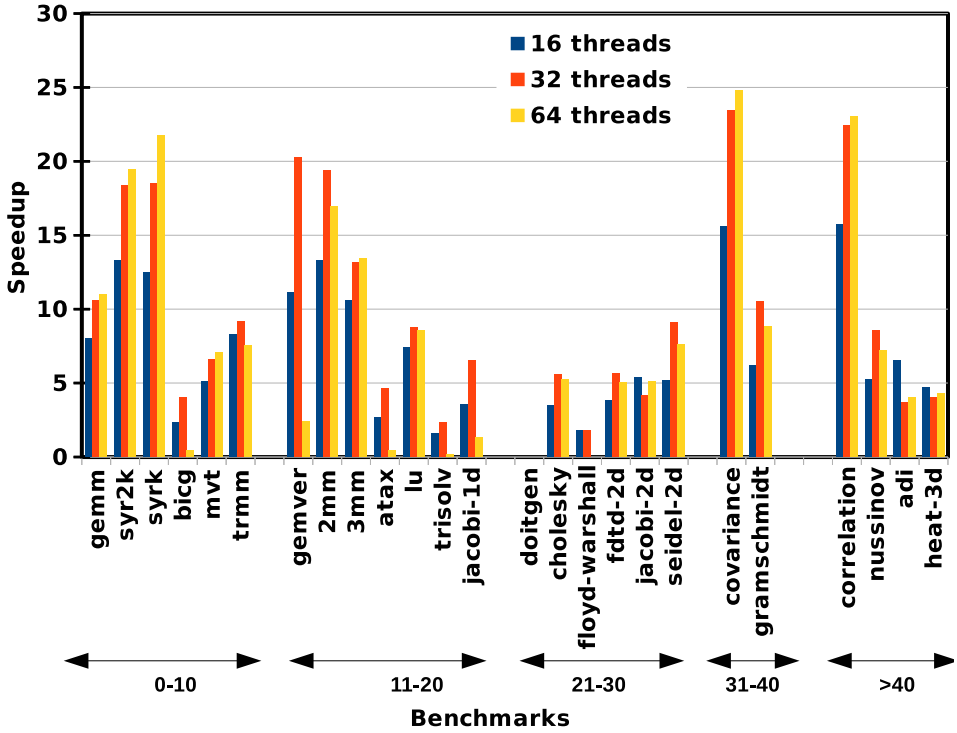
Figure 4.4 depicts the variation in speedup between individual **Dep** range of all the five frameworks. It is observed that for a large number of dependences, the speedup is usually relatively smaller. However, as expected, the speedup does not always reduce with increasing number of static dependences. Therefore, the number of static dependences alone cannot be conclusively used to predict the parallel performance.

Second, the maximum speedup is achieved by ICC (up to 34.5× for **Dep** between 0 and 10). Besides, on an average, the best speedup is achieved by ICC (average 11.1×). Pluto shows a considerable difference in performance across static dependence ranges. On the other hand, Cetus, Par4all, and Rose are relatively less sensitive to this value. This clearly indicates that the dependences are modeled more prominently in ICC and Pluto transformations. Among Cetus, Par4all and Rose frameworks, Cetus neither transforms many benchmarks nor achieves good parallelism on the ones which it successfully transforms. The performance benefits of Par4all and Rose are overall comparable but are significantly lesser than those of ICC and Pluto.

(a) ICC



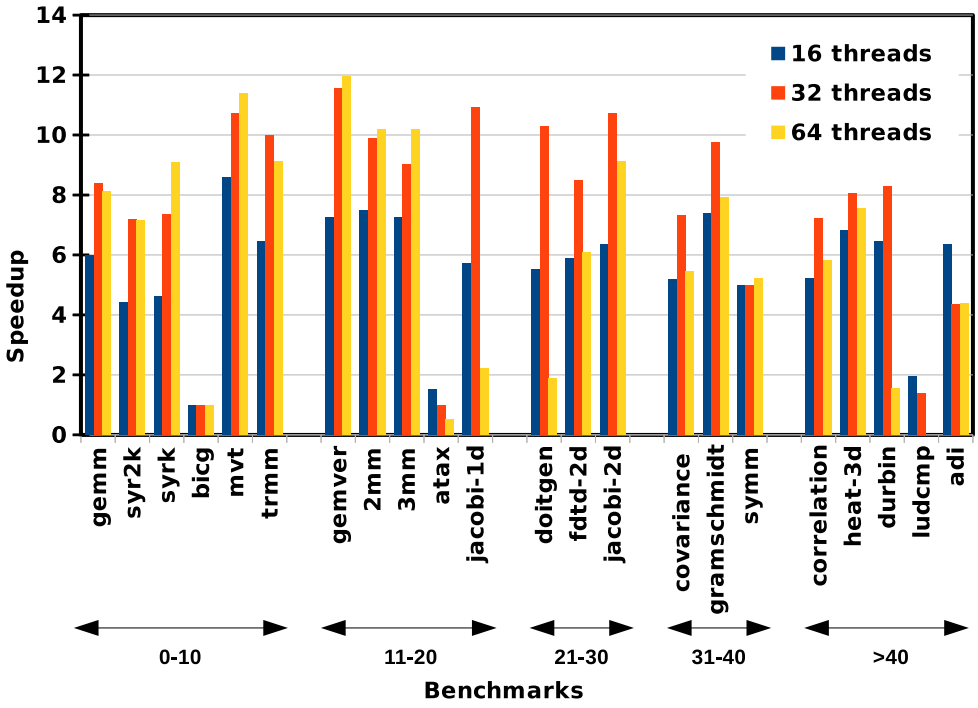
(b) Pluto



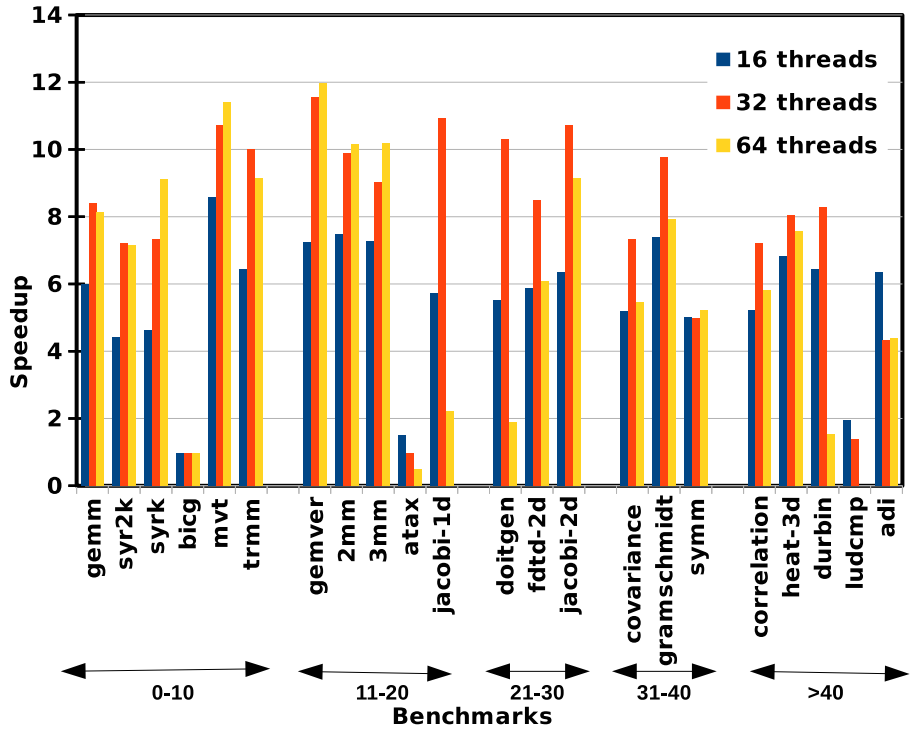
The variation in speedup (Y-axis) between individual Dep range (X-axis) of all the five frameworks is shown. Note that Y-axes use different scales across plots.

Figure 4.4: Effect of static dependences on PolyBench benchmark: A speedup analysis (a) ICC (b) Pluto, (continued in next page)

(c) Par4all



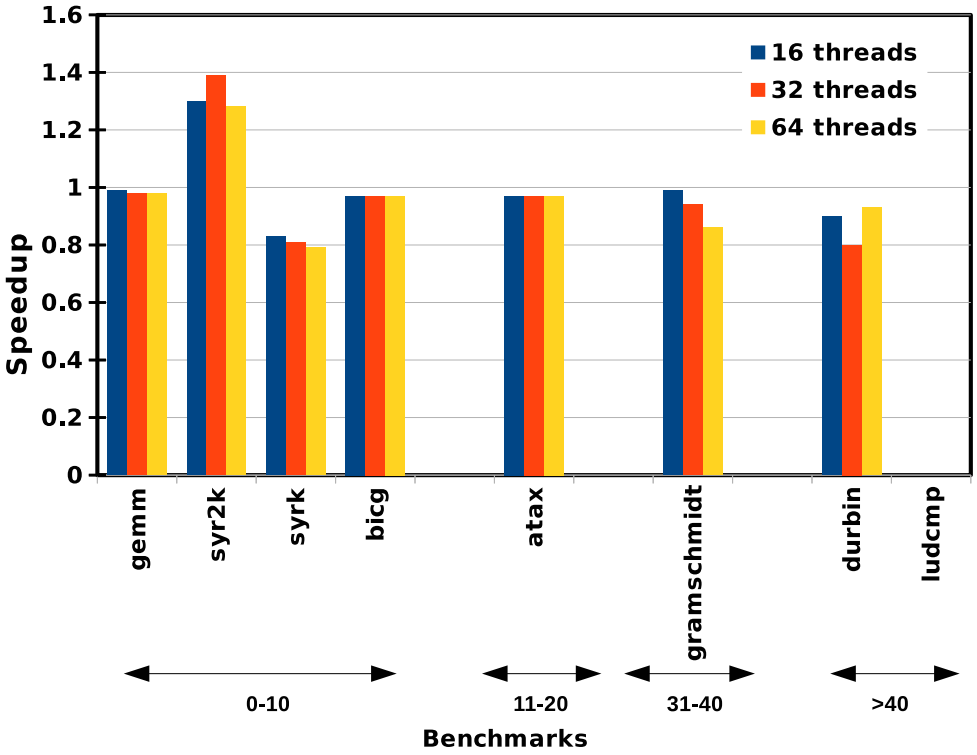
(d) Rose



The variation in speedup (Y-axis) between individual Dep range (X-axis) of all the five frameworks is shown. Note that Y-axes use different scales across plots.

Figure 4.4: Effect of static dependences on PolyBench benchmark: A speedup analysis (c) Par4all (d)Rose, (continued in next page)

(e) Cetus



The variation in speedup (Y-axis) between individual Dep range (X-axis) of all the five frameworks is shown. Note that Y-axes use different scales across plots.

Figure 4.4: Effect of static dependences on PolyBench benchmark: A speedup analysis (e) Cetus (Contd.)

4.5 Effect of individual techniques on parallelizers parallelized code

Hitherto, in the present study, auto-parallelizers with its default techniques had shown considerable performance benefits on PolyBench codes. This section describes the effect of individual techniques listed in Table 3.1 in Chapter 3 on the five frameworks. From Table 3.1, four different loop transformation techniques viz. (T1 - Privatization, T4 - Reduction, T6 - Tiling, T7 - Unrolling)[v] are chosen to study the performance that can be attributed when used across different frameworks. This examination is carried out to understand that additional tuning techniques applied to the default parallelization may lead to better speedup.

[v]The selection criteria is these techniques are supported by maximum parallelizers

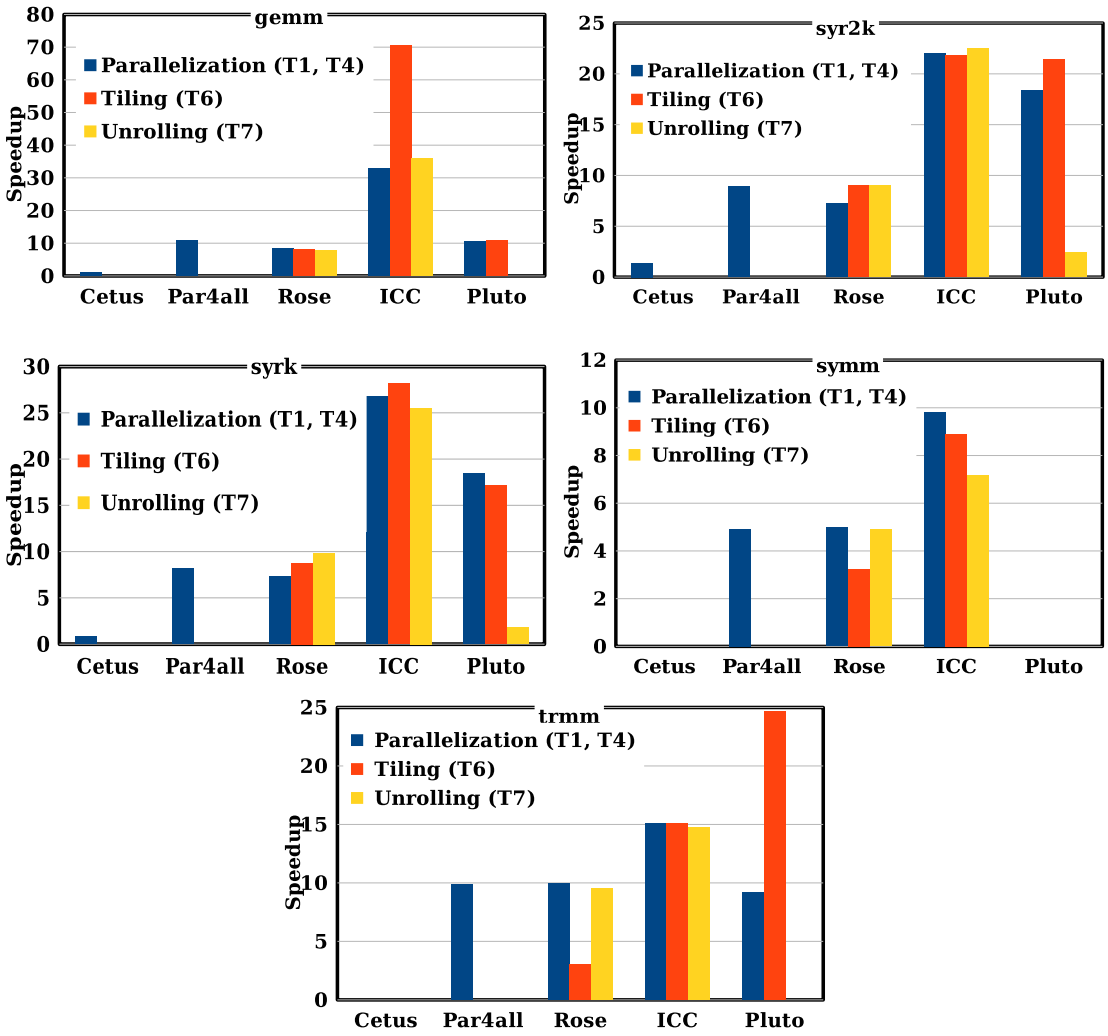
4.5.1 Effect of Tiling and Unrolling loop transformation techniques

Figure 4.5 shows the performance variation of T1, T4, T6, and T7 techniques on different parallelizers. T1 and T4 are performed by default when parallel flag is enabled. Throughout this section, effects of T1 and T4 are termed as Parallelization. T6 and T7 are supported by Rose, ICC, and Pluto. Although literature quotes T6 and T7 are supported by Par4all (Table 3.1), enabling Unrolling did not perform any transformation on the PolyBench suite. Also, to the best of our knowledge, documentation to enable Tiling is unavailable. Hence, for this study, effects of Par4all on T6 and T7 are excluded. Cetus does not perform Tiling and Unrolling.

The performance result shows that Tiling applied on Pluto has shown good speedup for `syr2k` 21.38×, and `trmm` 24.67× when compared to Parallelization and Unrolling. Similarly, ICC has shown good Tiling results for `gemm` 70.54×, and `syrk` 28.16×. Rose compiler has shown best Tiling results on two benchmarks on `syr2k` 9.00×, and `syrk` 8.78×), but however lesser than ICC and Pluto. Overall Tiling performs good in ICC and Pluto.

It was observed that Unrolling performed on Rose has shown considerable performance `syr2k` 9.00×, and `syrk` 9.86×. For ICC, there was no performance difference observed when Unrolling is performed. Unrolling performed on Pluto showed poor speedup than parallelized and tiled code.

Since Tiling and Unrolling are performed well on ICC and Pluto, we have applied these two techniques on 21 PolyBench benchmarks to analyze their efficacy.



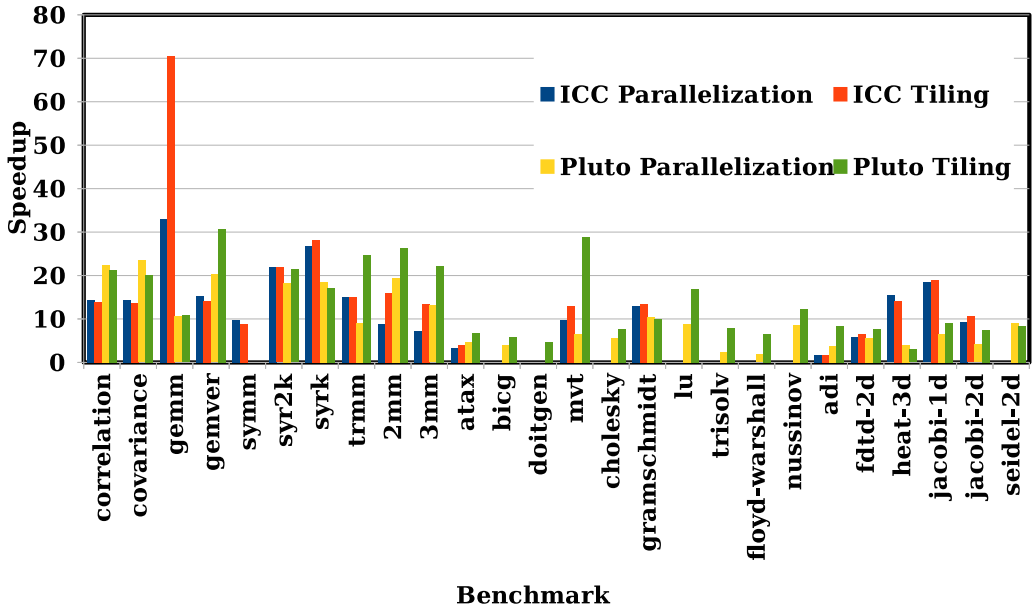
Parallelization (T1, T4) - This is the default option where either privatization/reduction performs automatically when parallel flag is enabled.
Tiling (T6) - Loop Tiling is enabled along with parallel flag (Tile size: 32)
Unrolling (T7)- Loop Unrolling is enabled along with parallel flag (Unroll factor: 8)

Figure 4.5: Effect of loop transformation techniques across different parallelizers : Speedup measured with 32 threads on PolyBench codes (`gemm`, `syr2k`, `syrk`, `symm`, and `trmm`)

Effect of Tiling over Parallelization Figure 4.6 depicts that ICC Tiling has shown performance improvement in 11 out of 18 codes than ICC Parallelization. ICC Tiling outperforms (50% more performance improvement than ICC Parallelization) for 4 benchmarks (speedup with 32 threads: `gemm` 70.54×, `2mm` 15.91×, `3mm` 13.47×, and `mvt` 13.03×).

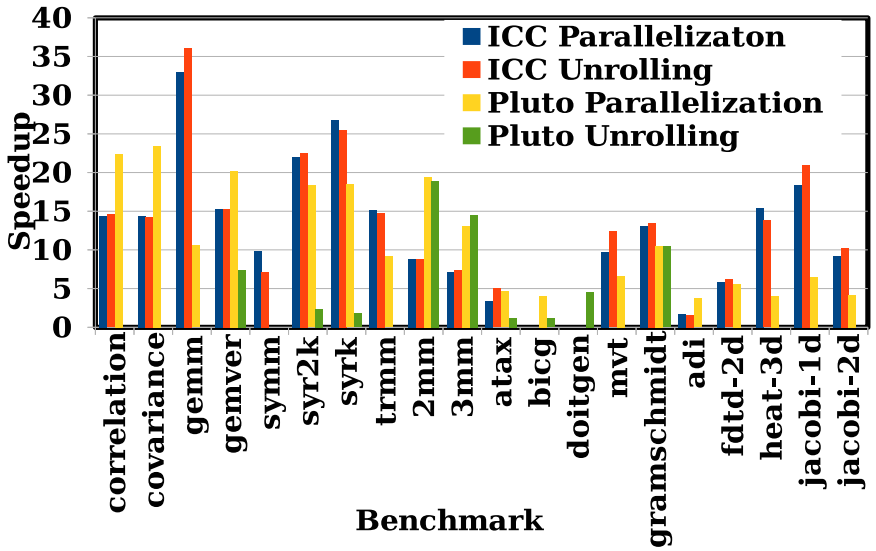
Figure 4.6 also shows that the usage of Tiling is well implemented in Pluto as the tool shows good speedup for 17 out of 25 benchmarks than Pluto Parallelization. Some of the good speedup results when executed with 32 threads are: `gemver` 30.79×, `trmm` 24.67×, `3mm` 22.21×, `mvt` 28.80×. Pluto Tiling outperforms ICC over 18 number of benchmarks, except the `gemm`, where ICC shows 70.54× speedup.

Effect of Unrolling over Parallelization Figure 4.7 shows that ICC gets benefited using Unrolling, whereas Pluto performs poorer due to this flag. ICC Unrolling has shown performance improvement for 10 out of 18 benchmarks (some of the speedup with 32 threads: `gemm` 36.09×, `atax` 5.02×, and `mvt` 12.38×). Pluto Unrolling is supported only for 9 out of 25 benchmarks. Some of the cases exhibited erroneous transformation and reasons could not be faithfully deduced, while several other cases do not perform Unrolling. Pluto Unrolling has degraded the performance than Pluto Parallelization, except for two benchmarks (speedup with 32 threads: `3mm` 14.47× and `doitgen` 4.57 ×).



ICC Parallelization / Pluto Parallelization- Only parallel flag enabled
ICC Tiling / Pluto Tiling - Loop Tiling + parallel flag enabled

Figure 4.6: Effect of Tiling on Pluto and ICC on PolyBench codes: Speedup measured with 32 threads



ICC Parallelization / Pluto Parallelization- Only parallel flag enabled
ICC Unrolling / Pluto Unrolling - Loop Unrolling + parallel flag enabled

Figure 4.7: Effect of Unrolling on ICC and Pluto on PolyBench codes: Speedup measured with 32 threads

4.6 Summary of performance

PolyBench codes were used to carry out the empirical analyses on five different auto-parallelizers. The benchmarks inherently possess various dependence structures. It was observed that there were no PolyBench benchmarks that could not be parallelized using any of these frameworks. The benchmarks were categorized based on two dependence types namely, (i) *loop-independent* dependence, and (ii) *loop-carried* dependence. Following were the observations made:

1. All five auto-parallelizers considered for the study were functionally supporting parallelization of *loop-independent* dependence problems.

2. Although differences occurred in the support matrix of the PolyBench codes by distinct auto-parallelizers, Pluto could parallelize more number of benchmarks.

3. Exclusively, Pluto successfully parallelized applications with complex *loop-carried* dependence using loop peeling. The limitation of Pluto is that it failed in parallelizing

loops with a scalar.

Table 4.2 summarizes the overall results of 16, 32, and 64 threads from Figures 4.2 and 4.3. It lists the average speedup achieved by all the auto-parallelization frameworks under study on PolyBench benchmark. The average speedup is calculated using the geometric mean equation (eqn. 4.1) for all five parallelizers. The geometric mean is calculated for individual parallelizers by taking the n^{th} root of the product of n numbers i.e. x_1, x_2,...x_n (speedup of individual benchmark); n denotes the total number of benchmarks parallelized by each parallelizer.

$$\Big(\prod_{i=1}^{n} x_i\Big)^{(1/n)} = \sqrt[n]{x_1 x_2 ... x_n} \tag{4.1}$$

Table 4.2: Performance summary of auto-parallelizers on PolyBench codes (with 16, 32 and 64 threads)

No. of threads	Cetus	Par4all	Rose	ICC	Pluto
16	0.98	5.48	5.01	12.47	5.94
32	0.97	7.00	6.37	11.09	7.82
64*	0.96	5.70	5.02	10.22	5.18

The table denotes the geometric mean calculated for parallelized benchmarks across five different parallelizers. The execution times of the benchmarks resulting in a timeout were excluded.
*Though there are 32 physical cores, we experiment with 64 threads as sometimes context switching becomes beneficial for applications that are memory intensive and are I/O bound.

Performance analysis shows that among the five frameworks, ICC stands out as it does vectorization besides parallelization and additional optimization. Pluto and Par4all resulted in significant performance improvement. It showed that polyhedral optimization exhibited by Pluto, and array privatization evinced by Par4all showed considerable benefits.

Cetus supported the least number of benchmarks due to dependence issues and lack of support for intra-procedural optimization. It performs inferior to all other frameworks.

4.6.1 Discussion

This section discusses the effect of inlining on Cetus performance. To understand and analyze the Cetus parallelization, function inlining was enabled. In total, 8 benchmarks were used to bring out the differences. The benchmarks that were not parallelized with default option of Cetus are parallelized after inlining is enabled. Figure 4.8 shows that all the 8 benchmarks had shown good performance improvement. This examination was performed to complement Cetus as the literature revealed that in most cases the tool had performed well. However, to have uniformity in our analysis, default options of the parallelizing compilers were used for this study.

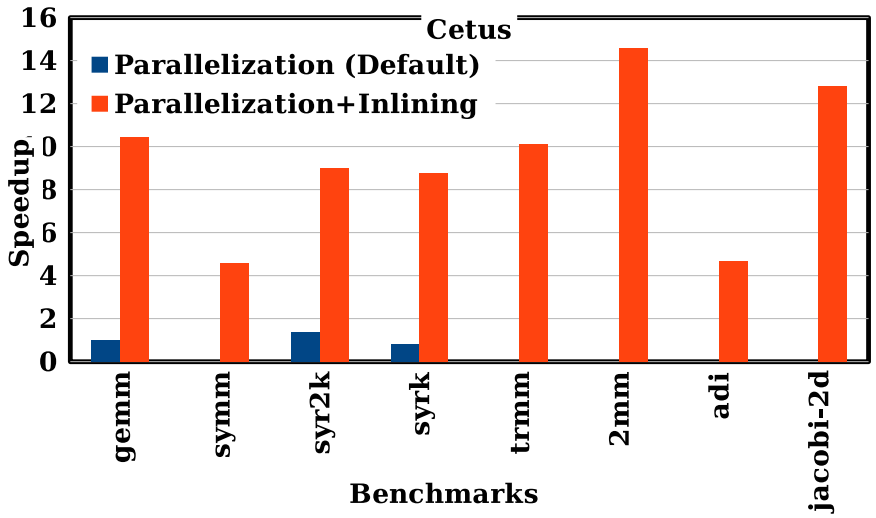


Figure 4.8: Effect of function inlining on Cetus: Speedup measured with 32 threads

CHAPTER 5

Quantitative Analysis of Parallelization Frameworks using NAS Parallel Benchmarks

This chapter explains the effectiveness of frameworks in parallelizing NAS Parallel benchmarks (NPB). The successful, semi-successful and unsuccessful parallelization of benchmarks are studied in detail. It discusses the pre- and post-transformation problems encountered by the frameworks during parallelization. A detailed analysis on the NPB results is carried out.

5.1 Background

Empirical analysis of five different auto-parallelizers namely Cetus, Par4all, Rose, Intel C compiler (ICC), and Pluto are carried out using NAS parallel benchmarks (NPB) [105, 106, 107, 108]. In this chapter, the frameworks are examined for the performance of their transformed codes. Popular benchmark suite NPB was used as a workload to carry out the experiments.

NPB is a suite of ten programs, and the benchmarks are primarily derived from the domain of computational fluid dynamics (CFD). It is a widely-used suite for assessing the effectiveness of large-scale parallel executions and in literature for evaluation of parallelization proposals. It includes applications dealing with unstructured adaptive mesh, parallel I/O, multi-zone applications, and computational grids. Drozdov et al. [109] tested different optimization techniques in low level virtual machine (LLVM) compiler using NPB and also compared the performance of different compilers. Griebler et al. [108] provided an efficient C++ version of NPB benchmark kernels by replacing the Fortran code.

This chapter focuses on the parallelization of NPB-C benchmark suite using auto-parallelizers. Unlike PolyBench, each NPB benchmark comprises of different dependence structures including *loop-independent* and *loop-carried* due to a scalar/vector. Also, it has several functions involving computationally intensive loops with complex coding style (which matters for source-to-source translators).

5.1.1 Experimental configuration

The experimental configuration is illustrated in Chapter 4. For empirical analysis, the number of threads are varied as 64, 32 and 16, and compared the speedup against that of the sequential version. The achieved speedup depends upon several factors such as the amount of total work, load imbalance across threads, overhead of thread creation, synchronization, etc. It is often smaller than the ideal speedup (K$\times$ with K threads). To have uniformity in the configuration concerning other frameworks, the default behavior of all the frameworks was retained during evaluation.

5.2 Auto-parallelization of NAS parallel benchmarks (NPB)

The NAS parallel benchmarks (NPB3.3-SER-C) are executed through the five frameworks under study. All ten NPB benchmarks are used for the study.

5.2.1 Differences in parallelization

Table 5.1 presents the effectiveness of the frameworks in auto-parallelizing the benchmarks. Description of individual benchmark is shown in Table 5.2. It illustrates the differences in parallelization by frameworks.

Successful parallelization symbolizes two aspects: (i) code transformation happened, and (ii) the transformed code is equivalent to the serial version in terms of its output. The correctness check is programmatically performed by the test case of the individual benchmark. It is observed that most of the benchmarks were successfully parallelized (✓).

