uc3m | Universidad Carlos III de Madrid

PhD Thesis

Parallel source code transformation techniques using design patterns

Author:
David del Río Astorga

Advisor:
José Daniel García Sánchez

Computer Science and Technology

Leganés, July 2018.

PhD Thesis

Parallel source code transformation techniques using design patterns

Author: David del Río Astorga

Advisor: José Daniel García Sánchez

Computer Science and Technology

Presidente D. .....

Vocal D. .....

Secretario D. .....

Realizado el acto de defensa y la lectura de tesis en .....en el dia .....de .....del año 2018.

Calificación: .....

El Presidente El Secretario

Los Vocales

Abstract

In recent years, the traditional approaches for improving performance, such as increasing the clock frequency, has come to a dead-end. To tackle this issue, parallel architectures, such as multi-/many-core processors, have been envisioned to increase the performance by providing greater processing capabilities. However, programming efficiently for this architectures demands big efforts in order to transform sequential applications into parallel and to optimize such applications. Compared to sequential programming, designing and implementing parallel applications for operating on modern hardware poses a number of new challenges to developers such as data races, deadlocks, load imbalance, etc.

To pave the way, parallel design patterns provide a way to encapsulate algorithmic aspects, allowing users to implement robust, readable and portable solutions with such high-level abstractions. Basically, these patterns instantiate parallelism while hiding away the complexity of concurrency mechanisms, such as thread management, synchronizations or data sharing. Nonetheless, frameworks following this philosophy does not share the same interface and users require understanding different libraries, and their capabilities, not only to decide which fits best for their purposes but also to properly leverage them. Furthermore, in order to parallelize these applications, it is necessary to analyze the sequential code in order to detect the regions of code that can be parallelized that is a time consuming and complex task. Additionally, different libraries targeted to specific devices provide some algorithms implementations that are already parallel and highly-tuned. In these situations, it is also necessary to analyze and determine which routine implementation is the most suitable for a given problem.

To tackle these issues, this thesis aims at simplifying and minimizing the necessary efforts to transform sequential applications into parallel. This way, resulting codes will improve their performance by fully exploiting the available resources while the development efforts will be considerably reduced. Basically, in this thesis, we contribute with the following. First, we propose a technique to detect potential parallel patterns in sequential code. Second, we provide a novel generic C++ interface for parallel patterns which acts as a switch among existing frameworks. Third, we implement a framework that is able to transform sequential code into parallel using the proposed pattern discovery technique and pattern interface. Finally, we propose mechanisms that are able to select the most suitable device and routine implementation to solve a given problem based on previous performance information. The evaluation demonstrates that using the proposed techniques can minimize the refactoring and optimization time while improving the performance of the resulting applications with respect to the original code.

Resumen

En los últimos años, las técnicas tradicionales para mejorar el rendimiento, como es el caso del incremento de la frecuencia de reloj, han llegado a sus límites. Con el fin de seguir mejorando el rendimiento, se han desarrollado las arquitecturas paralelas, las cuales proporcionan un incremento del rendimiento al estar provistas de mayores capacidades de procesamiento. Sin embargo, programar de forma eficiente para estas arquitecturas requieren de grandes esfuerzos por parte de los desarrolladores. Comparado con la programación secuencial, diseñar e implementar aplicaciones paralelas enfocadas a trabajar en estas arquitecturas presentan una gran cantidad de dificultades como son las condiciones de carrera, los deadlocks o el incorrecto balanceo de la carga.

En este sentido, los patrones paralelos son una forma de encapsular aspectos algorítmicos de las aplicaciones permitiendo el desarrollo de soluciones robustas, portables y legibles gracias a las abstracciones de alto nivel. En general, estos patrones son capaces de proporcionar el paralelismo a la vez que ocultan las complejidades derivadas de los mecanismos de control de concurrencia necesarios como el manejo de los hilos, las sincronizaciones o la compartición de datos. No obstante, los diferentes frameworks que siguen esta filosofía no comparten una única interfaz lo que conlleva que los usuarios deban conocer múltiples bibliotecas y sus capacidades, con el fin de decidir cuál de ellos es mejor para una situación concreta y como usarlos de forma eficiente. Además, con el fin de paralelizar aplicaciones existentes, es necesario analizar e identificar las regiones del código que pueden ser paralelizadas, lo cual es una tarea ardua y compleja. Además, algunos algoritmos ya se encuentran implementados en paralelo y optimizados para arquitecturas concretas en diversas bibliotecas. Esto da lugar a que sea necesario analizar y determinar que implementación concreta es la más adecuada para solucionar un problema dado.

Para paliar estas situaciones, está tesis busca simplificar y minimizar el esfuerzo necesario para transformar aplicaciones secuenciales en paralelas. De esta forma, los códigos resultantes serán capaces de explotar los recursos disponibles a la vez que se reduce considerablemente el esfuerzo de desarrollo necesario. En general, esta tesis contribuye con lo siguiente. En primer lugar, se propone una técnica de detección de patrones paralelos en códigos secuenciales. En segundo lugar, se presenta una interfaz genérica de patrones paralelos para C++ que permite seleccionar la implementación de dichos patrones proporcionada por frameworks ya existentes. En tercer lugar, se introduce un framework de transformación de código secuencial a paralelo que hace uso de las técnicas de detección de patrones y la interfaz presentadas. Finalmente, se proponen mecanismos capaces de seleccionar la implementación más adecuada para solucionar un problema concreto basándose en el rendimiento obtenido en ejecuciones previas. Gracias a la evaluación realizada se ha podido demostrar que uso de las técnicas presentadas pueden minimizar el tiempo necesario para transformar y optimizar el código a la vez que mejora el rendimiento de las aplicaciones transformadas.

Agradecimientos

Esta tesis cierra una etapa de mi vida la cual he compartido con muchas personas a las que quiero agradecer su apoyo y compañía. Estos agradecimientos van para todas estas personas que han estado ahí en todo momento, tanto en el ámbito profesional como personal.

En primer lugar, me gustaría agradecer todo el esfuerzo y apoyo proporcionado por mis compañeros de trabajo, tanto en el propio trabajo como aguantándome durante este tiempo. A Manuel por los días de duro trabajo para sacar adelante las publicaciones antes de las fechas límite, por enseñarme a como ser mejor en nuestra profesión y por las risas en los momentos de distensión. A Javi Doc por compartir su sabiduría y conocimiento cada día sobre todos los temas existentes. A Luismi que me ayudo en los primeros pasos y animando el ambiente con sus chistes (exageradamente malos). No sin mencionar a todos los compañeros con los que he compartido laboratorio en los días buenos y malos: Javi "Kernel", Andrés, Mario, Guille y Javi Prieto. Gracias a vosotros he podido disfrutar del trabajo realizado. También quiero agradecer la acogida que se me ha dado en este grupo y a mi director José Daniel por darme la oportunidad de trabajar en este proyecto.

Por otro lado, quiero agradecer a mi familia el haberme dado el apoyo durante todo este periodo universitario, desde que inicie mis estudios en Ingeniería Informática hasta el día de hoy en que finalizo esta tesis. También quiero agradecer a Noelia el haber estado para mí en los momentos buenos y en los no tan buenos, siendo la mejor compañera que haya podido conocer en todos los aspectos. Finalmente, no puedo olvidarme de todos los amigos que me han acompañado en este largo camino y con los que tanto he compartido. A Javi, Juan, Raúl, Iván y un largo etcétera. Muchas gracias a todos.

Contents

Abstract iii

Resumen v

Agradecimientos vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objetives . . . 2
- 1.3 Document structure . . . 3

2 State of the art 5
- 2.1 Parallel architectures . . . 5
- 2.2 Parallel Programming Models . . . 7
  - 2.2.1 Types of parallelism . . . 7
  - 2.2.2 Low-level parallel frameworks . . . 8
  - 2.2.3 High-level parallel frameworks . . . 9
- 2.3 Parallel Patterns . . . 11
  - 2.3.1 Data parallel patterns . . . 12
  - 2.3.2 Task parallel patterns . . . 12
  - 2.3.3 Stream parallel patterns . . . 13
  - 2.3.4 Stream operators . . . 14
  - 2.3.5 Advanced stream parallel patterns . . . 16
- 2.4 Parallel region detection techniques . . . 17
  - 2.4.1 Dynamic approaches . . . 18
  - 2.4.2 Static approaches . . . 19
- 2.5 Routine implementation selection techniques. . . . 19
- 2.6 Summary . . . 20

3 Static parallel pattern detection 23
- 3.1 LLVM infrastructure and the Abstract Syntax Tree . . . 23
- 3.2 Parallel Pattern Annotation Specificaction . . . 24
- 3.3 Parallel Pattern Analyzer Tool . . . 25
  - 3.3.1 Pipeline Detection Module . . . 27
  - 3.3.2 Farm Detection Module . . . 28
  - 3.3.3 Map Detection Module . . . 28
- 3.4 Evaluation . . . 29
  - 3.4.1 Reference platform . . . 29
  - 3.4.2 Results for the benchmarks suites . . . 29
  - 3.4.3 Analysis of the Fastflow use case . . . 31
- 3.5 Summary . . . 32

4 Generic Parallel Pattern Interface **33**
4.1 Execution policies . . . 33
4.2 Communication channels . . . 35
4.3 Description of the pattern interfaces . . . 37
4.3.1 Data patterns . . . 37
4.3.2 Stream patterns . . . 38
4.3.3 Stream operators . . . 40
4.4 Advanced patterns . . . 42
4.5 Pattern composability . . . 43
4.6 Evaluation . . . 49
4.6.1 Reference platform . . . 51
4.6.2 Analysis of the usability . . . 51
4.6.3 Performance analysis of pattern compositions . . . 53
4.6.4 Performance analysis of stream vs data patterns . . . 53
4.6.5 Performance analysis on heterogeneous configurations . . . 55
4.6.6 Performance analysis of the FM-Radio . . . 55
4.6.7 Performance analysis of the Stream-Pool pattern . . . 56
4.6.8 Performance analysis of the Windowed-Farm pattern . . . 56
Analysis of the *count-based* windowing policy . . . 57
Analysis of the *delta-based* windowing policy . . . 58
Analysis of the *time-based* windowing policy . . . 58
Analysis of the *punctuation-based* windowing policy . . . 59
4.6.9 Performance analysis of the Stream-Iterator pattern . . . 60
4.7 Summary . . . 61

5 Automated Patern-based Refactoring **63**
5.1 Parallel Pattern Refactoring Tool . . . 63
5.2 Pipeline Stage Balancing Algorithm . . . 64
5.2.1 The brute-force search . . . 66
5.2.2 The heuristic approach . . . 66
5.2.3 The hybrid approach . . . 68
5.2.4 Finding the optimal concurrency degree . . . 68
Iterative search . . . 69
Greedy iterative search . . . 70
5.3 Evaluation . . . 72
5.3.1 Reference platform . . . 72
5.3.2 Analysis of the Pipeline stage balancing algorithm . . . 73
5.3.3 Analysis of the optimal concurrency degree search algorithms . . . 73
5.3.4 Evaluation with VIDEO-BENCH, a video streaming application . . . 75
Performance evaluation . . . 76
Fine-grain analysis . . . 78
5.3.5 Evaluation of LANE-DETECTION . . . 80
5.4 Summary . . . 81

6 Automatic implementation selection techniques **83**
6.1 Compiletime Implementation Selection . . . 84
6.1.1 Formal definition of the selection algorithm . . . 84
6.1.2 Description of the framework . . . 85
6.1.3 The profile-guided selection algorithm . . . 87
6.1.4 Working example . . . 89
6.2 Evaluation . . . 90

6.2.1 Reference platform 91
6.2.2 Analysis with the GEMM use case 91
6.2.3 Analysis with the HARDI use case 92
6.2.4 Comparison with alternative approaches 93
6.3 Summary 94

7 Conclusions and future work **95**
7.1 Contributions 95
7.2 Disemination 96
7.3 Future work 98

Bibliography **99**

List of Figures

2.1 Data parallel patterns. . . . 13
2.2 Stream parallel patterns. . . . 15
2.3 Stream operators. . . . 16
2.4 Advanced parallel patterns. . . . 17

3.1 Example of Abstract Syntax Tree for a given source code. . . . 24
3.2 Workflow diagram of PPAT. . . . 25
3.3 Execution time of sequential, transformed PPAT code and OpenMP versions of `Rodinia` benchmark. . . . 31

4.1 Schema of the circular buffer queue. . . . 36
4.2 Schema of the Split-Join communication channels. . . . 41
4.3 Example of Pipeline-Stream-Iterator-Pipeline composition in GRPPI. . . . 44
4.4 Pattern composition and implementation of the FM-Radio in GRPPI. . . . 45
4.5 Pipeline compositions of the video application. . . . 50
4.6 Lines of code and cyclomatic complexity of the use cases w.r.t different programming models. . . . 52
4.7 GRPPI and Intel TBB implementations of the FM-Radio. . . . 53
4.8 FPS w/ and w/o using GRPPI along with the different frameworks and Pipeline compositions. . . . 54
4.9 FPS for different frameworks and Pipeline compositions with stream and data patterns. . . . 54
4.10 FPS for the Pipeline composed of two Stencil patterns on different heterogeneous configurations. . . . 55
4.11 Speedup of the FM-Radio with varying number of bands. . . . 56
4.12 Stream-Pool speedup varying with varying number of threads and problem size. . . . 57
4.13 Windowed-Farm speedup with varying number of threads and window size using *count-based* windows. . . . 58
4.14 Windowed-Farm speedup with varying number of threads and window size using *delta-based* windows. . . . 59
4.15 Windowed-Farm speedup with varying number of threads and window size using *time-based* windows. . . . 60
4.16 Windowed-Farm speedup with varying number of threads using *punctuation-based* windows . . . 60
4.17 Stream-Iterator speedup with varying number of threads and image size. . . . 61

5.1 Workflow of the Parallel Pattern Refactoring Framework. . . . 63
5.2 Analysis of the maximum concurrency degree using representative pipelines. . . . 69
5.3 Automatic approaches for finding the optimal concurrency degree of PIBA balanced Pipelines. . . . 70

5.4 Time-to-solution of the basic PiBA algorithms. . . . 73
5.5 Speedup of the of Pipelines in Pipe-Bench balanced with the basic PiBA algorithm variants. . . . 74
5.6 Comparative of the speedup, number of iterations and accuracy of the basic PiBA w.r.t the extended versions for finding the optimal concurrency degree. . . . 74
5.7 Speedup of the of Pipelines in Pipe-Bench balanced with the extended PiBA algorithms for oversubscribed scenarios. . . . 75
5.8 Frames per second obtained by the different Pipeline variants of Video-Bench using C++ threads and Intel TBB GrPPI backends and the basic and extended PiBA algorithms. . . . 77
5.9 Task trace and number of active threads during the execution of the application without PiBA. . . . 78
5.10 Task trace and number of active threads during the execution of the application with the basic PiBA algorithm. . . . 79
5.11 State time percentage per thread/stage of the Video-BenchPipeline w/o and w/ the basic PiBA algorithm. . . . 79
5.12 Task trace and number of active threads during the execution of the application with the extended PiBA algorithm. . . . 79
5.13 State time percentage per thread/stage of the Video-BenchPipeline using the extended PiBA algorithm. . . . 80
5.14 Lane-Detection application workflow. . . . 80
5.15 Speedup obtained by Lane-Detection using C++ threads and Intel TBB GrPPI back ends and the basic and extended PiBA algorithms. . . . 81

6.1 The hybrid static-dynamic implementation workflow. . . . 86
6.2 Example of the behavior of an hypothetical function interface `func` offering three different implementations (`func1`, `func2` and `func3`). Note the cutoffs for problem sizes 30 and 80 between the implementations 1–2 and 2–3, respectively. . . . 89
6.3 Execution time of the `dgemm` kernel for different square matrix sizes and implementations. . . . 91
6.4 Progress of the accuracy of the selector and `dgemm` performance through training iterations. . . . 92
6.5 Progress of the accuracy using a range of sizes. . . . 93
6.6 Execution progress of two 30-iteration loops computing the `dgemm` kernel using HIS and OmpSs. . . . 94

List of Tables

3.1 REPHRASE attributes. 24
3.2 Results for the Rodinia benchmark suite. P, F and M stand for the number of Pipeline, Farm and Map patterns detected, respectively. . . . 30
3.3 Results for the NAS benchmark suite. P, F and M stand for the number of Pipeline, Farm and Map patterns detected, respectively. 30

4.1 Parallel patterns compositions in GRPPI. 48
4.2 Percentage of increase of lines of code w.r.t. the sequential version for the video application. 51

5.1 PIBA working example. 68

Chapter 1

Introduction

In recent years, the traditional approaches for improving CPU performance, such as increasing clock speed, has come to a dead-end because of physical constraints. To tackle this issue, the appearing of parallel architectures, such as multi-/many-core processors, allows increasing the performance by providing greater processing capabilities. Furthermore, with the emerging trend of heterogeneous platforms, comprising nodes with multi-/many-core CPUs, coprocessors, and accelerators, developers have started to leverage the advantages provided by the different computing units as these platforms allow to improve performance and energy efficiency better than other alternatives.