Semi-successful parallelization indicates that either the code transformation or the semantic equivalence was not achieved. The former occurs due to pre-transformation issues (α) such as timeout or syntactical errors during conversion. The latter is due to post-transformation issues (β) which result in erroneous code transformation. The issues in

Table 5.1: Complexity measurement of NPB using chunky analyzer for dependences in loops (CANDL) and the parallelization results by various frameworks

Bm.	Stmts	Dep	RAW	WAW	WAR	Seq. Exec. Time T_S (S)	Cetus	Par4all	Rose	ICC	Pluto
BT	419	2412	-	-	-	2591.08	✗[h]	✓	β^{d} ◪	✓	$\alpha^{a}\beta^{g}$ ◪
CG	11	47	18	18	11	387.17	✓	✓	✓	✓	✓
DC	5	3	2	1	0	911.11	✓	$\alpha^{b}\beta^{c}$ ◪	$\alpha^{j}\beta^{c}$ ◪	✗[i]	β^{c} ◪
EP	13	58	26	20	12	616.47	✓	✓	β^{f} ◪	✓	β^{c} ◪
FT	9	5	2	2	1	668.01	☒ [a]	✓	☒[j]	✓	β^{g} ◪
IS	4	12	4	4	4	20.17	✓	✓	☒ [a]	✓	✓
LU	1282	15708	-	-	-	2931.64	✓	✓	$\alpha^{j}\beta^{c}$ ◪	✓	$\alpha^{a}\beta^{g}$ ◪
MG	11	47	18	18	11	256.44	✓	✓	✓	✓	β^{g} ◪
SP	230	884	305	263	316	2568.85	✓	✓	β^{d} ◪	✓	β^{g} ◪
UA	19	20	8	6	6	2081.15	✓	β^{c} ◪	☒ [e,g]	✓	☒ [e,g]
10	2003	19196	383	332	370	13032.09	8	10	7	9	9

* Problem size CLASS C is used for all benchmarks except `DC`. For `DC`, CLASS B is used

Bm. Benchmark, ✗ non-parallelized, Stmts = The number of potentially parallelizable statements, Dep = The number of static dependences, RAW = Read-After-Write, WAW = Write-After-Write, WAR = Write-After-Read, ☒ Unsuccessful, ✓ Successful, ◪ Successful after subjecting to changes, α Changes for pre-transformation issues, β Changes for post-transformation issues; for BT and LU, Candl invoked by Pluto timed out and hence some statistics are not displayed

[a] Timeout after an hour
[b] Due to recursion
[c] Erroneous code transformation
[d] Privatization error
[e] Parallelized code output is not equivalent to serial code
[f] Reduction error
[g] Error *after* code transformation and the Source of the error could not be deduced
[h] Lack of support for inter-procedural optimizations
[i] Loops are not parallelization candidates, there exists dependence across iterations, and also insufficient computational work
[j] Error *during* code transformation and the source of error could not be deduced

Table 5.2: Description of NAS parallel benchmarks (NPB)

Benchmark	Description
BT	Block Tri-diagonal solver
CG	Conjugate Gradient, irregular memory access and communication
DC	Data Cube
EP	Embarrassingly Parallel
FT	Discrete 3D fast Fourier Transform, all-to-all communication
IS	Integer Sort, random memory access
LU	Lower-Upper Gauss-Seidel solver
MG	Multi-Grid on a sequence of meshes, long and short distance communication, memory intensive
SP	Scalar Penta-diagonal solver
UA	Unstructured Adaptive mesh, dynamic and irregular memory access

the semi-successful codes were manually rectified. The codes that were addressed using the pre-transformation (α) and the post-transformation (α) changes are marked as ☑ in the Table 5.1. A few semi-successful benchmarks could be successfully parallelized using Par4all, Rose and Pluto after this manual intervention (e.g., `DC` by Par4all), denoted as α for pre-transformation and β for post-transformation (discussed more later in this section). After applying all the changes, Par4all could parallelize all ten NPB benchmarks. ICC and Pluto were a close second parallelizing 9 of them. Cetus could successfully parallelize eight, whereas Rose parallelized seven of them.

Despite the manual fixes, few of the benchmarks under study were *unsuccessful* and are represented in the table using symbol ☒. Pluto and Rose produced semantically incorrect output for `UA`, but the source of the error is not deduced. Cetus and Rose failed to parallelize `FT` and `IS` respectively due to timeout[i]. Rose could not parallelize `FT` due to an error *during* code transformation. There are non-parallelized benchmarks exhibited by a few of the tools and are shown in the table using symbol ✗. ICC did not parallelize

[i]Timeout indicates codes taking more than an hour for program transformation.

DC and reported that some of the loops are not parallelization candidates either because there exists dependence across iterations, or insufficient computational work. On the other hand, Cetus failed to parallelize BT due to lack of support for inter-procedural analysis (see Section 2.3.1 in Chapter 2). Interestingly, CG is the only benchmark which is successfully transformed by all the frameworks without any changes. The above discussed observation poses a serious question-mark over the expressivity and generality of auto-parallelization frameworks, which is one of the takeaways of this study.

5.2.2 Details of the transformation errors: manual changes on pre-transformation (α) and post-transformation (β) issues

This section first discusses about two pre-transformation issues (α) namely, (i) timeout, and (ii) error *during* code transformation in Table 5.1, and the manual changes to fix those issues.

(i) **Timeout**: For BT, Pluto timed out[ii] for the files *x_solve*, *y_solve*, and *z_solve*. Similarly, Pluto timed out in transforming file *erhs* of LU. This is due to more number of statements within the parallelizable region. Files that caused timeout were excluded from further evaluation.

(ii) **Error *during* code transformation**: For DC, auto-parallelization of files *jobcntl* and *adc* by Rose was unsuccessful, and the source of the error could not be deduced. Likewise, Rose produced error *during* transformation of file *ssor* of LU. So for the timeout and the error cases, the original sequential code was used for the evaluation. For DC, Par4all fails to transform due to recursive functions *WriteViewToDisk(), WriteViewToDiskCS(), computeChecksum()*, and *WriteChunkToDisk()*. Hence, the recursive functions was eliminated during transformation. Figure 5.1(a) shows the snippet of the recursive code from DC which forbid parallelization.

This section next illustrate the applied changes to the post-transformation problems (β), namely, (i) illegal transformation, (ii) incorrect transformation, and (iii) unrecognized problems. These are listed in the last five columns of Table 5.1.

[ii]Timeout indicates codes taking more than an hour for program transformation.

(a) Par4all fails to parallelize DC due to recursive WriteViewToDisk()

```
int32 WriteViewToDisk(ADC_VIEW_CNTL *avp, treeNode *t){
if(!t) return ADC_OK;
if(WriteViewToDisk( avp, t->left)) return ADC_WRITE_FAILED;
for(i=0;i<avp->nm;i++)
  avp->mSums[i] += t->nodeMemPool[i];
WriteToFile(t->nodeMemPool,avp->outRecSize,1,avp->viewFile,avp->logf);
if(WriteViewToDisk( avp, t->right)) return ADC_WRITE_FAILED;
return ADC_OK;
}
```

(b) Invalid privatization + reduction transformation by Rose (left) and valid manual transformation (right): EP benchmark

Rose

```
#pragma omp parallel for private
    (gc,i) reduction (+:gc)
  for (i = 0; i <= 9; i += 1)
    gc = gc + q[i];
```

Manual

```
#pragma omp parallel for private
    (i) reduction (+:gc)
  for (i = 0; i <= 9; i += 1)
    gc = gc + q[i];
```

(c) Rose produces illegal transformation (return within parallel pragma): DC benchmark

```
#pragma omp parallel for private (i)
  for (i = 0; i <= n - 1; i += 1)
    if (a[i] < b[i]) return - 1;
    else if (a[i] > b[i]) return 1;
```

(d) Serial DC code and incorrect for loop initialization by Par4all

Serial

```
for(pn=0,gbi=0,ii=nViews-1;ii>=0
    ;ii--)
.....
```

Par4all

```
for(ii = ii; ii >= 0; ii += -1)
.....
```

Figure 5.1: Pre- and post-transformation issues (a) Par4all fails to parallelize DC due to recursive WriteViewToDisk() (b) Invalid privatization + reduction transformation by Rose (left) and valid manual transformation (right) (c) Rose produces illegal transformation (return within parallel pragma) (d) Serial DC code and incorrect for loop initialization by Par4all, *(continued in next page)*

(e) Erroneous code transformation by Pluto (left) and correct manual transformation (right): `DC` benchmark

Pluto

```
if (numDims >= 1)
{
  lbp=0;
  ubp=numDims-1;
  #pragma omp parallel for
      private(lbv, ubv)
  for (t1=lbp;t1<=ubp;t1++)
    SetOneBit(binRepTuple,selTuple
        [t1]-1 );
}
void SetOneBit(uint64 *s, int32
    pos)
{
 uint64 ob = MLB;
 ob >>= pos;
 *s |= ob;
}
```

Manual

```
if (numDims >= 1)
{
  lbp=0;
  ubp=numDims-1;
#pragma omp parallel for private
    (lbv, ubv)
 for (t1=lbp;t1<=ubp;t1++)
 {
  uint64 ob = MLB;
  ob >>= selTuple[t1]-1;
#pragma omp atomic
  *binRepTuple |= ob;
 }
}
```

(f) Serial `EP` code (left) and incorrect code transformation by Pluto

Serial

```
t1 = A;
 for (i = 0; i < MK + 1; i++)
    t2 = randlc(&t1, t1);
an = t1;
```

Par4all

```
tt1 = A;;
an = tt1;;
if (MK >= 0)
  for (t2=0;t2<=MK;t2++)
    tt2 = randlc(&tt1, tt1);;
```

Figure 5.1: Pre- and post-transformation issues (d) Erroneous code transformation by Pluto (left) and correct manual transformation (right) (e) Serial `EP` code (left) and incorrect code transformation by Pluto (Contd.)

(i) Illegal transformation: For `BT` and `SP`, Rose-transformed code resulted in syntax error in function *compute_rhs()* as the tool applies privatization twice to the same index variable. For evaluation, the duplicate privatization was removed. Similarly, for `EP`, the converted code from Rose encountered syntax error as the framework performs reduction and privatization to the same variable **gc**. Once again, for correct output, the duplicate conversion was removed. Figure 5.1(b) demonstrates the valid and invalid transformations respectively.

For `DC`, Rose produced illegal parallelization in function *KeyComp()* as depicted in Figure 5.1(c). Having a return statement within parallel pragma is not allowed. Hence, OpenMP reports invalid branch to/from an OpenMP structured block. For `DC`, Pluto applied incorrect parallelization by inserting parallel directive to a for-loop comprising of a function call with dependence.

(ii) Incorrect syntactical/logical transformation: For `DC`, Par4all exhibited two incorrect transformations in functions *PrefixedAggregate()* and *MultiFileProcJobs()*. In *PrefixedAggregate()*, Par4all by default converts switch-case to if-else statements. During this process, it creates a flag variable suffixed and prefixed as '$-$' which results in incorrect identifier syntax. During the manual intervention, the '$-$' symbols was removed from the variable name. In *MultiFileProcJobs()*, Par4all modified the for-loop initialization incorrectly, so the serial version was retained as depicted in Figure 5.1(d).

Figure 5.1(e) illustrates Pluto's transformation in *CreateBinTuple()*, where the shared variable *s must be atomically updated. Pluto could not correctly identify this scalar dependence across function call and resulted in incorrect transformation. In the manually transformed code, inlining of function was applied and then atomically updated the variable *s. However, for evaluation purpose, the OpenMP pragma was removed to produce the correct output.

For `EP`, Pluto failed to preserve a dependence while applying data locality optimization, resulting in incorrect output. Hence, the dependence was fixed manually. Figure 5.1(f) shows the original and the incorrectly transformed codes. In case of `LU` benchmark, Rose performed incorrect transformation in *l2norm()* by replacing one of its arguments, (`double v[][ldy/2*2+1][ldx/2*2+1][5]` as `double v[][][][5]`). This

led to syntax error due to the array having incomplete element type. Hence it was replaced with the original array definition. For UA, the parallelized code by Par4all resulted in a segmentation fault. The investigation revealed that closing braces were wrongly inserted at multiple places in file *mason*, which retained a valid C syntax, but changed the control-flow. So, reverted to the original file for evaluation.

(iii) Unrecognized problems: For few of the incorrect transformations by Pluto, though the source of the error is not deduced, the function which led to such a pitfall could be identified. For BT, the incorrect output is due to the code segment in function *initialize()*. Similarly, for benchmarks FT and MG, the problematic region is *compute_initial_conditions()* of FT and *zran3()* of MG. For LU, the incorrect transformation was due to *error(), setbv()* before parallelization. For SP, *rhs_norm()* results in incorrect output. The problematic functions was excluded from further parallelization and replaced them with the corresponding serial versions. While such a replacement would reduce performance, it ensures correct output.

5.3 Experimental results: NPB result analysis

This section discusses the quantitative effectiveness of various parallelization frameworks on NPB benchmarks. Table 5.1 shows the description and the static characteristics of the benchmarks, obtained using the chunky analyzer for dependences in loops (CANDL). The problem size of Class C was used for all the benchmarks except for DC, for which Class B was used as input[iii]. The classes mainly differ in the sizes of the arrays and, in turn, the number of iterations [110]. Column 8 lists the sequential run time in seconds for each benchmark.

Table 5.3 lists the average speedup achieved by various frameworks on NPB. The average speedup is calculated using the equation 4.1 (illustrated in Chapter 4). It was observed that relative to PolyBench, NPB benchmarks are less amenable to parallelization with these frameworks, and ICC produces the best overall speedup.

[iii]The problem size for DC benchmark was predefined up to Class B.

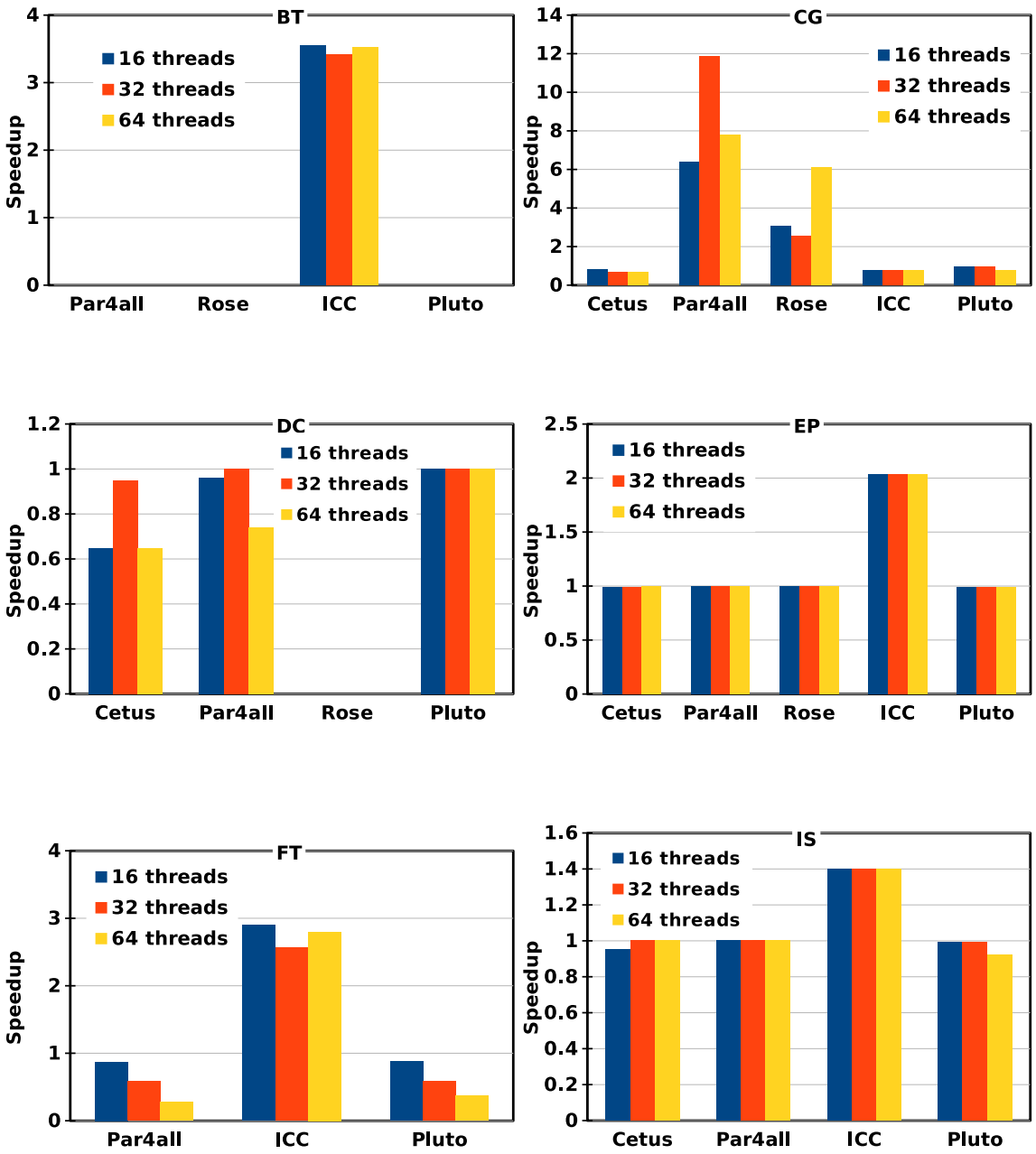
Table 5.3: Performance summary of auto-parallelizers on NPB codes (with 16, 32 and 64 threads)

No. of threads	Cetus	Par4all	Rose	ICC	Pluto
16	0.9	1.4	1.3	2.7	0.8
32	0.9	1.2	1.1	2.0	0.7
64	0.9	1.1	1.4	2.6	0.8

The execution times of the benchmarks resulting in a timeout were excluded

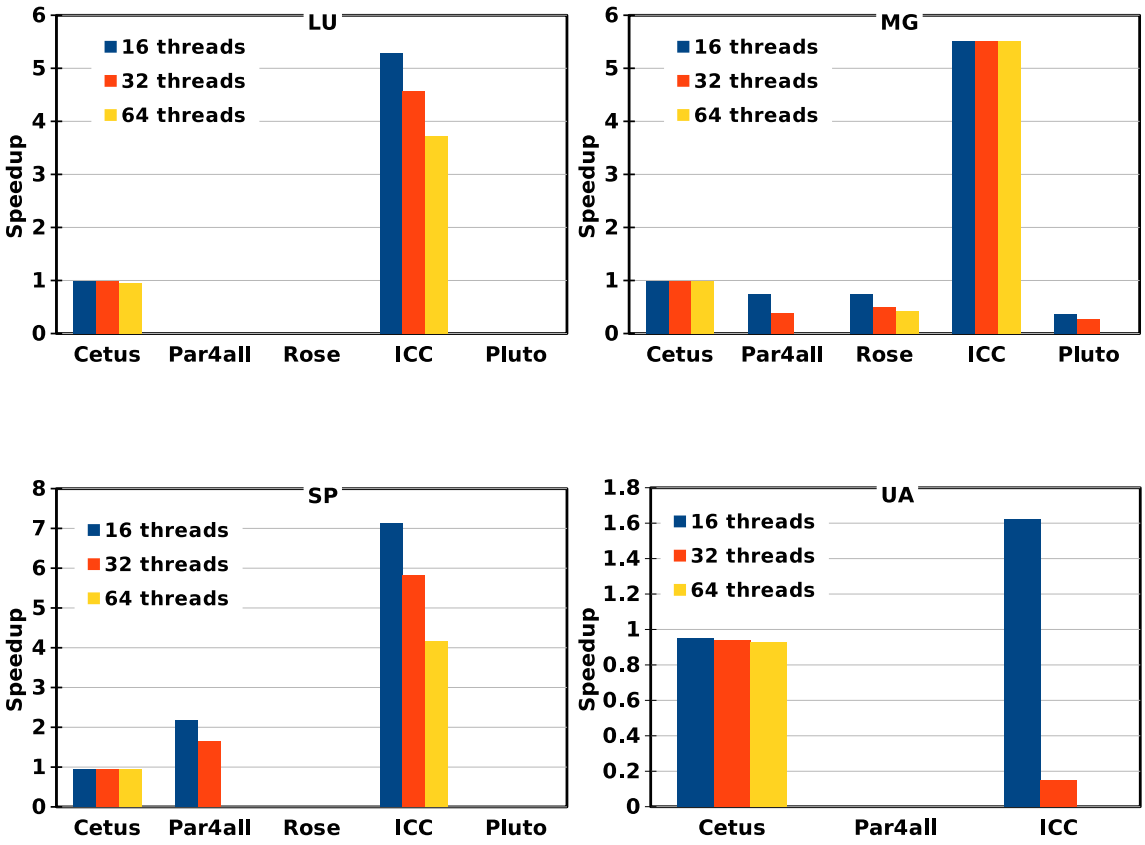
Figure 5.2 shows the speedup achieved by the frameworks on each benchmark. The results indicate that the performance delivered by the frameworks in parallelizing the NPB benchmarks was largely abysmal, except that by ICC. ICC could achieve small but better performance improvement in most of the benchmarks. To analyze the reasons behind this behavior of the frameworks, the number of loops parallelized by each framework is compared against the base version of NPB which is manually parallelized; and is called as Original. The reasons help us deduce the problems in the non-parallelized loops listed in Table 5.4.

Since the amount of parallelism achieved depends upon the number of compute-intensive portions of the program, an analysis is carried out in Table 5.5 that reveals the number of loops parallelized in the compute-intensive functions. GNU profiler (Gprof) was used to find the most time-consuming functions for each benchmark. The third column in Table 5.5 lists the percentage of total running time of the program used by the compute-intensive functions. The loop counts of the serial code parcount was calculated from Original code and the frameworks' transformed codes. Categorized parcount into four cases to understand the complexity of loop parallelization: (i) single loop (S), (ii) perfectly nested loop (N), (iii) single loop within complex nested loop (CS), and (iv) complex nested loop (CN) which includes nested loops comprising one or multiple single/nested loops as well as imperfectly nested loops. Listed these counts in Table 5.5 for Serial (non-transformed), for Original (manually parallelized NPB), and for the auto-parallelized codes output by the five



The speedup plots depict the performance of the auto-parallelizers in parallelizing the NPB benchmarks. Note different scales for y-axes

Figure 5.2: Quantitative analysis of NAS parallel benchmarks (NPB) (`BT`, `CG`, `DC`, `EP`, `FT`, `IS`), (continued in next page)



The speedup plots depict the performance of the auto-parallelizers in parallelizing the NPB benchmarks. Note different scales for y-axes

Figure 5.2: Quantitative analysis of NAS parallel benchmarks (NPB) (`LU`, `MG`, `SP`, `UA`) (Contd.)

Table 5.4: Problems identified in non-parallelized loops of NPB codes for individual parallelizers

Bm	Cetus	Par4all	Rose	ICC	Pluto
BT	Complex, FC	MS, FC, MO	FC, MO	NPC, ICW, PDEP, NOP, MI	NAL, SL, MS
CG	MO, LC, IF, EXIT	MO, LC, IF, EXIT	MO, LC, IF, EXIT	NPC, ICW, PDEP, NOP	LC, NAL, SL, IF, EXIT
DC	MO IF, EXIT, FC, while, FILE, DECL	MO, IF, EXIT, FC, while, FILE, DECL	MO, IF, EXIT, FC, while, FILE, DECL	NPC, ICW, PDEP	MO, IF, EXIT, while, FILE, DECL NAS, NAL, SL
EP	IF, FC, EXIT, ICW, LC	IF, FC, EXIT, LC	IF, FC, EXIT, LC	ICW, PDEP	IF, EXIT, SL
FT	Complex, MS, FC, LC	IF, FC, LC	ERR	NPC, ICW, PDEP	IF, SL
IS	ICW, IF, FC, LC, AU, LS, RS, switch-case	IF, FC, AU, LS, RS LC	Timeout	ICW, PDEP	LC, IF, switch-case, LS, RS, NAS, SL, SI
LU	Complex, ICW, UL	MO	MO	NPC, ICW, PDEP, NOP, MI	IF, MS, SL, MO
MG	Complex, IF, FC, UL, LC, MO	IF, FC, UL, LC, MO	IF, FC, UL, LC, MO	ICW, PDEP, NOP, MI	IF, UL, MO, SL
SP	Complex, MO, ICW, UL	MO, UL	MO, UL	NPC, ICW, PDEP, NOP, MI	MO, UL, NAL, SL
UA	Complex, ICW, IF, FC	MO, IF, FC	MO, IF, FC	NPC, ICW, PDEP, NOP, MI	IF, SL

(Small loops, Nested parallelism problem, Scalar and non-affine issues, Trivial parallelization issues, Dependence problems, Error and Timeout cases)

Complex - Complex loops, FC - Function call, MS - More number of statements, MO - Missed outer-loop parallelism, MI - Missed inner-loop parallelism, IF - `if` constructs in parallel region, NPC - Not a parallelization candidate, ICW - Insufficient computational work, PDEP - Existence of parallel dependence, NOP - No optimization reported, LC - Loop-carried dependence, NAL - Non-affine loop bound, NAS - Non-affine array subscript, SL - Scalar variable in ℓ-value of a statement, EXIT- exits and returns, DECL - Declarations within `for`-loop, `while` - while statement, FILE - File I/O operations, AU - Array updation, LS/RS - Left-/Right-shift operator, ERR - Error *during* auto-parallelization, Timeout - Timeout during auto-parallelization, UL - Upper-to-lower bound, SI - Scalar increment

frameworks. The speedups are computed relative to Serial, and expects the frameworks to approach the performance of Original (The speedup with 32 threads is listed below each benchmark in a separate row).

ICC shows the best performance improvement among all the frameworks for these benchmarks. During parallelization, ICC targets only a few loops for parallelization (see Table 5.5), yet achieves good speedup. To understand this, further analysis was carried out. Eighth column of Table 5.5, titled ICC, also shows values: P32 (speedup with 32 threads) and P1 (single-threaded speedup). The performance of P32 is same as P1 i.e. speedup does not improve with increasing threads. ICC does not exploit parallelism and hence it is similar to other frameworks. Due to the inherent optimization (ICC baseline uses optimized level -O2) apart from parallelization, nominal speedup were observed for benchmarks (with 32 threads: `BT` 3.4×, `EP` 2.0×, `FT` 2.6×, `LU` 4.6×, `MG` 5.5×, and `SP` 5.8×). Note that ICC works at the IR level, while the other frameworks are source-to-source translators.

Among the other four frameworks, Par4all shows performance improvement in two of the benchmarks (with 32 threads: `CG` 11.9× and `SP` 1.7×). It was noted that Par4all performs better for `CG` when compared to ICC. **This indicates that array privatization can lead to significant performance improvement.**

Although the value of parcount for most of the frameworks equals that of Original, the speedup is poor due to the non-trivial issues, which is discussed next in the subsections below.

5.3.1 Execution overhead problems

Three common scenarios was observed that caused overhead in execution time of the auto-parallelized codes: (i) shortcomings in the usage of implicit barriers (Figure 5.3(a)), (ii) parallelization of insignificant loops (Figure 5.3(b)), and (iii) inefficient parallelization (Figure 5.3(c)).

Table 5.5: Performance measurement based on parallelism achieved in the compute-intensive functions

Bm	Functions	Time %	Serial				Cetus				Par4all				Rose				ICC				Pluto				Original			
			SL	N	CS	CN	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p
BT	binvcrhs	20.34	No loops				No loops				No loops				No loops				No loops				No loops				No loops			
	compute_rhs	17.12	-	30	4	22	-	0	0	0	-	30	4	22	-	30	4	22	-	8	0	2	-	7	4	6	-	30	4	22
	z_solve	16.52	-	-	3	5	-	-	0	0	-	-	0	2	-	-	2	2	-	-	0	0		T			-	-	3	5
	y_solve	16.24	-	-	3	5	-	-	0	0	-	-	0	2	-	-	2	2	-	-	0	0		T			-	-	3	5
	x_solve	15.02	-	-	3	5	-	-	0	0	-	-	0	2	-	-	2	2	-	-	0	0		T			-	-	3	5
	matmul_sub	10.76	No loops				No loops				No loops				No loops				No loops				No loops				No loops			
Speedup with 32 threads							-				0.00				0.00				P32-3.41; P1-3.33				0.00				9.94			
CG	conj_grad	93.36	3	-	4	5	1	-	1	3	3	-	4	4	3	-	4	4	0	-	0	0	0	-	2	0	3	-	4	4
	sparse	6.59	4	2	-	8	1	0	-	0	2	0	-	1	2	0	-	1	0	0	-	0	2	0	-	0	1	0	-	2
Speedup with 32 threads							0.69				11.89				2.53				P32-0.78; P1-0.78				0.97				12.41			
DC	KeyComp	62.98	1	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-
	TreeInsert	29.69	No loops				No loops				No loops				No loops				No loops				No loops				No loops			
	SelectToView	5.70	1	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-
Speedup with 32 threads							0.95				1.00				-				P32-1.76; P1-1.76				1.00				14.99			
EP	vranlc	75.44	1	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-
	main	24.71	4	-	2	1	1	-	0	0	3	-	0	0	3	-	0	0	1	-	0	0	2	-	0	0	3	-	2	1
Speedup with 32 threads							0.99				1.00				1.00				P32-2.04; P1-2.03				0.99				21.29			

Continued on next page

Table 5.5 – Continued from previous page

Bm	Functions	Time %	Serial				Cetus				Par4all				Rose				ICC				Pluto				Original			
			SL	N	CS	CN	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p
FT	Swarztrauber	81.94	-	-	-	9	T				-	-	-	2	ERR				-	-	-	0	-	-	-	2	NA			
	fftXYZ	9.86	-	-	2	12	T				-	-	0	4	ERR				-	-	0	0	-	-	0	0	NA			
	evolve	4.92	-	3	-	-	-	0	-	-	-	3	-	-	ERR				-	0	-	-	-	3	-	-	NA			
Speedup with 32 threads							0.98				0.58				-				P32-2.56; P1-2.76				0.58				13.89			
IS	rank	49.65	9	-	-	-	1	-	-	-	3	-	-	-	T				0	-	-	-	1	-	-	-	NA			
	randlc	41.90	2	-	-	-	0	-	-	-	2	-	-	-	T				0	-	-	-	0	-	-	-	NA			
	fully_verify	4.92	3	-	-	-	1	-	-	-	1	-	-	-	T				1	-	-	-	1	-	-	-	NA			
Speedup with 32 threads							1.00				1.00				-				P32-1.40; P1-1.34				0.99				11.23			
LU	rhs	22.39	-	-	14	25	-	-	0	0	-	-	14	21	-	-	14	21	-	-	0	4	-	-	opt	opt	-	-	14	25
	buts	22.00	-	3	2	2	-	0	0	0	-	3	0	2	-	3	0	2	-	0	0	0	-	3	0	2	-	0	0	0
	blts	18.68	-	3	2	2	-	0	0	0	-	3	0	2	-	3	0	2	-	0	0	0	-	3	0	2	-	0	0	0
	jacld	17.57	-	2	-	-	-	0	-	-	-	2	-	-	-	2	-	-	-	1	-	-	-	0	-	-	-	0	-	-
	jacu	17.37	-	2	-	-	-	0	-	-	-	2	-	-	-	2	-	-	-	1	-	-	-	0	-	-	-	0	-	-
	ssor	1.67	2	4	9	2	0	0	0	0	2	4	9	2	ERR				0	1	0	2	4	0	6	0	2	4	9	2
Speedup with 32 threads							0.99				0.00				0.00				P32-4.58; P1-3.99				0.00				12.30			
MG	resid	50.34	-	-	2	2	-	-	0	0	-	-	2	0	-	-	2	0	-	-	0	0	-	-	2	0	-	-	2	2
	psinv	28.58	-	-	2	2	-	-	0	0	-	-	2	0	-	-	2	0	-	-	0	0	-	-	2	0	-	-	2	2
	interp	7.73	-	-	13	8	-	-	0	0	-	-	13	6	-	-	12	7	-	-	0	0	-	-	opt	opt	-	-	13	8
	rprj3	5.96	-	-	2	2	-	-	0	0	-	-	2	0	-	-	1	0	-	-	0	0	-	-	0	0	-	-	2	2
	vranlc	3.73	1	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-
	norm2u3	1.67	-	3	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-	1	-	-	-	0	-	-	-	3	0	0
Speedup with 32 threads							0.99				0.39				0.50				P32-5.52; P1-5.05				0.28				6.38			

Continued on next page

Table 5.5 – Continued from previous page

Bm	Functions	Time %	Serial				Cetus				Par4all				Rose				ICC				Pluto				Original			
			SL	N	CS	CN	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p	S_p	N_p	CS_p	CN_p
SP	compute_rhs	37.39	-	30	4	22	-	0	0	0	-	30	4	22	-	27	4	22	-	7	0	3	-	21	3	14	-	30	4	22
	z_solve	18.13	-	-	11	16	-	-	0	1	-	-	11	11	-	-	11	11	-	-	0	3	-	-	7	8	-	-	11	16
	y_solve	18.03	-	-	11	16	-	-	0	1	-	-	11	11	-	-	11	11	-	-	0	3	-	-	6	2	-	-	11	16
	x_solve	17.44	-	-	11	16	-	-	0	1	-	-	11	11	-	-	11	11	-	-	0	4	-	-	6	2	-	-	11	16
Speedup with 32 threads							0.93				1.65				0.00				P32-5.83; P1-5.03				0.00				9.31			
UA	laplacian	51.30	-	15	-	-	-	0	-	-	-	14	-	-	-	14	-	-	-	5	-	-	-	15	-	-	-	0	-	-
	convect	14.05	2	-	16	16	0	-	0	0	0	-	0	0	2	-	16	0	0	-	0	2	2	-	0	3	1	-	16	16
	diffusion	9.93	1	4	3	11	0	0	0	0	0	0	0	0	1	4	3	10	0	2	0	1	2	4	0	0	1	4	3	10
	transf	6.83	-	-	4	29	-	-	0	0	-	-	0	0	-	-	0	5	-	-	0	1	ERR				-	-	4	29
	transfb	6.28	-	-	4	29	-	-	0	0	-	-	0	0	-	-	1	10	-	-	0	0	ERR				-	-	4	29
Speedup with 32 threads							0.94				0.00				-				P32-0.15; P1-4.96				-				11.3			

Bm Benchmark, Function Name of the compute-intensive function, Time % Percentage of the total running time of the program used by the function, Original Original parallel version of NPB code, S Single loop count, N Perfect Nested loop count, CS Single loop in a Complex Nested loop count, CN Complex Nested loop count, ERR Error *during* auto-parallelization, NA Not Applicable since the original code is different from serial, T Timeout during auto-parallelization, opt more optimized code P32 Speedup using `-parallel` with 32 threads, P1 Speedup using `-parallel` with 1 thread)

Non-parallelized code

Parallelized code is not equivalent to the Serial code

Not Applicable due to Error and Timeout cases

Shortcomings in the usage of implicit barriers

In Table 5.5, parcount values of Par4all and Rose are approximately equal to that of Original for BT, LU and SP. This indicates that the two frameworks were able to identify almost the same number of loops as those that are manually parallelized. Also, Pluto's parcount is minimum when compared to Par4all and Rose but is non-zero. However, Original (BT 9.9×, LU 12.3×, SP 9.3×) outperforms other frameworks. The auto-parallelizers caused timeout execution for BT, LU, and SP. These benchmarks contain multiple, independent, consecutive loops within the parallel region. To these loops, Original uses the synchronization qualifier nowait at the appropriate places, thereby removing barrier synchronization and improving the load balancing. Auto-parallelizers lack such a sophistication which requires high-level understanding about the use of implicit barriers or needs to identify dependences across codes beyond the loop body. A code fragment that depicts the usage of **nowait** clause is shown in Figure 5.3(a).

Parallelization of insignificant loops:

Parallelization of small loops may lead to performance drop or execution overhead. One such scenario is depicted in Figure 5.3(b). For EP, Par4all (1.00×), Rose (1.00×) and Pluto (0.99×) insert parallel pragma to shorter loops, which results in performance overhead (for instance, the result of Pluto that causes overhead is shown). Cetus applies conditional parallelism (model-based profitability test as mentioned earlier in previous Section 2.3.1 in Chapter 2) which eliminates the parallel execution of shorter loops. However, the speedup of Cetus (0.99×) proves inferior compared to all the other frameworks. It is obvious from the study that its parcount is minimum across all the benchmarks, reducing opportunities for parallelism. ICC inherently avoids parallelization of computationally less intensive loops and reports insufficient computational work.

Inefficient parallelization:

A significant improvement is been observed from Par4all in the parallelization of CG. Although the numbers of loops parallelized by Par4all and Rose are comparable to Original

(Table 5.5), Par4all speedup (11.89×) is significantly better than Rose (2.53×). Rose inserts `parallel pragma` naïvely for all the loops of the nested-for which eventually leads to execution overhead as observed in Figure 5.3(c). This example outlines that since the tool targets the loops in the compute-intensive function (*conj_grad()* taking 93.4% of the execution time) for parallelization, a good speedup is achieved by Par4all. Pluto does several transformations for `LU` and `MG` which lead to timeout and poor speedup.

5.3.2 Nested parallelism problems

Referring at the counts of Serial in Table 5.5, there are several perfectly nested loops (N), as well as several imperfectly or Complex nested loops (CN). With the exception of Cetus, all the frameworks support parallelization of both N and CN types. However, the way in which each framework applies the parallelism is distinct. Cetus does not support nested parallelism in many instances due to its complexity (Section 2.3.1 in Chapter 2).

In case of the nested parallelism perfectly nested loops (N), and imperfectly or Complex nested loops (CN), it is noticed that ICC parallelizes only the outer loop, and Pluto parallelizes only the inner loop. Par4all and Rose miss outer loop parallelism while handling CN. Figure 5.3(d) illustrates how Original handles the CN in *resid()* of `MG` when compared to Rose. Performance of Original (6.38×) is better than that achieved by Rose (0.50×).

5.3.3 Scalar and non-affine issues in Pluto

As previously illustrated in Chapter 4, Pluto does not parallelize the code when the ℓ-value of the statement is a scalar variable. Most of the compute-intensive loops in the NPB encompass scalar reduction, which is not handled by Pluto. Also non-affine expressions are not parallelized by this polyhedral framework. In NPB, there occur more challenges such as non-affine loop bounds, non-affine array subscripts, and scalar variables as shown in

Table 5.4. In particular, Pluto could not parallelize the compute intensive function *conj_grad()* which takes 93.4% of `CG`'s execution time. This is due to the presence of scalar variable *sum*, and non-affine loop bound *k = rowstr[j]; k<=rowstr[j+1]-1* in the inner loop, as shown in Figure 5.3(c).

(a) Parallelization of `BT`: code snippet from compute_rhs(). Original uses nowait clause, which improves concurrent processing

```
#pragma omp parallel for private(j,i,rho_inv)
for(k=0;k<=grid_points[2]-1;k+=1)
for(j=0;j<=grid_points[1]-1;j+=1)
for(i=0;i<=grid_points[0]-1;i+=1)
{
rho_inv = 1.0/u[k][j][i][0];
.....
}
#pragma omp parallel for private(j,i,m)
for(k=0;k<=grid_points[2]-1;k+=1)
for(j=0;j<=grid_points[1]-1;j+=1)
for(i=0;i<=grid_points[0]-1;i+=1)
for(m=0; m<=4; m+=1)
rhs[k][j][i][m] = forcing[k][j][i][m];
#pragma omp parallel for private(j,i,uijk)
for(k=1;k<=grid_points[2]-2;k+=1)
{
for(j=1;j<=grid_points[1]-2;j+=1)
for(i=1;i<=grid_points[0]-2;i+=1)
{
uijk = us[k][j][i];
}
.....
}
```

Par4all

```
#pragma omp parallel default(shared) private(i,j,k,m,rho_inv,uijk)
{
#pragma omp for schedule(static) nowait
for(k=0;k<=grid_points[2]-1;k++)
for(j=0;j<=grid_points[1]-1;j++)
for(i=0;i<=grid_points[0]-1;i++){
  rho_inv = 1.0/u[k][j][i][0];
  .....
}
#pragma omp for schedule(static)
for (k=0;k<=grid_points[2]-1;k++)
for (j=0;j<=grid_points[1]-1;j++)
for (i=0;i<=grid_points[0]-1;i++)
for (m=0; m<5; m++)
rhs[k][j][i][m] = forcing[k][j][i][m];
#pragma omp for schedule(static) nowait
for(k=1;k<=grid_points[2]-2;k++){
for(j=1;j<=grid_points[1]-2;j++)
for(i=1;i<=grid_points[0]-2;i++){
uijk = us[k][j][i];
}
.....
}
}
```

Original

(b) Parallelization of `EP`: code snippet from main() - Parallelization of insignificant loop by Pluto

```
%Disabled due to low profitability:#pragma omp parallel for
for (i=0; i<10; i ++ )
{
 q[i]=0.0;
}
```

Cetus

```
if (NQ >= 1) {
  lbp=0; ubp=NQ-1;
#pragma omp parallel for private(lbv,ubv,t3)
for(t2=lbp;t2<=ubp;t2++){
   q[t2] = 0.0;;
  }
```

Pluto

```
for (i=0; i<NQ; i++)
{
q[i] = 0.0;
}
```

Original

Figure 5.3: NPB result analysis (a) Parallelization of `BT`: code snippet from compute_rhs(). Original uses nowait clause, which improves concurrent processing (b) Parallelization of `EP`: code snippet from main() - Parallelization of insignificant loop by Pluto, (continued in next page)

(c) Parallelization of CG: code snippet from conj_grad(). Rose naïvely targets all the loops, while Par4all judiciously chooses loops for parallelization

Par4all

```
.....
#pragma omp parallel
    for private(sum, k
    )
for(j = 0; j <=
    lastrow-firstrow+1
    -1; j += 1) {
sum = 0.0;
for(k = rowstr[j]; k <
    = rowstr[j+1]-1; k
     += 1)
sum = sum+a[k]*p[
    colidx[k]];
q[j] = sum;
}
.....
```

Rose

```
.....
#pragma omp parallel
    for private (sum,j
    ,k)
for(j=0;j<=lastrow -
    firstrow+1-1;j+=1)
     {
sum = 0.0;
#pragma omp parallel
    for private(k)
    reduction(+:sum)
for (k=rowstr[j];k<=
    rowstr[j+1]-1;k+=
    1) {
sum = sum+a[k]*p[
    colidx[k]];
}
q[j] = sum;
}
.....
```

Original

```
#pragma omp parallel
    default(shared)
    private(j,k,suml)
{
.....
#pragma omp for
for(j=0;j<lastrow-
    firstrow+1;j++) {
suml = 0.0;
for (k=rowstr[j]; k<
    rowstr[j+1]; k++)
    {
suml=suml+a[k]*p[
    colidx[k]];
}
q[j] = suml;
}
.....
}
```

(d) Parallelization of MG: code snippet from resid() - parallelization of complex nested (CN) loop (missed outer-loop parallelism)

Rose

```
for (i3=1; i3<=n3-1-1; i3+=1) {
for (i2=1; i2<=n2-1-1; i2+=1) {
#pragma omp parallel for private
    (i1)
for (i1=0; i1<=n1-1; i1+=1) {
u1[i1] = u[i3][i2 - 1][i1] +
    .....;
u2[i1] = u[i3 - 1][i2 - 1][i1] +
    .....;
}
#pragma omp parallel for private
    (i1)
for (i1=1; i1<=n1-1-1; i1+=1) {
r[i3][i2][i1] = v[i3][i2][i1] -
    .....);
}}}
```

Original

```
#pragma omp parallel for default(
    shared) private(i1,i2,i3,u1,u2
    )
for (i3 = 1; i3 < n3-1; i3++) {
for (i2 = 1; i2 < n2-1; i2++) {
for (i1 = 0; i1 < n1; i1++) {
u1[i1] = u[i3][i2-1][i1] + .....;
u2[i1] = u[i3-1][i2-1][i1] +
    .....;
}
for (i1 = 1; i1 < n1-1; i1++) {
r[i3][i2][i1] = v[i3][i2][i1] -
    .....;
}}}
```

Figure 5.3: NPB result analysis (c) Parallelization of CG: code snippet from conj_grad(). Rose naïvely targets all the loops, while Par4all judiciously chooses loops for parallelization (d) Parallelization of MG: code snippet from resid() - Parallelization of CN loop (missed outer-loop parallelism), (continued in next page)

(e) Parallelization of `EP`: code snippet from main() - Par4all prohibit parallelization due to non-supported programming construct in complex nested (CN) loop

Par4all

```
for(k = 1; k <= np; k += 1)
{
 kk = k_offset+k;
 .....
 if (!(i<=100))
 goto _break_5;
 ik = kk/2;
 if (2*ik!=kk)
 t3 = randlc(&t1, t2);
 if (ik==0)
 goto _break_5;
 .....
 for(i = 0; i <= (1<<16)-1; i += 1)
 {
 .....
 sx = sx+t3;
 sy = sy+t4;
 .....
 }
}
```

Original

```
#pragma omp parallel default(shared) private(k,kk,t1,t2,t3,i,ik)
{
 #pragma omp for reduction(+:sx,sy) nowait
 for (k=1; k<=np; k++)
 {
  kk = k_offset + k;
  .....
  for (i=1; i<=100; i++)
  {
   ik = kk / 2;
   if ((2 * ik) != kk)
    t3 = randlc(&t1, t2);
   if (ik == 0)
   break;
   .....
  }
  .....
  for (i=0; i<NK; i++)
  {
  .....
  sx = sx + t3;
  sy = sy + t4;
  .....
  }
 }
}
```

Figure 5.3: NPB result analysis (e) Parallelization of `EP`: code snippet from main() - Parallelization of non-supported programming construct in complex nested (CN) loop (Contd.)

Another significant problem that prevents frameworks from parallelization is *loop-carried* dependence (LC) which is studied in-detail in Sections 3.2 in Chapter 3 and Section 4.3 in Chapter 4. Table 5.4 shows that NPB benchmarks comprise of *loop-carried* dependence (LC) problems. This could be handled only by Pluto by applying loop peeling and loop distribution optimization. But due to some of the non-affine issues, scalar problems, and other non-trivial issues, Pluto perform inferior to Original.

Apart from the non-trivial parallelization problems above, there are several other issues that prevent the frameworks from successful and correct transformation which are discussed below.

5.3.4 Other parallelization issues

Table 5.4 depicts the problems that terminate the frameworks in parallelizing some of the loops. Most common problems prevailing in the frameworks are if construct (IF), exit and return statements (EXIT) that hinder the tool to perform loop parallelization. Figure 5.3(e) illustrates how the non-supported programming constructs if and break in `EP` affect the parallelization by Par4all ($1.00\times$). Automatic parallelization of irregular scientific application have been carried out in the earlier works [111]. Original ($21.29\times$) performs better than all the other frameworks. It applies the implicit barrier **nowait** and scalar reduction besides parallelization. Cetus, Par4all, and Rose do not support inter-procedural transformation via a function call (FC). Though ICC discloses details about its unsuccessful behavior, these are generic reasons. For further investigation, the auto-parallelized loops are examined and it was found that ICC could support IF construct, and function call (FC). Furthermore, usage of few of the programming constructs (highlighted in Table 5.4 as trivial parallelization issues) inhibit the parallelization of several compute-intensive parts of the code segment that, in turn, deteriorates performance. These challenges hint at support for more high-level constructs, sophisticated analyses, and user-driven parallelization. Earlier work by Chatarasi et al. [80] addressed the problem of explicit parallelism and unanalyzable data accesses that prohibit parallelism.

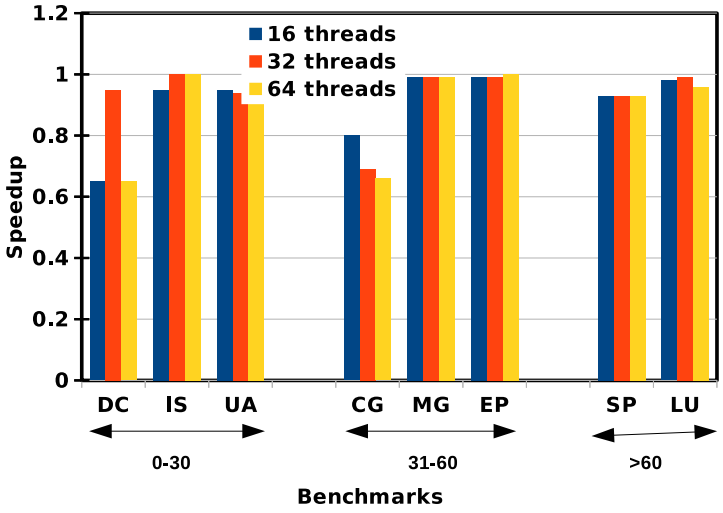
5.4 Effect of static dependences

In Table 5.1, the third column (named **Dep**) shows the number of static dependences in each NPB benchmark. Since dependences affect parallel performance, the analyses is carried out for each framework. In particular, bucketized the range of **Dep** values: 0-30, 31-60, and >60, and observe the speedup of benchmarks within each range. As described in the PolyBench study, the study expect the speedup to be smaller when the dependence is more.

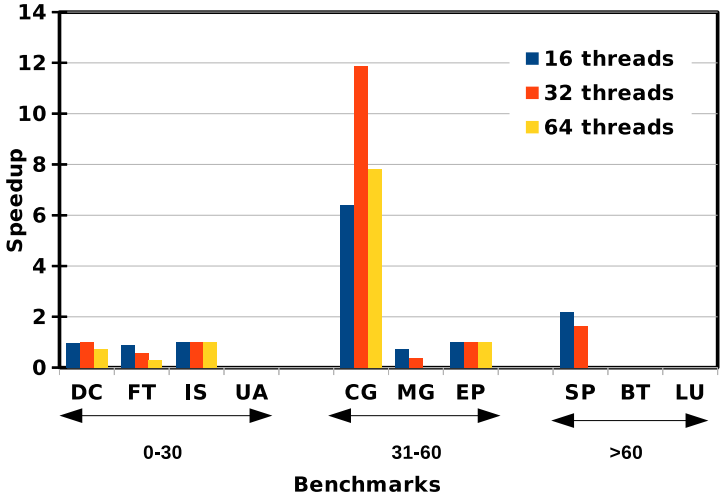
Figure 5.4 summarizes the impact of **Dep** range affecting the speedup for all the five frameworks. The frameworks show a varied behavior for the same set of benchmarks. Cetus performs almost the same across various **Dep** values, and in several cases, results in increased time compared to the single-threaded version (speedup less than unity). Par4all has a similar behavior except for `CG` where the speedup is the largest of all the frameworks. This occurs as most of the loops are dependence-free and Par4all parallelizes the maximum number of loops in `CG` compared to other frameworks.

Rose could only parallelize half of the benchmarks, so a conclusive remark could not be made. However, it is observed that its parallelization improvement of NPB is limited to `CG` alone. Similar to other frameworks, Pluto also is not effective on NPB and results in performance degradation on various benchmarks. On the other hand, ICC shows consistent improvement on most of the benchmarks. Surprisingly, the only benchmark for which ICC results in less-than-one speedup is `CG`. This occurs due to ICC's cost model which marks most of the loops to be *not parallelization candidates* due to insufficient computational work predicted. But more surprisingly, the speedup ICC achieves is more for larger values of **Dep**. This is a clear indicator that in case of NPB, the number of static dependences or the number of RAW dependences cannot faithfully capture the parallelization effect, and other parameters (such as the dynamic dependences, opportunities for loop transformations, etc.) dominate performance.

(a) Cetus



(b) Par4all



(c) Rose

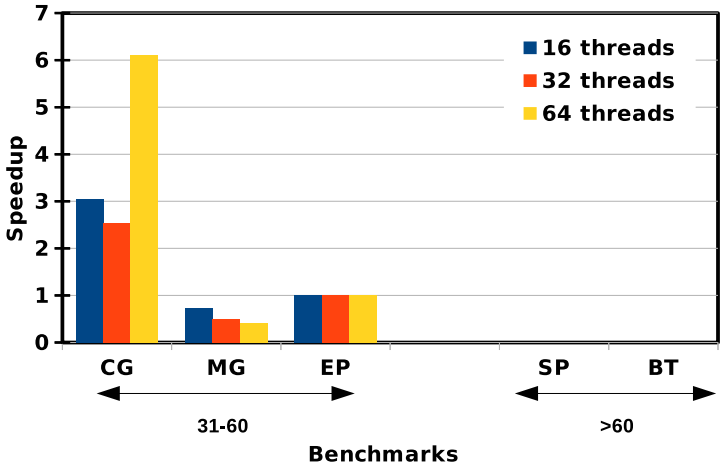
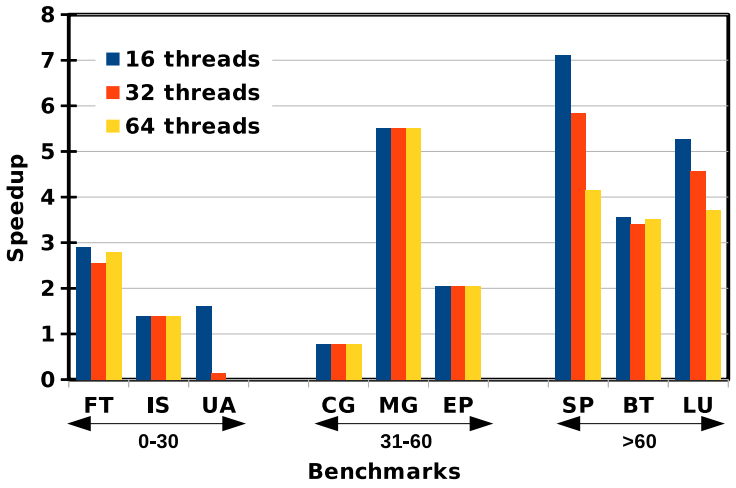


Figure 5.4: Effect of static dependences on NPB Benchmarks: A speedup analysis (a) Cetus (b) Par4all (c) Rose, (continued in next page)

(d) ICC



(e) Pluto

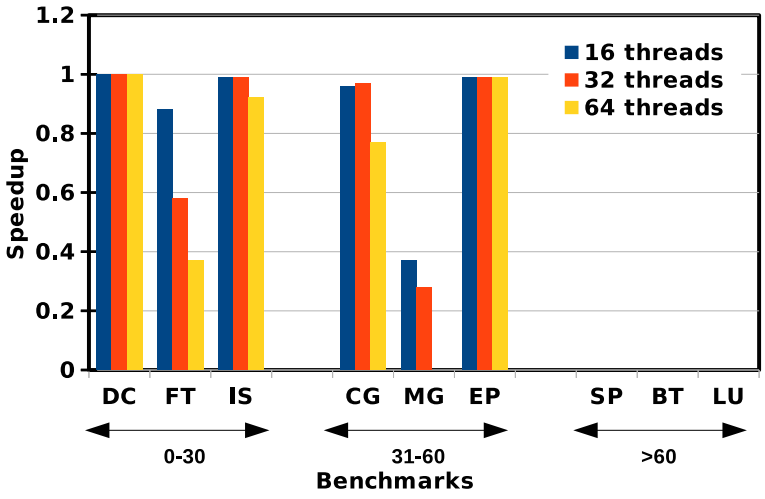


Figure 5.4: Effect of static dependences on NPB benchmarks: A speedup analysis (d) ICC (e) Pluto (Contd.)

5.5 Summary

The performance behavior of NPB codes parallelized by different frameworks was studied. For affine codes such as PolyBench suite, the parallelizers mainly, ICC and Pluto exhibited better speedup. However, in the case of NPB suite which has irregular and non-affine patterns with complex coding structures, all the frameworks face issues during parallelization. Some of the significant issues are listed below:

1. Excluding ICC and Cetus, all the other frameworks produce erroneous code transformation that leads to incorrect output or syntactical error.

2. ICC does not exploit parallelism for NAS parallel benchmarks (NPB) codes, and the performance improvement was observed due to its inherent baseline optimization (-O2).

3. Rose is the only framework that causes an error *during* transformation.

4. Par4all, Rose, and Pluto perform inefficient parallelization which causes overhead in the execution time.

5. Imperfectly or complex nested loops are not handled well by the frameworks.

6. Scalar and non-affine issues lead to ineffective parallelization by Pluto.

The above-mentioned issues are quite serious and forbid the widespread acceptance of tools. Several of the compute-intensive functions in the NPB are dependence-free, yet the complex code structure prohibits the frameworks from achieving parallelization. Although Pluto and ICC perform optimization besides parallelization for regular codes with an analyzable pattern, these tools do not fetch benefit and become ineffective for the codes comprising irregular coding style. In the forthcoming chapters (Chapter 6 and 7), two approaches have been proposed to improve parallelization by overcoming several of these irregular coding styles in Pluto, an open-source tool.

CHAPTER 6

Elimination of Auto-parallelization Issues in Irregular and General-purpose Programs in Pluto

Parallelizer based on the polyhedral model namely, Pluto, is found to be efficient in performing loop optimizations and parallelizing loop with dependences. This chapter explores the non-affine issues and general limitations in irregular and general-purpose programs that forbid parallelization using Pluto. It presents analyses of three distinct real-world problems namely, Green-Marl, Rodinia, and NAS parallel benchmarks (NPB) and their issues. This chapter introduces a method to eliminate the auto-parallelization issues and makes the code amenable to parallelization.

6.1 Background

In this section, a brief introduction on polyhedral model is illustrated.

6.1.1 Polyhedral model

Polyhedral model is a framework used for performing loop transformations. It is based on three main concepts, viz. iteration domain, scattering function, and access function [83]. A program part that can be represented using the polyhedral model is called a static control part (SCoP) [112, 113].

Iteration domain

The key aspect of the polyhedral model is to consider statement instances. The outer loop counters, i.e., *iterators* are the instance of a statement which is enclosed inside a loop. The ordered list of iterators (ordered from the outermost iterator to the innermost iterator) is called the *iteration vector*. For instance, in Figure 6.1(a) the *iteration vector* for $S3$ is (i, j).

(a) An example illustration 1

```
1 for (i = 2; i <= N; i++)
2 for (j = 2; j <= N; j++)
3 S3: A[i] = pi;
```

(b) 2D Iteration Space

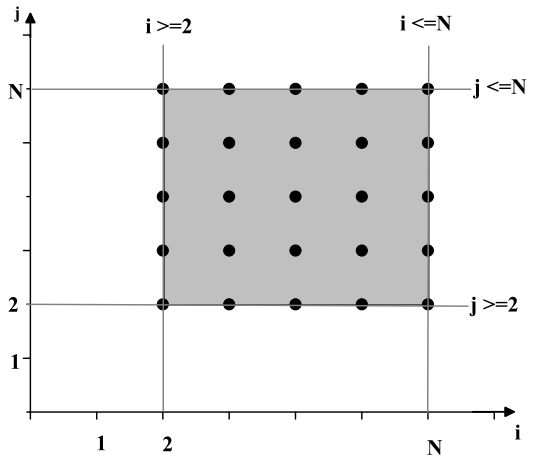


Figure 6.1: Iteration domain: (a) An example illustration-1 (b) 2D Iteration space

It is difficult to know at compile time how many time $S3$ will be executed. Such a loop is said to be *parametric* and the parameter here is N.

Iteration domain is the set of all possible values of *iteration vector* which is a compact way to represent all the instances of a given statement. Equation 6.1 shows the set of *iteration vector* (i, j). Figure 6.1(b) shows the iteration domain is a part of 2-dimension space Z^2 which is specified to a set of constraints. If the constraints are affine, the set of constraints then defines a polyhedral model.

$$D_{S3} = \{(i,j) \in Z^2 | 2 \leq i \leq N \wedge 2 \leq j \leq N\} \tag{6.1}$$

To analyze the loops are affine, a matrix representation is used. For instance, for the statement $S3$ in Figure 6.1(a), the set of constraints is below:

$$\begin{cases} i - 2 \geq 0 \\ -i{+}N \geq 0 \\ j - 2 \geq 0 \\ -j{+}N \geq 0 \end{cases}$$

```
1 for (i = 2; i <= 4; i++)
2 for (j = 2; j <= 4; j++)
3 S4: P[i+j] += A[i] + B[j];
```

Figure 6.2: Scattering function: An example illustration-2

Lastly, the constraint system is translated to the form (*domain matrix * iteration vector* $\geq$ *0*) and the matrix representation is:

$$\begin{bmatrix} 1 & 0 & 0 & -2 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -2 \\ 0 & -1 & 1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Scattering function

Iteration domain does not provide ordering information. Such information in the polyhedral model is called *scattering*. There exist many kinds of ordering, such as allocation, scheduling, chunking, etc. All these types are expressed using logical stamps. However, they differ in semantics.

Scattering functions are affine functions of the outer loop counter and the global parameters. The iteration domain for the statement $S4$ (Figure 6.2) is shown in equation 6.2

$$D_{S4} = \{(i,j) \in Z^2 | 2 \leq i \leq 4 \wedge 2 \leq j \leq 4\} \tag{6.2}$$

In order to schedule the instances of the statement S_4, the following scheduling function (Equation 6.3) is used to find the logical dates of each instance:

$$\Theta_{S4}(i,j) = (j+2, 3*i+j) \tag{6.3}$$

It is to be noted that in the polyhedral model, users do not have to take care of the generation of target code. The chunky loop generator (CLooG) tool generates the target code using scattering function.

The matrix form of scattering function of any statement S is of the form:

$$\Theta_S\big(iterationvector\big) = scatteringmatrix * iterationvector \tag{6.4}$$

For instance, for equation 6.3 the matrix form is:

$$\Theta_{S4} = \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} = \begin{bmatrix} 0 & 0 & 2 \\ 3 & 1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \\ 1 \end{pmatrix}$$

Access functions

It is necessary to analyze the correctness of the transformation such that the semantics of the original code is maintained. This is achieved in the polyhedral model when all the memory accesses are made through arrays.

For instance, there are three dimensions in the array access A[2*i+j][j][i+N]. Each subscript dimension is an affine form of some outer loop iterators (i and j) and global parameter (N). Hence it corresponds to an acceptable array access to be analyzed in the polyhedral model. Access functions are used depending on the *iteration vector* to describe an array access. The access function for the statement A[2*i+j][j][i+N] is:
F_A(i, j) = (2 * i + j, j, i + N).

The matrix representation is of the form F_A*(iteration vector) = access matrix * iteration vector* which is shown below

$$\mathrm{F}_A \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} = \begin{bmatrix} 2 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix}$$

6.2 Limitations of polyhedral model

In general, the polyhedral model is powerful and helps in performing various optimizations and transformations for improved performance. However, the tools developed using the polyhedral model expects the loop to meet certain requirements for parallelization [32]. Consequently, if it fails to meet any one of the conditions, then the parallelizing tool stops by throwing an error or warning. Polyhedral frameworks transform selected regions in the input program into static control part (SCoP) and capture precise dependence information among statement instances in the form of dependence polyhedron over the iterators and global parameters [56]. If there are any unanalyzable or non-affine constructs in the SCoP, then there may be hurdles in constructing the dependence polyhedron.

The positive report is that decades of research has led to a great expansion of programs that can be considered analyzable by polyhedral frameworks, thereby reducing the impact of these limitations [32, 114]. The main remaining constraints include restrictions on pointer usage, recursion, and unstructured control flow [115]. One of the primary downsides of the current polyhedral frameworks is that there are still many unanalyzed reasons leading to SCoP rejection.

Based on the following study on three real-world benchmark suites, namely GreenMarl, Rodinia and NPB using Pluto [20, 21], we have identified 9 non-affine patterns within the SCoP that may be considered unanalyzable by some polyhedral frameworks:

1. Scalar variables in ℓ-value of a statement (SL)
2. Constructor calls (CC)
3. Function calls (FC)
4. Declaration statements (DECL)
5. Non-affine loop bound (NAL)
6. Non-affine IF construct (NIF)
7. Scope resolution operator (SR)
8. Non-affine array subscripts (NAS)
9. FILE I/O operations (FILE)

```
1  for(col=0; col<public_cols; col++)
2  {
3   for(row=0; row<public_rows; row++)
4   {
5   ori_row = row + in2_rowlow − 1;                          // SL
6   diff_prv = diff_prv +  std::abs((val − G_pg_rank[row]));  // SR
7   private.d_in2[col*public_rows+row] = temp;                // NAS
8   }
9  }
10
11 for (node_t n = 0; n<G.num_nodes(); n ++)                  // NAL
12 {
13 int32_t __S1 = 0 ;                                        // DECL
14 __S1 = 0 ;
15  gm_common_neighbor_iter n_I(__S1);                       // CC/FC
16   if (G_age[n] > K)                                       // NIF
17  {
18  __S2_prv = __S2_prv + G_teen_cnt[n] ;
19  }
20 }
21
22 for(i=0; i < edge_list_size ; i++)
23  fscanf(...);                                             // FILE
```

Highlighted ▢ are the non-affine constructs (NAC). Acronyms of individual problems are described earlier in Section 6.2

Figure 6.3: An example illustrating all nine non-affine constructs (NAC) faced by Pluto (code snippets from Rodinia suite)

Figure 6.3 depicts an example that discuss all the nine auto-parallelization problem. Note: The abbreviations defined above are used in this figure to depict the non-affine patterns.

6.3 Summary of the proposed approach

Potentially parallelizable code with presence of any one of the aforementioned nine non-affine construct (NAC) prohibits parallelizaton by Pluto. To overcome the issue, in this chapter, an elimination based approach is proposed to make the non-affine code amenable to Pluto parallization. The contributions to the proposed approach are as follows:

- Identified the occurrence of the nine non-affine constructs (NAC) in three set of benchmarks suite namely, Green-Marl, Rodinia, and NAS parallel benchmarks (NPB) (refer Table 6.1, 6.2, and 6.3). The summary of the NAC in individual benchmark

suite that prohibit pluto parallelization is shown in Table 6.4.