However, although most of the current computing hardware has been envisioned for parallel computing, much of the prevailing production software is still sequential [83]. In other words, a large portion of the computing resources provided by modern architectures is basically underused. In order to exploit these resources, it is necessary to refactor sequential software into parallel. However, platforms comprising diverse devices are notoriously more difficult to program effectively, since they demand distinct frameworks and programming interfaces [47].

This chapter introduces the background of this thesis. Section 1.1 explains the motivation for this work in the aforementioned context. Section 1.2 defines the goals and expected contributions of this work. Finally, this section outlines the structure of the rest of the document.

1.1 Motivation

As mentioned, the computational elements used in heterogeneous platforms provides performance improvements thanks to their parallel capabilities. However, programming efficiently for these architectures demands big efforts in order to transform sequential applications into parallel and to optimize such applications. Compared to sequential programming, designing and implementing parallel applications for operating on modern hardware poses a number of new challenges to developers [5]. Communication overheads, load imbalance, poor data locality, improper data layouts, contention in parallel I/O, deadlocks, starvation or the appearance of data races in threaded environments are just examples of those challenges. Besides, maintaining and migrating such applications to other parallel platforms demands considerable efforts. Thus, it becomes clear that programmers require additional expertise and endeavor to implement parallel applications, apart from the knowledge needed in the application domain.

To tackle this issue, several solutions in the area, such as parallel programming frameworks, have been developed to efficiently exploit parallel computing architectures [77]. Indeed, multiple parallel programming frameworks from the state-of-the-art benefit from shared memory multi-core architectures, such as OpenMP,

Cilk or Intel TBB; distributed platforms, such as MPI or Hadoop; and some others especially tailored for accelerators, as e.g., OpenCL and CUDA. Basically, these frameworks provide a set of parallel algorithmic skeletons (e.g. parallel_for or parallel_reduce) representing recurrent algorithmic structures that allow running pieces of source code in parallel. Nevertheless, only a small portion of production software is using these frameworks. Although all these skeletons aim to simplify the development of parallel applications, there is not a unified standard [32]. Therefore, users require understanding different frameworks, not only to decide which fits best for their purposes but also to properly use them. Not to mention the migration efforts of applications among frameworks, which becomes as well an arduous task. In this sense, a good practice to implement more robust, readable and portable solutions is the use of design patterns that provide a way to encapsulate (using a building blocks approach) algorithmic aspects with such a high-level of abstraction. Examples of applications coming from multiple domains (e.g., financial, medical and mathematical) and improving their performance through parallel programming design patterns, can be widely found in the literature [17, 45, 59].

However, in order to parallelize these applications, it is necessary to analyze the sequential code in order to detect the regions of code that can be parallelized using parallel patterns. A solution to parallelize these codes is to manually translate into parallel code, however this task results, in most cases, cumbersome and very complex for large applications. Another solution is to use refactoring tools, applications that advice developers or even semi-automatically transform sequential code into parallel [12]. Although source codes transformed using these techniques do not often get the best performance, they aid in reducing necessary refactoring time [57].

Unfortunately, refactoring tools found are still premature, not yet being fully adopted by development centers. In fact, many of them are human-supervised, being the developer the only responsible for providing specific sections of the code to be refactored. Although these tools relieve the burden of the source-to-source transformation, this process still remains semi-automatic. Key components for turning this process from semi- to full-automatic are parallel pattern detection tools. This situation motivates the purpose of this thesis: design a toolchain to automatically transform sequential codes into parallel and ease the development and optimization of parallel applications.

1.2 Objetives

In light of the growing necessity of transforming sequential applications into parallel, as well as reducing the complexity of modern parallel programming models, we define the following goal for this Thesis:

Goal: The main goal of this Thesis is to provide a tool-chain capable of detecting parallel patterns in sequential code and transform them into optimized parallel code by leveraging a unified interface, designed to act as a layer between developers and different parallel programming frameworks.

This goal can be divided in the following specific goals:

1. **To develop automated techniques to detect parallel patterns.** This tool will be able to detect parallel patterns in sequential code, diminishing the efforts required to analyze the code. The result will be an annotated source code determining the potential candidates to be refactored.

2. **To define a unified interface for parallel patterns.** This interface will act as a layer between the users and different parallel programming frameworks and, therefore, easing the development of parallel applications. Furthermore, parallel patterns supported by this interface will also be composable among them in order to build more complex constructions.

3. **To develop automated transformation techniques.** This transformation will take the annotated code in order to generate parallel code using the generic and unified interface proposed in this thesis. With this tool, the refactoring process will become automatic and, thus, reducing the efforts to transform the code.

4. **To define a mechanism to optimize parallel constructions**. During the refactoring process from sequential to parallel source code, some constructions could be further improved by rearranging compositions or by modifying execution parameters such as the concurrency degree. Therefore, this goal is to introduce a way to semi-automatically evaluate and generate better configurations and compositions of parallel patterns.

5. **To develop a mechanism to select the most suitable implementations.** This mechanism will allow selecting among different routine implementations depending on the problem size in order to provide, with a single interface, the most suitable version on each case.

1.3 Document structure

The rest of this document is structured as follows: Chapter 2 reviews the state-of-the-art about existing technologies and techniques researched in the same line as the work proposed in this thesis. Chapter 3 explains in detail the parallel pattern analyzer tool that is able to automatically detect and annotate parallel patterns in sequential codes. Chapter 4 defines the generic and reusable parallel pattern interface. Chapter 5 describes the automatic refactoring and optimization of parallel constructions for some stream-oriented parallel patterns. Chapter 6 describes the proposed mechanisms to select among different routine implementations. Finally, Chapter 7 enumerates some concluding remarks and future works.

Chapter 2

State of the art

This chapter provides the background and previous works that act as the base for the rest of the work presented in this thesis.

Firstly, Section 2.1 introduces the definition of parallel computing and presents some of the current architectures and processors. In order to efficiently exploit the resources provided by these platforms, Section 2.2 revisits different parallel frameworks from the state of the art that provide a way to express parallelism while minimizing the efforts of the parallelization task. Afterward, Section 2.3 describes the parallel patterns that will be supported by the proposed generic parallel pattern interface, one of the key points of this thesis. These patterns encapsulate algorithmic aspects allowing to ease the task of designing and developing parallel applications.

As mentioned in the previous chapter, transforming sequential code into parallel is a time-consuming and error-prone task, which demands an additional expertise and endeavor from developers. Therefore, another key point of this Thesis is the automated parallel region detection and code transformation. In Section 2.4, we discuss the state-of-the-art regarding code transformation techniques and tools.

Lastly, since different implementations of the same routine behave differently on concrete architectures, it is necessary to select which version provides the best performance in each case. In this sense, Section 2.5 reviews the research about multiple implementations selection techniques.

2.1 Parallel architectures

With the end of the traditional improvements in computing architectures, such as the increase of clock frequency, has led to the widespread adoption of parallel architectures that provides a way to increase the number of operations that can be performed at the same time. In this sense, following Flynn's taxonomy, modern architectures leverages two major strategies to provide these parallel capabilities: Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD)[19].

- **Single Instruction Multiple Data**

  This strategy is based on applying the same instruction or operation at the same time on a given data set. In this case, the parallel units of a given architecture share the same instructions but apply them to different data elements [71]. Usually, this kind of architectures employs a control unit that emits the same instruction stream to multiple execution units that applies the same instruction to different data in a synchronous manner. This way a same copy of the code can be executed simultaneously, and thus, reducing the necessary instruction bandwidth and space. Examples of technologies that take advantage of this kind of architectures are the vector instructions implemented on modern processors and the Graphic Processing Units (GPUs).

Focusing on vector instructions we can find the Advanced Vector Extensions (AVX), the successor of the Streaming SIMD Extensions (SSE) [38], implemented on modern Intel and AMD processors. In general AVX are extensions to the instruction sets of x86 architectures that allow applying the same operation over a multiple data elements. These instructions leverage registers of 256-bit that may contain eight 32-bit single-precision or four 64-bit double precision floating point numbers. Additionally, these registers and instruction have been extended to 512-bit (AVX-512) in recent Intel architectures such as Intel Xeon Phi Knight Landing and some Intel Skylake CPUs [36].

- **Multiple Instruction Multiple Data** On the contrary to the previous strategy, in this case, the parallel units do not share instructions or data. This way, each parallel unit may perform different operations over different data at the same time. In other words, processors in these architectures work in an autonomous way. However, the software exploiting these architectures requires to leverage techniques to synchronize the work performed by each process, e.g. by accessing some shared data on shared memory architectures or by passing messages via interconnection networks. Basically, we can distinguish two main architecture designs that follow this strategy: shared memory and distributed memory.

  On the one hand, shared memory architectures share the main memory, and in some architectures even cache memory, among the different cores or CPUs in the same platform. This way, processes working in these architectures are able to modify information in a memory that can be seen by other processes in the platform in order to communicate them. Depending on the distances to the memory to the different processing units, we can distinguish two kind of memory sharing: Uniform Memory Access (UMA) in which the distance to the memory from each unit is the same, i.e. the latency of accessing the memory is the same; and Non-uniform Memory Access (NUMA) where the memory access time depends on the memory location relative to a given processor. Some examples of these architectures are the multi-/many-core processors as an example of UMA architectures and multi-socket or NUMA-multiprocessor architectures as an example of NUMA architectures.

  On the other hand, distributed systems are similar to a multi-processor architecture in that they do not share memory among the different processors (nodes) in the system. In contrast to shared-memory architectures, data cannot be directly shared among processors but transmitted through an interconnection network. This interconnection network among the different nodes in the distributed system is usually implemented by using high-speed standard communication protocols, such as Gigabit Ethernet or infiniband. However, data communication has non-negligible overheads due to network communication latencies. Thus the topology of the network becomes really important to diminish these latencies. On the other hand, these architectures can be easily grown by just adding new nodes to the system. Additionally, these systems also use shared-memory in each of the nodes comprising the distributed system. A representative example of these architectures are clusters, where each node interconnected to a specific network can be equipped with one or more shared-memory processors.

Up to this point, we have discussed the most common paradigms or strategies (SIMD and MIMD), however current platforms are equipped with multiple devices

that take advantage of both paradigms. These platforms are usually named heterogeneous platforms that are comprised of one or multiple CPUs, each of them potentially with a vector unit, GPUs, accelerators and/or co-processors. This way the parallel resources have been widely increased and multiple operations can be performed at the same time by a given application. However, most of the production software is still sequential, thus even if the resources are available to be exploited by parallel applications they are still under-used. The main reason for this situation is that a big portion of that software consists of legacy applications, and transforming them into parallel is an expensive task which requires additional expertise in parallel programming by the developers.

2.2 Parallel Programming Models

As commented, sequential legacy applications do not take advantage of current parallel architectures and, therefore, under-exploit its parallel resources. However, transforming sequential code into parallel present major challenges to developers, since it requires additional expertise in parallel programming and it does not always provide performance improvements. This is mainly due to the inherent complexities of parallel programming such as synchronizations, data locality or load balance. To tackle these issues, multiple parallel programming models have been developed to alleviate the burden of parallelization.

In this section, we revisit some parallel programming models from the state-of-the-art developed to ease the development of parallel programming. Concretely, Section 2.2.1 classifies the different ways in which the parallelism is expressed in the programming models. Finally, Sections 2.2.2 and 2.2.3 provide a survey of different well-known programming models from the state of the art classified as low-level and high-level frameworks.

2.2.1 Types of parallelism

This section provides a classification of the parallel programming models from the state-of-the-art depending on the parallelism that can be expressed. Specifically, we distinguish three different types of parallelism: *Task-parallelism*, *Data-parallelism* and *Stream-parallelism*.

Task-parallelism This type of parallelism consists on executing distinct and independent computations on different execution entities (e.g. threads). This computations, namely tasks, can be communicated by sending the data among them constituting a task dependency graph. Usually, programming models that provide this kind of parallelism implement a runtime to execute the task when its dependencies have been satisfied.

Data-parallelism is based on dividing a data set among the different parallel entities and processing them independently. This kind of parallelism requires that the computations of each data element should be stateless, i.e, the processing of a data element cannot have any dependencies of any other element and the output value should not depend on state values.

Stream-parallelism This type of parallelism is based on computing in parallel the processing of elements arriving into an input stream. This parallel processing is

based on splitting the whole computation into different tasks that can be parallel tasks if the task can be applied in parallel to multiple elements, or sequential task if it is stateful and only one element at a time can be processed. Note that, in this case, as a major difference with *Data-parallelism*, the whole data set is not usually available at the beginning and the data elements might become available over time, e.g. readings from a sensor or requests received from a network connection.

2.2.2 Low-level parallel frameworks

As introduced, we classify the parallel programming into two categories: *low-level* and *high-level* parallel frameworks. *Low-level* parallel frameworks cover those that do not provide abstractions over the parallel platform. In this sense, those frameworks demand deeper knowledge of the underlying platform and usually require to manually introduce synchronization primitives, explicitly determine mutual exclusion regions or manage the data sharing.

Next, we introduce some well-known *low-level* parallel programming models from the state-of-the-art.

ISO C++ Threads The C++ thread class, incorporated in the C++11 standard [40], represents an individual thread that executes a sequence of instructions concurrently with any other thread in the application on multithreaded environments. This class encapsulates the creation of a thread hiding away the actual call that depends on the specific operating system, e.g. `pthread_create()` on Unix like systems. However, to communicate and/or synchronize threads using this feature requires to explicitly incorporate the concurrency mechanisms, e.g. mutexes or atomic variables. In this sense, programming parallel applications become complex due to the potential data dependencies, data races or deadlocks that should be managed by the developer.

Compute Unified Device Architecture CUDA is a set of tools and a compiler, developed by nVidia, that allows developers to express parallel applications targeted to Graphics Processing Units (GPUs) [63]. This framework uses an extension of the C++ language to leverage the GPU for coprocessing algorithms in parallel with the CPUs. Thanks to this framework is possible to take advantage of the SIMD capabilities provided by these architectures. In this sense, GPGPU programming is suitable for data-parallelism.

However, in order to properly leverage these architectures, is it necessary to explicitly determine the data communication between the "Host" (CPU) and the "Device" (GPU). This communication leads to overheads related to the data transfers since both CPU and GPU do not share the same memory address space. Therefore, dealing with these devices requires an additional effort to determine if the communication overheads are paid off by the performance improvement provided by them.