- Proposed a technique to eliminate the identified NAC from benchmarks and make them analyzable by Pluto (Section 6.5). The elimination part is decomposed into three categories viz. Pre-elimination, In-elimination, and Post-elimination (refer Figure 6.4). The high-level idea is to remove the minimum amount of NAC from the original sequential source such that the parallelizer is able to transform the modified program (Pre- and In-elimiantion). Obviously, after removing the NACs, the code is functionally not equivalent to the original. Therefore, we re-add the removed NAC code judiciously to the parallelizer parallelized code (Post-elimination), to obtain semantically equivalent parallel code. This set of transformations were done carefully to not compromise correctness. To the best of our knowledge, no other framework improves the effectiveness of parallelizing compilers using such a method.

- Finally, after ensuring the correctness, the Pluto parallelized codes are evaluated for performance (Section 6.6).

6.4 Analysis on general-purpose and irregular programs

The general limitations and the non-affine issues discussed above were identified by analyzing three general-purpose and irregular set of benchmarks namely, Green-Marl, Rodinia, and NAS parallel benchmarks (NPB). This section explains the study of these benchmarks. **Throughout this chapter non-affine constructs (NAC) is referred as the combination of general limitations and non-affine issues.**

6.4.1 Green-Marl analysis

Green-Marl is a domain-specific language that is specially designed for graph data analysis [116]. The compiler `gm_comp` produces an equivalent, efficient and parallelized C++ function for Green-Marl procedure. This package is comprised of classical graph algorithms implemented in Green-Marl `(*.gm)` as well the generated C++ implementation

(`*.cc`). There are totally 21 different C++ graph algorithms, out of which we use 14 benchmarks for our analysis (Table 6.1 lists the benchmarks and their non-affine issues).

The reason for eliminating the remaining seven benchmarks from the analysis are: (i) The benchmark `random_walk_sampling_with_random_jump` has no loop to parallelize (ii) (`bc` and `bc_random`) show segmentation fault during serial execution and the source of the error could not be deduced (iii) The usage of input argument vector for `bc_adj, birdir_dijkstra, sssp_dijkstra`, and `sssp_path_adj` is missing during execution and hence the output is not reproduced.

The remaining 14 benchmarks were subjected on Pluto for parallelization. **Surprisingly, none of these benchmarks were parallelized by Pluto as the benchmarks have too many NAC.** In total, six non-affine constructs (NAC) were identified in the Green-Marl benchmark that forbid the auto-parallelization using Pluto. Table 6.1 shows the existence (✓) and non-existence (✗) of these issues in each benchmark.

6.4.2 Rodinia analysis

Rodinia benchmark suite [117] [118] is a collection of parallel applications and kernels with unique characteristics which target heterogeneous computing, viz. multi-core CPUs and GPUs. It cover a wide range of application domains, parallel communication patterns, synchronization techniques and power consumption. Rodinia have been implemented for GPUs, multi-core CPUs, and accelerators using CUDA, OpenMP, and OpenCL respectively. The benchmarks exhibit various types of parallelism, data access patterns, and data sharing characteristics.

There are in total 19 different real-world OpenMP implemented applications that cover data mining, physics simulation, pattern recognition, graph algorithms, image processing, etc. We used all the 19 OpenMP implementations for our analysis and were subjected to parallelization using Pluto. After the investigation, in total, seven frequently occurring NAC were identified within the SCoP that prohibit Pluto parallelization and are listed in Table 6.2.

Table 6.1: Non-affine constructs (NAC) that prohibit Pluto parallelization in Green-Marl

Bm.	Non-affine Constructs					
	SL	**CC**	**DECL**	**NAL**	**NIF**	**SR**
adamicAdar	✓	✓	✓	✓	✗	✗
avg_teen_cnt	✗	✗	✓	✓	✓	✗
communities	✗	✗	✓	✓	✓	✗
hop_dist	✓	✗	✓	✓	✓	✗
kosaraju	✗	✗	✓	✓	✗	✗
pagerank	✓	✗	✓	✓	✗	✓
potential_friends	✗	✗	✗	✓	✓	✗
random_degree_node_sampling	✓	✗	✓	✓	✓	✗
random_node_sampling	✓	✗	✓	✓	✓	✗
sssp	✓	✗	✓	✓	✓	✗
sssp_path	✓	✗	✓	✓	✓	✗
triangle_counting	✗	✗	✗	✓	✓	✗
v_cover	✗	✗	✓	✓	✓	✗
conduct	✗	✗	✓	✓	✓	✗
14	7	1	12	14	11	1

Bm. Benchmark, ✗ Absence of NAC, ✓ Existence of NAC, SL = Scalar Variable in the ℓ-value of a statement, CC = Constructor Call, DECL = Declaration Statements, NAL = Non-affine Loop bound, NIF = Non-affine IF construct, SR = Scope Resolution Operator

Table 6.2: Non-affine constructs (NAC) that prohibit Pluto parallelization in Rodinia

Bm.	Non-affine constructs identified using Pluto					
	SL	DECL	NAL	NIF	NAS	FILE
backprop	✓	✗	✗	✗	✗	✗
b+tree	✓	✓	✓	✓	✓	✗
bfs	✗	✗	✓	✓	✗	✓
cfd	✓	✓	✗	✓	✓	✓
hotspot	✓	✓	✓	✓	✓	✓
hotspot3D	✓	✗	✗	✓	✓	✓
heartwall	✓	✓	✓	✗	✓	✗
lavaMD	✓	✗	✓	✓	✓	✓
lud	✓	✓	✓	✓	✓	✓
kmeans	✓	✗	✗	✓	✗	✗
nn	✓	✓	✗	✓	✗	✓
nw	✗	✓	✓	✓	✓	✗
srad	✓	✗	✗	✓	✓	✗
mummergpu	✗	✓	✗	✓	✓	✗
streamcluster	✓	✓	✓	✓	✓	✓
pathfinder	✗	✗	✗	✓	✗	✗
particlefilter	✓	✓	✗	✓	✓	✗
myocyte	✗	✗	✗	✓	✓	✓
leukocyte	✓	✓	✓	✓	✗	✗
19	14	11	9	17	13	9

Bm. Benchmark, ✗ Absence of NAC, ✓ Existence of NAC, SL = Scalar variable in the ℓ-value of a statement, DECL = Declaration statements, NAL = Non-affine Loop bound, NIF = Non-affine IF construct, IRR = Irregular loops, NAS = Non-affine Array Subscript, FILE = FILE I/O operations

Table 6.3: Non-affine constructs (NAC) that prohibit Pluto parallelization in NAS parallel benchmarks (NPB)

Bm.	NAC identified using Pluto					
	SL	FC	NAL	NIF	NAS	FILE
BT	✓	✗	✓	✓	✓	✗
CG	✓	✗	✓	✓	✓	✗
DC	✓	✗	✗	✓	✓	✓
EP	✓	✗	✗	✓	✗	✓
FT	✓	✓	✗	✓	✓	✗
IS	✓	✗	✗	✓	✓	✗
LU	✓	✗	✗	✗	✓	✗
MG	✓	✗	✗	✓	✓	✓
SP	✓	✓	✓	✗	✓	✗
UA	✓	✓	✓	✓	✓	✓
10	10	3	4	8	8	4

Bm. Benchmark, ✗ Absence of NAC, ✓ Existence of NAC, SL = Scalar variable in the ℓ-value of a statement, NAL = Non-affine Loop bound, NIF = Non-affine IF construct, FC = Function Call, NAS = Non-affine Array Subscript, FILE = FILE I/O operations

6.4.3 NAS parallel benchmarks (NPB) analysis

As previously discussed in Chapter 5, NPB are derived from computational fluid dynamics (CFD) applications and consists of ten kernels. Each of these benchmarks comprises of complex coding style and too many non-affine constructs (NACs) that affect the parallelization process. In total, six common NACs were identified during parallelization using Pluto and are listed is in Table 6.3.

6.4.4 Overall auto-parallelization issues in Pluto

In total, nine auto-parallelization issues in Pluto have been identified by examining three benchmark suites namely, Green-Marl, Rodinia and NAS parallel benchmarks (NPB), which are listed in Table 6.4. Among which SL, DECL, NIF, and NAL are frequently

Table 6.4: Overall auto-parallelization issues in Pluto

Benchmarks	Tested	SL	CC	FC	DECL	NAL	NAS	NIF	SR	FILE
Green-Marl	14	7	1	-	12	14	-	11	1	-
Rodinia	19	14	-	-	11	9	8	17	-	9
NPB	10	10	3	3	-	4	8	8	-	4
	43	21	4	3	23	27	16	36	1	13

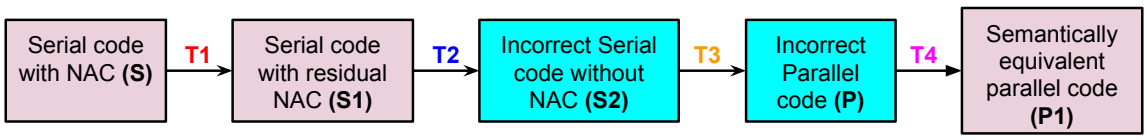
Bm. Benchmark, SL = Scalar variable in the ℓ-value of a statement, CC = Constructor Call, FC = Function Call, DECL = Declaration statements, NAL = Non-affine Loop bound, NIF = Non-affine IF construct, SR = Scope Resolution Operator, FILE = FILE I/O Operations

occurring problems. Each benchmark has at least one non-affine constructs (NAC) that forbids parallelization in Pluto.

The codes with these NAC are made amenable to parallelization using elimination method which will be a pre-parser and post-parser to the Pluto tool. The elimination method is illustrated next.

6.5 Method to eliminate Non-affine constructs (NAC) in Pluto

In this section, the proposed approach in making the input code analyzable by polyhedral frameworks is illustrated. For such an evaluation, we use polyhedra-based parallelizer, Pluto. From our earlier studies, the two frameworks that performed well are ICC and Pluto. ICC is a commercial compiler while Pluto is open-source tool. Therefore, we focus on Pluto in this chapter.



T1 Pre-elimination, **T2** In-elimination, **T3** Parallelize S2 using Pluto, **T4** Post-elimination

Figure 6.4: Method to eliminate non-affine constructs (NAC)

Table 6.5: Classification of identified non-affine constructs (NAC) and proposed solution

Problems	Descriptions	Solutions	Classification
P1	Scalar variables in ℓ-value	Scalar expansion	Pre-elimination
P2	Declaration statements	Loop-invariant code motion	
P3	Non-affine loop bound		
P4	Function Calls	Remove NAC from SCoP	In-elimination
P5	Constructor Calls		
P6	Non-affine IF construct		
P7	Non-affine array subscript		
P8	Scope resolution operator		
P9	FILE I/O operations	Remove parallel pragma	Post-elimination

Figure 6.4 illustrates the method to eliminate NAC in the input code and make them amenable to parallelization by preserving the semantics of the code. The elimination part is decomposed into three categories viz. **Pre-elimination**, **In-elimination**, and **Post-elimination**. Table 6.5 lists the problems that are solved in this chapter. Currently, the method is manually applied, but it can be automated. Our goal here is to check if temporary removal of NAC improves effectiveness of auto-parallelizers.

Algorithm 1 illustrates the 3-stage elimination process (manual method) which is performed to input code `INP`. The transformed code under each phase is stored in variable `T1`. After eliminating NAC, the parallelized code is checked for correctness with respect to the original sequential code. After retaining the semantics of the code, the final `T1` is targeted for evaluation. The 3-stage elimination process is explained subsequently in this section.

Algorithm 1: Elimination of NAC from SCoP

Input: INP := FOR loops with NAC
Output: Semantically correct parallel code T1 for evaluation

```
1  begin
       /* Pre-elimination                                          */
2      if Scalar in ℓ-value then
3          T1 := Scalar_Expansion(INP);
       /* Pre-elimination                                          */
4      if DECL within T1 || NAL within T1 then
5          T1 := Loop_Invariant(T1);
       /* In-elimination                                           */
6      if FC || CC || NIF || NAS || SR then
7          T1 := Remove_NAC(T1, Stmt);
       /* Incorrect Parallelization                                */
8      T1 := Pluto_Parallelize(T1);
       /* Post-elimination                                         */
9      Stmt := Renaming_Index(Stmt);
10     T1 := Re_add(T1, Stmt);
11     if FILE_IO within SCoP then
12         T1 := Remove_Pragma(T1);
```

6.5.1 Pre-elimination

Three NAC within the SCoP are addressed in this stage namely, scalar variables in the ℓ-value of statement (P1), declaration statement (P2), and non-affine loop bound (P3) (Table 6.5). In this stage two non-trivial sub-tasks are performed as part of elimination process (scalar expansion and loop-invariant code motion). However, during such transformation the semantics of the code is preserved.

Scalar expansion

Pluto worked with vectors but did not parallelize for-loops when the ℓ-value of the statement is a scalar variable. Hence, we convert the scalar to vector variable. However, we need to preserve the semantics of the code, (i) by checking the loop-carried data dependence due to scalar and preserving it (iii) by looking into if the scalar variables are used in function call/constructor call. A valid scalar expansion is illustrated in Figure 6.5(a) for single and nested for-loop.

Two major issues should be taken into account during scalar expansion:

- During such transformation, it is obvious that the vector size may dynamically grow which can lead to *memory bandwidth bottlenecks*. To avoid such scenario, we allocate and deallocate the array variable using malloc() within the scope as illustrated in Figure 6.5(b)

- Scalar expansion cannot be performed to a reduction variable in a single for-loop since it comprises *loop-carried* dependence, if vectorized then it would lead to false output. Figure 6.5(c) shows that scalar expansion is not feasible in case of single loop (T_1) when there is a reduction. The code snippets of (T_2) reveals that it is invalid to perform vectorization in a nested loop, when the reduction variable is globally defined. In case of nested loop, vectorization can be applied to the reduction variable in the inner loop, iff the value is updated within the outer loop.

Figure 6.6(a) shows the parallelization of `adamicAdar`, where scalar expansion (SV) is performed (shown in code fragments in T_2) i.e. scalar `_S0` is replaced as vector variable outer loop counter e as `_S0[e]` (transformation is highlighted in red). Similarly, the variables `from` and `to` are replaced as `from[e]` and `to[e]` respectively. The transformation is performed even within the function call. All the scalar variables within the SCoP are replaced with vector variable only if there is no loop-carried data dependence due to scalar variable.

Loop-invariant code motion

Pluto does not allow declaration of variables with user-defined/derived/primitive data types within the static control part (SCoP). Hence, these declarations are moved outside the SCoP by performing loop-invariant code motion (LI) without affecting the semantics of the original code. In Figure 6.6(a), lines 2-3 of T_1 is brought outside the SCoP by performing loop-invariant code motion (LI). However, this mechanism is performed by preserving the semantics of the code, hence initialization of variable `_S0` is retained within the SCoP.

(a) Valid Scalar expansion

```
{
    for (x=0; ...)
    {
     temp = 0;
     for (...)
     {
      temp = temp + ... ;
     }
    }
}
```

Original code (T_1)

```
{
    for (x=0; ...)
    {
     temp[x] = 0;
     for (...)
     {
      temp[x] = temp[x] + ... ;
     }
    }
}
```

Modified code (T_2)

(b) Eliminating the memory bandwidth bottleneck

```
{
    int *arr = (int *)malloc(n * sizeof(int));
    for (...) {
        ...
    }
    free(arr);
}
```

(c) Two cases where scalar expansion is invalid

```
{
    for (...)
    {
     temp = temp + ... ;
    }
}
```

Original code (T_1)

```
{
    temp = 0;
    for (...)
    {
     for (...)
     {
      temp = temp + ... ;
     }
    }
}
```

Modified code (T_2)

Figure 6.5: Scalar expansion issues and solution (a) Valid scalar expansion (b) Eliminating the memory bandwidth bottleneck (c) Two cases where scalar expansion is invalid

(a) Parallelization of `adamicAdar`

```
for(edge_t e = 0; e < G.
      num_edges(); e ++) {
node_t from; node_t to;
double _S0 = 0.0 ;
from = G.node_idx_src[e] ;
to = G.node_idx[e] ;
_S0 = ((float)(0.000000)) ;
gm_common_neighbor_iter n_I
      (G, from, to);
for(node_t n = n_I.get_next(); n
      != gm_graph::
      NIL_NODE ; n = n_I.
      get_next()) {
_S0 = _S0+1/log((G.begin[n+1]
      -G.begin[n]));
}
G_aa[e] = _S0 ;
}
```

T_1 (Original Code)

```
edge_t e;
auto X1 = G.num_edges();
double _S0[X1];
node_t from[X1];
node_t to[X1];
node_t n;
for (e = 0; e < X1; e ++) {
_S0[e] = 0.0 ;
from[e]=G.node_idx_src[e] ;
to[e] = G.node_idx[e] ;
gm_common_neighbor_iter n_I
      (G, from[e], to[e]);
for(n = n_I.get_next(); n !=
      gm_graph::NIL_NODE ; n
      = n_I.get_next()) {
_S0[e]=_S0[e]+1/log((G.begin[
      n+1]-G.begin[n])) ;
}
G_aa[e] = _S0[e] ;
}
```

T_2 (Pre- and In-elimination)

```
if (X1 >= 1) {
lbp=0; ubp=X1-1;
#pragma omp parallel for private(
      lbv,ubv,t2)
for (t1=lbp;t1<=ubp;t1++) {
_S0[t1] = 0.0 ;
from[t1]=G.node_idx_src[t1];
to[t1] = G.node_idx[t1] ;
gm_common_neighbor_iter n_I(G,
      from[t1], to[t1]);
for(node_t n = n_I.get_next(); n !=
      gm_graph::NIL_NODE ; n =
      n_I.get_next()) {
_S0[t1]=_S0[t1]+1/log((G.begin[n
      +1]-G.begin[n]));
}
G_aa[t1] = _S0[t1] ;
}
}
```

T_3 (Post-elimination)

Pre-elimination = SV (highlighted in red) and LI performed in lines 1-6 of T_2

In-elimination = remove NAC: CC, NAL ■

Post-elimination = RI (highlighted in violet), append NAC □

(b) Parallelization of `b+tree`

```
for(i = 0; i < count; i++){
start[i]=(rand()/(float)RAND_MAX)*size;
end[i] = start[i]+rSize;
if(end[i] >= size){
start[i] = start[i] - (end[i] - size);
end[i]= size-1;
   }
}
```

T_1 (In-elimination)

```
if (count >= 1) {
lbp=0; ubp=count-1;
#pragma omp parallel for private(lbv,ubv,t2)
for (t1=lbp;t1<=ubp;t1++) {
start[t1]=(rand()/(float)RAND_MAX)*size;
end[t1] = start[t1]+rSize;
if(end[t1] >= size){
start[t1] = start[t1]-(end[t1]-size);
end[t1]= size-1;
    }
  }
}
```

T_2 (Post-elimination)

In-elimination = remove NAC: NIF ■ , Post-elimination= append NAC □

Figure 6.6: Making codes amenable to parallelization to port via Pluto (a) Parallelization of `adamicAdar` *(b) Parallelization of* `b+tree`*, (continued in next page)*

All upper and lower loop bounds in the SCoP need to be affine expressions. Assuming if the loop bounds are non-affine, then the code becomes unanalyzable by the frameworks. Such cases were observed in the tested program. One such scenario is shown in Figure 6.6(a). In the code snippets of T_1, the key issue is with the conditional part of for-loop 'e < G.num_edges()', where the *r*-value is a function call which is non-affine. Hence, we eliminate the non-affine issues, by applying loop-invariant code motion (LI) method, where the declaration is moved outside the SCoP. Also, the *r*-value is assigned to a new temporary variable `X1` defined outside the SCoP and the *r*-value of for-loop condition is replaced with variable `X1` (line 2 and 7 in T_2 of Figure 6.6(a)).

The key issue to be noted before applying loop-invariant code motion in case of solving non-affine loop bound are as follows:

- In case the for-loop is not a canonical loop i.e. when there are multiple initialization or conditional statements, and when the condition of the for-loop (!=, ==) is invalid, then loop-invariant code motion is not performed. Such cases prevail in Green-Marl and Rodinia suites namely, `adamicAdar`, `b+tree`, `nw`, `sradd`, and `mummergpu`.

- In the nested for-loop, if the *r*-value of the initialization/conditional statement depends on the outer loop counter, then the loop-invariant code motion is avoided. This scenario exists in Green-Marl benchmarks namely, `avg_teen_cnt`, `pagerank`, `communities`, `potential_friends`, `sssp`, and `sssp_path`.

6.5.2 In-elimination

In total, five non-affine issues are elucidated and addressed in this stage namely, function calls (P4), constructor calls (P5), non-affine IF construct (P6), non-affine array subscript (P7), and scope resolution operator (P8) (problem illustrated in Table 6.5). This process involves the elimination of the above-discussed five NAC from the input code which forces the code to be incorrect. Following are the illustrations on each non-affine issues.

Function call (FC)/Constructor call (CC) Constructor call within SCoP made the code not amenable for parallelization. Hence, the constructor call was removed before porting to parallelization. For instance in Figure 6.6(a), T_2 shows the constructor call in line 11 removed from the benchmark `adamicAdar` of Green-Marl (highlighted in ■).

Non-affine IF construct (NIF) Non-affine IF constructs inside the SCoP stop the polyhedral framework to perform parallelization and hence is removed before sending the code to parallelizer. In Figure 6.6(b), T_1 reveals that the non-affine IF (NIF) construct (highlighted in ■) is removed from the input code and after obtaining the parallelized code the removed code is appended. The results of `avg_teen_cnt` shows the code snippets in T_2, where the removed IF construct has been appended to the parallelized code (highlighted in □).

Non-affine array subscript (NAS) On parallelization of `heartwall` benchmark of Rodinia, the system found that due to presence of non-affine array subscript (NAS), the code was not amenable to parallelization. Hence, the problematic code segment was removed from further parallelization. After this the code was feasible for parallelization. Figure 6.6(c) shows that presence of NAS in lines 5-7 of T_1 forbids parallelization, and hence after our elimination method the code was parallelizable. However, the incorrect code was appended with the removed statements (lines 10-12 in T_2) and the semantics of the code was preserved.

Scope resolution operator (SR) Presence of scope resolution operator (::) was found in Green-Marl applications. For instance, in the `pagerank` benchmark, the statement with scope resolution operator was removed and then appended back after obtaining the parallelized code.

6.5.3 Post-elimination

After obtaining the incorrect/correct parallelized code from Pluto, two significant checks are performed to validate the correctness of the program. Assuming that the parallelized

(c) Parallelization of `heartwall`

T_1 (In-elimination)

```
for(col=0; col<public.in2_cols; col++){
for(row=0; row<public.in2_rows; row++){
ori_row = row + in2_rowlow − 1;
ori_col = col + in2_collow − 1;
temp = public.d_frame[ori_col*public.frame_rows+
    ori_row];
private.d_in2[col*public.in2_rows+row] = temp;
private.d_in2_sqr[col*public.in2_rows+row] = temp
    *temp;
}
}
```

T_2 (Post-elimination)

```
auto X1 = public.in2_cols;
auto X2 = public.in2_rows;
if ((X1 >= 1) && (X2 >= 1)) {
for (t1=0;t1<=X1+X2−2;t1++) {
lbp=max(0,t1−X1+1); ubp=min(t1,X2−1);
#pragma omp parallel for private(lbv,ubv)
for (t2=lbp;t2<=ubp;t2++) {
ori_row[t2] = t2 + in2_rowlow − 1;
ori_col[t2] = (t1−t2) + in2_collow − 1;
temp = public.d_frame[ori_col[t2]*public.
    frame_rows+ori_row[t2]];
private.d_in2[(t1−t2)*public.in2_rows+t2] = temp;
private.d_in2_sqr[(t1−t2)*public.in2_rows+t2] =
    temp*temp;
  }
 }
}
```

In-elimination = remove NAC : NAS ■ , Post-elimination = append NAC □

Figure 6.6: Making codes amenable to parallelization by elimination method to port via Pluto (c) Parallelization of `heartwall` *(Contd.)*

code is incorrect due to two issues namely, (i) parallelization of loops with File I/O operations (ii) index variable mismatch during scalar expansion. Following are the ways in which the two issues are solved.

- Pluto inserts parallel directive to loops with File I/O operations. In general, reading and writing from a file needs to follow sequential semantics. Hence to maintain correctness, before evaluation, we remove the 'parallel pragma' from the loop.
- The scalar expansion applied to non-affine issues (such as function call (FC), non-affine array subscript (NAS), etc.) which are then removed during parallelization causes mismatch in the index variable. This is because Pluto performs polyhedral transformation in which it creates temporary variable and replaces the old variable. To overcome this, after appending the removed non-affine issues to the parallelized

code, we apply renaming of index variable (RI) to make a behavior preserving transformation. Figure 6.6(a) shows the usage of renaming of index variable (RI) in T_3 code snippets (lines 8 and 10, highlighted in violet), where the variables `from[e]` and `to[e]` is replaced as `from[t1]` and `to[t1]`.

6.6 Result analysis

Figures 6.7, 6.8 and 6.9 show the speedup results of three benchmark suites namely, Green-Marl, Rodinia and NAS parallel benchmarks (NPB) compared against Original, Pluto, and Pluto_M (Original is the speedup of the manually parallelized individual benchmark, Pluto is the speedup of the parallelized code *before* applying the elimination method, Pluto_M is the speedup of the parallelized code obtained *after* applying the elimination method). In the case of Green-Marl and Rodinia, the correctness of Pluto_M transformation was evaluated by comparing the output of the parallelized version with the sequential version. In the case of NPB, the correctness check is programmatically performed by the test case of the individual benchmark.

6.6.1 Experimental configuration

The experimental configuration is illustrated in Chapter 4. For empirical analysis, the number of threads is varied as 64, 32 and 16, and the speedup was compared against that of the sequential version.

6.6.2 Performance impact of elimination method on Green-Marl

Out of 14 benchmarks listed in Table 6.1, in total 12 benchmarks were amenable to parallelization using $Pluto\_M$. The unresolved two problems were `potential_friends` and `triangle_counting`, since after undergoing the elimination process, there were no parallelization candidate within the for-loop.

Figure 6.7 depicts the speedup results of Green-Marl codes. Pluto was not able to parallelize Green-Marl applications, as there were too many NAC which forbid parallelization. Hence, for evaluation purpose, the speedup of Pluto was assumed as unity, same as the sequential version. Pluto_M outperforms Pluto for 5 benchmarks (with 32 threads:

adamicAdar 11.58×, avg_teen_cnt 9.54×, communities 1.57×, pagerank 9.07×, and sssp_path 1.05×).

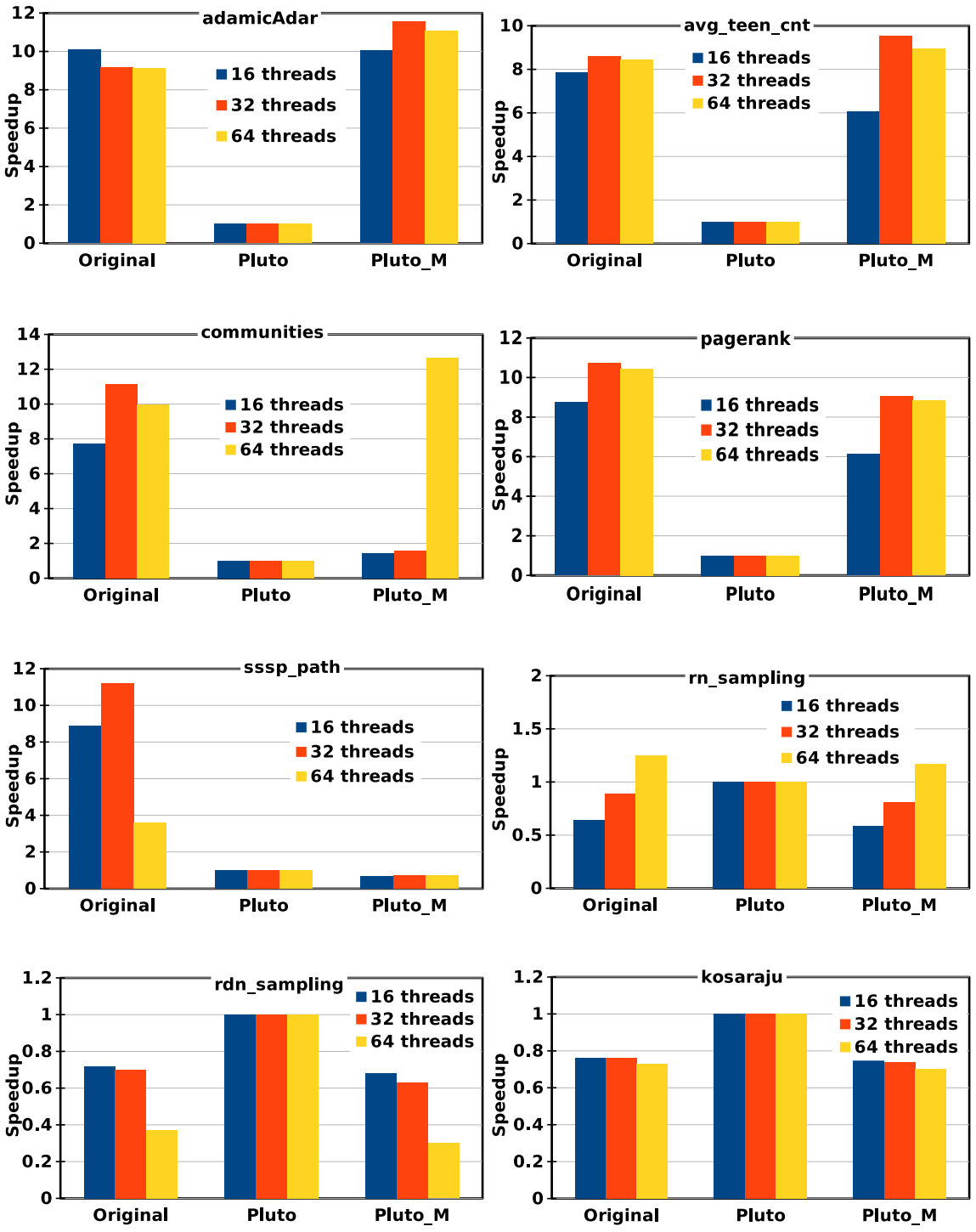
Interestingly, Pluto_M showed performance improvement than Original for adamicAdar and avg_teen_cnt. However, Pluto_M exhibits speedup less than the sequential version for few of the codes due to lack of usage of implicit barriers besides parallelization.

6.6.3 Performance impact of elimination method on Rodinia

In Rodinia, out of 19 listed in Table 6.2, on the whole 8 benchmarks were acquiescent to parallelization by Pluto_M. In the remaining 11 benchmarks, 9 codes were not feasible for parallelization as there were no parallelization candidate within the for-loop after undergoing the elimination process; 2 of the benchmarks showed semantically incorrect output and had left clueless of the source of the transformation error.