Open Computing Language Open Computing Language (OpenCL) is a parallel framework that allows implementing application targeted to heterogeneous platforms comprised of multi-/many-core processors, GPUs, digital signal processors (DSPs) and/or field programmable gate arrays (FPGAs) [82]. This framework, similar to CUDA, follows a host-device approach. In other words, thanks to this framework the host (CPU) is able to launch compute functions, called "kernels", on the different computing devices available in the platform. The main advantage of this

programming framework is that the same source code can run on multiple architectures without modifying it. However, it is necessary to explicitly manage the data transfers between host and device and it requires to tune the application in order to better exploit the available resources. Furthermore, the original sequential code requires deep refactoring in order to leverage OpenCL for improving the performance of a given application.

Message Passing Interface Message Passing Interface (MPI) is a standard that defines the syntax and semantics of the function provided by a given library implementing it [26]. This interface defines a way to effectively program concurrent applications on multiprocessing environments such as "clusters". Several implementations of this standard can be found on the state-of-the-art such as MPICH [33] or OpenMPI [27]. These frameworks provide synchronization and communication between processes that may run in different processors, e.g. cores in the same CPU or different CPUs on different machines. This way, using these mechanisms, multiple processes can run concurrently on the same application. However, the communication and synchronization should be explicitly incorporated in the application by means of sending or receiving messages to/from other processes. Therefore, it becomes clear that efficient use of these mechanisms requires a profound knowledge of both the framework and the target application in order to determine when and what should be passed between processes.

2.2.3 High-level parallel frameworks

In the previous section, we have discussed *low-level* parallel frameworks that allow developers to implement parallel applications. This kind of frameworks provide a way to express parallelism in a low-level way, and consequently, allow to fully optimize the code to the target platform and better exploit the available resources. However, these benefits come with a wide number of challenges, it is necessary to know the target platform, properly use synchronization and communication primitives, be concerned about data sharing, etc. For these reasons, the resulting code is usually tied to the target platform which makes it difficult to migrate the application between different architectures [5]. Furthermore, when dealing with heterogeneous platforms comprised by multiple and diverse devices, it becomes even more complex to develop portable and maintainable code.

To tackle these issues, high-level approaches provide abstractions that allow implementing parallel applications hiding away the aforementioned complexities. Therefore, the resulting applications have improved portability and maintainability compared to those implemented using *low-level* frameworks. However, *high-level* frameworks usually do not provide the best possible performance due to the inherent abstraction overheads, but performs reasonably well in a wide range of parallel platforms [32].

In the following, we review different *high-level* frameworks from the state of the art.

Open Multi Processing Open Multi Processing (OpenMP) is an application programming interface targeted to shared memory platforms [67]. This programming framework is based on "pragma" annotations to determine code regions that can be processed in parallel. Basically, OpenMP abstracts the aspects related to thread and data management thanks to the different clauses that can be introduced along with the "pragmas". In this sense, OpenMP provides some higher-level abstractions

easing the parallelization task, however, it still requires to identify data dependencies, the scope of the variables in a parallel region and restrict the access of critical sections.

Intel Cilk Intel Cilk is an extension of C and C++ languages that allows taking advantage of multi-core architectures and vector instructions [8]. Similarly to the OpenMP framework, this extension provides support for task and data parallelism that is enabled via "pragma" annotations (e.g. `simd`) and specific keywords (e.g. `_Cilk_for` and `_Cilk_spawn`). Thanks to these keywords and "pragmas", the inherent complexities of parallel programming are alleviated. Additionally, this C++ language extension provides a collection of objects that allow protecting shared variables while maintaining the sequential semantics of the application (e.g. `reducers`). However, this programming model lacks high-level abstractions and it still requires to explicitly identify synchronization points.

Intel Threading Building Blocks Intel Threading Building Blocks (TBB) is a C++ template library targeted to multi-core processors [75]. This task-parallel library provides a set of algorithmic skeletons (e.g parallel_for and pipeline) that hides away the complexities of thread management and synchronizations in such parallel constructions. Additionally, TBB incorporates a runtime scheduler that permits to execute the different task respecting the dependencies and to balance the parallel workload by leveraging a work-stealing approach. This way, this framework eases the development of parallel applications while decouples the underlying architecture from the source code.

Fastflow FastFlow is a structured data and stream parallel programming framework targeted to multi-core and GPU architectures [3]. This framework provides a set of algorithmic skeletons that models different parallel patterns such as Pipeline, Farm or Map. Basically, these constructs, implemented as C++ classes, allows encapsulating algorithmic features while hiding away complex thread communication and synchronization mechanisms. Additionally, the patterns supported by the framework can be composed among them in order to build more complex algorithmic constructs. In general, this framework reduces the parallel design and development efforts while improves the performance thanks to its high-level abstractions.

SkePU SkePU is a programming framework targeted to multicore and GPU architectures based on C++ templates [21]. This framework provides a collection of predefined generic components that implement specific computation patterns and data dependencies known as "skeletons". These skeletons receive the sequential user code encapsulating low-level and platform-specific details such as synchronizations, data management, and several optimizations. Additionally, this framework also supports multiple backends for sequential, OpenMP, OpenCL and CUDA.

Parallel Standard Template Library The parallel STL is a novel feature of C++17 [42] that provides parallel implementations of the algorithms present in the C++ standard library. These parallel algorithms support multiple execution policies that are used as an extra parameter with respect to the original STL algorithms to determine how the algorithm is computed, i.e. in sequential, parallel or vectorized. This extension is currently supported by the Intel Compiler and Microsoft Visual Studio Compiler [13].

High Performance ParallelX High Performance ParallelX (HPX) is a general purpose C++ runtime for parallel and distributed application [44]. This library provides a set of parallel algorithms that extend the C++ standard library algorithms to be computed in parallel (Parallel STL) along with other utilities such as new container types or hardware counters support. The interface for the high-level algorithms is similar to that designed for this thesis and provides support for multiple execution policies. However, this interface lacks high-level patterns targeted to stream processing applications such as the farm or pipeline patterns.

Muesli The Münster Skeleton Library (Muesli) is a C++ programming framework targeted to heterogeneous and distributed platforms [22]. In this sense, this framework is able to generate different binaries aimed at multiple heterogeneous clusters and hides away the complexities of using specific frameworks, e.g. OpenMP, MPI or CUDA. This library provides support for multiple data parallel skeletons and for the farm pattern. However, it lacks stream processing skeletons.

CUDA Thrust CUDA thrust is a C++ template library based on the standard template library (STL) [66]. Basically, this framework allows implementing algorithms targeted to nVidia GPUs by using the collection of supported data parallel algorithms. Thus, using this interface similar to the STL, eases the development of applications targeted to GPUs without requiring additional expertise and knowledge about CUDA programming. However, this framework still requires to explicitly determine the host-device data transfers and to annotate the kernels with macros.

SYCL SYCL is a high-level programming framework that introduces an abstraction layer between the users and the OpenCL framework using an interface similar to the STL [46]. This library provides an implementation of the parallel STL by defining a new execution policy to support OpenCL. This way, the code developed using this framework becomes portable and cross-platform thanks to the OpenCL environment.

2.3 Parallel Patterns

Patterns can be loosely defined as commonly recurring strategies for dealing with particular problems. This methodology has been widely used in multiple areas, such as architecture, object-oriented programming, and software architecture [53]. In this Thesis, we focus on patterns for parallel software design, as it has been recognized to be one of the best codifying practices [29]. This is mainly because patterns provide a mechanism to encapsulate algorithmic features, making them more robust, portable and reusable, while if tuned, they can achieve better parallel scalability and data locality. In general, parallel patterns can be categorized in three groups: data parallel patterns, e.g., Map, Reduce and MapReduce; task parallel patterns, e.g. Divide&Conquer; and stream parallel patterns, e.g., Pipeline, Farm and Filter [56]. Additionally, when dealing with stream processing, different kind of constructs, denoted as stream operators, can be found in the state-of-the-art in order to modify the stream flow, e.g. Window, Split-Join [6]. However, in some situations, these constructs do not match or need to be composed in a very complex way. Focusing on these situations, the advanced patterns model some domain-specific algorithms that cannot be

represented using basic patterns or simple compositions [73]. Following this classification, in the next sections, we describe formally the parallel patterns and stream operators leveraged in this Thesis.

2.3.1 Data parallel patterns

In this section, we describe formally the data parallel patterns Map, Reduce, Stencil, and MapReduce.

Map This data parallel pattern computes the function $f : \alpha \rightarrow \beta$ over the elements of the input data collection, where the input and output elements are α and β types, respectively (see Figure 2.1a). The output result is the collection of elements $y_1, y_2, \ldots, y_N$, where $y_i = f(x_i)$ for each $i = 1, 2, \ldots, N$ and x_i is the i-th element of the input collection. The only requirement of the Map pattern is that the function f should be pure, i.e. the function has no side effects.

Reduce This data parallel pattern aggregates the elements of the input data collection of type α using the binary function $\oplus : \alpha \times \alpha \rightarrow \alpha$, that is usually associative and commutative. Finally, the result of the pattern is summarized in a single element y of type α that is obtained performing the operation $y = x_1 \oplus x_2 \oplus \ldots x_N$, where x_i is the i-th data item of the input data collection (see Figure 2.1b). The main constraint of this pattern is that the binary function should be pure.

Stencil This pattern is a generalization of the Map pattern in which an elemental function can access, not only to a single element in an input collection but also to a set of neighbors (see Figure 2.1c). The function $f : \alpha* \rightarrow \alpha$ used by the Stencil pattern receives the input item and a set of neighbors ($\alpha*$) and produces an output element of the same type. The main requirement of this pattern is that the function f should be pure.

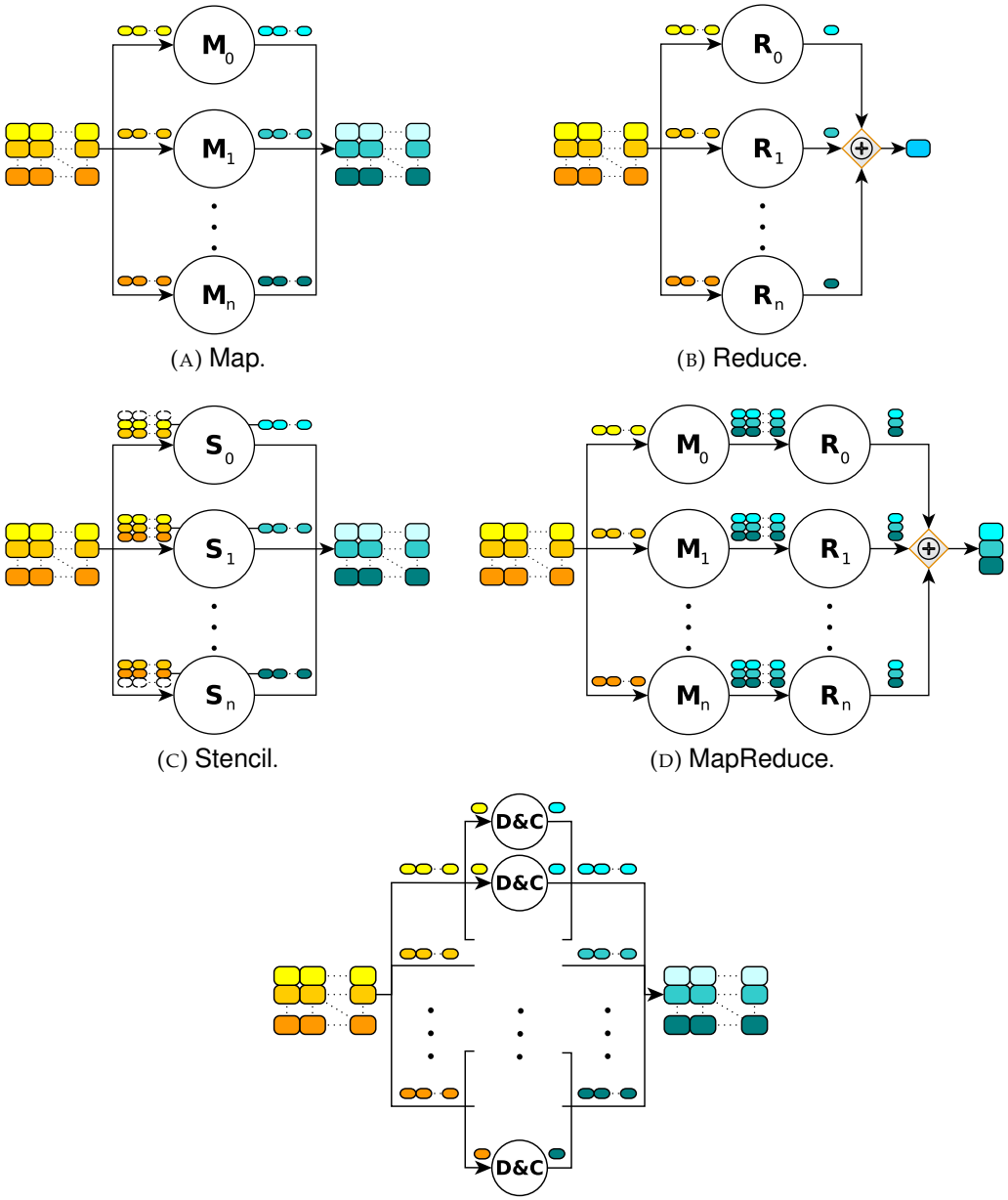
MapReduce This pattern computes, in a first stage a Map-like pattern, a key-value function over all the elements of an input collection, and delivers, in a second stage a Reduce-like pattern, a set of unique key value pairs where the value associated to the key is the "sum" of the values output for the same key (see Figure 2.1d). To do so, the MapReduce pattern computes in the Map function $f : \alpha \rightarrow \{Key, \alpha\}$ the elements in the input collection; afterwards it uses the Reduce binary function $\oplus : \beta \times \beta \rightarrow \beta$ to sum up the partial results with the same key. The result of this pattern is a collection of data elements of type β, one per key. The requirements of the MapReduce pattern is that both Map and Reduce-related functions should be pure.

2.3.2 Task parallel patterns

In this section, we describe formally the task parallel pattern Divide&Conquer.

Divide&Conquer This pattern computes a problem by means of breaking it down into two or more subproblems of the same kind until the base case is reached and solved directly. Afterward, the solutions of the subproblems are merged to provide a solution to the original problem (see Figure 2.1e). In other words, this pattern applies the function $f : \alpha* \rightarrow \beta*$ on a collection of elements of type α and produces a collection of elements of type β. A divide function $\mathcal{D}$ is used first to split the collection into distinct partitions up to the size of the base problem, which can be solved

directly applying f. Finally, the partial results of the base problems are combined according to a merge function $\mathcal{M}$ in order to build the final output collection. The requirements of the Divide&Conquer pattern are that the functions f, $\mathcal{S}$ and $\mathcal{M}$ should be pure.



(A) Map. (B) Reduce. (C) Stencil. (D) MapReduce. (E) Divide&Conquer.

FIGURE 2.1: Data parallel patterns.

2.3.3 Stream parallel patterns

In this section, we describe formally the stream parallel patterns Pipeline, Farm, Filter, and Reduce.

Pipeline This pattern processes the items appearing on the input stream in several parallel stages (see Figure 2.2(a)). Each stage of this pattern processes data produced by the previous stage in the pipe and delivers results to the next one. Provided that

the i-th stage in a n-staged Pipeline computes the function $f_i : \alpha \rightarrow \beta$, the Pipeline delivers the item x_i to the output stream applying the function $f_n(f_{n-1}(\ldots f_1(x_i)\ldots))$. The main requirement of this pattern is that the functions related to the stages should be pure, i.e., they can be computed in parallel without side effects.