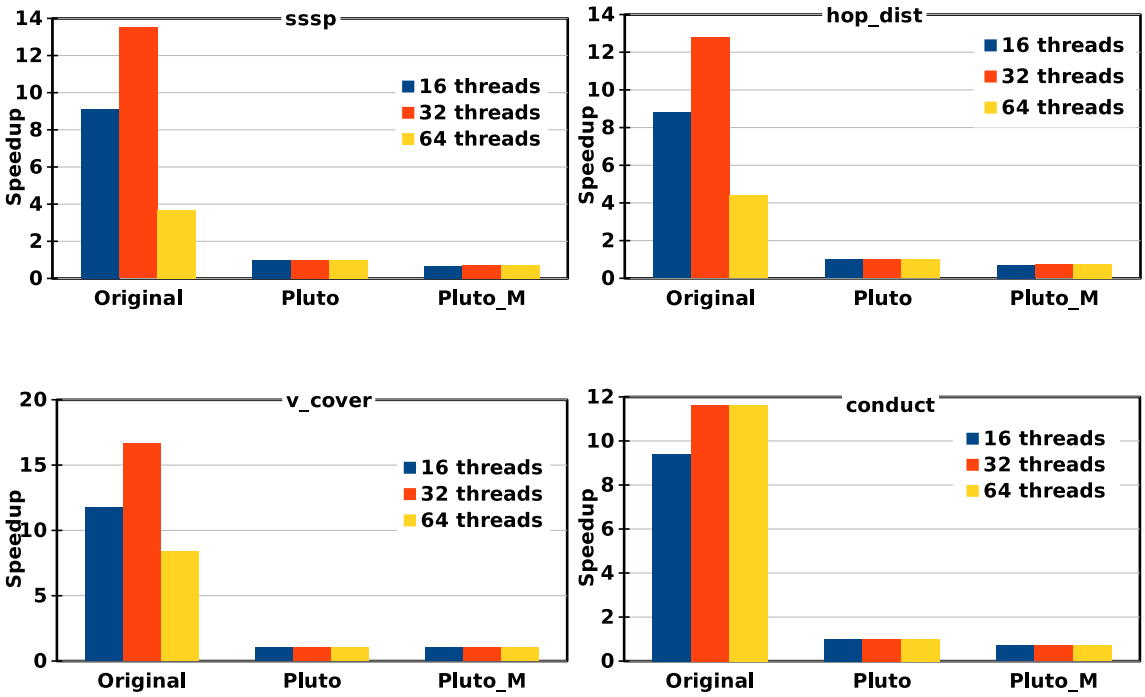
Figure 6.8 depicts the speedup results of eight of the benchmarks across Original, Pluto, and Pluto_M. The results reveal that Pluto_M performs better than Pluto for three of the applications (with 32 threads: heartwall 10.27×; kmeans 48.61×; particlefilter 1.11×; Pluto and Pluto_M show similar speedup results for b+tree). However, Pluto_M showed lesser speedup than Pluto for one of the benchmarks myocyte due to additional optimization.

Pluto_M outperforms Original for kmeans. The speedup of Pluto_M for few of the benchmarks is less when compared to Original due to some of the unresolved problems such as several other non-affine constructs; nonfeasible parallelization as there were no parallelization candidates after elimination process; lack of usage of implicit barriers and synchronization constructs.



Original = Parallelized Green-Marl version, Pluto = Before applying elimination method, Pluto_M = After applying elimination method

Figure 6.7: Speedup analysis of Green-Marl (`adamicAdar`, `avg_teen_cnt`, `communities`, `pagerank`, `sssp_path`, `rn_sampling`, `rdn_sampling`, `kosaraju`), (continued in next page)

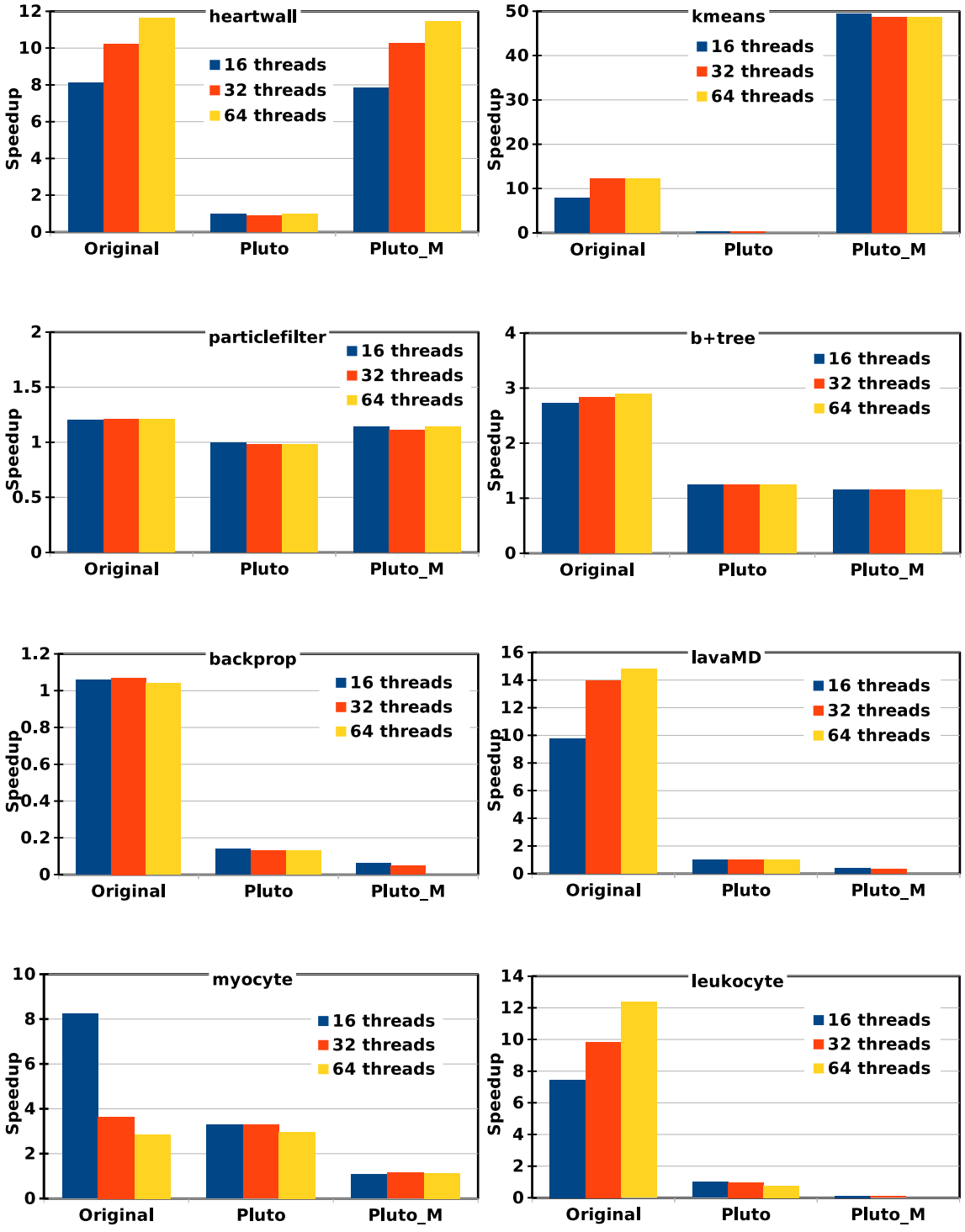


Original = Parallelized Green-Marl version, Pluto = Before applying elimination method, Pluto_M = After applying elimination method

Figure 6.7: Speedup analysis of Green-Marl (`sssp`, `hop_dist`, `v_cover`, `conduct`) (Contd.)

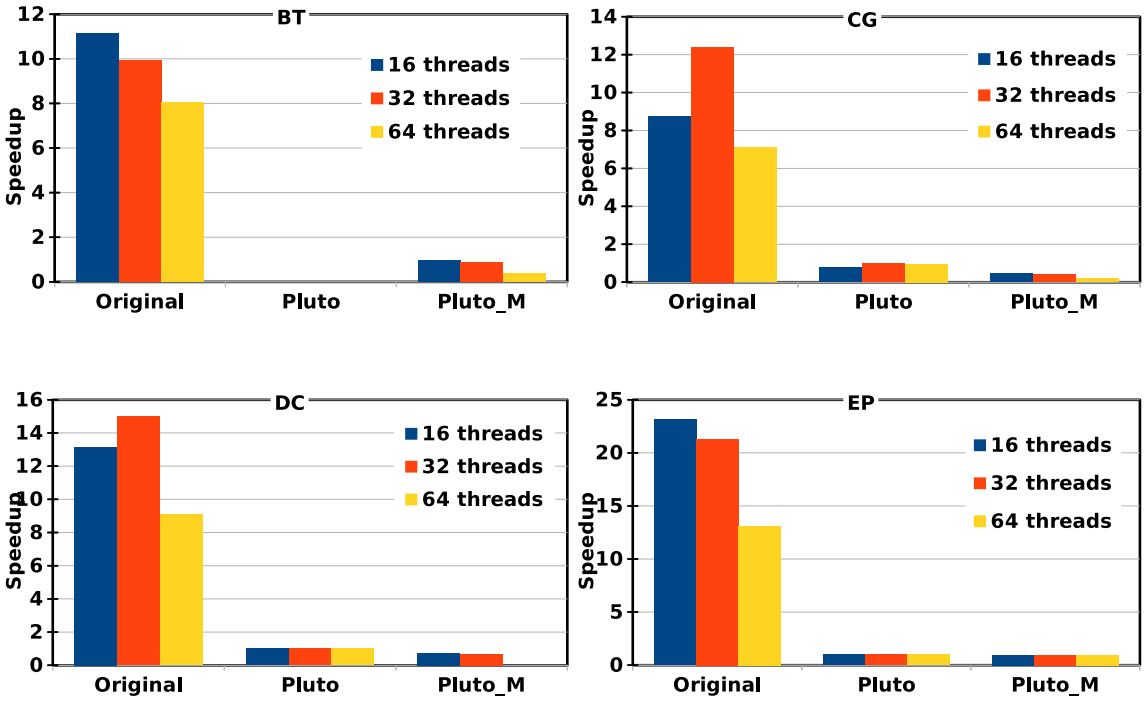
6.6.4 Performance impact of elimination method on NPB

Figure 6.9 shows the speedup results of Original, Pluto, and Pluto_M. Among 10 NPB codes listed in Table 6.3, nine applications were amenable to parallelization by subjecting to Pluto_M. For the remaining one benchmark, there were no parallelization candidate after the elimination process. The results illustrate that the parallelized NPB codes obtained after subjecting to the elimination process, Pluto_M does not show performance improvement when compared to Pluto and is less than the sequential version. The degradation happened as the removal of NAC leads to a no parallelization candidate case.



Original = Parallelized Green-Marl version, Pluto = Before applying elimination method, Pluto_M = After applying elimination method

Figure 6.8: Speedup analysis of Rodinia (`heartwall`, `kmeans`, `particlefilter`, `b+tree`, `backprop`, `lavaMD`, `myocyte`, `leukocyte`)

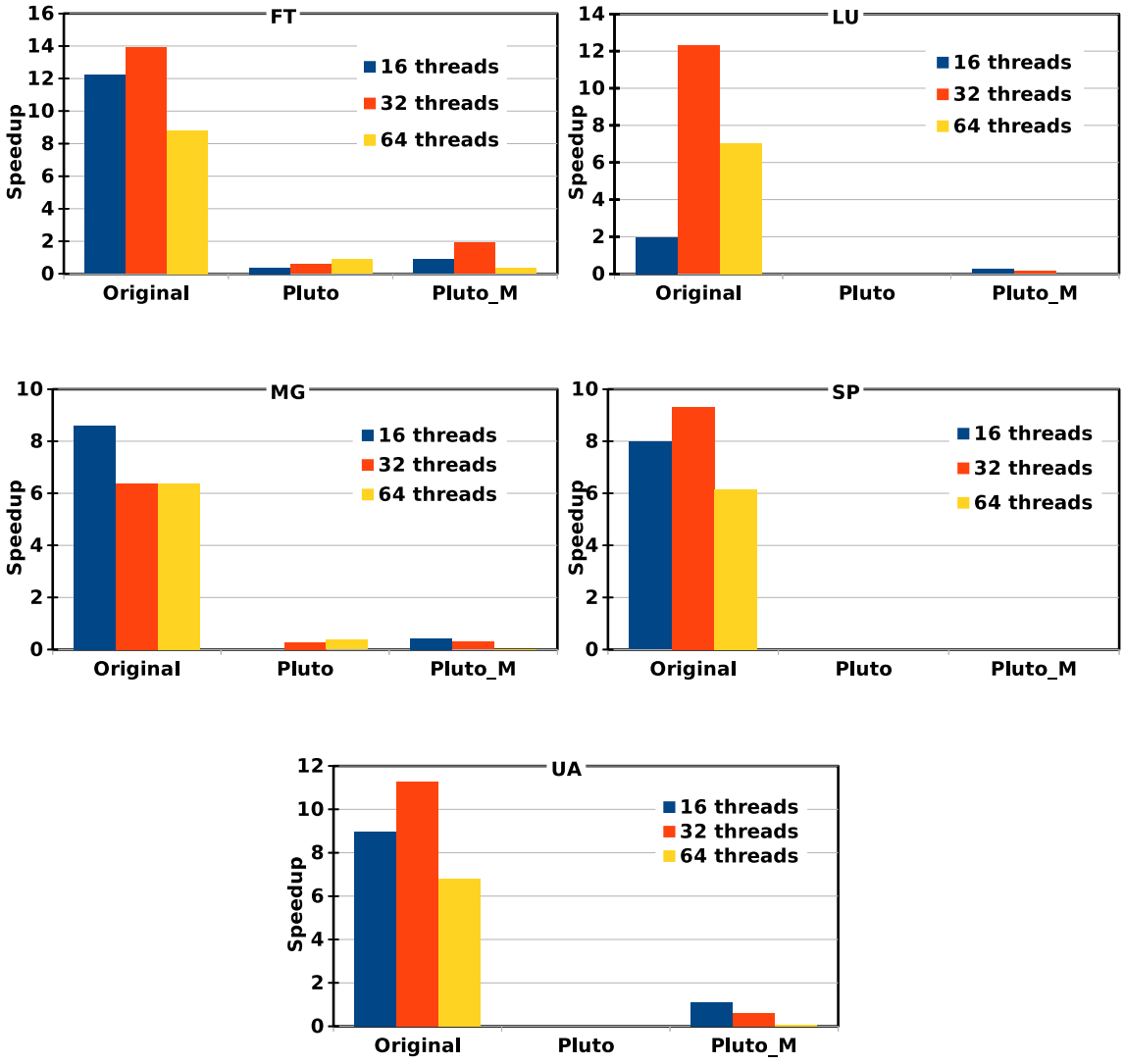


Original = Parallelized Green-Marl version, Pluto = Before applying elimination method, Pluto_M = After applying elimination method

Figure 6.9: Speedup analysis of NPB (`BT, CG, DC, EP`), (continued in next page)

It is also evident from Chapter 5 that the NPB benchmarks were less amenable to parallelization by auto-parallelizing frameworks due to inherent complex style. Further, it was reported in Chapter 5 that Pluto has performed inefficient parallelization in many instances viz. parallelization of the insignificant loop and too many optimizations. Also, present-day parallelizing compilers lag in the usage of implicit barriers and synchronization constructs.

The poorer results of Pluto tool on NPB show that modern parallelizing compilers should be tuned to parallelize real-world problems by making the code more amenable to parallelization. Besides the percentage of acceptance of code for parallelization, the frameworks should be modeled to perform efficient parallelization and inherently apply synchronization constructs wherever necessary.



Original = Parallelized Green-Marl version, Pluto = Before applying elimination method, Pluto_M = After applying elimination method

Figure 6.9: Speedup analysis of NPB (`FT`, `LU`, `MG`, `SP`, `UA`) (Contd.)

6.7 Summary

In total nine non-affine constructs (NAC) were identified by exploring three set of irregular and general-purpose programs namely Green-Marl, Rodinia, and NPB on parallelization with Pluto. A manual method, named Pluto_M, was introduced to eliminate the NAC from the codes, and several of them were acquiescent to parallelization. The manual elimination process comprised of three phases namely pre-elimination, in-elimination, and post-elimination.

The correctness of the transformed code was ensured. The resulting analysis reveals that several of the Green-Marl and Rodinia codes of Pluto_M showed performance improvement over Pluto and about 67% of the codes were amenable to parallelization. Pluto_M had shown better results over Original for few of the benchmarks. The elimination method's poorer outcomes on NPB codes portray the significance of manual intervention in case of real-world problems.

CHAPTER 7

Solution-focused Auto-parallelization Mechanism of Sequential Codes

This chapter introduces a new method to address the complications in the polyhedral parallelizer. Overall, five pitfalls are explored and resolved in this chapter. It explains the solution-focused mechanism that analyzes the compute-intensive function, determines the problems and provides an automated solution for parallelization. It discusses three phases of the proposed method namely, Profiling, Analysis and Code Transformation (ACT), and Parallelization. It also presents the validation results of the mechanism using PolyBench benchmarks.

7.1 Background

In Chapter 6 the limitations of polyhedral auto-parallelizer, Pluto, an open-source tool, were discussed and addressed using the elimination method. In this chapter, a solution is devised that focuses towards addressing various drawbacks in Pluto. The solution mechanisms are fully automatic. To effectively use the tool, a solution-focused automatic parallelization mechanism has been developed which utilizes profiling and code transformation to make the code more amenable to parallelization.

The three essential frameworks that contribute to the automated solutions are namely, GNU profiler (Gprof) [119], Rose [9] [60] , and Pluto [20].

Gprof [119] is a performance analysis tool for LINUX applications to keep track of the execution times of functions in the program.

Rose [9] [60] is also an open-source compiler framework to build source-to-source transformation and analysis tools for large-scale application[i] to improve its performance. It helps to perform the code transformations, analyses, and optimizations . Rose uses two

[i]Application that is developed in C (C89 and C98), C++ (C++98 and C++11), UPC, Fortran (77/95/2003), OpenMP [11], Java, Python, and PHP

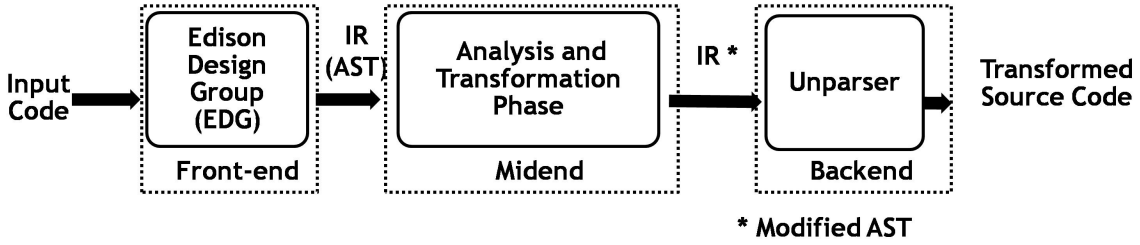


Figure 7.1: Schematic representation of Rose compiler framework

methodologies namely abstract syntax tree (AST) query, and AST traversal mechanism for code conversion. Figure 7.1 depicts the schematic representation of Rose and its components. The front-end, 'Edison Design Group (EDG)' [120] addresses the language specific parsers/front-end issues. The mid-end, 'analysis and transformation phase' is used to perform the analysis and code transformation by processing the AST information. The backend, 'unparser' handles the code generation part and generates the converted code by unparsing the source code.

Pluto as discussed earlier is implemented based on the polyhedral model which performs automatic optimization and parallelization. It is a mathematical framework for loop optimization in program optimization [83].

7.2 Proposed method

In this section, a solution-focused automatic parallelization mechanism is proposed as a pre-parser to Pluto which alleviates the parallelization pitfalls. It combines the profiler, Gprof [119] along with AST [121] based analysis and code transformation (using Rose compiler framework [9] [60]) and polyhedral parallelizer (Pluto). Based on the studies, problems are identified pertaining to the Pluto tool and are listed in Table 7.1. Problems P1 and P2 were discussed in Chapter 3 and in-depth analysis on P1 problem was discussed in Chapter 6. Figure 7.2 illustrates the problems P3-P5. Solutions are derived for these issues and are automated.

In this manner, the proposed method includes the following three phases; *Profiling*, *Analysis and Code Transformation (ACT)*, and *Parallelization*. The *Profiling* phase utilizes

Table 7.1: Problems and suggested solutions for static control part (SCoP) rejection in Pluto

No.	Problems	Suggested Solutions
P1	while loop	Convert the while-loop to for-loop
P2	Scalar Variables in the ℓ-value of a statement	Scalar expansion
P3	Reverse for-loop	Convert the upper bound to lower bound
P4	Updation of for-loop by more than one time step	Convert the increment step to one
P5	Irregular loop[a]	Transform to a regular loop

[a] In this chapter, Irregular loop* refers to the code fragments comprising incrementing/Decrementing of loop index variable such as i++, i–, i=i+n, i=i-n, i+=n, and i-=n with the for-loop.

the Gprof to extract the function name and execution time of hotspot[ii].

The *ACT* phase takes the output from the *Profiling*, and with the aid of the abstract syntax tree (AST) query mechanism, identifies the parallelizable region of the hotspots comprising of the pitfalls. Once the problems are determined, to make the code amenable to parallelization, automatic code transformation is applied. *ACT* modifies the intermediate representation (IR[iii]) of the code and makes it acquiescent to parallelization.

The key contribution is in finding the presence of non-affine codes in benchmark programs using *Profiling* and applying the code transformation using ACT to make the code amenable to parallelization using Pluto. Hence, the phases, *Profiling* and *ACT* are a pre-pass for Pluto (refer Figure 7.3(a)).

Finally, the polyhedral parallelizer, Pluto, parses the translated code and generates the paralleized code.

[ii] Compute-intensive functions of input C code
[iii] IR is modified using the Rose compiler framework

(a) Reverse for-loop

```
for(i = n; i >= 0; i--)
 a[i] = b[i] + c[i];
```

Original code

```
for(i = 0; i < n; i++)
 a[n-i] = b[n-i] + c[n-i];
```

Reformed code

(b) Updation of for-loop by more than one time step

```
for(i = 0; i < n; i += 3)
 a[i] = b[i] + c[i];
```

Original code

```
for(i = 0; i < n; i++)
{
 a[i] = b[i] + c[i];
 i += 2;
}
```

Reformed code

*(c) Irregular loop**

```
for(i = 0; i < n; i++)
{
 a[i] = b[i] + c[i];
 i += 2;
}
```

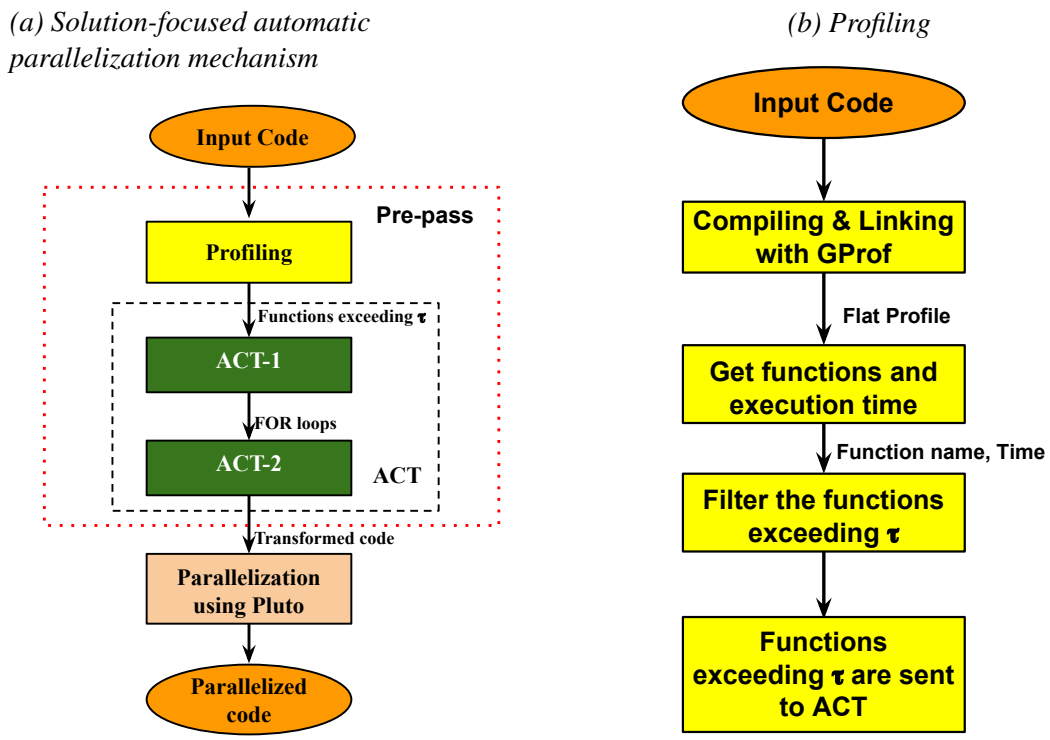
Original code

```
if (n % (1 + 2)==0)
 _zB = n / (1 + 2);
if (n % (1 + 2)!=0)
 _zB = n/(1 + 2)+1;
for(i = 0; i < n; i++)
{
 a[(i+1) * 0 + i * (2+1)] = b[(i+
     1) * 0 + i * (2+1)] + c[(i+1)
      * 0 + i * (2+1)];
}
```

Reformed code

In this chapter, Irregular loop* refers to the code fragments comprising incrementing/Decrementing of loop index variable such as $i++$, $i--$, $i = i + n$, $i = i - n$, $i+ = n$, and $i- = n$ with the for-loop. This example has $i+ = 2$ within for-loop which is unsupported by Pluto.

Figure 7.2: Suggested solution for the static control part (SCoP) rejection problems in Pluto (a) Reverse for-loop (b) Updation of for-loop by more than one time step (c) Irregular loop



The symbol τ is the threshold time. The execution time of a function that exceeds τ are termed as compute-intensive.

Figure 7.3: Proposed method for alleviating pitfalls in Pluto (a) Solution-focused automatic parallelization mechanism (b) Profiling

7.3 Implementation

As discussed earlier, the phase-by-phase approach of the proposed method includes *Profiling*, *Analysis and Code Transformation* (*ACT*), and *Parallelization*. Gprof is used to implement the *Profiling* stage. Rose compiler framework is used to perform the *ACT*. Pluto, a polyhedral parallelizer is used for *parallelization* of the transformed code.

Figure 7.3(a) depicts the schematic representation of the proposed solution-focused automatic parallelization mechanism to alleviate pitfalls faced by Pluto. Following sections provide a detailed description of the proposed method.

7.3.1 Profiling

Code profiling is one of the most critical aspects of the software development as it lets you identify the bottlenecks, dead codes, bugs, and regions that take more execution time.

Hence this phase of the proposed method is dedicated to finding the compute-intensive functions (hotspots) so that the transformation is avoided in trivial applications. Figure 7.3(b) depicts the processes involved in *Profiling*. The input code was first compiled and linked with Gprof [iv]. Gprof helps to determine the execution time of a function in the program. The flat profile is extracted from the Gprof which is used to extract the function name and execution time. The threshold timing is set as τ i.e. the name of the function whose execution time is exceeding τ are termed as compute-intensive and is sent to the *ACT* phase to identify the pitfalls (Table 7.1) in the parallelizable region of hotspots and is subjected to further coding changes.

7.3.2 Analysis and code transformation (ACT)

The hotspots obtained from the *Profiling* phase is fed to the *ACT* phase. In this stage, the pitfalls that forbid parallelism are identified and resolved. Table 7.1 lists the problems that cause the rejection of loops while parallelizing static control part (SCoP) along with the proposed solutions. *Analysis and Code Transformation* contains two phases: the first phase, *Analysis and Code Transformation Phase-1* (*ACT-1*) involves identifying the while-loop (P1) within hotspots and converting to for-loop. The second phase, *Analysis and Code Transformation Phase-2* (*ACT-2*) includes detecting the pitfalls (P2, P3, P4, and P5) in the for-loop within the hotspot and resolving it for further parallelization. The identified pitfalls are solved by modifying the AST. It is implemented using the AST query mechanism by changing the intermediate representation (IR) of the Rose compiler using the SageInterface and SageBuilder APIs of Rose. It primarily converts one AST to another version of AST. The process comprises of addition, deletion, and modification of the information stored in the AST. The code transformation implemented has the following steps:

- Search for the AST nodes to be transformed.
- Perform the transformation action on the obtained AST nodes.
- Update the existing AST nodes.
- After rewriting the AST, the code is unparsed to get the transformed code which can be subjected to parallelization.

[iv] using '-pg' flag along with the GNU compiler

The sections 7.4 and 7.5 briefly describe the *ACT-1* and *ACT-2* which illustrate the mechanism of addressing the problems listed in Table 7.1.

7.3.3 Parallelization

The transformed code from *ACT-1* and *ACT-2* is subjected to Pluto, since the identified predicaments are related to polyhedral parallelizer. Based on our earlier study on different auto-parallelizers, Pluto is a good parallelizer which parallelizes SCoPs with loop-carried dependences. The polyhedral parallelizer, Pluto is very efficient in performing optimization and parallelization. It supports C with OpenMP parallelization. Hence this mechanism aims at improving Pluto parallelization.

7.4 Analysis and code transformation phase-1 (ACT-1)

ACT-1 helps to detect and transform the while-loop (P1 in Table 7.1). Pluto does not support while-loop parallelization and hence, it is converted to for-loop by rewriting the AST using AST query mechanism.

Figure 7.4(a) and 7.4(b) portrays the AST representation of while and for-loop for the example illustrated in Figure 7.6(a). The control flow of while-loop is different from that of for-loop. Four essential components for such a transformation are: (i) initialization (ii) condition (iii) updation (iv) basic block.

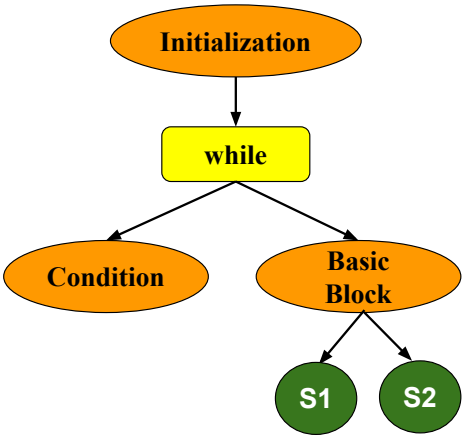
Figure 7.5 depicts the implementation of analysis and code transformation phase-1 (*ACT-1*) for transforming while-loop to for-loop.

7.4.1 Auto-conversion of while-loop to for-loop

Algorithm 1 shows the algorithm of transforming while fragment to for. The example depicted in Figure 7.6(a) illustrates the algorithm. For each W_j the initialization (`INIT`, $i = 0$); condition (α, $i < n$); updation (`UPD`, i++) and basic block `B1`, i.e. statements S_1 and S_2 are obtained to build the for-loop.

The more challenging aspects are getting `INIT` and `UPD` of W_j. Note that finding `INIT` is hard for a particular W_j as the initialization of variable i is unknown before the loop begins. A mechanism is found to retain the semantics of the code by assigning (α_L) to a temporary variable (`VAR`, $\_Cy$), which forms the `INIT` portion $\_Cy = i$. Algorithm 2

(a) while-loop



(b) for-loop

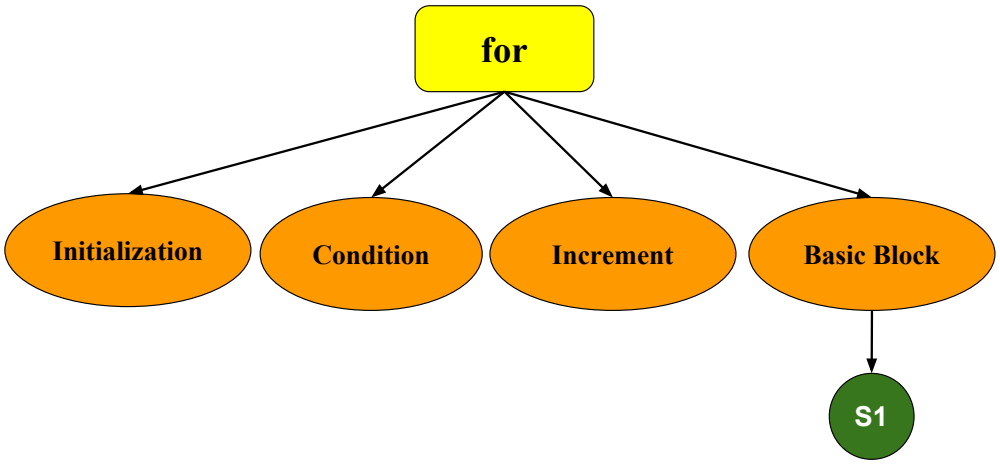


Figure 7.4: Abstract syntax tree (AST) graph of (a) while-loop (b) for-loop

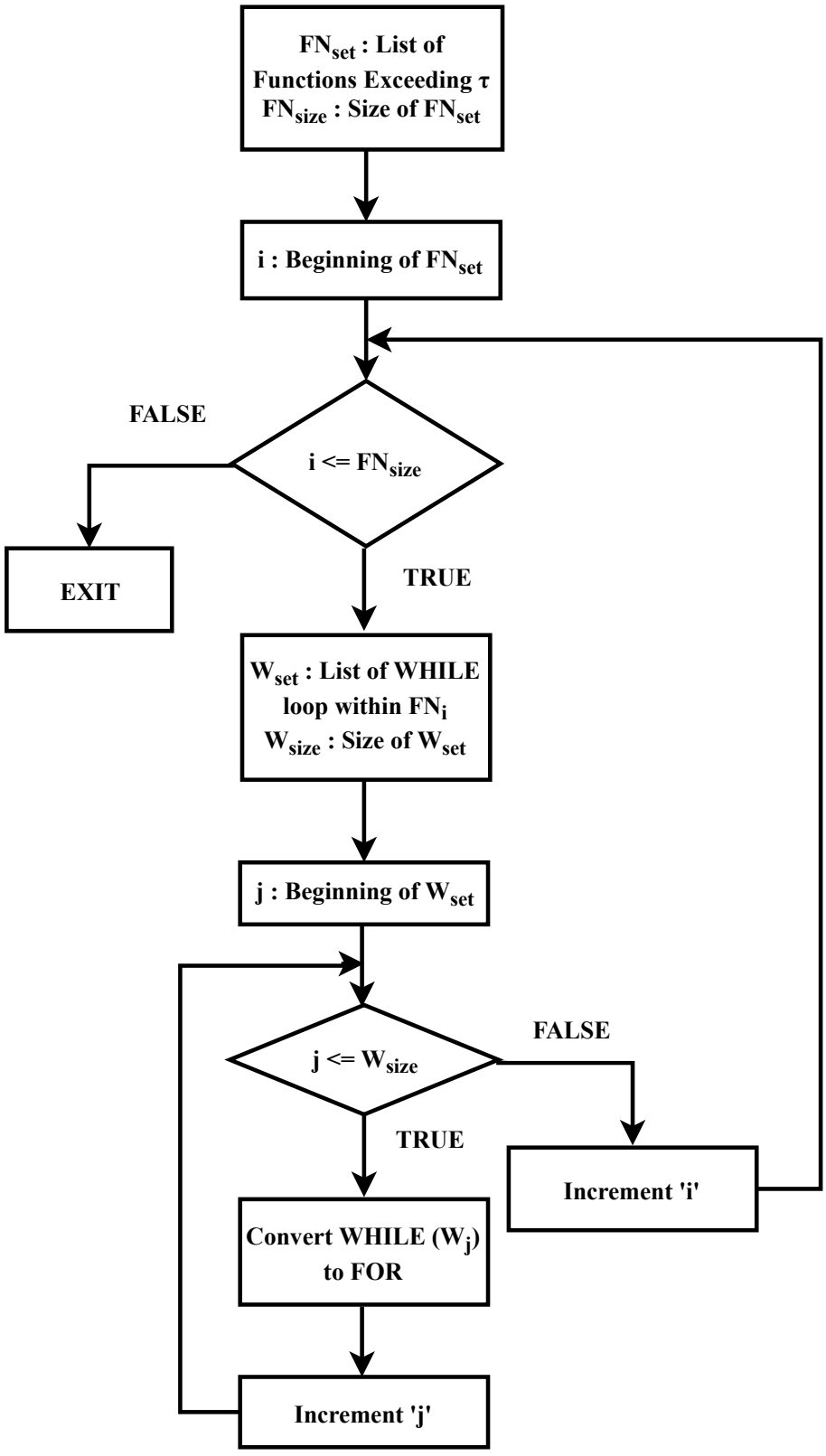


Figure 7.5: Flowchart representation of the implementation of analysis and code transformation phase-1 (ACT-1)

```
int i;
i = 0;
while(i < n)
{
S1: c[i]=a[i]+b[i];
S2: i++;
}
```

Original Code

```
int i;
i = 0;
int _Cy = 0;
_Cy = i;
for (i=_Cy;i<n;i++)
{
c[i] = a[i] + b[i];
}
```

Reformed Code

```
if (_Cy <= n-1)
{
lbp=_Cy;
ubp=n-1;
#pragma omp parallel for
     private(lbv,ubv)
 for (t1 = lbp; t1 <=
     ubp; t1++)
 {
 c[t1]=a[t1] + b[t1];;
 }
}
```

Parallelization of Reformed Code

(a) Updation at end of the basic block of while*-loop*

```
int i;
i = 0;
while(i < n)
{
a[i] = i;
b[i] = i + 2;
c[i] = a[i] + b[i];
i++;
d[i] = c[i-1] + 5;
}
```

Original Code

```
int i;
i = 0;
int _Bp = 0;
_Bp = i;
for (i=_Bp;i<n;i++)
{
a[i] = i;
b[i] = i + 2;
c[i] = a[i] + b[i];
d[i+1] = c[i+1-1]+5;
}
```

Reformed Code

```
if (_GC <= n-1)
{
lbp=_GC;
ubp=n-1;
#pragma omp parallel for
     private(lbv,ubv,t2,
    t3)
 for (t1 = lbp;t1 <= ubp
     ;t1++)
 {
 a[t1] = t1;;
 b[t1] = t1 + 2;;
 c[t1] = a[t1]+b[t1];;
 d[t1+1] = c[t1+1-1] 5;;
 }
}
```

Parallelization of Reformed Code

(b) Updation at middle of the basic block of while*-loop*

Figure 7.6: Auto-conversion of while*-loop to* for*-loop (a)Updation at end of the basic block of* while *(b) Updation at middle of the basic block of* while

Algorithm 1: Conversion of while to for : whileToFor

Input: W_{set} : Set of while-loop
Output: F_{set} : Set of for-loop

```
1  begin
2      for each W_j do
           /* Get the Conditional Part                                  */
3          α := Conditional Statement of W_j;
4          α_L := LHS variable of α;
           /* Get the Initialization Part                               */
5          T := Data type of α_L;
6          V_set := Set of Variables of W_j;
7          for each V_i do
8              Insert V_i in set X;
9          VAR := createTemp(X);
10         DECL := Declaring the Variable VAR of datatype T;
11         STMT := Building a statement by assigning α_L to VAR;
12         INIT := Building a statement by assigning VAR to α_L;
13         Insert DECL, STMT before W_j;
           /* Get the Updation Part                                     */
14         S_set := Set of Statements from bottom of Basic Block B;
15         COUNT := 0;
           /* isUnaryOperator returns TRUE if S_k contains unary
           operator                                                     */
16         for each S_k do
17             S_{k+1} := Get the next statement of S_k;
18             if isUnaryOperator(S_k) then
19                 β := Get the Operand;
20             else
21                 β := Get the LHS Operand;
22             if β == α_L then
23                 COUNT := COUNT +1;
24                 if COUNT == 1 and S_{k+1} == NULL then
25                     UPD := S_k;
26                     Delete(S_k);
27                 else if COUNT == 1 and S_{k+1} ≠ NULL then
28                     UPD := S_k;
29                     EXP := Build expression using S_k;
30                     modifyStmt(S_k,EXP,α);
31         B_1 := Get the Basic Block after modification of Statements;
32         Build the FOR using INIT,α,UPD,B_1;
```

illustrates the creation and validation of VAR.

Algorithm 2: Creation of temporary variable : createTemp

```
   Input: X : Set of variables of W_j
   Output: VAR : Temporary variable is created
1  begin
2      Temp := Create a temporary variable using random function;
3      if ∃(TEMP) ∉ X then
4          Insert TEMP in X;
5          VAR := TEMP;
6      else
7          createTemp(X);
```

Algorithm 3: Modify basic block of for-loop : modifyStmt

```
   Input: S_k : Statement,
   EXP : New expression,
   α : Condition
   Output: S_set : Set of Statements within B are modified
1  begin
2      INDEX := 0;
3      while S_{k+1} ≠ NULL do
4          α_L := LHS of α;
5          S_{k(L)} := LHS of S_k;
6          if α_L ≠ S_{k(L)} then
7              V_set := Set of Variable of S_k;
8              for each V_i do
9                  if V_i == α_L then
10                     replace variable V_i with EXP;
11             if INDEX == 0 then
12                 Delete(S_k);
13             INDEX := INDEX+1;
14         S_{k+1} := Get the next statement of S_k;
```

Two different scenarios of while to for transformations are discussed here. If S_k is found to be an update part, then there can exist two possible cases (explained in Algorithm 3):

- Case (i): If S_{k+1} is equal to NULL, S_k gets copied to `UPD`, and the node is deleted. For instance in Figure 7.6(a), updation i++ of the index variable i occurs at the end of the basic block.

- Case (ii): IfS_{k+1} is not equal to NULL, S_k gets copied to `UPD`, and further modifications are done to basic block B. For instance in Figure 7.6(b), i++ occur in the middle of the basic block.

After conversion of all while-loop to for, the F_{set} (set of for-loop in compute-intensive function) is sent to *ACT-2* for identifying and solving the remaining pitfalls.

Constraints in while to for-loop transformation: This transformation is performed only if:
(i) the for-loop conditional operator is $<$, $>$, $<=$, and $>=$.
(ii) the increment/decrement statement is of the form i++, i–, i=i+n, i=i-n, i+=n, and i-=n, where i is the loop index variable.
(iii) there is no nested while loop.

7.5 Analysis and code transformation phase-2 (ACT-2)

The static control part (SCoP) should be an affine expression to be supported by polyhedral parallelizer [122] [123]. *ACT-2* addresses the non-affine issues (P2, P3, P4, and P5) listed in Table 7.1. Figure 7.7 illustrates the *ACT-2* mechanism. After resolving the pitfalls, the transformed code is obtained which is ready for parallelization.

7.5.1 Automatic scalar expansion

As discussed in the previous chapters, scalar variable in the ℓ-value of a statement has been a serious issue prohibiting parallelization by Pluto for many real-world problems. Hence, automatic scalar expansion [112] is performed.

Let us consider the code snippet from Figure 7.8(a). The scalar w of the statement S_1 within the SCoP forbids parallelization. Hence, a mechanism is introduced to convert all the scalar w to a vector $w[i]$. The essential region to be modified is only the statements within the basic block of for.

Algorithm 4 emphasizes the approach before performing scalar expansion. Algorithm 5 illustrates retrieving the block information of F_i i.e., basic block `B`, `LHS` (α_L) and `RHS` (α_R)

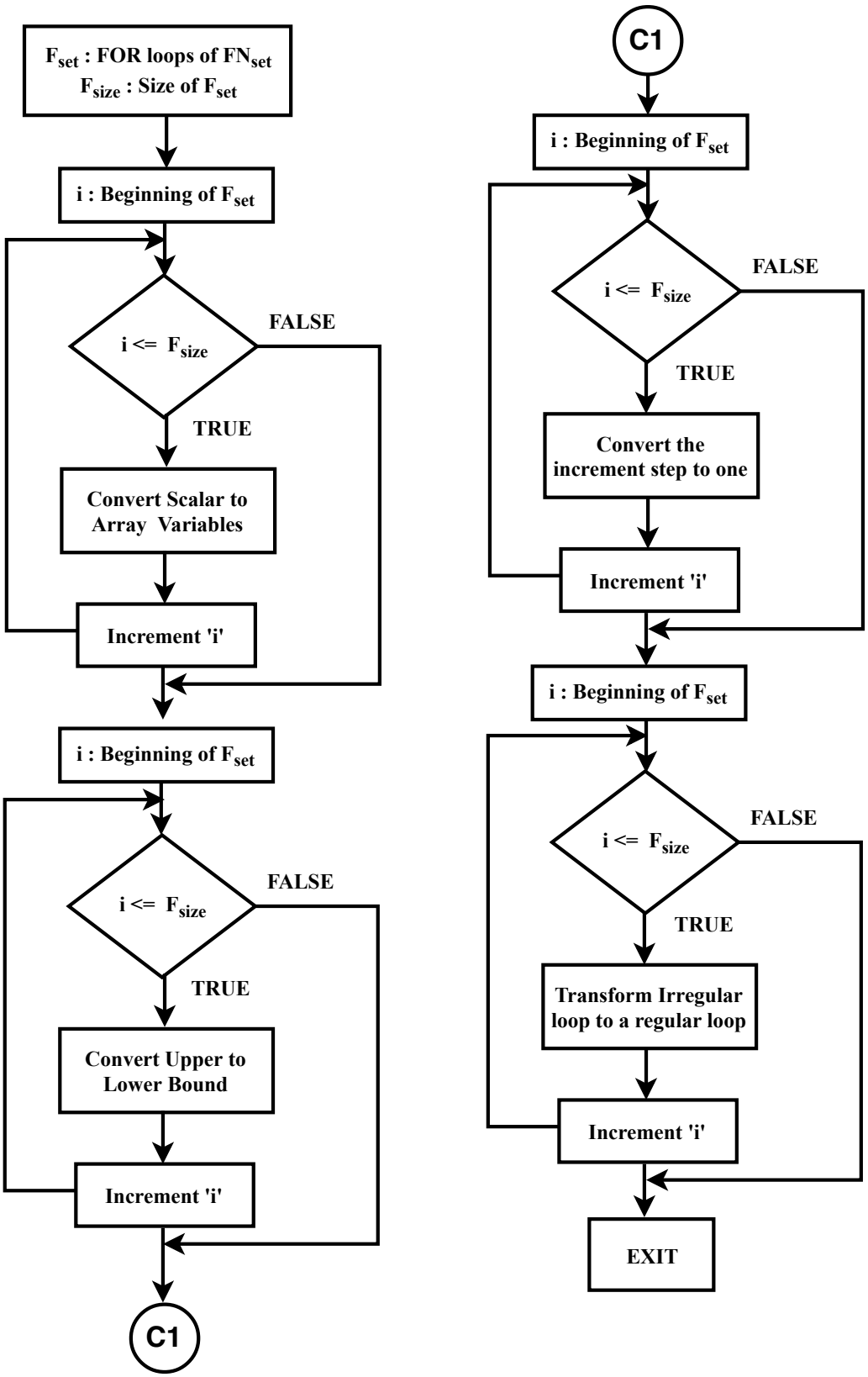


Figure 7.7: Flowchart representation of the implementation of analysis and code transformation phase-2 (ACT-2)

(a) Data dependence

```
int w;
for(i=0;i<n;i++) {
S1:w = i + 1;
S2:a[i] = w * 2;
S3:b[i] = i + 2;
S4:c[i]=w+a[i]+b[i];
}
```
Original Code

```
int w[n];
for(i=0;i<n;i++) {
w[i] = i + 1;
a[i] = w[i] * 2;
b[i] = i + 2;
c[i]=w[i]+a[i]+b[i];
}
```
Reformed Code

```
int w[n];
if (n >= 1) {
lbp=0; ubp=n-1;
#pragma omp parallel for
    private(lbv,ubv,t2)
for(t1=lbp;t1<=ubp;t1++)
{
w[t1]=t1 + 1;;
b[t1]=t1 + 2;;
a[t1]=w[t1] * 2;;
c[t1]=w[t1]+a[t1]+b[t1];
 }
}
```
Parallelization of Reformed Code

(b) Illustration for unsupported scalar variables: Loop-carried dependence

```
for(i = 0; i < n ; i++) {
w = w + 1;
a[i] = w * 2;
b[i] = i + 2;
c[i] = w + a[i] + b[i];
}
```
Single loop

```
int w = 0;
for(i = 0; i < n ; i++) {
 for(j = 0; j < n; j++) {
  w = w + 1;
  A[i][j] = w * 2;
 }
}
```
Nested loop

(c) Anti dependence

```
int w;
for(i=1;i<n;i++){
S1 : a[i] = w;
S2 : c[i] = w+a[i];
S3 : w = i + 2;
}
```
Original Code

```
int w[n];
for(i=1;i<n;i++) {
a[i] = w[i-1];
c[i] = w[i-1]+a[i];
w[i] = i + 2;
}
```
Reformed Code

```
int w[n];
if (n >= 2) {
a[1] = w[1 - 1];;
c[1] = w[1 - 1]+a[1];;
lbp=2;  ubp=n-1;
#pragma omp parallel for
    private(lbv,ubv,t2,
    t3)
for(t1=lbp;t1<=ubp;t1++)
    {
 w[t1-1] = t1-1 + 2;;
 a[t1] = w[t1 - 1];;
 c[t1]=w[t1 -1]+a[t1];;
 w[n-1] = n-1 + 2;;
 }
}
```
Parallelization of Reformed Code

Figure 7.8: Example illustration of automatic scalar expansion(a) Data dependence (b) Illustration for unsupported scalar variables: Loop-carried dependence (c) Anti dependence, (continued in next page)

(d) Output dependence

```
int w[n];
for(i=0;i<n;i++) {
w[i] = i;
a[i] = w[i];
b[i] = i * 2;
w[i] = i + 4;
c[i]=a[i]+b[i]+w[i];
}
```

Original Code

```
int w[n];
for(i=0;i<n;i++) {
w[i] = i;
a[i] = w[i];
b[i] = i * 2;
w[i] = i + 4;
c[i]=a[i]+b[i]+w[i];
}
```

Reformed Code

```
int w[n];
if (n >= 1) {
lbp=0;
ubp=n-1;
#pragma omp parallel for
     private(lbv,ubv,t2,
    t3,t4)
for(t1=lbp;t1<=ubp;t1++)
     {
 w[t1] = t1;;
 b[t1] = t1 * 2;;
 a[t1] = w[t1];;
 w[t1] = t1 + 4;;
 c[t1] = a[t1] + b[t1] +
     w[t1];;
 }
}
```

Parallelization of Reformed Code

(e) Nested loop with scalar variable in `LHS` *of* S_1

```
int w;
for(i=0; i<n; i++)
{
S_i^1: w = 0;
 for(j=0; j<n; j++)
 {
 S_j^1: w = i + 1;
 S_j^2: A[i][j] = w*2;
 }
}
```

Original Code

```
int w[n];
for (i = 0; i < n; i
    ++)
{
w[i] = 0;
 for (j = 0; j < n;
     j++)
 {
 w[i] = i + 1;
 A[i][j] = w[i] * 2;
 }
}
```

Reformed Code

```
int w[n];
if (n >= 1)
{
lbp=0;
ubp=n-1;
#pragma omp parallel for
     private(lbv,ubv,t3,
    t4)
for(t2=lbp;t2<=ubp;t2++)
w[t2] = 0;;
lbp=0;
ubp=n-1;
#pragma omp parallel for
     private(lbv,ubv,t3,
    t4)
for(t2=lbp;t2<=ubp;t2++)
 {
 for(t3=0;t3<=n-1;t3++)
  {
  w[t2] = t2 + 1;;
  A[t2][t3] = w[t2]*2;;
  }
 }
}
```

Parallelization of Reformed Code

Figure 7.8: Example illustration of automatic scalar expansion (d) Output dependence (e) Nested loop with scalar variable in `LHS` *of* S_1 *(Contd.)*

of condition (α).

Algorithm 4: Scalar expansion : scalarToArray

Input: F_{set} : Set of for-loop with scalar in ℓ-value of statement
Output: F_{set} : Set of for-loop after performing scalar expansion

```
1  begin
2      B,α_L,α_R := NULL;
       /* Hash table is created to identify the valid scalar to be
       modified                                                    */
3      FLAG := 0;
4      for each F_i do
5          DEPTH := 0;
6          blockInfo(F_i,B,α_L,α_R);
7          F_i = singleNested(B, FLAG);
       /* Scalar expansion is applied here                        */
8      FLAG := 1;
9      for each F_i do
10         Execute line 5 to 7
```

Algorithm 5: Retrieving for block attribute : blockInfo

Input: F_i : for-loop
Output: All information about block F_i

```
1  begin
2      B := Basic Block of F_i;
3      α := Conditional statement of F_i;
4      α_(L) := LHS variable of α;
5      α_(R) := RHS variable of α;
```

In order to preserve the semantics of the code during scalar expansion, the code is subjected to undergo two passes:

1. The first pass reveals if there are any problematic statements contradicting the conversion (see Algorithm 6). For instance, in Figure 7.8(b) (single loop), the scalar variable w is not feasible for conversion since the code fragment comprises loop-carried dependence $w = w + 1$.

2. The second pass contributes to the transformation if there exists no dependence.

Algorithm 6: Finding the type (Single/Nested) of loop : singleNested

Input: B : Basic Block of F_i
Output: Forming the HashTable HT with scalar variables and its boolean value

```
1  begin
2      S_set := Set of Statements of B; Create an empty HashTable HT[string,bool];
       /* isForStatement returns TRUE if S_j is a FOR                    */
       /* isStatement returns TRUE if S_j is a statement                 */
       /* isPntrArrRefExp returns TRUE if S_j(L) is an array variable    */
       /* isPlusAssignOp/isMinusAssignOp is TRUE if S_j has +=/-=        */
       /* isAssignInitializer returns TRUE for an assign initializer     */
3      for each S_j do
4          if isForStatement(S_j) then
5              DEPTH := DEPTH + 1; B := Basic Block of S_j; F_i := singleNested(B);
6          else if isStatement(S_j) then
7              S_j(L) := LHS of S_j;
8              if !isPntrArrRefExp(S_j(L)) then
9                  S_j(R) := RHS of S_j; V_set := Set of variables of S_j(R);
10                 for each V_k do
11                     if FLAG == 0 and DEPTH < 1 then
12                         if S_j(L) ≠ α_(L) then
13                             HT[S_j(L)] := TRUE;
14                             if S_j(L) == V_k or isPlusAssignOp(S_j) or isMinusAssignOp(S_j)
                               then
15                                 HT[S_j(L)] := FALSE;
16                     else if FLAG == 0 and DEPTH > 0 then
17                         if S_j(L) ≠ α_(L) then
18                             if S_j(L) == V_k or isPlusAssignOp(S_j) or isMinusAssignOp(S_j)
                               then
19                                 P := Get previous statement of F_i;
20                                 if isStatement(P) then
21                                     P_(L) := LHS of P;
22                                 V1_set : Set of Variables of SRC;
23                                 for each V1_l do
24                                     if V1_l == P_(L) then
25                                         INITOR := Get the Initializer of V1_l;
26                                     if isAssignInitializer(INITOR) or V_k == P_(L) then
27                                         HT[P_(L)] = FALSE;
28             if FLAG == 1 and DEPTH < 1 then
29                 modifyScalToArr(S_j(L), S_j(R), HT);
30             else if FLAG == 1 and DEPTH > 0 then
31                 modifyScalToArr_1(S_j(L), S_j(R), HT);
```

The implementation of the transformation algorithm is different for single/nested loops. Figure 7.9 depicts the abstract syntax tree (AST) graph of nested for-loop for the test case referred in Figure 7.8(e). Algorithm 6 shows the algorithm to find the type (single/nested) of loops and creation of hash table HT to verify the valid scalar variable for conversion. There may occur two scenarios during scalar expansion:

1. If S_j is a for, then the loop is nested. Hence DEPTH is incremented, and the scalar expansion is recursively called until S_j is a statement (see Algorithm 6).

2. If S_j is a statement, then HT is created comprising scalar V_k and the boolean value TRUE/FALSE is associated. If HT$[V_k]$ is TRUE, then scalar to vector is valid. If HT$[V_k]$ is FALSE, then the conversion is not feasible.

The scalar expansion is performed on synthetic codes with different dependences. Figures 7.8(a), 7.8(c), 7.8(d) illustrates the transformation mechanism for single for-loop comprising data dependence, anti-dependence and output dependence [101] respectively. Figure 7.8(e) shows the nested loop transformation.

Algorithm 7 depicts the scalar expansion process for single/nested loop for the example code shown in Figure 7.8(e). Following is the description of the algorithm:

1. When DEPTH > 0 (nested loop) and HT$[V_k]$ is TRUE (scalar expansion is valid), then array variable PTR is formed with (V_k), w and (α_L) i and replace V_k with PTR $w[i]$.

2. When DEPTH < 1 (single loop) and HT$[V_k]$ is TRUE (scalar expansion is valid), then it is checked if (V_k) w exists in the set X (Figure 7.8(a) and Figure 7.8(d)). If it is present, then w is transformed to $w[i]$. If the variable V_k w does not exist in set X (Figure 7.8(c)) then the array variable PTR is built with (V_k) w and ($\alpha_L - 1$) $i - 1$ and replaced V_k with PTR 'w[i-1]'.

Algorithm 7: Modifying the basic block of single/nested loop : modifyScalToArr

Input: $S_{j(L)}$: LHS of S_j,
$S_{j(R)}$: RHS of S_j.
HT: Hash table with scalar variables and its boolean value
Output: Transformed scalar variable to array variable and modified basic block B

```
1  begin
2      Create an empty set X;
3      if HT[S_j(L)] == TRUE then
4          if ∃S_j(L) ∉ X then
5              insert S_j(L) to X;
6              T := Datatype of S_j(L);
7              Get and remove the variable declaration of S_j(L);
8              ARRDECL := Declaring the variable S_j(L) of type T of array size α_R;
9              Insert ARRDECL before F_i;
10         PTR := Build array variable with S_j(L) and α_L;
11         Replace S_j(L) with PTR;
12     V_k := Set of variables of S_j(R);
13     for each V_k do
14         if HT[V_k] == TRUE and DEPTH < 1 then
15             if ∃V_k ∉ X then
16                 PTR := Build array variable with V_k and (α_L - 1);
17                 Replace V_k with PTR;
18             else
19                 PTR := Build array variable with V_k and α_L;
20                 Replace V_k with PTR;
21         if HT[V_k] == TRUE and DEPTH > 0 then
22             PTR := Build array variable with V_k and α_L;
23             Replace V_k with PTR;
```

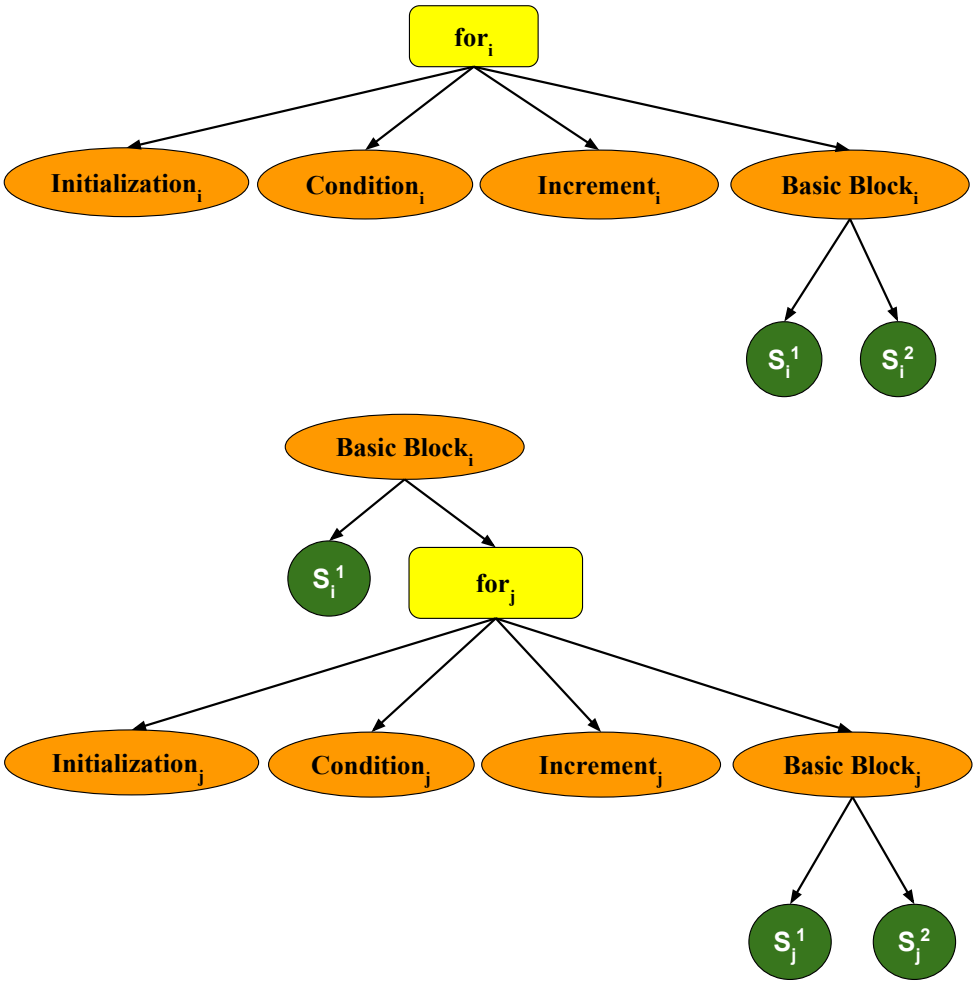


Figure 7.9: Abstract syntax tree (AST) graph of nested for*-loop*

Constraints in scalar to vector transformation This transformation is performed only if:

(i) there is no loop-carried dependence in loop. (ii) there is no function call or if-constructs within for-loop.

7.5.2 Automatic conversion of upper to lower bound

This section briefs about solving the reverse for-loop by transforming the loop bound from upper to lower bound. Figure 7.10(a) and 7.10(b) illustrates the transformation mechanism.

Algorithm 8 depicts the process before applying the transformation to the reverse for-loop. It involves two steps,

1. At first, the for-loop is checked for the presence of reversed for-loop using function *canonicalCheck* (Algorithm 9).

2. If reverse for exists, then the second step involves applying *modUpperToLower* to find the type (single/nested) of the loop to apply code transformation (Algorithm 10).

Algorithm 8: Upper to Lower Bound Conversion : upperToLower

Input: F_{set} : Set of FOR loop

```
1 begin
2     for each F_i do
3         canonicalCheck(F_i,B);
4         F_i = modUpperToLower(B);
```

Output: F_{set} : Set of FOR loop after modification of reversed FOR loops

Consider the illustration in Figure 7.10(a) used to explain the Algorithm 10 involving the process of basic block modification if there exists a reversed for. The essential part required for verification is the initialization part (`INIT`, $i = n$); condition part (α, $i > 0$); and update part (`UPD`, $i - -$). The *canonicalCheck* proves that the loop is reversed for; hence the `LHS` of (α) and `RHS` of `INIT` are stored in hashtable `HT`. A new FOR structure 'for(i = 0; i $<$ n; i++)' is built with modified `INIT`, (α), and `UPD`.

Further the basic block modification is performed by iterating variables V_k of statement S_j and is cross-validated with `HT` (see Algorithm 9). If the V_k matches with variables in `HT`, the transformation happens. For instance, the variable i is replaced as $n - i$ i.e. the variable V_k is replaced with the newly created expression EXP. Figure 7.10(b) illustrates transformation of reversed FOR loop structure in a nested innermost loop.