Farm This pattern computes in parallel the function $f : \alpha \rightarrow \beta$ over all the items appearing in the input stream (see Figure 2.2(b)). Thus, for each item x_i on the input stream the Farm pattern delivers an item to the output stream as $f(x_i)$. In this pattern, the computations performed by f for the items in the input stream should be completely independent of each other, otherwise, they cannot be processed in parallel.

Filter This pattern computes in parallel a filter over the items appearing on the input stream, passing only to the output stream those items satisfying the boolean "filter" function (or predicate) $\mathcal{P} : \alpha \rightarrow \{true, false\}$ (see Figure 2.2(c)). Basically, the pattern receives a sequence of input items $\ldots, x_{i+1}, x_i, x_{i-1}, \ldots$ and produces a sequence of output items of the same type but with different cardinality. The evaluation of the filtering function on an input item should be independent to any other, i.e., the predicate should be a pure function.

Reduce This pattern collapses items appearing on the input stream and delivers these results to the output stream (see Figure 2.2(d)). The function used to collapse item values $\oplus$ should be a pure binary function of type $\oplus : \alpha \times \alpha \rightarrow \alpha$, being usually associative and commutative. Basically, the pattern computes the function $\oplus$ over a finite sequence of input items $\ldots, x_{i+1}, x_i, x_{i-1}, \ldots$ to produce a collapsed item on the output stream. The number of elements to be accumulated depends on the window size set as a parameter.

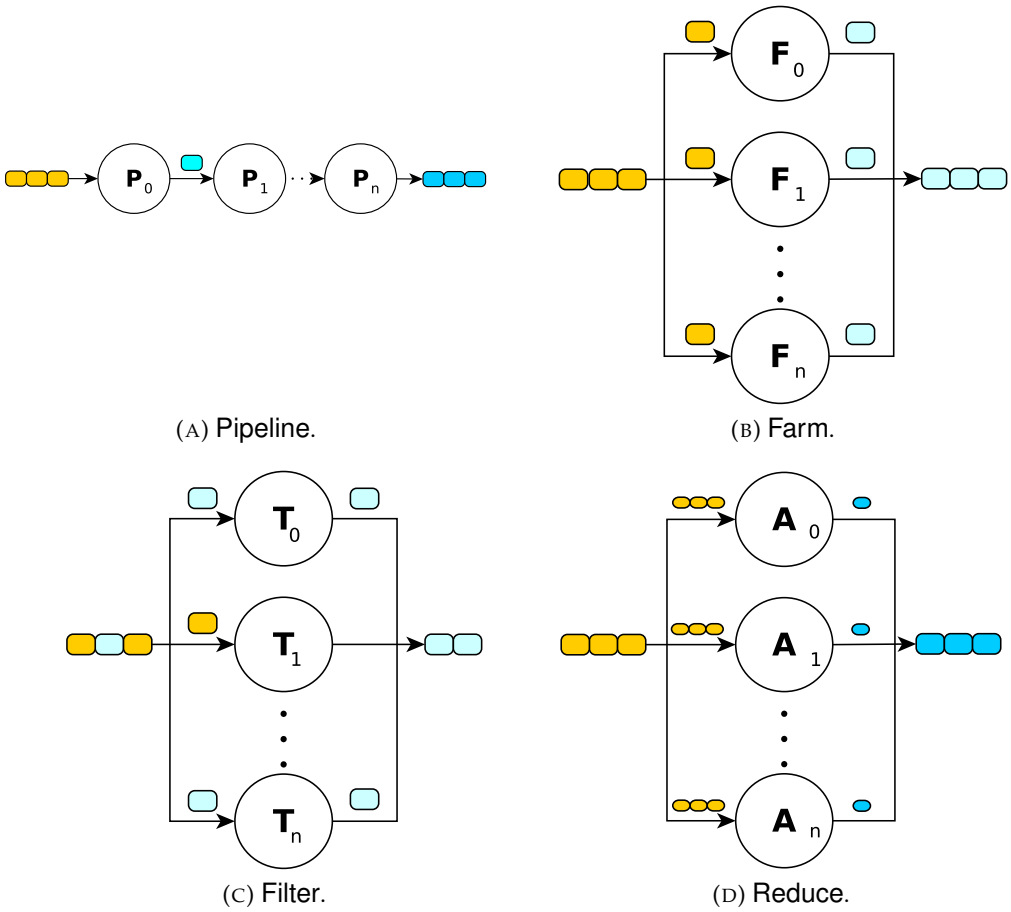
2.3.4 Stream operators

The stream operators are designed to work cooperatively with other patterns in order to provide a way of expressing more complex constructions. Specifically, they are intended to modify the stream flow in different ways. In this section, we describe the Split-Join and the Window stream operators along with their different configurations.

Split-Join This stream operator distributes, in a first Split phase, a data stream into different substreams which can be processed in parallel applying different transformations. Afterwards, a Join phase combines the substreams into a single one (see Fig. 2.3(a)). This operator is characterized by the different distribution and combining policies that can be applied in both Split and Join phases. The supported policies are the following two:

Duplication This policy duplicates the data items appearing on the input stream to each of the different substreams. In other words, when an item of type α arrives at the input stream, this is copied into every substream. By definition, this policy can only be applied in the Split phase of the Split-Join operator.

Round-robin This policy can be applied in both Split and Join phases. If applied on the Split phase, the data items of type α appearing on the input stream are delivered following a round-robin policy onto the substreams. On the other



(A) Pipeline. (B) Farm. (C) Filter. (D) Reduce.

FIGURE 2.2: Stream parallel patterns.

hand, if used on the Join phase, the items delivered at the end of the sub-streams are combined together into the main data stream employing the same policy. In this distribution policy the slice size, i.e. the number of consecutive items that should be taken from the source stream, can be freely configured.

Window This stream operator takes the data items from the input stream and delivers collections of items (windows) to the output stream (see Fig. 2.3(b)). Depending on how the windows are internally managed by this operator, different windowing policies can be set. The following policies can be portrayed based on what and how many items are part of the same window.

Count-based This policy is characterized by managing windows of fixed size, i.e., capable of holding up a maximum number of items. Note that the user should specify the window size. The rationale of this policy is the following: when a new item of type α arrives at the input stream, this is included in the window and the oldest is flushed. As soon as the window is complete, it is delivered to the output stream.

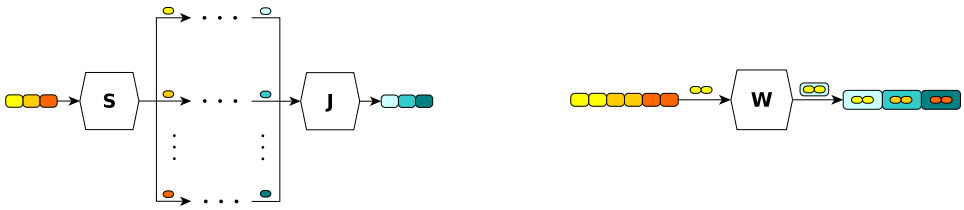
Delta-based This policy requires a δ threshold value and a monotonic increasing (Δ) attribute included in the input items. With these parameters, the *delta-based* policy is able to build a window of variable size. The items conforming a window are only those whose difference between the Δ attribute of the latest item and the current one is less or equal than the given δ threshold. Similar

to the *count-based* policy, when a window is built it is forwarded to the output stream.

Time-based This policy keeps an internal wall-clock and labels each input item with a timestamp indicating their arrival time. Additionally, it requires the users to specify a time threshold (τ). This information allows the policy to build windows including the items that arrived in the last time, as specified by τ. For instance, a threshold of 60 s would conform windows including items that arrived at the last minute.

Punctuation-based This policy needs a *punctuation* value/symbol that indicates the end of a window. In other words, it delivers a new window each time a new item matching the *punctuation* value is received. For instance, considering a stream of words belonging to a text and using the "." character as for the punctuation symbol, this policy would conform windows containing the sentences in the text.

Note that the count-, delta- and time-based windowing policies support an overlap factor, i.e., the number of items in the window w_i that are also part of the window w_{i+1}.



(A) Split-Join. (B) Window.

FIGURE 2.3: Stream operators.

2.3.5 Advanced stream parallel patterns

In this section, we describe some advanced stream parallel patterns, designed for those scenarios in which the basic patterns do not match any of these constructs or have to be composed in a very complex way.

Stream-Pool This pattern models the evolution of a population of individuals matching many evolutionary computing algorithms in the state-of-the-art [4]. Specifically, the Stream-Pool pattern is comprised of four different functions that are applied iteratively to the individuals of type α belonging to a population P managed as a stream (see Figure 2.4(a)). First, the *selection* function S: $\alpha^* \rightarrow \alpha^*$ selects a subset of individuals belonging to P. Next, the selected individuals are processed by means of the *evolution* function E: $\alpha^* \rightarrow \alpha^*$, which may produce any number of new or modified individuals. The resulting set of individuals, computed by E, are filtered through a *filter* function F: $\alpha^* \rightarrow \alpha^*$, and eventually inserted into the input stream (population). Finally, the *termination* function T: $\alpha^* \rightarrow \{true, false\}$ determines in each iteration whether the evolution process should be finished or continued. To guarantee the correctness of the parallel version of this pattern, the functions E, F and T should be pure, i.e., they can be computed in parallel with no side effects.

Windowed-Farm This pattern delivers "windows" of processed items to the output stream. Basically, this pattern applies the function F over consecutive contiguous collections of x input items of type α and delivers the resulting windows of y items of type β to the output stream (see Figure 2.4(b)). Note that this pattern simplifies the composition of the Window and Farm patterns. Also, the windows produced by this pattern benefit from the same windowing policies provided by the Window stream operator. The parallelization of this pattern requires a pure function F: $\alpha^* \rightarrow \beta^*$ for processing windows.

Stream-Iterator This pattern is intended to recurrently compute the pure function F: $\alpha \rightarrow \alpha$ on a single stream input item until a specific condition, determined by the boolean function T: $\alpha \rightarrow \{true, false\}$, is met. Additionally, in each iteration the result of the function F is delivered to the output stream, depending on a boolean output guard function G: $\alpha \rightarrow \{true, false\}$ (see Figure 2.4(c)). Note that this pattern, due to its nature, does not provide any parallelism degree by itself and can be classified as a pattern modifier. Therefore, the parallel version of this construct is only achieved when it is used in cooperation with some other core stream pattern, e.g., using Farm or Pipeline as for the function F. An example of Stream-Iterator composed with a Farm pattern is shown in Figure 2.4(d).

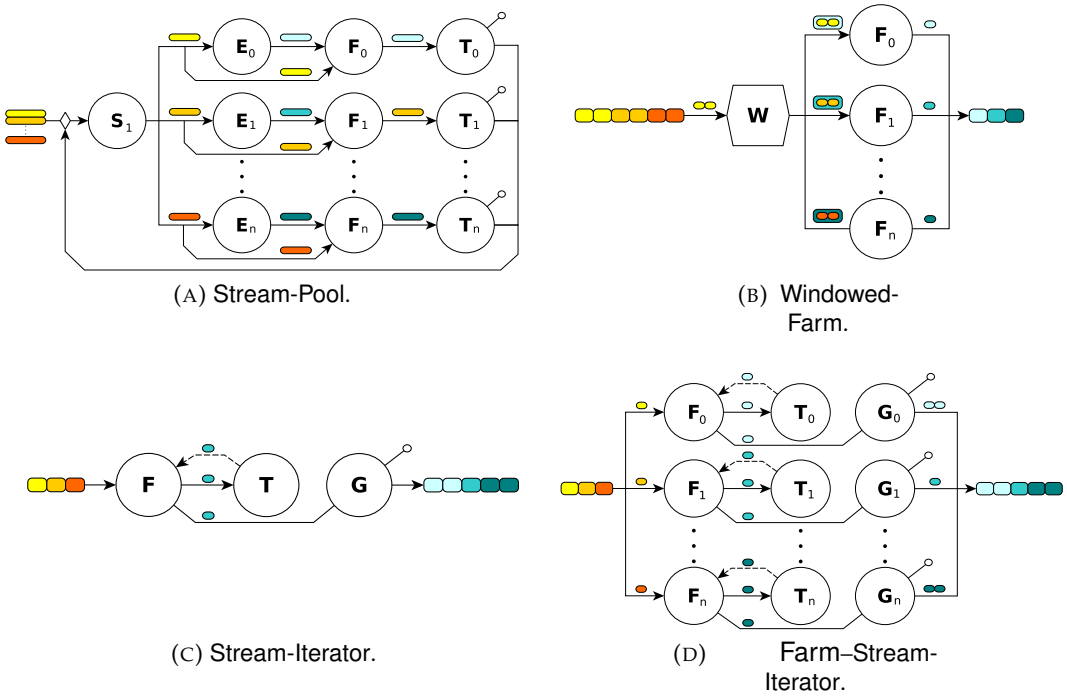


FIGURE 2.4: Advanced parallel patterns.

2.4 Parallel region detection techniques

Although parallel patterns and high-level programming frameworks alleviate the burden of developing parallel applications, this task is still time-consuming and error-prone. In this sense, automatically detecting regions that can be parallelized and its corresponding transformation from sequential to parallel are key points to reduce the complexities of parallel programming [1]. In general, in order to achieve

this automatic parallelization, we classify the approaches from the state-of-the-art into two different groups: *i)* dynamic approaches based on profiling and code instrumentation; and *ii)* static approaches that rely on analyzing the code directly.

On the one hand, the dynamic approaches usually depend on executing the application that has been previously instrumented. These tools collect information about different characteristics of the application, e.g. memory accesses or write operations, and determines data dependencies. Next, with this information, they are able to determine if a region can be parallelized by analyzing data dependencies occurred during the execution. However, these approaches have some limitations: *i)* they have non-negligible overheads due to instrumentation and the required execution to perform the postmortem analysis; *ii)* since the analysis relies on the execution profile, if a data dependency is not revealed in a given run, it may lose it and give an erroneous analysis; and *iii)* the instrumentation techniques usually lose semantic information of the original code, and thus, making harder to link the profile with the source code.

On the other hand, static approaches rely on analyzing the code without executing it. In other words, they leverage the structures generated during the compilation to analyze data dependencies and determine if a code region can be parallelized. In this sense, these approaches, since they do not use profiling techniques, require less time to perform their analysis. Additionally, as they use the structures generated at compile time, they can also leverage semantic information present in the source code and it is easier to perform source-to-source transformations. However, these techniques are usually based on approximations, so they have to deal with false-positives/negatives. For instance, some data dependencies can be difficult to detect at compile time, e.g. aliasing or access to double-indexed matrices, and it can be determined as a dependency or not depending on which approximation is used.

In this section, we revisit some research works in the state-of-the-art that addresses the detection of potential parallel codes and refactoring processes.

2.4.1 Dynamic approaches

Focusing on dynamic approaches for detecting potential parallel regions, multiple works can be found in the state-of-the-art. For example, the approach developed by Sean Rul et al. [76] leverages LLVM to instrument loops in the sequential code and performs an LLVM-IR profiling analysis to decide whether a loop is a pipeline or not. After that, it transforms the code to produce a parallel source code. However, this tool presents some shortcomings: it needs to execute the target application several times and profile it. Also, it is tied to the C programming language which allows some simplifications that cannot be made in C++. Similarly, Intel Advisor [37] performs profiling analysis in order to detect regions of code (loops) that can benefit from parallelization via threading or vectorization. Using the data collected during the profiling phase, it uses a Cache-Aware Roofline Model [51] in order to estimate the attainable performance of a given source code. Afterward, it provides a comparison for different parallel implementation alternatives.

Orthogonally, some works take advantage of functional languages. For instance, István Bozó et al. [9] develop a tool that analyzes and detects parallel patterns in applications written in Erlang in a semi-automated fashion. In a first stage, the tool performs an static analysis to detect potential parallel patterns and, afterward, it decides which pattern suits best for a given problem by using profiling techniques.