(a) Nested upper bound

```
for(i=n; i>0 ; i--)
 for(j=0; j<n; j++)
  A[i][j] = i+j;
```
Original Code

```
for (i=0; i<n; ++i)
 for (j=0; j<n; j++)
  A[n-i][j] = n-i+j;
```
Reformed Code

```
if (n >= 1)
{
lbp=0;
ubp=n-1;
#pragma omp parallel for
     private(lbv,ubv,t2,
    t3)
for(t1=lbp;t1<=ubp;t1++)
 for(t2=0;t2<=n-1;t2++)
  A[n - t1][t2] = n - t1
      + t2;;
}
```
Parallelization of Reformed Code

(b) Nested inner replacement

```
for(i=0; i<n ; i++)
 for(j=n; j>0; j--)
  A[i][j] = i+j;
```
Original Code

```
for(i=0; i<n; i++)
 for (j=0; j<n; ++j)
  A[i][n-j] = i + (n
     -j);
```
Reformed Code

```
if (n >= 1)
{
lbp=0;
ubp=n-1;
#pragma omp parallel for
     private(lbv,ubv,t2,
    t3)
for(t1 = lbp;t1 <= ubp;
    t1++)
 for(t2=0;t2<=n-1;t2++)
  A[t1][n-t2] = t1 + (n-
      t2);;
}
```
Parallelization of Reformed Code

Figure 7.10: Example illustration for automatic conversion of upper to lower bound (a) Nested upper bound (b) Nested inner replacement

Algorithm 9: Checking for Reversed for : canonicalCheck

Input: F_i : FOR loop

begin

B : Basic Block of F_i;

α, α_L, α_R := NULL;

blockInfo(F_i,B,α_L,α_R);

INIT := Initialization part of F_i;

$INIT_R$:= RHS of INIT;

UPD := Updation part of F_i ;

V_{set} := Set of Variables of $INIT_R$;

Create a Hash Table HT[string,string];

for *each* V_j **do**

  Var := Search V_j in HT;

  if $\exists$*(Var)* $\in$ *HT* **then**

    Exp := Build subtract expression with Var→first and Var→second;

    replace V_j with Exp;

V_{set} := Set of Variable of α_R;

for *each* V_k **do**

  Var := Search V_k in HT;

  if $\exists$*(Var)* $\in$ *HT* **then**

    Exp := Build subtract expression with Var→first and Var→second;

    replace V_k with Exp;

/* isGreaterThanOp returns TRUE if α contains greater than operator */

/* isGreaterOrEqualOp returns TRUE if α contains greaterorequal operator */

/* isMinusMinusOp returns TRUE if α contains decrement operator */

if *isGreaterThanOp(α) or isGreaterOrEqualOp(α) and isMinusMinusOp(UPD)* **then**

  HT[α_L] = $INIT_R$;

  α := Build conditional statement with less than operator using α and $INIT_R$;

  UPD := Build PlusPlusOp updation statement using α;

  INIT := BUILD INIT statement using α_L and α_R;

  B := Get basic block from F_i;

  Build New FOR loop using INIT, α, UPD and B;

Output: F_i : Modified FOR initialization, condition and updation

Algorithm 10: Modifying the basic block of Single/Nested loop: modUpperToLower

```
   Input: B : Basic Block of F_i
1  begin
2      S_set : Set of Statements of B;
3      for each S_j do
4          if isForStatement(S_j) then
5              F_i ++;
6              canonicalCheck(F_i,B);
7              F_i := modUpperToLower(B);
8          else if isStatement(S_j) then
9              V_set := Set of Variables of S_j;
10             for each V_k do
11                 Var := Search V_k in HT;
12                 if ∃(Var) ∈ HT then
13                     Exp := Build subtract expression with Var→first and Var→second;
14                     replace V_k with Exp;
15     return F_i;
   Output: F_i : Modified FOR loop feasible for parallelization
```

Constraints in conversion of upper to lower bound This transformation is performed only if:

(i) the loop is a canonical loop (ii) inner loop index variable is not dependant on the outer loop index variable in case of nested loop. (iii) there is no if construct and function call.

```
1 for (i=0; i<n; i+=2) {
2 a[i] = i;
3 b[i] = i + 2;
4 c[i] = a[i] + b[i];
5 }
```

Original Code

```
1 for (i=0; i<n; ++i) {
2 a[i] = i;
3 b[i] = i + 2;
4 c[i] = a[i] + b[i];
5 i += 1;
6 }
```

Transformed Code

Figure 7.11: Converting updation by time step one

7.5.3 Automatic conversion of loop increment with step size one

Polyhedral transformation requires the update part `UPD` in the for to be incremented by step size one. Hence if step size > 1, code transformation is applied to make it to unity. The transformation are confined to for-loop increment/decrement viz. i=i+n, i=i-n, i+=n, and i-=n.

Let us consider the example in Figure 7.11 that shows `UPD` i+=2 violates the above rule. The procedure in Algorithm 11 and 12 illustrates the transformation process for such a scenario. The `UPD` is obtained, and its `RHS` contains `VALUE` 2. Hence, if `VALUE` > 1, then it is decremented by one and expression (`EXP`, $i+ = 1$) is created, and the `UPD` is replaced with `EXP`. Now, there is a small modification to the end of the basic block. The decremented value is then used to form expression `NEWEXP` with `LHS` of `UPD`, i.e., $i+ = 1$. The new expression is appended next to the last statement `STMT` of the basic block `B`. The transformed code in Figure 7.11 is still not feasible for parallelization since it forms an irregular loop.

Algorithm 11: Converting the increment step to one : StepToOne

Input: F_{set} : Set of for-loop
Output: F_{set}: Modified for-loop with increment step size as unity

```
1  begin
2      for each F_i do
3          blockInfo(F_i,B,α_L,α_R);
4          UPD := Updation part of F_i;
           /* isPlusPlusOp returns TRUE if UPD contains increment
           operator                                                  */
           /* isMinusMinusOp returns TRUE if UPD contains
           minusminus operator                                       */
5          if isPlusPlusOp(UPD) or isMinusMinusOp(UPD) then
6              UPD_L := Get Operand of UPD;
7          else if isBinaryOp(UPD) then
8              UPD_L := Get LHS Operand of UPD;
9          if α_L == UPD_L then
10             if ((isGreaterThanOp(α) or isGreaterOrEqualOp(α) and isBinaryOp(UPD))
                then
11                 if isMinusAssignOp(UPD) or isAssignOp(UPD) then
12                     VALUE := RHS of UPD;
13                     if VALUE > 1 then
14                         EXP := Build expression with α_L;
15                         modifyIncr(EXP,UPD);
16             else if isPlusAssignOp(UPD) or isAssignOp(UPD) then
17                 Execute line 12 to 15;
```

Algorithm 12: Modify increment part : modifyIncr

Input: EXP : Expression built for replacement in UPD,
UPD : Updation part of F_i

```
1  begin
2      VALUE1 = VALUE - 1;
3      B : = Basic block of F_i;
4      SCOPE := Scope of B;
5      STMT := Last statement of SCOPE;
6      if isMinusAssignOp(UPD) or isSubtractOp(UPD) then
7          NEWEXP := Build Expression with α_L and VALUE1;
8      if isPlusAssignOp(UPD) or isAddOp(UPD) then
9          NEWEXP := Build Expression with α_L and VALUE1;
10     replace the expression of UPD with NEWEXP;
11     Insert Statement NEWEXP after STMT;
```

Output: F_{out} : Reformed Output FOR block

7.5.4 Automatic conversion of irregular loop to regular loop

Figure 7.11 in the previous section portrays the transformation of the update part UPD, where the step size was more than one. The reformed code is considered as an irregular loop and not amenable to parallelization since the updation of index variable occurs within the basic block of for-loop.

In this section, irregular loop refers to the code fragments comprising incrementing/Decrementing of loop index variable such as i++, i–, i=i+n, i=i-n, i+=n, and i-=n with the for-loop. The irregular loop transformation are confined to the aforementioned statements. This example has i+=1 within for-loop which is unsupported by Pluto. Algorithm 13 addresses the irregular loop and the transformation is shown in Figure 7.12. The statements S_j within for-loop F_j is iterated to detect if UPD exists. The value of UPD is obtained and the COUNT is incremented by 1 and the statement S_j is removed. The iteration of S_j continues until it reaches the end of the basic block and the UPD value is obtained. The COUNT is incremented/decremented and the final value is stored in variable Y. The total update part is calculated in Y1. An expression EXP is created using the value Y and Y1 and the variable i is replaced with the same.

Algorithm 13: Solving Irregular Loop : solveIrregularLoop

```
   Input: SRC : Source Code,
          F_set : Set of FOR loops
1  begin
2      α, B, α_L, α_R = NULL;
3      FLAG := 1;
4      for each F_i do
5          blockInfo(F_i,B,α_L,α_R);
6          S_set := Set of Statements of F_i;
7          for each S_j do
8              if isPlusPlusOp(S_j) or isMinusMinusOp(S_j) or isBinaryOp(S_j) then
9                  OP_L := LHS operand of S_j;
10                 if α_L == OP_L then
11                     Calculate COUNT;
12                     remove S_j;
13                     continue;
14             FLAG := 0;
15             if isBinaryOp(S_j) and FLAG == 0 then
16                 X := COUNT;
17                 NEXT := Next statement of S_j;
                   /* Y1 : Calculation of total updation part where
                   α_L is equal to OP_L                              */
18                 repeat
19                     if isPlusPlusOp(S_j) or isMinusMinusOp(S_j) or isBinaryOp(S_j) then
20                         Calculate Y;
21                         NEXT1 := Next statement of NEXT;
22                     else
23                         NEXT1 := Next statement of NEXT;
24                 until NEXT != NULL;
25                 Y1 := Y + X;
26                 if Y1 != 0 or X != 0 then
27                     if α_L != OP_L then
28                         EXP := Build new expression to replace all α_L;
29                         V_set := Set of Variables of S_j;
30                         for each V_k do
31                             if α_L == V_k then
32                                 Replace V_k with EXP;
33             else if FLAG == 0 then
34                 Y2 = Y1 - COUNT;
35                 Execute line 27 to 32;
```

```
for (i=0; i<n; ++i)
{
a[i] = i;
b[i] = i + 2;
c[i] = a[i] + b[i];
i += 1;
}
```

Original Code*

```
int _zB = 0;
if (n % (1 + 1)==0)
 _zB = n / (1 + 1);
if (n % (1 + 1)!=0)
 _zB = n/(1 + 1)+1;
for(i = 0; i < _zB; ++i)
{
 a[(i+1) * 0 + i * (1+1)] = (i+1) * 0 + i * (1+1);
 b[(i + 1) * 0 + i * (1 + 1)] = (i + 1) * 0 + i * (1 + 1) + 2;
 c[(i + 1) * 0 + i * (1 + 1)] = a[(i + 1) * 0 + i * (1 + 1)] + b[(i + 1) * 0 + i * (1 + 1)];
}
```

Reformed Code

```
int _zB = 0;
if (n % (1 + 1) == 0)
 _zB = n / (1 + 1);
if (n % (1 + 1) != 0)
 _zB = n / (1 + 1) + 1;
if (_zB >= 1)
{
lbp=0;
ubp=_zB-1;
#pragma omp parallel for private(lbv,ubv,t2,t3)
for(t1 = lbp;t1 <= ubp;t1++)
 {
 a[(t1+1) * 0 + t1 * (1+1)] = (t1+1) * 0 + t1 * (1+1);;
 b[(t1+1) * 0 + t1 * (1+1)] = (t1+1) * 0 + t1 * (1+1) + 2;;
 c[(t1+1) * 0 + t1 * (1+1)] = a[(t1+1) * 0 + t1 * (1+1)] + b[(t1+1) * 0 + t1 * (1+1)];;
 }
}
```

Parallelization of Reformed Code

*Irregular loop is incrementing/decrementing loop variable inside for-loop

Figure 7.12: Example illustration for automatic conversion of irregular loop to regular loop

7.6 Application exploration

Three application programs of PolyBench suite namely, `symm`, `ludcmp`, and `adi` were transformed using solution-focused automatic parallelization mechanism. In this chapter, the analysis is restricted to three PolyBench codes which comprise specific problems discussed. Pluto was not effective in parallelizing these codes. The key aspect is to analyze the performance improvement after applying the transformation.

The *Profiling* phase identifies that all three benchmarks are compute-intensive. There exists no while-loop in the codes. Hence, the codes are sent to analysis and code transformation phase-2 (*ACT-2*), and found that `symm` and `ludcmp` have scalar in the ℓ-value of statement. It was then subjected to scalar expansion. The benchmarks `ludcmp` and `adi` comprise reverse for-loop, and the transformation is applied to convert them into normal for-loops. After the code is modified, it became amenable to parallelization. The modified code is then subjected to Pluto to obtain the parallelized version. The semantic equivalence of the code is verified by testing their output with the serial output.

The experimental configuration is described in Chapter 6. The parallelized codes are executed for problem size N=1.0E4 on different number of cores (16, 32, and 64).

7.6.1 Result analysis

This section illustrates the performance of Pluto_A on the three PolyBench benchmarks (`symm`, `ludcmp`, and `adi`). Pluto_A is the parallelized code after applying to the solution-focused automatic parallelization mechanism. Pluto parallelizer did not parallelize the codes, hence we exclude it from the speedup graph. This section illustrates the automatic code transformation applied to individual benchmarks and its performance impact. Figure 7.13 illustrates the transformation applied to `symm`. Here the scalar expansion is performed on scalar $temp2$ and replaced as vector $temp2[i]$. Usage of $temp2[i]$ over $temp2[j]$ reduces latency due to locality of reference i.e. temporal locality. The transformed code was then amenable to parallelization. Figure 7.16(a) depicts the parallelization of `symm` by Pluto that showed better performance (with 32 threads: 7.1×).

```
for (i = 0; i < _PB_M; i++)
{
 for (j = 0; j < _PB_N; j++ )
 {
 temp2 = 0;
  for (k = 0; k < i; k++)
  {
  C[k][j]+=alpha*B[i][j]*A[i][k];
  temp2 += B[k][j] * A[i][k];
  }
 C[i][j] = beta * C[i][j] + alpha
     *B[i][j] * A[i][i] + alpha *
     temp2;
 }
}
```

Original Code

```
for (i = 0; i < m; i++)
{
 for (j = 0; j < n; j++)
 {
 temp2[i] = 0;
  for (k = 0; k < i; k++)
  {
  C[k][j]+=alpha*B[i][j]*A[i][k];
  temp2[i] += B[k][j] * A[i][k];
  }
 C[i][j] = beta * C[i][j] + alpha
      * B[i][j] * A[i][i] + alpha
     * temp2[i];
 }
}
```

Reformed Code

Figure 7.13: Automatic code transformation applied to `symm` benchmark

The benchmark `ludcmp` failed in parallelizing the loop due to two problems namely (i) scalar variable in the ℓ-value (ii) reverse for-loop. Figure 7.14 illustrates the transformations applied to `ludcmp`. The *ACT-2* converted the scalar variable w to an array variable $w[i]$ and replaced the loop bound from lower to upper. Automatic transformation of ludcmp benchmark leads to automatic parallelization with Pluto. The experimental result in Figure 7.16(b) shows that `ludcmp` showed good speedup using Pluto (speedup with 32 threads: 4.6×).

Figure 7.15 depicts the code transformation of `adi`. It was found that the benchmark failed to apply parallelization due to the presence of reverse for-loop $for(j = \_PB\_N - 2; j >= 1; j--)$. After applying code transformation, `adi` showed lesser speedup due to additional optimization (Figure 7.16(c)).

```
for (i = 0; i < _PB_N; i++)
{
 for (j = 0; j <i; j++)
 {
 w = A[i][j];
  for (k = 0; k < j; k++)
  {
  w -= A[i][k] * A[k][j];
  }
 A[i][j] = w / A[j][j];
 }
 for (j = i; j < _PB_N; j++)
 {
 w = A[i][j];
  for (k = 0; k < i; k++)
  {
  w -= A[i][k] * A[k][j];
  }
 A[i][j] = w;
 }
}
for (i = 0; i < _PB_N; i++)
{
w = b[i];
 for (j = 0; j < i; j++)
 w -= A[i][j] * y[j];
y[i] = w;
}
for (i = _PB_N-1; i >=0; i--)
{
w = y[i];
 for (j = i+1; j < _PB_N; j++)
 w -= A[i][j] * x[j];
x[i] = w / A[i][i];
}
```

Original Code

```
for (i = 0; i < n; i++)
{
 for (j = 0; j < i; j++)
 {
 w[i] = A[i][j];
  for (k = 0; k < j; k++)
  {
  w[i] -= A[i][k] * A[k][j];
  }
 A[i][j] = w[i] / A[j][j];
 }
 for (j = i; j < n; j++)
 {
 w[i] = A[i][j];
  for (k = 0; k < i; k++)
  {
  w[i] -= A[i][k] * A[k][j];
  }
 A[i][j] = w[i];
 }
}
for (i = 0; i < n; i++)
{
w[i] = b[i];
 for (j = 0; j < i; j++)
 {
 w[i] -= A[i][j] * y[j];
 }
y[i] = w[i];
}
for (i = 0; i <= n - 1; ++i)
{
w[n-1-i] = y[n-1-i];
 for (j = n-1-i+1; j < n; j++)
 {
 w[n-1-i] -= A[n-1-i][j] * x[j];
 }
x[n-1-i] = w[n-1-i] / A[n-1-i][n-
    1-i];
}
```

Reformed Code

Figure 7.14: Automatic code transformation applied to `ludcmp` benchmark

```
for (t=1; t<=_PB_TSTEPS; t++)
{
 for (i=1; i<_PB_N-1; i++)
 {
 v[0][i] = SCALAR_VAL(1.0);
 p[i][0] = SCALAR_VAL(0.0);
 q[i][0] = v[0][i];
  for (j=1; j<_PB_N-1; j++)
  {
  p[i][j] = -c / (a*p[i][j-1]+b);
  q[i][j]=(-d*u[j][i-1]+(
      SCALAR_VAL(1.0)+SCALAR_VAL
      (2.0)*d)*u[j][i] - f*u[j][i+
      1]-a*q[i][j-1])/(a*p[i][j-1]
      +b);
  }
 v[_PB_N-1][i] = SCALAR_VAL(1.0);
  for (j=_PB_N-2; j>=1; j--)
  {
  v[j][i] = p[i][j] * v[j+1][i] +
       q[i][j];
  }
 }
 for (i=1; i<_PB_N-1; i++)
 {
 u[i][0] = SCALAR_VAL(1.0);
 p[i][0] = SCALAR_VAL(0.0);
 q[i][0] = u[i][0];
  for (j=1; j<_PB_N-1; j++)
  {
  p[i][j] = -f / (d*p[i][j-1]+e);
  q[i][j]=(-a*v[i-1][j]+(
      SCALAR_VAL(1.0)+SCALAR_VAL
      (2.0)*a)*v[i][j] - c*v[i+1][
      j]-d*q[i][j-1])/(d*p[i][j-1]
      +e);
  }
 u[i][_PB_N-1] = SCALAR_VAL(1.0);
  for (j=_PB_N-2; j>=1; j--)
  {
  u[i][j] = p[i][j] * u[i][j+1] +
       q[i][j];
  }
 }
}
```

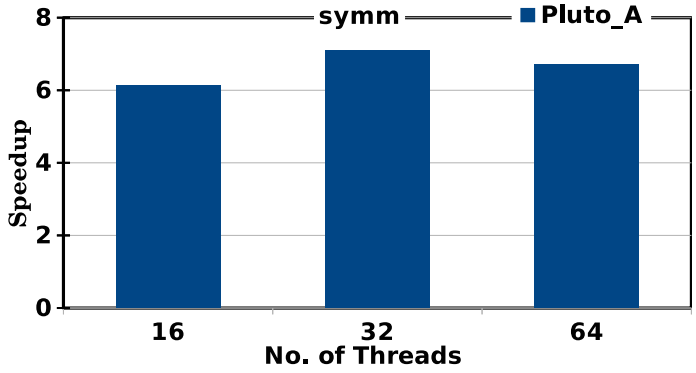
Original Code

```
for (t=1; t<=_PB_TSTEPS; t++)
{
 for (i=1; i<_PB_N-1; i++)
 {
 v[0][i] = SCALAR_VAL(1.0);
 p[i][0] = SCALAR_VAL(0.0);
 q[i][0] = v[0][i];
  for (j=1; j<_PB_N-1; j++)
  {
  p[i][j] = -c / (a*p[i][j-1]+b);
  q[i][j] = (-d*u[j][i-1]+(
      SCALAR_VAL(1.0)+SCALAR_VAL
      (2.0)*d)*u[j][i] - f*u[j][i+
      1]-a*q[i][j-1])/(a*p[i][j-1]
      +b);
  }
 v[_PB_N-1][i] = SCALAR_VAL(1.0);
  for (j = 0; j < _PB_N - 2; ++j)
  {
  v[_PB_N - 2 - j][i] = p[i][
      _PB_N - 2 - j] * v[_PB_N - 2
       - j + 1][i] + q[i][_PB_N -
      2 - j];
  }
 }
 for (i=1; i<_PB_N-1; i++)
 {
 u[i][0] = SCALAR_VAL(1.0);
 p[i][0] = SCALAR_VAL(0.0);
 q[i][0] = u[i][0];
  for (j=1; j<_PB_N-1; j++)
  {
  p[i][j] = -f / (d*p[i][j-1]+e);
  q[i][j] = (-a*v[i-1][j]+(
      SCALAR_VAL(1.0)+SCALAR_VAL
      (2.0)*a)*v[i][j] - c*v[i+1][
      j]-d*q[i][j-1])/(d*p[i][j-1]
      +e);
  }
 u[i][_PB_N-1] = SCALAR_VAL(1.0);
  for (j = 0; j < _PB_N - 2; ++j)
  {
  u[i][_PB_N - 2 - j] = p[i][
      _PB_N - 2 - j] * u[i][_PB_N
      - 2 - j + 1] + q[i][_PB_N -
      2 - j];
  }
 }
}
```

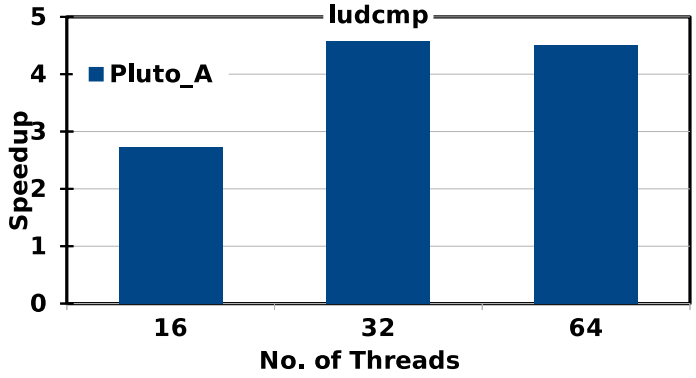
Reformed Code

Figure 7.15: Automatic code transformation applied to `adi` benchmark

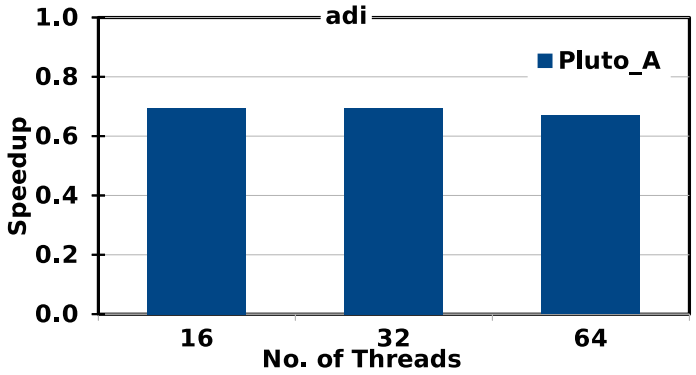
(a) symm



(b) ludcmp



(c) adi



Pluto parallelizer did not parallelize the codes, hence we exclude it from the speedup graph. Pluto_A is the parallelized code after applying to the solution-focused automatic parallelization mechanism

Figure 7.16: Parallelization speedup of PolyBench benchmarks after applying to solution-focused automatic parallelization mechanism (a) `symm` *(b)* `ludcmp` *(c)* `adi`

7.7 Summary

Totally five pitfalls were identified and addressed in this chapter. An automated solution-focused mechanism was developed comprising three phases namely, *Profiling*, *Analysis and Code Transformation* (ACT), and *Parallelization*. *Profiling* was performed using Gprof. The code transformation was performed to remove the Pluto problems using AST query mechanism with the aid of Rose compiler framework. After that the transformed code was amenable to parallelization by Pluto tool.

Three PolyBench benchmarks (`symm`, `ludcmp`, and `adi`) were used for the analysis, and the investigation shows that applying the benchmarks directly to Pluto did not parallelize these codes due to irregular coding patterns. However, after subjecting to solution-focused automatic parallelization mechanism (Pluto_A), the codes were successfully parallelized using Pluto. The automatic method (Pluto_A) was tested using PolyBench benchmarks, and it was observed that the transformation was successful. The result analysis has proven better speedup (`symm` 7.1×), (`ludcmp` 5.6 ×) adding merits to the proposed approach. Partially this transformation helps the real-world problem. However, the remaining non-affine issues have to be addressed manually to make the code amenable to parallelization. The future work can focus on performing the solution-focused mechanism for many trivial non-affine issues to make utilization of Pluto's powerful polyhedral and additional optimization.

CHAPTER 8

Conclusion and Scope for Future Investigations

The present chapter summarizes the research work carried out, qualitative analysis, empirical study, methods developed, conclusions arrived at, and future work that can be pursued.

8.1 Summary and conclusion

Automatic parallelization has been the topic of research for several decades. Although, manual parallelization have always been useful and usually had performed better. Researchers in the present era focus on developing a fully automated tools. The key contribution of automatic parallelization aims to enhance the hand-optimized code. Earlier studies on parallelizing compilers have shown several merits and demerits in achieving the ideal speedup. Some of the cons were related to parallelization issues such as dependences across iterations. Apart from dependence issues, parallelizers does not support codes with complex coding style. Studies on modern auto-parallelizers, their capabilities and limitations have been sorely missing. This thesis has focused on alleviating the pitfalls faced by widely-used popular parallelizers and had brought out interesting findings.

8.1.1 Conclusion derived from qualitative capabilities of auto-parallelizers

The modern auto-parallelizing frameworks namely, Cetus, Par4all, Rose, Intel C compiler (ICC), and Pluto and their methodologies were critically analyzed in Chapter 2. A qualitative analysis of the five frameworks was carried out in Chapter 3. The study revealed the following capabilities of auto-parallelizers:

1. All the frameworks inherently support loop parallelization but differ in the techniques they perform transformation in the parallelizable region.

2. Cetus performs a profitability test that eliminates parallelization of insignificant loops. This compensates for the thread-creation overhead. Similarly, ICC reported few of the non-parallelized loops having insufficient computational work. Hence, Cetus and ICC forbid parallelization of smaller loops which is a positive aspect.

The effect of loop transformation techniques employed by various tools on different types of dependences were examined using synthetic codes in Chapter 3, and the key findings are:

1. All frameworks under study parallelized loops with *loop-independent dependence* by OpenMP private clause.

2. Pluto does not parallelize loop with *loop-carried dependence due to scalar*. This is because Pluto is a polyhedral parallelizer that supports the transformation of array accesses. All other frameworks parallelize such cases using reduction clause.

3. Only Pluto and ICC perform parallelization on loops with *loop-carried dependence due to vector*. Parallelization of code with inter-iteration dependence is supported by Pluto with the aid of loop peeling, and ICC with the help of loop peeling or loop fission.

Besides dependence issues, auto-parallelizers forbid parallelization of loops with complex coding style. Different programming constructs (in total 76) were tested, and the support matrix of these features by individual framework was evaluated (acceptance ratio before applying solutions in %: Cetus 63.2, Par4all 67.1, Rose 26.3, Parallware 72.4, ICC 71.1, and Pluto 65.8). Based on the solutions, through minimal coding changes, the codes

were acceptable for parallelization. It was found that the parallelizers effectiveness was improved on the whole by 10% (acceptance ratio after applying solutions in %: Cetus 76.3, Par4all 78.9, Rose 39.5, Parallware 82.9, ICC 80.3, and Pluto 77.6). Overall, commercial compilers viz. Parallware and ICC have the maximum acceptance ratio. This study showed that the effectiveness of the open-source tools namely, Cetus, Par4all, Rose, and Pluto could be improved by applying manual coding changes.

8.1.2 Conclusion derived from quantitative analysis of auto-parallelizers using PolyBench benchmarks

Empirical analysis of different frameworks using PolyBench suite was emphasized in Chapter 4. The benchmarks were categorized based on the presence and absence of loop-carried dependence. Following are the observed results after performance analysis:

1. All five frameworks parallelize benchmarks with *loop-independent dependence*. In addition, these tools support *loop-carried dependence due to vector* occurring in either inner or outer loop of nested for. Except Pluto, all other tools parallelize benchmarks with *loop-carried dependence due to scalar*. The reason for Pluto's poor support for scalar variables is because it is primarily targeted towards polyhedral transformations of array accesses, rather than individual shared items. However, Pluto parallelized benchmarks with complex *loop-carried dependence due to vector*.

2. In terms of speedup, Pluto and ICC outperform all other frameworks (Pluto average best speedup with 32 threads: 10.9×) (ICC average best speedup with 32 threads: 19.2×). This is because ICC performs privatization and vectorization, and Pluto performs polyhedral optimization besides parallelization.

3. Par4all and Rose perform better for most of the cases due to array privatization (Par4all average best speedup with 32 threads: 10.2 ×) (Rose average best speedup with 32 threads: 9.4 ×).

4. All 28 codes which are part of PolyBench suite could be parallelized at least by any one of the frameworks.

8.1.3 Conclusion derived from quantitative analysis of auto-parallelizers using NAS parallel benchmarks (NPB)

Quantitative analysis of various frameworks using NAS parallel benchmarks (NPB) was analyzed, and the observations were reported in Chapter 5. NPB benchmarks were less amenable to parallelization by auto-parallelizers. Auto-parallelizers faced pre- and post-transformation difficulties during parallelization of NPB codes which was fixed by manual intervention. When examining the speedup, ICC outperforms all other frameworks due to its inherent optimization (with 32 threads: `BT` 3.4×, `EP` 2.0×, `FT` 2.6×, `LU` 4.6×, `MG` 5.5×, and `SP` 5.8×). However, poorer speedup was noted for all other tools. The reasons for such behavior was dug out by comparing the parcount (number of loops parallelized) of Original (manually parallelized NPB) with parcount of all other frameworks. Although the parcount for several of the benchmarks were approximately equal, the speedup was observed to be less. Further observations were made by examining the parallelized loops that could explain the following findings.

1. Par4all, Rose, and Pluto perform inefficient parallelization and parallelize insignificant loops which cause overhead in execution.

2. Parallelizers lag in the usage of implicit barriers for multiple, independent and consecutive loops which reported execution overhead problems.

3. The auto-parallelizers did not handle Imperfectly or complex nested loops. Parallelizers namely, Par4all, Rose, and Pluto missed outer loop parallelism.

4. There were several non-affine issues observed by Pluto such as non-affine loop bound (NAL), non-affine subscript (NAS), if construct (IF) and general trivial parallelization

limitations such as the use of function call (FC), switch-case, recursive function, that became the non-parallelizable part for auto-parallelizers.

8.1.4 Overall conclusions derived from qualitative and quantitative analyses of auto-parallelizers

The qualitative and quantitative studies in Chapters 3, 4, and 5 emphasized that ICC and Pluto outperformed all other frameworks. However, the examined results over real-world problems such as NPB codes elucidated the importance of handling complex coding to improve the effectiveness of tools. The observations revealed that Pluto works well on affine codes but lag in support of non-affine codes and several trivial limitations. **As ICC is a closed-source compiler, the thesis focused towards enhancing the efficacy of Pluto tool by two distinct methods.**

8.1.5 Conclusions derived from elimination of auto-parallelization issues in irregular and general-purpose programs

The first new approach was a manual elimination method which acts as a pre-parser and post-parser to Pluto. Following illustrates the key points of this approach: Three real-world benchmarks suites were explored namely, Green-Marl, Rodinia, and NPB. In total, nine non-affine constructs (NAC) including non-affine issues and general trivial issues were identified in Chapter **??**. The non-affine constructs were removed using the elimination method, and the codes were amenable to parallelization. This approach has three different stages namely, pre-, in-, and post-elimination. The parallelized code was validated for the semantics of the code. Elimination of non-affine constructs (NAC) helped the Pluto parallelizer in parallelizing most of the Green-Marl and Rodinia benchmarks with good amount of speedup improvement (best speedup results of Green-Marl with 32 threads: `adamicAdar` 11.58×, `avg_teen_cnt` 9.54×, `communities` 1.57×, `pagerank` 9.07×, and `sssp path` 1.05×) (best speedup results of Rodinia with 32

threads: `heartwall` 10.27×; `particlefilter` 1.11×; and `kmeans` 48.61×). However, NPB was not benefited using this approach due to its inherent complexity which requires several optimizations and explicit parallelism.