2.4.2 Static approaches

Other contributions, such as the work by Molitorisz et al. [60], detect statically potential parallel patterns, nevertheless they do not check for dependencies, so the correctness of the resulting parallel application cannot be guaranteed. Instead, they need a subsequent execution to discover potential data races and dependencies. Likewise, PoCC [72], a flexible source-to-source compiler using polyhedral compilation, is able to detect and parallelize loops, however, it does not take into account high-level parallel patterns. On the other hand, we also find tools that detect parallel patterns using only profiling techniques. For example, DiscoPoP leverages dependency graphs in order to detect parallel patterns [50]. Nevertheless, this tool has an important drawback: the profiling techniques have a non-negligible execution time and memory usage. A similar approach, presented by Tournavitis et al. [87], detects and transforms sequential code into parallel introducing parallel pipeline patterns. Alternatively, FreshBreeze [49], a dataflow-based execution, and programming model, leverages static loop detection techniques that analyze dependencies and transform parallelizable loops using a task tree-structured memory model. It is important to remark that, approaches based on static analysis are not very extended in the area since analyzing data dependencies becomes much more complex at compile time.

However many of the refactoring techniques for parallel programming presented in the literature are to some extent limited [11]. In a first inspection, we notice that many of these approaches are focused to specific structural rearrangements of the code (e.g. loop optimizations), which have been lately included in recent compiler optimizer modules using polyhedral or unimodular transformations [34, 79]. However, transformations applying parallel design patterns or algorithmic skeletons, have not been yet widely explored. In this line, we encounter works proposing pattern rewriting rules [43], commercial frameworks to introduce parallelism using structural refactoring steps [10, 68] and projects that aim to develop advanced refactoring frameworks [35]. Other works combine the refactoring and optimization techniques. For instance Aldinucci et al. [2] propose a technique to minimize the service time of stream parallel pattern compositions by applying systematic rewritings. Also, some of the rewriting and tuning techniques in the literature have been embedded into skeleton-based programming frameworks [52].

2.5 Routine implementation selection techniques.

Since heterogeneous platforms have spread across the scientific community, different implementations of the same algorithm targeted to specific processor architectures have been developed. For example, several libraries comprising highly-tuned numeric kernels (e.g. BLAS or LAPACK implementations), are available for different processors, e.g., cuBLAS [65] for nVidia GPUs, Intel MKL [39] for multi-/many-core processors, etc. This situation reveals as a new challenge the selection of the most suitable device and routine implementation to solve a given problem. To tackle this issue, two different approaches have been traditionally taken: *i)* runtime schedulers that are able to map kernels from multiple libraries on processors available in a heterogeneous platform, and *ii)* static tools that select at compile time the most appropriate implementation according to past knowledge.

Some research works using static approaches can be found in the literature. For instance, the work presented by Jun et al. [84] proposes an automatic system based on source code analysis, which maps user calls to optimized kernels. Additionally, Jie Shen et al. [80] propose an analytic system for determining which hybrid

programming configuration is optimal for a given problem. Likewise, the work by Zhong et al. [89] proposes a solution that uses functional performance models (FPMs) of processing elements and FPM-based data partitioning algorithms to obtain ideal data partitions among the processing units in a heterogeneous platform. The approach in this thesis, however, differs from the latter by the fact that it bypasses data partitioning techniques but selects the kernel implementation that performs best on any of the processing units.

On the other hand, dynamic approaches are also greatly extended in the community. Particularly, the OmpSs [20] programming framework leverages an extended set of OpenMP-like pragmas to support asynchronous parallelism and exploit task-parallelism of applications via data-dependencies. Concretely, among the available pragmas, the `target` directive allows developers to select the target device in a heterogeneous platform. Together with this directive, the `implements` clause lets users specify that the annotated code is an alternate implementation of a given function. This feature allows its *versioning* runtime scheduler to freely map the same task onto different devices. Other works in this line, like the extension for the SkePu framework, presented in [21], take advantage of machine learning techniques to automatically select the most appropriate implementation of a given function. These models basically carry out a tuning phase for estimating the ranges in which different implementations perform better than others. Following a similar approach, the implementation selector framework presented in this thesis gets hints from the user-code C++ attributes in order to select among implementations and processors available in the heterogeneous platform. Using the dynamic mode, applications compiled with our framework are able to select the most appropriate implementation at run-time based on a decision tree that is generated at compile time. For that purpose, the proposed approach requires a previous training phase in order to find out which implementation performs best for a given problem size.

2.6 Summary

In general, we have identified several difficulties when dealing with parallel programming and transforming existing sequential application into parallel. First, in order to transform an existing sequential application into parallel requires analyzing the original source code in order to detect the potential candidate regions. This task has been recognized as time-consuming and error prone task by multiple authors. Secondly, parallel programming presents some challenges related to identify the target architecture and to select the most suitable parallel framework for such architecture. Additionally, programming parallel applications efficiently require additional expertise and a deeper knowledge of the existing frameworks. Finally, the use of already existing parallel and optimized algorithms reveals that multiple implementations of the same routine behave differently depending on the target architecture and problem size. In these situations, deciding which implementation alternative becomes important in order to achieve an increased application performance.

This thesis aims to minimize the impact of these challenges by:

1. To propose automated techniques to detect parallel patterns in sequential code, diminishing the efforts required to analyze the code. This way, the potential candidates to be refactored via parallel patterns will be annotated in the source code.

2. To define a unified interface for parallel patterns that acts as a layer between the users and the different existing parallel programming framework and, therefore, easing the development of parallel applications.

3. To develop automated transformation and optimization techniques in order to generate parallel and optimized code using the generic and unified interface proposed in this thesis. With this tool, the refactoring process will become semi-automatic and, thus, reducing the efforts to transform the code.

4. To develop a mechanism to select the most suitable implementations among different routine implementations depending on the problem size in order to provide, with a single interface, the most suitable version on each case.

Chapter 3

Static parallel pattern detection

As previously introduced, one of the main challenges when transforming sequential code into parallel code is to analyze the code and identify the code regions that can be potentially parallelized. Indeed, this task requires big efforts in terms of time and it is error-prone. Although there are tools that are able to detect some of these regions, they lack high-level parallel pattern detection. This chapter describes one of the main contributions of this Thesis, a tool that is able to detect and annotate parallel patterns in sequential C/C++ source codes. Specifically, Section 3.1 introduces the Clang compiler part of the LLVM infrastructure and the abstract syntax tree employed to detect parallel patterns. Afterward, Section 3.2 describes the annotation format specification for the detected patterns. Section 3.3 describes the workflow and techniques for detecting and annotating parallel patterns in sequential codes. Finally, Section 3.4 shows the evaluation results for PPAT using different benchmarks.

3.1 LLVM infrastructure and the Abstract Syntax Tree

This section presents the main components leveraged to implement the parallel pattern detection tool: *LLVM infrastructure* and the *Abstract Syntax Tree* (AST). The Low Level Virtual Machine (LLVM) compiler infrastructure project is a collection of compiler and tools used to develop different compiler front ends and back ends [48]. In this sense, the tools provided by this infrastructure provide a way to implement a wide range of utilities ranging from compile-time analysis tools and optimizers to run-time instrumentation analysis.

In general, this infrastructure provides three different components: *i)* LLVM Intermediate Representation, a platform independent representation, that allows applying transformations, optimizations and to instrument the code; *ii)* Compiler-rt, a runtime library that provides the necessary components to instrument the code; *iii)* Clang, a compiler that supports multiple languages, such as C, C++, Objective-C and Objective-C++. In this thesis, we take advantage of the Clang compiler since this component provides a collection of tools that allows performing static analysis of the source code and to instrument the LLVM IR during its generation. Specifically, the Clang compiler performs multiple actions from the original source code until generating an unoptimized IR, that will be optimized applying multiple transformations by the LLVM IR module. The generation of this unoptimized IR can be divided into three different stages: *i)* preprocessing, parsing and lexical analysis; *ii)* semantic analysis and generation of the Abstract Syntax Tree (AST); and *iii)* generation of the IR from the AST, namely CodeGen. In order to develop static analyzers, this compiler provides a collection of utilities that can be used to retrieve the result of this process at a given point. For instance, in order to analyze the source code and perform source-to-source transformations, Clang provides a set of classes and functions in order to obtain the AST from the source code and traverse this tree in

different ways. Thanks to this features, Clang can be used to develop the parallel pattern detection tool proposed in this Thesis by traversing and analyzing the AST generated by Clang.

In order to explain the parallel pattern detection techniques implemented in this Thesis, it becomes necessary to detail the Abstract Syntax Tree. The AST is a syntactic structure representation of the source code in a tree model [62]. This tree is composed of different nodes that represent each of the different statements that are present in the source code, e.g. variable declarations, function calls, loops, etc. This nodes contains different semantic and syntactic information about these statements such as the variable names, implicit castings from *r-value* to *l-value*, function names or location in the source. Figure 3.1 shows an example of source code along with its associated AST.

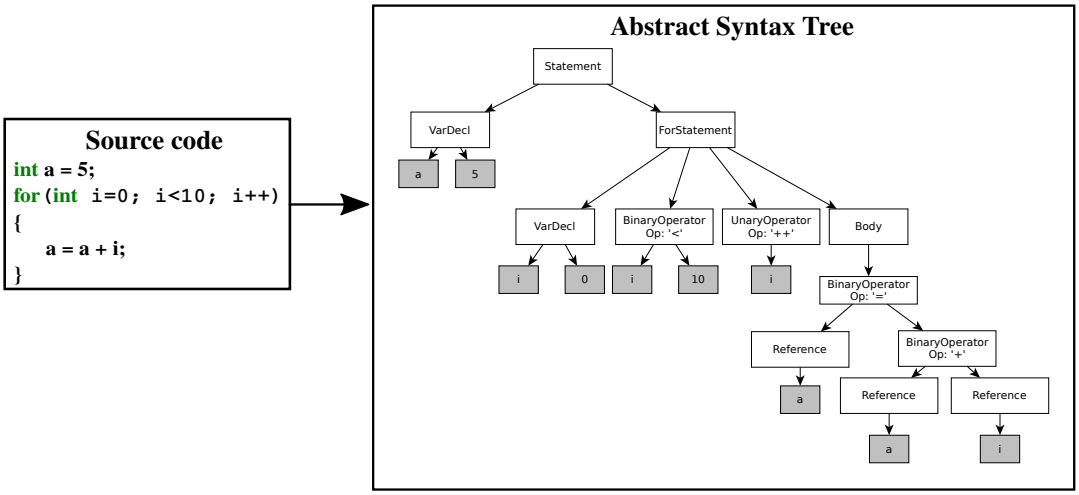


FIGURE 3.1: Example of Abstract Syntax Tree for a given source code.

3.2 Parallel Pattern Annotation Specificaction

In order to annotate parallel patterns using custom C++11 attributes [41], we have extended the set of attributes defined for the projects REPARA [23] and REPHRASE [24]. Table 3.1 describes the attributes used for annotating the Pipeline, Farm and Map parallel patterns.

TABLE 3.1: REPHRASE attributes.

REPHRASE Attribute	Description
`rph::pipeline`	It identifies a pipeline pattern.
`rph::stream`	This attribute identifies the data streams used across stages of a `rph::pipeline`.
`rph::stage`	It identifies a code section as a pipeline stage.
`rph::plid`	It is associated to `rph::stage` and includes the pipeline ID.
`rph::farm`	This attribute specifies the farm pattern.
`rph::map`	It determines the map pattern.
`rph::in`	This attribute references the input variables of a pattern.
`rph::out`	It references the pattern output variables.

Thanks to these attributes, a refactorization tool would have enough information to transform annotated code regions into parallel. Also, they allow to split the detection and transformation processes, so different tools can be used in these stages.

3.3 Parallel Pattern Analyzer Tool

In this section, we describe the Parallel Pattern Analyzer Tool (PPAT). This tool takes advantage of the Clang library to generate the Abstract Syntax Tree. Then, it walks through it in order to collect relevant information about the source code and identify parallel patterns.

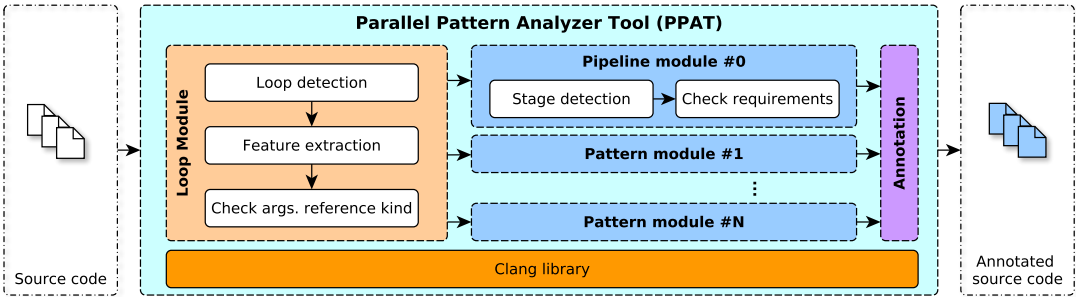


FIGURE 3.2: Workflow diagram of PPAT.

Figure 3.2 depicts the general workflow diagram of PPAT. First, the tool receives the sequential source code files that should be analyzed. Next, the following steps are executed:

1. *Loop detection.* This step is in charge of traversing the AST generated by clang. During this process, the tool gathers information of different AST nodes (e.g. variables, function calls, conditional statements) and identifies loop-related subtrees for the pattern analysis. Additionally, it collects information about the functions implemented in the source code. To illustrate the detection process, Listing 3.1 shows a code example with two loops. This code snippet multiplies each value stored in a matrix by a value `k` that is incremented in each row. In this stage, the tool gathers the information related from the different AST nodes and generates an internal representation of the AST that only contains the relevant information. This transformation eases the analysis performed in the next steps and allows to maintain the AST among compile units. Additionally, detected loop sub-trees are passed to the next phase in order to properly mark the loops locations.

LISTING 3.1: Code example.

```
print_value(auto & value);
...
for(int i=0;i<100;i++){
  for(int j=0;j<100;j++)
  {
    matrix[i][j] = k * matrix[i][j];
  }
  print_value(k);
  k++;
}
```

2. *Feature extraction.* This step leverages the structures collected in the previous step in order to extract specific features about variable declarations, references, function calls, inner loops, memory accesses, operations, etc. Next, for each

statement, it stores information about the location on the original code, variable and functions name, reference kind (Write or Read) and global storage references. This step is similar to the aforementioned information collection but only for the loop sub-trees, that are marked in order to ease the analysis of each pattern module. For instance, on the inner loop's body of the example code, the tool will detect two references to `matrix` and one to `k`. For both `matrix` references, the tool will also store the information about array subscripts operators to determine that are accessing to the position (i, j) of the array. Afterwards, analyzing the binary operators the first reference to `matrix` will be annotated as Write since it is on the left hand of the ≝öperator. On the other hand, both `k` and `matrix` references are marked as Read references.

3. *Check arguments reference kind.* The last step checks whether the kinds of variable references passed as arguments in functions can be determined or not. In some cases, it is not possible to know statically if the kind of arguments passed by reference is read or written. When this occurs, the tool performs the following actions:

   (a) If the function code is available or the function is implemented in the user code, it is possible to check the set of arguments and assign the right variable kind (Write or Read). If an argument is not modified, it is considered as Read. In contrast, if there exist write accesses to the variable, Write is assigned as the variable kind. Alternatively, if there is a read-after-write (RAW) dependency on a variable, the kind is set to Write/Read, since the argument can generate potential feedbacks among iterations.

   Coming back to the example in Listing 3.1, in the outer loop the function `print_value` receives a reference to a type as a parameter. If this function is defined in the user code, it can be analyzed to detect if the value is modified and, if so, the parameter is annotated to let the analysis modules know that it may produce side effects. Otherwise, if no write operation is performed on the parameter in the function code, it will be annotated as a read-only parameter.

   (b) On the contrary, if the function cannot be accessed, it is not possible to check the actual variable kind. Thus, the tool takes a conservative decision: it sets the arguments kinds always to Write/Read. Despite this, it inserts the function name and parameter kinds into a dictionary file in order to improve the detection process in future analyses. Afterwards, the user can eventually modify this dictionary to set the right parameter kinds for these specific functions.

   In the example, if the `print_value` function is defined in a library and can not be accessed the parameter will be annotated as if it were modified and will be introduced in the dictionary. Then, the user can modify the dictionary to set the parameter to not modified for a future analysis.

Next, marked loops are passed to the different pattern analyzer modules. Finally, the parallel patterns found are forwarded to the annotation module responsible for inserting the above-described REPHRASE attributes in the corresponding loops. In the following sections, we describe the parallel pattern analyzer modules that are currently supported by the PPAT.

3.3.1 Pipeline Detection Module

In this section, we detail the internal workings of the Pipeline detection module. As defined in previous section, this pattern defines a code that can be split into stages and run in parallel by different threads, so that the output of a stage is the input of the next one. The requirements for a Pipeline to be detected are the following:

- *No global variables can be modified*. In other words, there should not exist instructions that write on global variables. For instance, if we consider `k` as global variable, the outer loop in Listing 3.1, cannot be treated as a parallel pattern since `k` is modified in line 9.

- *No feedback*. This requirement controls that no feedback exists between iterations of the loop. To do so, it checks if there are no variables written before they are read, i.e., there are no RAW dependencies. Focusing on the example, the outer loop will be discarded as potential Pipeline due to loop-carried dependencies in variable `k` which is read in line 6 and 8 and modified in line 9.

- *Multiple stages*. The last requirement checks whether the potential pipeline can be split in, at least, two stages. Otherwise, the loop cannot be treated as a parallel pipeline and PPAT discards it right away.

  The current strategy to split a loop into stages is to create a new stage each time a function call or an inner loop is found in the main loop. Afterwards, for each stage encountered, PPAT checks whether the stage is fed with, at least, a previous stage output. If this is not the case, the complete stage is merged with the previous one until all stages comply with the requirement. Note that this strategy assumes that each stage has a substantial amount of work, however, if function calls or nested loops inside a stage have negligible workloads, the tool may identify a Pipeline with unbalanced stages. Finally, PPAT checks for the presence of another parallel pattern in the stages codes. If so, the corresponding stages are annotated as well.

  In order to illustrate the stage splitting, Listing 3.2 shows a code in which each position of a vector is computed as the averaged sum of the corresponding position of an input vector multiplied by the values in a second vector. Afterwards, each position of `vector_out` and a counter are printed out by calling the function `print_value`. In this example, the loop will be initially divided into three stages: from line 2 to 5, line 6 and line 7. Afterwards, the tool will check if the output from the first stage is used in the second one. Since the second stage uses the `vector_out[i]` that is modified in the previous stage, the first stage will be maintained. Next, stages 2 and 3 are merged together because the second stage does not modify any variable used in the third stage. Finally, since all the Pipeline requirements are met, the main loop will be annotated as a Pipeline. Additionally, the tool will insert the corresponding annotations to identify the first stage (from line 2 to 5) and the second stage (lines 6 and 7).

LISTING 3.2: Pipeline code example.

```
1 for(int i=0;i<100;i++){
2   for(int k=0;k<100;k++)
3   {
4     vector_out[i] += vector2[k] * vector[i]/100.0;
5   }
6   print_value(vector_out[i]);
7   print_value(counter++);
8 }
```

3.3.2 Farm Detection Module

This pattern defines a loop that can be run in parallel by different threads over a data stream. In this case, the code analyzed should be equivalent to a pure function, i.e., there should not exist data dependencies producing potential side effects. This requirement is controlled using the following constraints:

- *No RAW dependencies*. There should not exist RAW dependencies of variables used within iterations of the loop.
- *No global variables are modified*. There should not exist instructions that modify global variables in the loop.
- *No break statements*. There should not exist break statements (i.e., continue, break or return) in the loop, as they cannot be parallelized. However, they may be placed in inner scopes of the main loop.

Additionally, if a pipeline stage detected by the previous module fulfill the aforementioned requirements, the stage is annotated as a farm pattern. In this cases, the given stage has been proved to not produce side effects nor having data dependencies and, therefore, it can be executed in parallel.

3.3.3 Map Detection Module

This section describes the implementation of the Map detection module within PPAT. Basically, the Map pattern represents a parallel code executing a pure function that is responsible for generating the output elements. Note that in this case the total number of input elements is known in advance. To ensure these requirements, the Map pattern adds the following two constraints over the requirements of the Farm pattern:

- *Known number of input elements*: The input data must be declared and allocated before the definition of the analyzed loop. In order to check this requirement, the tool analyzes the scope of the variables referenced in the loop and selects those that are declared in an outer scope. Next, these variables are classified as input, if all its references kind are Read, or as output, if in at least one reference modifies the value of the variable. Afterward, having this in mind, the tool evaluates if the loop condition depends on an input variable or if it depends on a function. If the condition depends on a variable, it cannot be modified during the computations of the loop. On the contrary, if the condition depends on a function, this function should return always the same value, i.e. the function should be pure and its arguments cannot be modified.
- *At least one output*: According to the previous Map definition, the pure function of the Map pattern processes an input to produce an output. So, the set of outputs for a given loop should not be empty.

To illustrate the Map detection module process, Listing 3.3 shows a code snippet that multiplies each value stored in a vector by two. First, the tool looks for the variables referenced in the loop's body that comes from an outer scope. In this case, variables `i` and `vector` are classified as inputs and `vector_out` as output since it is modified. Afterward, the tool checks the loop's condition. Since the condition depends on a literal, the number of iterations is known in advance and, thus, fulfilling

the first constraint. Finally, the tool checks for the second restriction that compares the inputs with the output dependencies. In this case, since the output (`vector_out[i]`) depends on an input value (`vector[i]`) the loop is annotated as a Map pattern.

LISTING 3.3: Pipeline code example.

```
1 for(int i=0;i<100;i++)
2 {
3   vector_out[i] = 2 * vector[i];
4 }
```

3.4 Evaluation

To evaluate PPAT, we used the sequential versions of the two scientific benchmark suites: `Rodinia` [81] and `NAS` Parallel Benchmarks (NPB) [7]. We also leverage a processing video application taken from the FastFlow framework [16] as a real use case. Our evaluation methodology is based on a comparison between a manual inspection and an automatic one, using PPAT, of the loops appearing in the benchmarks. To conduct a double-blind study, the manual inspection is performed before the automatic one, so that the manual results are not biased by those from PPAT. For each benchmark, we collect the number of loops and parallel patterns detected. Then, we discuss the results collected during the manual inspection with those obtained by PPAT in order to demonstrate the quality of the pattern detection process.

Moreover, we transform the sequential code of the `Rodinia` benchmark tests using the PPAT annotations, and then, compare the performance of the PPAT parallel versions with respect to the parallel ones provided by the benchmarks suites. Finally, we test PPAT on a real use case in order to evaluate the quality of the pattern detection. The results obtained for this test are contrasted with the FastFlow parallel version.

3.4.1 Reference platform

The evaluation has been carried out on a server platform comprised of 2× Intel Xeon Ivy Bridge E5-2695 v2 with a total of 24 cores running at 2.40 GHz, 30 MB of L3 cache and 128 GB of DDR3 RAM. The OS is a Linux Ubuntu 14.04.2 LTS with the kernel 3.13.0-57. This platform also incorporates a NVIDIA Tesla K40c with 12 GB and GeForce GTX680 GPUs with 2 GB of DDR5 RAM. These GPUs are denoted as `GPU0` and `GPU1`, respectively. The OS is a Linux Ubuntu 14.04.5 LTS with kernel 3.13.0-85.

3.4.2 Results for the benchmarks suites

As mentioned, the two benchmark suites used to evaluate PPAT are `Rodinia` and `NAS`. Note that we only employ the sequential versions of these benchmarks to detect potential parallel patterns. Tables 3.2 and 3.3 presents the results obtained by PPAT and manual inspection for both `Rodinia` and `NAS` benchmarks. As can be seen, the number of patterns detected manually and through PPAT are perfectly matching. Therefore, we observe that the pattern detection quality of PPAT is close to that performed by a human expert.

[ht]

Focusing on the differences between manual and automatic detection, as shown in Tables 3.2 and 3.3, the human expert is able to detect more Farm patterns than the

TABLE 3.2: Results for the Rodinia benchmark suite. P, F and M stand for the number of Pipeline, Farm and Map patterns detected, respectively.

		PPAT			Manual		
Test	**Loops**	**P**	**F**	**M**	**P**	**F**	**M**
b+tree	80	3	7	7	2	7	7
particlefilter	44	1	8	8	1	10	10
bfs	7	0	1	1	0	2	1
nw	12	0	6	6	0	6	6
cfd	78	16	12	12	15	13	13
lavaMD	10	0	1	1	0	2	2
heartwall	54	1	4	2	0	4	3
nn	2	0	0	0	0	0	0
backprop	28	0	2	2	0	5	5

TABLE 3.3: Results for the NAS benchmark suite. P, F and M stand for the number of Pipeline, Farm and Map patterns detected, respectively.

		PPAT			Manual		
Test	**Loops**	**P**	**F**	**M**	**P**	**F**	**M**
IS	16	1	8	8	0	9	9
LU	187	1	37	37	1	81	81
FT	41	0	7	7	3	20	20
EP	8	1	2	2	0	3	3
MG	80	1	26	26	1	44	44
UA	321	3	116	116	2	171	170
DC	30	2	5	5	1	7	7
SP	250	1	51	51	1	103	103
BT	181	1	46	46	1	78	78

tool for some of the tests. These differences mainly occur when the tool is not able to guarantee the parallel correctness of the pattern when shared variables are used. Listing 3.4 shows an example of this situation in a non-annotated Farm-like pattern. In this case, PPAT detects that the variable `new_dx` and iterators `j` and `k` are shared and, therefore, the tool cannot ensure that the code corresponds with a parallel pattern. However, PPAT lets the user know that, if these variables had been declared as local, the code would have corresponded with a parallel pattern. Listing 3.5 shows a version of the code in which we have used shared variables on purpose to demonstrate how the Farm and Map patterns would have been introduced. We observed in the annotated loops that this situation happens in many cases for the NAS benchmark tests, as the loop iterators are declared right before the loop sentences. Although the PPAT is not able to discover some parallel patterns, the most time consuming regions that matches with a parallel patterns have been detected by the tool·

In order to analyze the benefits of PPAT on the patterns detected, we have implemented parallel versions of the Rodinia tests following parallelization suggestions given by PPAT. Both Farm and Map patterns were implemented using OpenMP, while the Pipeline pattern was introduced using the corresponding Intel TBB construction using "serial in-order" stages. Note that we have transformed all parallel loops suggested by PPAT, even the nested ones. Figure 3.3 shows performance results of

LISTING 3.4: Original **backprop** snippet.

```
363
364
365 for (j = 1; j <= ndelta; j++)
366
367
368  for (k = 0; k <= nly; k++) {
369   new_dw = ((ETA * delta[j] * ly[k]) + (
          MOMENTUM * oldw[k][j]));
370   w[k][j] += new_dw;
371   oldw[k][j] = new_dw;
372  }
```

LISTING 3.5: Annotated **backprop** snippet.

```
[[rph::map, rph::farm,
  rph::in(nly,delta,ly,oldw,w), rph::out(w,oldw)]]
for (int j = 1; j <= ndelta; j++)
 [[rph::map, rph::farm,
   rph::in(delta,ly,oldw,w,j), rph::out(w,oldw)]]
 for (int k = 0; k <= nly; k++) {
  float new_dw = ((ETA * delta[j] * ly[k]) + (
       MOMENTUM * oldw[k][j]));
  w[k][j] += new_dw;
  oldw[k][j] = new_dw;
 }
```

the sequential, PPAT parallel and OpenMP versions for the `Rodinia` benchmark suite. In all cases, the tests were executed with the default input parameters, using 24 threads to fully populate the multi-core machine. For brevity, we only highlight some interesting cases. For the `lavaMD` test, we note that PPAT is not able to annotate one of the main application hotspot (or loop) as a parallel pattern. This is mainly because some of its instructions, operating on sparse datasets, rely on indirect memory accesses in the form of `A[B[i]]`. In these cases, PPAT is not yet able to detect, at compile time, such potential data dependencies among loop iterations. Regarding the `heartwall` test, the PPAT version detects more parallel loops, or patterns, than those parallelized originally in the OpenMP version. This situation leads to increased overheads due to loops without a considerable workload have been parallelized.

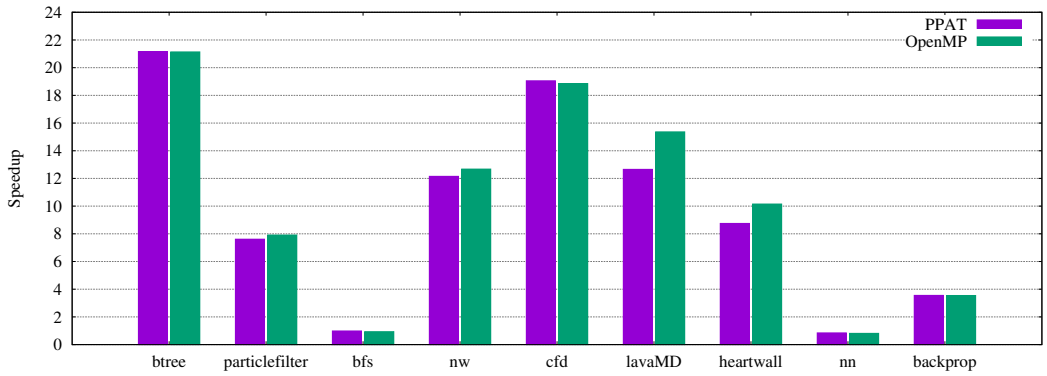


FIGURE 3.3: Execution time of sequential, transformed PPAT code and OpenMP versions of `Rodinia` benchmark.

After this study, we make the following observations: *i)* PPAT obtains close performance figures with respect to the OpenMP implementations, as the PPAT annotations correspond with the OpenMP original pragmas in most cases; and *ii)* PPAT annotated versions may add slight overheads, as they contain initialization loops that were annotated to run in parallel, while in the original versions were vectorized by the compiler optimizations.

3.4.3 Analysis of the Fastflow use case

Finally, we test PPAT using a video processing use case taken from the FastFlow framework. Basically, this application processes a stream of video frames captured by a camera and applies, to each of them, two different image processing filters: Gaussian blur and Sobel filters. Listing 3.6 shows the results of the analysis performed by PPAT. As observed, the tool is able to detect a Pipeline comprised of 4 stages that operate in the following way. The first stage detected captures the input video frames and forwards them to the next stage. The second and third stages are

responsible for applying, in series, the two image processing filters Gaussian blur and Sobel, respectively. The last stage takes care of delivering the frames processed to the user. Another observation is that PPAT is able to determine as well that the filtering stages can be parallelized using Farm patterns individually. Therefore, multiple threads can execute the filters over the frames concurrently without side effects, as these operations have been determined to be pure functions.