8.1.6 Conclusions derived from solution-focused auto-parallelization mechanism of sequential codes

The second new method was an automated solution-focused mechanism (discussed in Chapter 7) to alleviate a few of the problems faced by Pluto. In total, five pitfalls are explored, and automated solutions were derived. The proposed method involves three processes namely, *Profiling*, *Analysis and Code Transformation* (ACT), and *Parallelization*. Profiling benefited in filtering the compute-intensive functions for transformation. AST query mechanism using Rose compiler framework helped in achieving code transformation. Finally, the transformed code is amenable to parallelization. The correctness of the code was assured by comparing the parallel output with the sequential output. Performance improvement was observed on `symm` (with 32 threads: 7.1×) and `ludcmp` (with 32 threads: 4.6×) benchmarks.

8.1.7 New techniques to enhance auto-parallelization of open-source tool: Pluto

Auto-parallelization of real-world problems using polyhedral parallelizer, Pluto revealed several non-affine constructs (NAC) that forbid parallelization. Following are the key points of the two techniques $T1$ (discussed in Chapter 6) and $T2$ (disucssed in Chapter 7) introduced to improve the Pluto parallelization by alleviating several of its limitations (two approaches are depicted in Figure 8.1).

1. T1: A method was introduced to eliminate the NAC in Chapter 6. The elimination is performed *before* and *after* applying Pluto transformation (the method works as a pre-parser and post-parser to Pluto tool). It was found that 67% of the codes were feasible for parallelization after the removal of NAC and achieved reasonable speedup

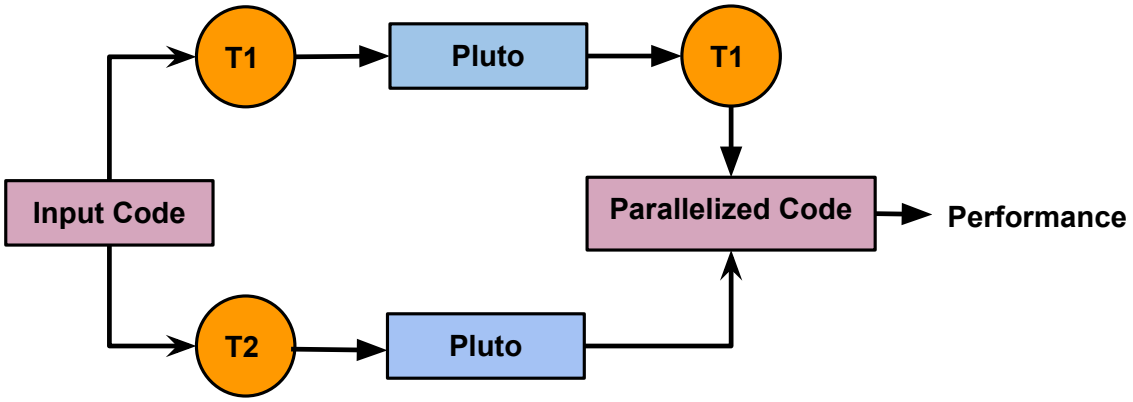


Figure 8.1: Two new approaches to enhance auto-parallelization of Pluto

for several of the benchmarks.

2. T2: A solution-focused mechanism is developed to improve the efficacy of Pluto in Chapter 7. This automatic method performs code transformation to remove the pitfalls faced by Pluto. It was observed that the applications were amenable to parallelization after applying this automatic method and have shown significant performance improvement.

The pitfalls resolved by techniques $T1$ and $T2$ are distinct except Pluto's scalar issue. Hence, the two approaches proposed in Chapter 6 and Chapter 7 can be combined to deliver more effective result.

8.2 Scope for the future work

1. Cetus performs a profitability test that helps in avoiding the execution of loops with smaller iterations which eventually eliminates the thread creation cost. Par4all in many instances (observed in Chapter 4) inserts parallel directives at appropriate places which had lead to better speedup. Pluto's ability to parallelize codes comprising loop-carried dependence using optimization techniques has shown a major benefit. Hence, further work can focus towards building a meta auto-parallelizer that provides combined benefits of these frameworks.

2. Although NAS parallel benchmarks (NPB) do not work well with implicit parallelism, they give a clear understanding that explicit parallelism plays a significant role and is highly necessary for performance improvement of real-world problems (discussed in Chapter 5). Hence, a profile-driven approach with manual intervention has to be developed to overcome such predicament.

3. NPB codes are more complex comprising multiple, independent, and consecutive loops. Parallelizing these loops requires a high-level understanding of the usage of implicit barriers and various synchronization constructs. Auto-parallelizers can be incorporated with a sophisticated provision to deal with more real-world problems like NPB.

4. The elimination of non-affine constructs described in Chapter 6 is a manual method. The results of Green-Marl and Rodinia benchmarks showed considerable benefits which pave the way for automation of this approach.

5. With the aim of making full-fledged automation, some more solutions have to be worked out for addressing the problems like switch-case, recursive functions, non-affine array subscripts and much more. Pluto's optimization created several unused temporary variables which are assumed to incur some cost. Necessary steps should be taken to remove these variables. Yet another way to take care of complex syntactic constructs is to work at the intermediate representation (IR) level (similar to ICC) or even at the assembly code level.

REFERENCES

[1] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/MC.2008.209 [*Cited on pages 1 and 31.*]

[2] S. P. Midkiff, *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2012. [Online]. Available: http://dx.doi.org/10.2200/S00340ED1V01Y201201CAC019 [*Cited on pages 1 and 3.*]

[3] W.-T. Lin and C.-P. Chu, "A fast and parallel algorithm for frequent pattern mining from big data in many-task environments," *International Journal of High Performance Computing and Networking*, vol. 10, no. 3, pp. 157–167, 2017. [*Cited on page 1.*]

[4] K. S. Hasan, J. K. Antonio, and S. Radhakrishnan, "A model-driven approach for predicting and analysing the execution efficiency of multi-core processing," *Int. J. Comput. Sci. Eng.*, vol. 14, no. 2, pp. 105–125, Jan. 2017. [Online]. Available: https://doi.org/10.1504/IJCSE.2017.082877 [*Cited on page 1.*]

[5] R. Eigenmann, J. Hoeflinger, and D. Padua, "On the automatic parallelization of the perfect benchmarks(r)," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 1, pp. 5–23, Jan 1998. [*Cited on pages 3 and 10.*]

[6] W. Schulte and N. Tillmann, "Automatic parallelization of programming languages: Past, present and future," in *Proceedings of the 3rd International Workshop on Multicore Software Engineering*, ser. IWMSE '10. New York, NY, USA: ACM, 2010, pp. 1–1. [Online]. Available: http://doi.acm.org/10.1145/1808954.1808956 [*Cited on page 3.*]

[7] C. Dave, H. Bae, S. J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, pp. 36–42, Dec 2009. [*Cited on pages 3, 5, 14, 15, 16, and 62.*]

[8] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. Mcmahon, F.-X. Pasquier, G. Péan, and P. Villalon, "Par4All: From

Convex Array Regions to Heterogeneous Computing," in *IMPACT 2012 : Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, Paris, France, Jan. 2012, 2 pages. [Online]. Available: https://hal-mines-paristech.archives-ouvertes.fr/hal-00744733 [*Cited on pages 3, 5, 14, and 17.*]

[9] D. Quinlan and C. Liao, "The rose source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, 2011, p. 1. [*Cited on pages 3, 5, 14, 17, 139, and 140.*]

[10] W. Blume and R. Eigenmann, "Performance analysis of parallelizing compilers on the perfect benchmarks programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 6, pp. 643–656, Nov 1992. [*Cited on pages 4 and 9.*]

[11] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. [*Cited on pages 4, 32, and 139.*]

[12] R. Eigenmann and W. Blume, "An effectiveness study of parallelizing compiler," in *Proceedings 20th International Conference Parallel Processing 1991*, vol. 2. CRC Press, 1991, p. 17. [*Cited on pages 4 and 9.*]

[13] D. Hisley, G. Agrawal, and L. L. Pollock, "Evaluating the effectiveness of a parallelizing compiler," in *Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, ser. LCR '98. London, UK, UK: Springer-Verlag, 1998, pp. 195–204. [Online]. Available: http://dl.acm.org/citation.cfm?id=648048.745865 [*Cited on pages 4 and 10.*]

[14] H. Nobayashi and C. Eoyang, "A comparison study of automatically vectorizing fortran compilers," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (Supercomputing '89)*, Nov 1989, pp. 820–825. [*Cited on pages 4 and 9.*]

[15] Z. Shen, Z. Li, and P. C. Yew, "An empirical study of fortran programs for parallelizing compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 356–364, Jul 1990. [*Cited on pages 4 and 9.*]

[16] T. Imai, "Detecting more independent loops across hierarchical structures," in *Proceedings ICCI '92: Fourth International Conference on Computing and Information*, May 1992, pp. 168–172. [*Cited on pages 4 and 11.*]

[17] C. Dave and R. Eigenmann, "Automatically tuning parallel and parallelized programs," in *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 126–139. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13374-9_9 [*Cited on pages 4 and 12.*]

[18] D. Mustafa, A. Aurangzeb, and R. Eigenmann, "Performance analysis and tuning of automatically parallelized openmp applications," in *Proceedings of the 7th*

International Conference on OpenMP in the Petascale Era, ser. IWOMP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 151–164. [Online]. Available: http://dl.acm.org/citation.cfm?id=2023025.2023041 [*Cited on pages 4 and 11.*]

[19] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–12. [*Cited on pages 4 and 11.*]

[20] 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*, ser. CC'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 132–146. [Online]. Available: http://dl.acm.org/citation.cfm?id=1788374.1788386 [*Cited on pages 4, 5, 14, 18, 115, and 139.*]

[21] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, pp. 101–113, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1379022.1375595 [*Cited on pages 4, 12, 14, 18, 68, and 115.*]

[22] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The paradigm compiler for distributed-memory multicomputers," *Computer*, vol. 28, no. 10, pp. 37–47, Oct 1995. [*Cited on page 4.*]

[23] 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 WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING*. Springer-Verlag, Berlin/Heidelberg, 1994, pp. 10–1. [*Cited on pages 4 and 14.*]

[24] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling fortran d for mimd distributed-memory machines," *Commun. ACM*, vol. 35, no. 8, pp. 66–80, Aug. 1992. [Online]. Available: http://doi.acm.org/10.1145/135226.135230 [*Cited on page 4.*]

[25] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the suif compiler," *Computer*, vol. 29, no. 12, pp. 84–89, Dec 1996. [*Cited on pages 4 and 14.*]

[26] S. Benkner, B. M. Chapman, and H. P. Zima, "Vienna fortran 90," in *Proceedings Scalable High Performance Computing Conference SHPCC-92.*, April 1992, pp. 51–59. [*Cited on page 4.*]

[27] D. Mustafa and R. Eigenmann, "Petra: Performance evaluation tool for modern parallelizing compilers," *International Journal of Parallel Programming*, vol. 43, no. 4, pp. 549–571, Aug. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10766-014-0307-8 [*Cited on pages 5, 10, and 12.*]

[28] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "Helix: Automatic parallelization of irregular programs for chip multiprocessing," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 84–93. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259028 [*Cited on pages 5 and 11.*]

[29] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su, "Intel® openmp c++/fortran compiler for hyper-threading technology: Implementation and performance." *Intel Technology Journal*, vol. 6, no. 1, 2002. [*Cited on pages 5, 14, and 18.*]

[30] R. Eigenmann, J. Hoeflinger, Z. Li, and D. A. Padua, "Experience in the automatic parallelization of four perfect-benchmark programs," in *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*. London, UK, UK: Springer-Verlag, 1992, pp. 65–83. [Online]. Available: http://dl.acm.org/citation.cfm?id=645669.665205 [*Cited on page 9.*]

[31] R. Eigenmann, "Toward a methodology of optimizing programs for high-performance computers," in *Proceedings of the 7th International Conference on Supercomputing*, ser. ICS '93. New York, NY, USA: ACM, 1993, pp. 27–36. [Online]. Available: http://doi.acm.org/10.1145/165939.165948 [*Cited on page 9.*]

[32] D. Göhringer and J. Tepelmann, "An interactive tool based on polly for detection and parallelization of loops," in *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, ser. PARMA-DITAM '14. New York, NY, USA: ACM, 2014, pp. 1:1–1:6. [Online]. Available: http://doi.acm.org/10.1145/2556863.2556869 [*Cited on pages 10 and 115.*]

[33] W. Blume and R. Eigenmann, "The range test: A dependence test for symbolic, non-linear expressions," in *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 528–537. [Online]. Available: http://dl.acm.org/citation.cfm?id=602770.602858 [*Cited on page 10.*]

[34] ——, "Nonlinear and symbolic data dependence testing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 12, pp. 1180–1194, Dec 1998. [*Cited on page 10.*]

[35] K. Psarris and K. Kyriakopoulos, "Nonlinear symbolic analysis for advanced program parallelization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, pp. 623–640, 2008. [*Cited on page 10.*]

[36] K. Kyriakopoulos and K. Psarris, "Efficient techniques for advanced data dependence analysis," in *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, Sept 2005, pp. 143–153. [*Cited on page 10.*]

[37] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 211–243, Feb 1993. [*Cited on page 11.*]

[38] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August, "Automatic speculative doall for clusters," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 94–103. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259029 [*Cited on page 11.*]

[39] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August, "Speculative parallelization using software multi-threaded transactions," *SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 65–76, Mar. 2010. [Online]. Available: http://doi.acm.org/10.1145/1735970.1736030 [*Cited on page 11.*]

[40] H. Vandierendonck, S. Rul, and K. De Bosschere, "The paralax infrastructure: automatic parallelization with a helping hand," in *Parallel Architectures and Compilation Techniques, 19th International Conference, Proceedings*. Association for Computing Machinery (ACM), 2010, pp. 389–400. [*Cited on page 11.*]

[41] L. Rauchwerger, "Run-time parallelization: Its time has come," *Parallel Comput.*, vol. 24, no. 3-4, pp. 527–556, May 1998. [Online]. Available: http://dx.doi.org/10.1016/S0167-8191(98)00024-6 [*Cited on page 11.*]

[42] Z. Liu, B. Chapman, T.-H. Weng, and O. Hernandez, "Improving the performance of openmp by array privatization," in *Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming*, ser. WOMPAT'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 244–259. [Online]. Available: http://dl.acm.org/citation.cfm?id=1761900.1761925 [*Cited on pages 11 and 16.*]

[43] Z. Li, "Array privatization for parallel execution of loops," in *Proceedings of the 6th International Conference on Supercomputing*, ser. ICS '92. New York, NY, USA: ACM, 1992, pp. 313–322. [Online]. Available: http://doi.acm.org/10.1145/143369.143426 [*Cited on pages 11 and 16.*]

[44] P. Tu and D. Padua, "Automatic array privatization," in *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 500–521. [*Cited on page 11.*]

[45] S. W. Kim, M. Voss, and R. Eigenmann, "Performance analysis of compiler-parallelized programs on shared-memory multiprocessors," *Proceedings of CPC2000 Compilers for Parallel Computers*, p. 305, 2000. [*Cited on page 11.*]

[46] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, Dec. 1994. [Online]. Available: http://doi.acm.org/10.1145/197405.197406 [*Cited on pages 11, 26, and 31.*]

[47] B. Blainey, C. Barton, and J. N. Amaral, "Removing impediments to loop fusion through code transformations," in *Proceedings of the 15th International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'02. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 309–328. [Online]. Available: http://dx.doi.org/10.1007/11596110_21 [*Cited on page 11.*]

[48] A. Fraboulet, K. Kodary, and A. Mignotte, "Loop fusion for memory space optimization," in *Proceedings of the 14th International Symposium on Systems Synthesis*, ser. ISSS '01. New York, NY, USA: ACM, 2001, pp. 95–100. [Online]. Available: http://doi.acm.org/10.1145/500001.500025 [*Cited on page 11.*]

[49] N. Manjikian and T. S. Abdelrahman, "Fusion of loops for parallelism and locality," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 2, pp. 193–209, Feb. 1997. [Online]. Available: http://dx.doi.org/10.1109/71.577265 [*Cited on page 11.*]

[50] K. Kennedy and K. S. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *Languages and Compilers for Parallel Computing: 6th International Workshop Portland, Oregon, USA, August 12–14, 1993 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 301–320. [Online]. Available: https://doi.org/10.1007/3-540-57659-2_18 [*Cited on page 11.*]

[51] J. W. Davidson and S. Jinturkar, "Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, ser. MICRO 28. Los Alamitos, CA, USA: IEEE Computer Society Press, 1995, pp. 125–132. [Online]. Available: http://dl.acm.org/citation.cfm?id=225160.225184 [*Cited on page 12.*]

[52] V. Sarkar, "Optimized unrolling of nested loops," *Int. J. Parallel Program.*, vol. 29, no. 5, pp. 545–581, Oct. 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1012246031671 [*Cited on page 12.*]

[53] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 40:1–40:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389051 [*Cited on page 12.*]

[54] L. Song, R. Glück, and Y. Futamura, "Loop peeling based on quasi-invariance/induction variables," *Wuhan University Journal of Natural Sciences*, vol. 6, no. 1, pp. 362–367, Mar 2001. [Online]. Available: https://doi.org/10.1007/BF03160270 [*Cited on page 12.*]

[55] L. Song and K. M. Kavi, "A technique for variable dependence driven loop peeling," in *Fifth International Conference on Algorithms and Architectures for Parallel Processing, 2002. Proceedings.*, Oct 2002, pp. 390–395. [*Cited on page 12.*]

[56] 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*, ser. CC'10/ETAPS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 283–303. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11970-5_16 [*Cited on pages 12, 19, 60, and 115.*]

[57] A. Simbürger, S. Apel, A. Grösslinger, and C. Lengauer, "The potential of polyhedral optimization: An empirical study," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 508–518. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693108 [*Cited on page 12.*]

[58] A. Bhattacharyya and J. N. Amaral, "Automatic speculative parallelization of loops using polyhedral dependence analysis," in *Proceedings of the First International Workshop on Code OptimiSation for MultI and Many Cores*, ser. COSMIC '13. New York, NY, USA: ACM, 2013, pp. 1:1–1:9. [Online]. Available: http://doi.acm.org/10.1145/2446920.2446921 [*Cited on page 12.*]

[59] T. Grosser, H. Zheng, R. A, A. Simbürger, A. Grösslinger, and L.-N. Pouchet, "Polly - polyhedral optimization in llvm," in *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, Apr. 2011. [*Cited on page 14.*]

[60] D. QUINLAN, "Rose: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626400000214 [*Cited on pages 14, 15, 17, 139, and 140.*]

[61] H. Gómez-Sousa, M. Arenaz, Ó. Rubiños-López, and J. Á. Martínez-Lorenzo, "Novel source-to-source compiler approach for the automatic parallelization of codes based on the method of moments," in *2015 9th European Conference on Antennas and Propagation (EuCAP)*, May 2015, pp. 1–6. [*Cited on pages 14 and 42.*]

[62] J. Lobeiras, M. Arenaz, and O. Hernández, "Experiences in extending parallware to support openacc," in *Proceedings of the Second Workshop on Accelerator Programming Using Directives*, ser. WACCPD '15. New York, NY, USA: ACM, 2015, pp. 4:1–4:12. [Online]. Available: http://doi.acm.org/10.1145/2832105.2832112 [*Cited on page 14.*]

[63] H. Bae, D. Mustafa, J.-W. Lee, Aurangzeb, H. Lin, C. Dave, R. Eigenmann, and S. P. Midkiff, "The cetus source-to-source compiler infrastructure: Overview and evaluation," *Int. J. Parallel Program.*, vol. 41, no. 6, pp. 753–767, Dec. 2013.

[Online]. Available: http://dx.doi.org/10.1007/s10766-012-0211-z [*Cited on pages 14 and 16.*]

[64] T. A. Johnson, S.-I. Lee, L. Fei, A. Basumallik, G. Upadhyaya, R. Eigenmann, and S. P. Midkiff, "Experiences in using cetus for source-to-source transformations," in *Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing*, ser. LCPC'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 1–14. [Online]. Available: http://dx.doi.org/10.1007/11532378_1 [*Cited on page 14.*]

[65] S.-I. Lee, T. A. Johnson, and R. Eigenmann, "Cetus – an extensible compiler infrastructure for source-to-source transformation," in *Languages and Compilers for Parallel Computing*, L. Rauchwerger, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 539–553. [*Cited on pages 14, 15, and 16.*]

[66] C. Ierotheou, H. Jin, G. Matthews, S. Johnson, and R. Hood, "Generating openmp code using an interactive parallelization environment," *Parallel Computing*, vol. 31, no. 10, pp. 999 – 1012, 2005, openMP. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819105001080 [*Cited on page 14.*]

[67] N. Ventroux, T. Sassolas, A. Guerre, B. Creusillet, and R. Keryell, "Sesam/par4all: A tool for joint exploration of mpsoc architectures and dynamic dataflow code generation," in *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, ser. RAPIDO '12. New York, NY, USA: ACM, 2012, pp. 9–16. [Online]. Available: http://doi.acm.org/10.1145/2162131.2162133 [*Cited on pages 14, 15, and 17.*]

[68] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, "Graphite: Loop optimizations based on the polyhedral model for gcc," in *Proc. of the 4þ GCC Developper's Summit*, Ottawa, Ontario, Unknown or Invalid Region, 2006, https://hal.archives-ouvertes.fr/hal-01257284 [*Cited on page 14.*]

[69] I. Bluemke and J. Fugas, "A tool supporting c code parallelization," in *Innovations in Computing Sciences and Software Engineering*. Dordrecht: Springer Netherlands, 2010, pp. 259–264. [Online]. Available: https://doi.org/10.1007/978-90-481-9112-3_44 [*Cited on page 14.*]

[70] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, "Parallel programming with polaris," *Computer*, vol. 29, no. 12, pp. 78–82, Dec. 1996. [Online]. Available: http://dx.doi.org/10.1109/2.546612 [*Cited on page 14.*]

[71] 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," Stanford University, Stanford, CA, USA, Tech. Rep., 1994. [*Cited on page 14.*]

[72] D. Kwon, S. Han, and H. Kim, "Mpi backend for an automatic parallelizing compiler," in *Fourth InternationalSymposium on Parallel Architectures, Algorithms, and Networks, 1999. (I-SPAN '99) Proceedings.*, 1999, pp. 152–157. [*Cited on page 14.*]

[73] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI (2Nd Ed.): Portable Parallel Programming with the Message-passing Interface*. Cambridge, MA, USA: MIT Press, 1999. [*Cited on page 13.*]

[74] P. T. J. and Q. R. W., "Antlr: A predicated ll(k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250705 [*Cited on page 15.*]

[75] S. C. Johnson, "Yet another compiler-compiler," in *in Unix Programmer's Manual, Seventh Edition*. Citeseer, 1979. [*Cited on page 15.*]

[76] A. A. Aaby, "Compiler construction using flex and bison," *Podręcznik dostępny pod http://cs. wwc. edu/˜ aabyan/464/Book*, vol. 5, 2005. [*Cited on page 15.*]

[77] R. Keryell, R. K. (presenting, C. Ancourt, B. Creusillet, F. Coelho, P. Jouvelot, and F. Irigoin, "Pips: a workbench for building interprocedural parallelizers, comilers and optimizers," PIPS4U, Tech. Rep., 1996. [*Cited on pages 15 and 17.*]

[78] M. Kim, H. Kim, and C.-K. Luk, "Prospector: A dynamic data-dependence profiler to help parallel programming," in *HotPar'10: Proceedings of the USENIX workshop on Hot Topics in parallelism*, 2010. [*Cited on page 15.*]

[79] S. Verdoolaege, "Isl: An integer set library for the polyhedral model," in *Proceedings of the Third International Congress Conference on Mathematical Software*, ser. ICMS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 299–302. [Online]. Available: http://dl.acm.org/citation.cfm?id=1888390.1888455 [*Cited on pages 15 and 19.*]

[80] P. Chatarasi, J. Shirako, and V. Sarkar, "Polyhedral optimizations of explicitly parallel programs," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct 2015, pp. 213–226. [*Cited on pages 15 and 105.*]

[81] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 7–16. [Online]. Available: https://doi.org/10.1109/PACT.2004.11 [*Cited on pages 15, 19, and 68.*]

[82] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD Conference*, 2008, pp. 1–2. [*Cited on page 15.*]

[83] C. Bastoul, "Openscop: A specification and a library for data exchange in polyhedral compilation tools," tech. rep., Paris-Sud University, France, Tech. Rep., 2011. [*Cited on pages 15, 19, 111, and 140.*]

[84] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. [*Cited on pages 16, 27, 28, and 34.*]

[85] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*. Cambridge, MA, USA: MIT Press, 1990. [*Cited on pages 16 and 26.*]

[86] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu, "Automatic detection of parallelism: A grand challenge for high-performance computing," *IEEE Parallel Distrib. Technol.*, vol. 2, no. 3, pp. 37–47, Sep. 1994. [Online]. Available: http://dx.doi.org/10.1109/M-PDT.1994.329796 [*Cited on pages 16 and 32.*]

[87] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. [*Cited on page 17.*]

[88] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010. [*Cited on page 17.*]

[89] M. Amini, C. Ancourt, F. Coelho, B. Creusillet, S. Guelton, F. Irigoin, P. p.-p. Jouvelot, R. Keryell, and P. Villalon, "Pips is not (just) polyhedral software adding gpu code generation in pips," in *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011) in conjonction with CGO 2011*, Chamonix, France, Apr. 2011, 6 pages. [Online]. Available: https://hal-mines-paristech.archives-ouvertes.fr/hal-00744312 [*Cited on page 17.*]

[90] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization: An overview of the pips project," in *Proceedings of the 5th International Conference on Supercomputing*, ser. ICS '91. New York, NY, USA: ACM, 1991, pp. 244–251. [Online]. Available: http://doi.acm.org/10.1145/109025.109086 [*Cited on page 17.*]

[91] B. Creusillet and F. Irigoin, "Interprocedural array region analyses," *Int. J. Parallel Program.*, vol. 24, no. 6, pp. 513–546, Dec. 1996. [Online]. Available: http://dx.doi.org/10.1007/BF03356758 [*Cited on page 17.*]

[92] L.-N. Pouchet, "Pocc: the polyhedral compiler collection," 2013. [*Cited on page 17.*]

[93] P. Di, D. Ye, Y. Su, Y. Sui, and J. Xue, "Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on gpus," in *2012 41st International Conference on Parallel Processing*, Sept 2012, pp. 350–359. [*Cited on page 17.*]

[94] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam, "Putting polyhedral loop transformations to work," in *Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 209–225. [Online]. Available: https://doi.org/10.1007/978-3-540-24644-2_14 [*Cited on pages 18 and 19.*]

[95] C. Bastoul, "Clan-a polyhedral representation extractor for high level programs," Tech. Rep., 2008. [*Cited on page 19.*]

[96] N. Ahmed, N. Mateev, and K. Pingali, "Synthesizing transformations for locality enhancement of imperfectly-nested loop nests," in *Proceedings of the 14th International Conference on Supercomputing*, ser. ICS '00. New York, NY, USA: ACM, 2000, pp. 141–152. [Online]. Available: http://doi.acm.org/10.1145/335231.335245 [*Cited on page 19.*]

[97] R. Strzodka, M. Shaheen, D. Pajak, and H. P. Seidel, "Cache accurate time skewing in iterative stencil computations," in *2011 International Conference on Parallel Processing*, Sept 2011, pp. 571–581. [*Cited on page 19.*]

[98] Y. Solihin, *Fundamentals of Parallel Multicore Architecture*, 1st ed. Chapman & Hall/CRC, 2015. [*Cited on pages 26, 29, and 32.*]

[99] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Commun. ACM*, vol. 29, no. 12, pp. 1184–1201, Dec. 1986. [Online]. Available: http://doi.acm.org/10.1145/7902.7904 [*Cited on page 26.*]

[100] J. Xue, *Loop Tiling for Parallelism*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. [*Cited on page 29.*]

[101] U. Banerjee, *Dependence Analysis*. Springer Publishing Company, Incorporated, 2013. [*Cited on pages 31 and 157.*]

[102] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. [*Cited on page 32.*]

[103] L. N. Pouchet, "PolyBench: The Polyhedral Benchmark suite." [*Cited on page 59.*]

[104] L.-N. Pouchet and T. Yuki, "Polybench/c 3.2," 2012. [*Cited on page 59.*]

[105] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The nas parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991. [Online]. Available: https://doi.org/10.1177/109434209100500306 [*Cited on page 83.*]

[106] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks—summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165. [Online]. Available: http://doi.acm.org/10.1145/125826.125925 [*Cited on page 83.*]

[107] S. Seo, J. Kim, G. Jo, J. Lee, J. Nah, and J. Lee, "Snu npb suite," 2016. [*Cited on page 83.*]

[108] D. Griebler, J. Loff, L. G. Fernandes, G. Mencagli, and M. Danelutto, "Efficient nas benchmark kernels with c++ parallel programming," in *Proceedings of the 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, Cambridge, United Kingdom, Jan 2018. [*Cited on page 83.*]

[109] A. Y. Drozdov, S. V. Novikov, V. E. Vladislavlev, E. L. Kochetkov, and P. V. Il'in, "Program auto parallelizer and vectorizer implemented on the basis of the universal translation library and llvm technology," *Program. Comput. Softw.*, vol. 40, no. 3, pp. 128–138, May 2014. [Online]. Available: http://dx.doi.org/10.1134/S0361768814030037 [*Cited on page 83.*]

[110] P. Wong and R. Der Wijngaart, "Nas parallel benchmarks i/o version 2.4," *NASA Ames Research Center, Moffet Field, CA, Tech. Rep. NAS-03-002*, 2003. [*Cited on page 91.*]

[111] M. Guo, "Automatic parallelization and optimization for irregular scientific applications," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, April 2004, pp. 228–. [*Cited on page 105.*]

[112] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, Feb 1991. [Online]. Available: https://doi.org/10.1007/BF01407931 [*Cited on pages 111 and 151.*]

[113] 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," *International Journal of Parallel Programming*, vol. 34, pp. 261–317, 2006. [*Cited on page 111.*]

[114] 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*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 287–298. [Online]. Available: https://doi.org/10.1109/SC.2014.29 [*Cited on page 115.*]

[115] 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 *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct 2015, pp. 138–149. [*Cited on page 115.*]

[116] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: A dsl for easy and efficient graph analysis," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating*

Systems, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 349–362. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151013 [*Cited on page 117.*]

[117] 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 *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54. [*Cited on page 118.*]

[118] K. Skadron, "Rodinia benchmark suite," http://lava.cs.virginia.edu/Rodinia/download_links.htm, 2018 (accessed May 31, 2018). [*Cited on page 118.*]

[119] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, Jun. 1982. [Online]. Available: http://doi.acm.org/10.1145/872726.806987 [*Cited on pages 139 and 140.*]

[120] M. Schordan and D. Quinlan, "A source-to-source architecture for user-defined optimizations," in *Modular Programming Languages*, L. Böszörményi and P. Schojer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 214–223. [*Cited on page 140.*]

[121] A. V. Aho and J. D. Ullman, *Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1977. [*Cited on page 140.*]

[122] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, 2012. [*Cited on page 151.*]

[123] A. Kumar and S. Pop, "Scop detection: A fast algorithm for industrial compilers," in *6th International Workshop on Polyhedral Compilation Techniques on IMPACT*, 2016. [*Cited on page 151.*]