LISTING 3.6: Example of annotated loop from the video use case.

```
[[rph::pipeline(0) , rph::stream(cap,frames,frame,frame1)]]
for(;;) {
 class cv::Mat frame1, frame;
 [[rph::stage(0), rph::plid(0), rph::in(cap), rph::out(frame1,frame,cap)]]{
  if(cap.read(frame) == false) break;
 }
 [[rph::stage(1), rph::plid(0), rph::farm, rph::in(frames,filter1,frame,frame1),
   rph::out(frames,frame1,frame)]]{
  frames++;
  if(filter1) {
   cv::GaussianBlur(frame, frame1, cv::Size(0, 0), 3);
   cv::addWeighted(frame, 1.5, frame1, -0.5, 0, frame);
  }
 }
 [[rph::stage(2), rph::plid(0), rph::farm, rph::in(filter2,frame),
   rph::out(frame)]]{
  if(filter2) Sobel(frame,frame,-1,1,0,3);
 }
 [[rph::stage(3), rph::plid(0), rph::in(outvideo,frame)]]{
  if(outvideo) { imshow("edges", frame); if(waitKey(30) >= 0) break;}
 }
}
```

In this particular case, we note that the parallel patterns detected are exactly the same as those used in the implementation of the original parallel version. Therefore, we believe that PPAT will be able to minimize the development costs of parallel C/C++ applications by means of detecting regions that can be represented as parallel patterns. However, since the presented techniques perform static analysis, they have limitations when dealing with aliasing or access to double-indexed matrices. In these situations, PPAT is not able to correctly detect data dependencies and, therefore, it may lead to patterns not found by the tool.

3.5 Summary

In this chapter, we have proposed a tool that is able to discover potential parallel patterns in sequential source codes. This way, the required efforts to analyze the original source code can be diminished thanks to PPAT. However, up to this point, the transformation from sequential to parallel is still manual, and its difficulties are still present. Transforming these codes requires additional expertise and knowledge from the developers in order to select the most suitable framework. Additionally, since there is not a unified interface for parallel programming migrating parallel code from one framework to another is not straightforward. To pave the way, in the next chapter, we propose a generic and reusable parallel pattern interface that acts as a switch between existing frameworks.

Chapter 4

Generic Parallel Pattern Interface

As mentioned, one of the best codifying practice in order to write robust, portable and efficient code is to use parallel patterns. In this sense, as studied in Chapter 2, although parallel programming frameworks aim to simplify the development of applications, there is not a unified standard [32]. In order to mitigate this situation, this Chapter presents GrPPI, a generic and reusable high-level C++ parallel pattern interface that comprises both stream and data-parallel patterns.

In general, the goal of GrPPI is to accommodate a layer between developers and existing parallel programming frameworks targeted to multi-core and heterogeneous platforms. Basically, GrPPI allows users to implement parallel applications without having a deep understanding of existing parallel programming frameworks or third-party interfaces and, thus, relieves the development and maintainability efforts. In contrast to other object-oriented implementations in the literature, we use C++ template meta-programming techniques in order to provide generic interfaces of the patterns, inspired by the Parallel STL, without incurring significant runtime overheads. Furthermore, the modularity of GrPPI permits to easily integrate new patterns, while combining them to arrange more complex ones. Thanks to this property, GrPPI can be used to implement a wide range of existing stream-processing and data-intensive applications with relatively small efforts, having as a result portable codes that can be executed on multiple frameworks.

Specifically, Section 4.1 and 4.2 describe in detail two of the main components of the GrPPI library, the execution policies and the communication channels. Afterward, Section 4.3 describes the interface for the supported parallel patterns and building blocks. Next, Section 4.5 demonstrates the composability feature of grppi. Finally, Section 4.6 shows an experimental analysis of GrPPI from the usability and performance points of view using different benchmarks.

4.1 Execution policies

This section presents the execution policies, a set of types designed to include the pattern implementation using different frameworks. In this sense, each type contains the framework-specific implementations along with the different configuration information. Then, these policies can be used as an argument of the parallel patterns interface in order to select the desired back-end. Thanks to these types, we are able to provide a unified interface hiding away the complexities of the framework used underneath. Furthermore, since this types are used to select a given framework, changing from one back-end to other only requires modifying a single argument in the pattern function call, thus, improving the portability of the application.

In the current version, we provide four different execution policies related to the different supported back-ends: sequential, C++ Threads, OpenMP and Intel TBB.

- **sequential_execution**: The sequential execution policy allows to improve the maintainability and code structure by expressing an application in terms of patterns, but without providing any parallelism. So, this mode is useful when using a parallel implementation does not enhance performance but the use of parallel patterns improves the readability. Additionally, this policy along with the parallel ones allows expressing parallel and sequential algorithms in the same way. Listing 4.1 shows the sequential_execution class definition along with its default constructor since this execution policy has no configuration parameters.

LISTING 4.1: Sequential execution policy.

```
class sequential_execution {
  public:
    constexpr sequential_execution() noexcept = default;
    ...
}
```

- **parallel_execution_native**: This policy leverages C++ threads to implement the different patterns. In this case, each thread is in charge of processing a different subset of the input data for data parallel patterns and processing a given stage or task for stream parallel patterns. This specific policy can receive an integer for establishing the number of threads that will be used in a pattern and maintains an internal thread ID table to allow users to ask for the ID of a given thread.

  Listing 4.2 shows the parallel_execution_native class definition along with its constructors. In this case, the native execution policy can be configured with two different arguments: tthe *concurrency degree* to limit the number of threads that will be used for executing a given pattern and the *ordering* to determine if the items in the output stream will be ordered.

LISTING 4.2: C++ threads execution policy.

```
class parallel_execution_native {
  public:
    parallel_execution_native() { ... }
    parallel_execution_native(int concurrency_degree, bool ordering=true) { ...}
    ...
}
```

- **parallel_execution_omp**: This execution policy uses the task-based parallel framework for implementing the different patterns. Specifically, this policy provides the parallelism by explicitly declaring OpenMP or by using the abstractions provided by this frameworks. For instance, stream patterns use explicit tasks for each of the stages, while some data patterns like the Map are implemented by using the parallel_for abstraction. Similar to the native policy, this policy may also receive the number of OpenMP threads that will be used for executing a given pattern. However, in this case, the policy does not maintain an ID table since these IDs are internally managed and provided by the OpenMP framework itself.

  Similar to the native execution policy (See Listing 4.3), this type has different constructors for defaulted values as for the configuration parameters and for user-defined values. However, the default concurrency, in this case, is defined by the environment variable `OMP_NUM_THREADS`.

LISTING 4.3: OpenMP execution policy.

```
class parallel_execution_omp {
 public:
   parallel_execution_omp() { ... }
   parallel_execution_omp(int concurrency_degree, bool ordering=true) { ...}
   ...
}
```

- **parallel_execution_tbb**: This policy takes advantage of the High-level parallel framework Intel TBB. In this case, the implementation of each pattern uses the corresponding implementation of Intel TBB. However, some patterns are not natively provided by TBB. On these cases, similar to the aforementioned back-ends, the implementation of such patterns is made by means of Intel TBB tasks. Focusing on the number of threads, in this case, the control of threads only corresponds with the employed tasks on those patterns that are not provided by TBB. On the contrary, as for the natively supported patterns, the thread management relies on the runtime scheduler provided by the framework. Listing 4.4 shows the constructors of the Intel TBB execution policy that share the behavior of the native policy.

LISTING 4.4: Intel TBB execution policy.

```
class parallel_execution_tbb {
 public:
   parallel_execution_tbb() { ... }
   parallel_execution_tbb(int concurrency_degree, bool ordering=true) { ...}
   ...
}
```

4.2 Communication channels

Since the lower-level back-ends (OpenMP and Native) does not provide any high-level mechanism to safely communicate data among threads, it is necessary to introduce a custom communication channels. This is important for those patterns in which data is not available at the beginning and should be transmitted among threads during the execution, i.e. for the stream parallel patterns. On the contrary, for data parallel patterns the distribution of the data is done by means of dividing the data into independent subsets. Focusing on the stream communication channels, we provide a First-In-First-Out (FIFO) queue implemented as a circular bounded buffer that supports concurrent accesses for multiple producers and multiple consumers inspired by the Michael and Scott's lock-free queue [58]. This queue provides the following set of function to access the queue:

- `push`: Enqueues the item into the buffer in the first empty position.
- `pop`: Removes and returns the first item in the buffer.
- `empty`: Returns true if the buffer is empty.
- `full`: Returns true if the buffer is full.

Figure 4.1 depicts the internal working of the circular buffer from the FIFO Multiple-Producer/Multiple-Consumer (MPMC) queue. Initially, `pread` and `pwrite` point to the initial position of the buffer, while some elements have been added at the terminal position through `push` calls and others removed from the head position by means of `pop` calls.

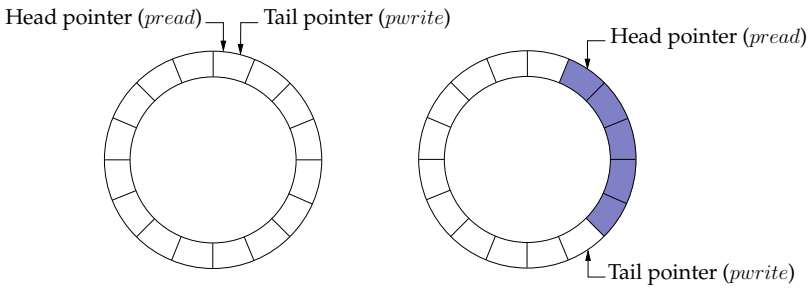


FIGURE 4.1: Schema of the circular buffer queue.

Listing 4.5 shows the GRPPI MPMC queue class definition. This class is defined as a template class to support any type as the item type. Additionally, this type is stored, similarly to the C++ standard container, in the value_type. This type is used in the pattern implementations to allow determining input and output types of the user functions.

LISTING 4.5: MPMC queue.

```
template <typename T>
class mpmc_queue{
 public:
   using value_type = T;
   mpmc_queue<T>(int q_size, queue_mode q_mode ) {...}
   bool is_empty () const noexcept;
   T pop ();
   bool push (T item);
}
```

In order to control the access for multiple producers and multiple consumers, we employ two different techniques to support both *blocking* and *lock-free* behaviors. This way, depending on the pattern construction, target platform and concurrency degree, developer can select the most suitable control mechanism. For instance, when dealing with oversubscription, i.e. using more threads that available cores, the *blocking* strategy may outperform the *lock-free* implementation since suspended threads leave the CPU available for other threads that may exploit such resources. Focusing on the *blocking* implementation of the queue, the synchronization among producers and consumers is performed by means of conditional variables and mutexes. In this specific case, if the queue is empty or full, the consumers and producers will wait until a new item is produced or consumed, respectively.

On the other hand, as for the *lock-free* implementation, we use atomics for both `pread` and `pwrite` pointers to avoid potential data races. Concretely, the access to the queue is controlled by Compare-And-Swap (CAS) operations. With these operations on both pointers, we can control the access of producers and consumers. In addition to these pointer, we define two new atomic internal pointers related to both consumers and producers. Thanks to these new pointers we can control the concurrent access for multiple producers and multiple consumers avoiding data races. For instance, when a consumer requires an item from the queue, first it gets a unique index by performing CAS operations over the internal `pread` pointer. Afterward, this consumer will use the index obtained to access the queue and it will check if the corresponding item is ready in the queue. If it is ready, it takes the element and using again CAS operations updates the value of the `pread` pointer.

Finally, in order to select the implementation of the queue and the size of the circular buffer, the execution policies provide the `set_queue_attributes` function

that allows selecting these parameters. This function receives two arguments, an integer for establishing the size of the circular buffer and a enumerate value (`queue_mode`) that can be *blocking* or *lock-free*. Note that this configuration can not be modified during the execution of a pattern and the queues involved in such pattern will keep using the configuration established before starting the computation of the pattern.

4.3 Description of the pattern interfaces

This section describes the interface carefully designed to allow composability and to support multiple implementation back-ends. In the current version, GRPPI offers stream, data and task parallel patterns with a single interface. Furthermore, GRPPI also supports some stream operators offering new semantics for stream applications and a set of advanced patterns simplifying some domain-specific or complex constructions.

4.3.1 Data patterns

This section describes in detail the interfaces for the data parallel patterns Map, Reduce, Stencil, MapReduce and Divide&Conquer supported by GRPPI.

Map The GRPPI interface for the Map pattern, shown in Listing 4.6, receives the following input parameters: the execution policy, references to the first and last elements of the input data collections and the kernel (`map`) function. After the computation, the result of the Map pattern is left in the corresponding position of the output data set. Given that each element in the input data collection is independent to each other, the parallel execution of the Map pattern can be performed in the following way. First, the input collection is divided equally among the available concurrent entities. Afterwards, these entities execute in parallel the kernel `map` function and write the results in the corresponding segments of the output data collection.

LISTING 4.6: Map interface.

```
1 template <typename EM, typename InIt, typename OutIt,
2         typename TaskFunc, typename ... MoreIn>
3 void map(EM m, InIt first, InIt last, OutIt firstOut,
4         TaskFunc &&map, MoreIn ... inputs );
```

Reduce The interface for the Reduce pattern, as described in Listing 4.7, takes the execution policy, a reference to the first and last elements of the input data collection and the reduce operator. The result of the reduction is written in the output parameter passed by reference. According to the properties of the reduce operator, the reduce computation can be performed in parallel. Thus, the input data collection is partitioned in N chunks and computed in parallel by N different concurrent entities that produce a set of partial results. Note that, the reduce operation should be commutative and associative, therefor, the initial value for each partial reduction will be the first element of the corresponding chunk. Finally, the result of the Reduce pattern is calculated in series by one of these entities.

LISTING 4.7: Reduce interface.

```
1 template <typename EM, typename InIt, typename Output, typename ReduceOperator>
2 void reduce(EM m, InIt first, InIt last, Output &out, ReduceOperator && redop);
```

Stencil The GRPPI interface for the Stencil pattern, presented in Listing 4.8, is quite similar to that for the Map pattern, with the exception that it additionally receives the neighborhood (`nh`) function. This function is responsible for accessing the neighbors in a given coordinate of the input data set. The parallel implementation of the Stencil pattern is analogous to that for the Map pattern. However, accessing the neighbors in the boundaries of a partitioned data set might require additional comparisons between the positions of the elements.

LISTING 4.8: Stencil interface.

```
template <typename EM, typename InIt, typename OutIt, typename TaskFunc,
        typename NFunc, typename ... MoreIn>
void stencil(EM m, InputIt first, InIt first, InIt last, OutIt firstOut,
         TaskFunc && stencil, NFunc && nh, MoreIn ... inputs);
```

MapReduce The interface for the MapReduce pattern combines internally calls to the Map and Reduce GRPPI pattern interfaces. As for input parameters, it receives the execution policy, references to the first and last elements of the input data collections, the kernel (`map`) function and the reduction operator for the Reduce pattern. The result is finally left in a reference to the first element of the output collection. The parallel implementation of this pattern in GRPPI exploits the parallelism offered internally by the Map and Reduce parallel patterns. The result of the Map operation is then shuffled and reduced in parallelAfterward, the global result is finally reduced in series by one of the concurrent entities.

LISTING 4.9: MapReduce interface.

```
template <typename EM, typename InIt, typename Output, typename MapFunc,
        typename ReduceOperator, typename ... MoreIn>
void map_reduce(EM m, InIt first, InIt last, Output &out, MapFunc &&map,
            ReduceOperator &&redop, MoreIn ... inputs);
```

Divide&Conquer The interface designed for the Divide&Conquer pattern consists of the following elements: the execution policy, a reference of the input data collection and the `divide`, `base_case` and `merge` functions. The result of this pattern is written to the output data collection passed by reference. The parallel implementation of this pattern in the GRPPI interface leverages first the divide kernel to steadily split the problem into smaller ones. This operation is performed by the available concurrent entities until the minimal problem dimension is reached and where the base-case solution kernel is applied. Taking the partial solutions generated, the concurrent entities merge the results in a tree-based structure until the global solution is obtained. Note that since the tree width can grow above the maximum number of concurrent entities specified, a pool of tasks is used instead in order to implement a dynamic scheduling approach.

LISTING 4.10: Divide&Conquer interface.

```
template <typename EM, typename Input, typename Output, typename DivFunc,
        typename TaskFunc, typename MergeFunc>
void divide_conquer(EM m, Input &problem, Output &out, DivFunc && divide,
        TaskFunc && base_case, MergeFunc && merge);
```

4.3.2 Stream patterns

The GRPPI stream parallel patterns include the Pipeline, Farm, Filter, and Reduce patterns. In this case, stream patterns have been designed to be easily composed

among them. Since all these patterns require a stream generator at the beginning and a stream consumer at the end, when they are composed, both stream generator and consumer are provided by the outer pattern. For this reason, we provide 2 different interfaces for the same pattern, one of them for composing them with other stream patterns and another to work as the outermost pattern.

Pipeline The GRPPI standalone interface designed for the Pipeline pattern receives the execution policy and the functions (`in` and `stages`) related to its stages. As shown in Listing 4.11, its C++ interface uses templates, making it more flexible and reusable for any data type. Note as well the use of variadic templates, allowing a Pipeline to have an arbitrary number of stages by receiving a collection of callable objects passed as arguments.

LISTING 4.11: Pipeline interface.

```
1 template <typename EM, typename InFunc, typename ... Transformers>
2 void pipeline( EM m, InFunc && in, Transformers ... stages );
```

In order to compose the Pipeline pattern inside any other pattern (See Listing 4.12, the interface only receives the function related to the stages without receiving the execution policy. Note that in the composed Pipeline is not necessary to provide a function to generate the stream elements since the stream is produced in the outermost pattern.

LISTING 4.12: Pipeline interface.

```
1 template <typename ... Transformers>
2 auto pipeline( Transformers ... stages );
```

Farm The Farm pattern interface, shown in Listing 4.13, receives the concurrency degree, i.e. the number of replicas used in the farm construction, and the `farm` function. Basically, this pattern is intended to be used along with an outer Pipeline pattern and it performs the following steps: *i)* consumes the items from the input stream coming from the previous stage, *ii)* processes them individually, and *iii)* delivers the results to the output stream. Note that the `farm` function will be executed in parallel by the different concurrent entities. In this case, the execution policy used depends on that used for the outer Pipeline.

LISTING 4.13: Farm interface.

```
1 template < typename TaskFunc >
2 auto farm( int concurrency_degree, TaskFunc &&farm );
```

Filter The interface for the Filter pattern, described in Listing 4.14, only receives the `predicate_op` function. Basically, this pattern reads items from the input stream and forwards them to the `predicate_op` function, which is responsible to determine whether an item should be accepted or not. Afterward, those items that satisfy the filtering routine are delivered to the output stream.Note that it is mandatory the `predicate_op` function to return a boolean expression. However, this pattern is by nature sequential and, in order to be parallel, this pattern should be included in a Farm pattern.

In this case, we provide two interface alternatives *keep* and *discard* depending on which items are filtered out. If the *keep* function is used only those item that the filter evaluates as true are delivered to the output stream. On the contrary, *discard* removes those items evaluated as true.

LISTING 4.14: Filter interface.

```
template <typename Predicate>
auto keep(Predicate && predicate_op);

template <typename Predicate>
auto discard(Predicate && predicate_op);
```

Reduce The stream-based Reduce pattern aims at reducing, using a specific reduction (`combine_op`) function, the sets of items appearing on the input stream. The Reduce interface, as shown in Listing 4.15 receives the window size, i.e., the number of items that will be part of each reduction operation; the offset, determining the number of overlapping items among windows; the initial value of the reductions; and the reduction operator (`combine_op`) function. In this case, the concurrent entities in the parallel implementation are responsible for processing individually the accumulation of the input stream windows.

LISTING 4.15: Reduce interface.

```
template <typename InitVal, typename Combiner>
auto reduce(int window_size, int offset,
              InitVal initial_value,
              Combiner && combine_op);
```

4.3.3 Stream operators

As stated in Section 2, the stream operator are not intended to work as stand-alone patterns since they transform or modify the stream data flow in different ways. Specifically, in GRPPI, these flow modifications are implemented by adding a logic layer in the communication channels to provide the semantics of such operators. This section describes in detail the proposed interfaces for the Split-Join and Window stream operators.

Split-Join The interface designed for the Split-Join pattern, shown in Listing 4.16, receives the split policy (`split_policy`) and the list of transformations (`transformers`) that should be applied to the different output streams. Note that, these transformations may be a single function or a composition of patterns and the number of output streams is given by the number of transformations. This pattern takes the elements from the input stream and splits them into the output streams following the policy received as an argument. Afterward, a different processing entity computes the transformation related to its stream. Finally, the results of each stream are joined into a single one following a round-robin policy. [1]

LISTING 4.16: Split-Join interface.

```
template <typename SP, typename ... T>
auto split_join(SP & split_policy, T ... transformers);
```

In the current implementation, we provide support for both duplicate and round-robin policies (See Listing 4.17). However, the set of policies can be easily extended by declaring a new object that implements two specific functions: `set_num_streams` and `get_next_stream`. The first function is intended to receive as argument the number of output streams, while the second one should return a vector containing the id of the streams that should receive the item. These ids are determined by the `transformers` order in the Split-Join interface.

[1] Note that while the current Split-Join pattern only supports round-robin as for the join policy, in the future, we plan to extend its interface to cover other policies.

LISTING 4.17: Split-Join policy interfaces.

```
auto duplicate();
auto round_robin(int number);
```

As commented, the stream operators in GRPPI introduce a new layer between the worker threads and the communication channels to implement the business logic of the stream operators. For the Split-Join, this layer is in charge of distributing the input items to the different workers depending on the policy stated by the user and joining the results into a single stream. Specifically, as for the split phase instead of having a single queue, this layer generates the same number of queues as the number of flows received in the function call. Afterward, when the input item arrives into the split phase, this is introduced in the queue associated with the corresponding queue. Focusing on the join phase, the logic layer is in charge of getting the elements from the corresponding results queue following a round-robin policy a delivering them to the worker/s of the next stage. Figure 4.2 shows a schema of the layers introduced in the communication channels in a Split-Join pattern. As can be seen, these layers are seen as a regular MPMC queues by the worker threads but modify how the actual queues are accessed. Furthermore, since the policy is separated from this layer, extending the set of policies does not require to modify the layer implementation.

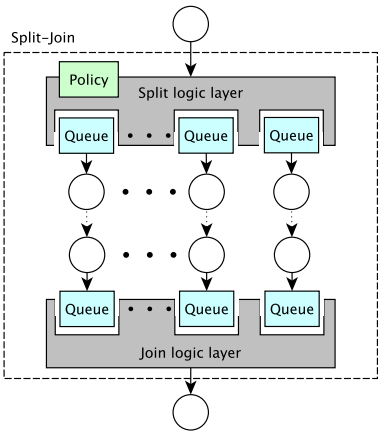


FIGURE 4.2: Schema of the Split-Join communication channels.

Window The interface for the Window pattern, described in Listing 4.18, receives a single argument that is the window policy. This building block takes the items arriving in the input stream and stores them into a buffer. Afterward, depending on the policy, when a window is ready the buffer is delivered to the output stream.

LISTING 4.18: Window interface.

```
template <typename WP>
auto window(WP & window_policy);
```

In the current implementation, we support four different window policies: count-based, time-based, delta-based and punctuation-based (See Listing 4.19). These policies require as a template argument the type of the incoming items to be grouped into the different windows. Additionally, as function arguments, this interface also receives the arguments that allow determining the behavior of the policy, e.g. window size, punctuation-value, sliding, etc. In order to extend the set of window policies, we have defined two functions that should be implemented by the new policies:

`add_item` and `get_window`. The first function should receive the input item and return a boolean to determine if the window is complete or not. On the other hand, the `get_window` function does not receive any argument and should return the window.

LISTING 4.19: Window policy interfaces.

```
template <typename ItemType>
auto count_based(int w_size, int slide);

template <typename ItemType>
auto time_based(int time_size, int slide);

template <typename ItemType>
auto delta_based(int delta_value, int slide);

template <typename ItemType>
auto punctuation(ItemType punctuation_value);
```

Similarly to the Split-Join pattern, this operator introduces a layer between the producers and the consumers to provide the business logic of the Window pattern. Specifically, this layer, seen as an actual queue by producers and consumers, has an internal buffer in order to manage the different queues. Thus, when an input item arrives, it is introduced in the buffer and checks if the window is ready. If so, the whole window is forwarded to the worker/s of the next function and the buffer is updated according to the window policy received as argument.

4.4 Advanced patterns

In this section, we describe the advance stream parallel patterns that provide simpler interfaces for complex pattern compositions to improve the readability and maintainability of the source code. Specifically, we provide support for the Stream-Pool, Windowed-Farm and Stream-Iterator parallel patterns.

Stream-Pool The GRPPI interface designed for the Stream-Pool pattern, shown in Listing 4.20, receives the execution policy, the population (`popul`), the selection (`select`), evolving (`evolve`), filtering (`filter`) and termination (`term`) functions. Initially, the selection takes individuals from the original population and introduces them into the input stream of the pattern. Afterward, the different processing entities take the individuals from the input stream and apply the evolve, termination and filter functions. If the termination condition is met, the pattern finalizes its execution and returns the population with the resulting individuals. Otherwise, depending on the filter function, an evolved or an original individual is introduced again into the input stream.

LISTING 4.20: Stream-Pool interface.

```
template <typename EM, typename P, typename S, typename E, typename F, typename T>
void stream_pool(EM exec_mod, P &popul, S &&select, E &&evolve, F &&filt, T &&term);
```

Windowed-Farm The interface for the Windowed-Farm pattern, described in Listing 4.21, receives the execution policy, the stream consumer (`in`), the Farm (`transformer`) and the producer (`out`) functions. This pattern also receives the windowing policy. In this sense, this pattern is a simplified interface for a GRPPI composition of a Pipeline containing a Window operator and a Farm pattern. Specifically, the `in` function reads from the input stream as many items as required to fill the window buffer. Next, this buffer is forwarded to one of the concurrent entities, which

will compute the function `task` in a Farm-like fashion. Therefore, the parallel implementation of this pattern is offered by the Farm construction. Finally, the items collections resulting from the `task` function are delivered to the output stream. Note that, depending on the user requirements, this pattern can deliver processed windows in an ordered way by properly configuring the execution policy.

LISTING 4.21: Windowed-Farm interface.

```
1 template <typename EM, typename I, typename F, typename O, typename WP>
2 void windowed_farm(EM exec_mod, I &&in, F &&transformer, O &&out, WP &window_policy);
```

Stream-Iterator The GRPPI interface for the Stream-Iterator pattern, detailed in Listing 4.22, takes the execution policy, the stream consumer (`in`), the kernel (`transformer`) and the producer (`out`) functions. This pattern also receives two boolean functions: the termination (`term`) and output guard (`guard`) functions. In the first step, the `in` function reads items from the input stream and a worker thread executes the kernel `transformer` function for each item. Next, the termination function `term` is evaluated with the resulting item to determine if the kernel should be re-executed on the same input item. Additionally, the output `guard` function decides whether an item should be delivered to the output stream or not.

LISTING 4.22: Stream-Iterator interface.

```
1 template <typename EM, typename I, typename F, typename O,
2         typename T, typename G>
3 void stream_iteration(EM exec_mod, I &&in, F &&transformer, O &&out,
4                       T &&term, G &&guard);
```

Note that interfaces presented for these patterns are intended to work as standalone functions, however, the GRPPI interface also offers composable interfaces that allow them to be used as part of other patterns, e.g. Pipeline or Farm. In those cases, the parameters related to the execution model, consumer and producer functions are inherited from the outer pattern.

4.5 Pattern composability

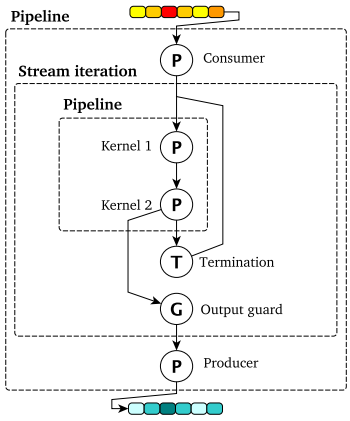
As mentioned in the introduction, the patterns offered by GRPPI can be composed among them to produce more complex structures and to match specific constructions present in both stream and data parallel applications. To demonstrate this feature we describe three examples of pattern composability tackling each of the feasible combinations of computational paradigms (stream and data) supported by GRPPI interface: stream-stream, data-data and stream-data compositions.

For the stream-stream pattern composability, the code in Listing 4.23 implements a Pipeline in which the second stage is a Farm pattern. The Pipeline stages, passed as lambda functions, perform the following tasks: *i)* read the lines of an input file with blank-separated values and pack them into a vector structure, *ii)* compute the maximum value from incoming vectors using the Farm pattern, and *iii)* print the maximum values of the vectors onto an output stream. Given that the Pipeline receives the OpenMP parallel execution policy (line 1), the stages are computed in parallel by the 3 worker threads. Similarly, the nested Farm pattern is executed by 6 OpenMP threads, being one of them the OpenMP thread related to the outer parallel pattern. Therefore, the total number of OpenMP threads for computing this

LISTING 4.23: Example of the Pipeline-Farm composition.

```
pipeline( parallel_execution_omp,
   // Stage 0: read values from a file
   [&]() -> optional<vector<int>> {
      auto r = read_list(is);
      return ( r.size() == 0 ) ? {} : r;
   },
   // Stage 1: takes the maximum value of the vector
   farm(6,
      []( vector<int> v ) {
         return ( v.size() > 0 ) ?
            max_element(v.begin(), v.end()) :
            numeric_limits<int>::min();
   }),
   //Stage 2: prints out the result
   [&os]( int x ) {
      os << x << endl;
   }
);
```

pattern composition will be 8. Note as well that `std::optional` variables, from the C++ Library Fundamentals Extensions (ISO/IEC 19568:2015), are used to mark the end of the streams with an empty value. We denote this Pipeline-Farm composition as `(p|f|p)`, being `p` and `f`, respectively, sequential and Farm-based stages. As shown, thanks to the use of metaprogramming techniques, templates, and lambda expressions, it is possible to easily compose GRPPI parallel patterns in order to build more complex ones.



(A) Composition schema.

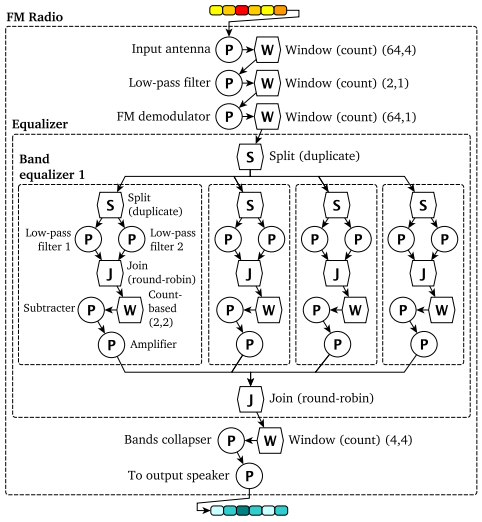
```
using namespace grppi;
...
pipeline(parallel_execution_thr{}
  // Consumer function
  [&]() -> optional<int> {
    auto value = read_value(is);
    return (value > 0) ? value : {};
  },
  stream_iteration(
    pipeline( // Kernel function
      [](int e){ return 3*e; },
      [](int e){ return e-1; }
    ),
    // Termination function
    [](int e){ return e<100; },
    // Output guard function
    [](int e){ return e%2==0;}
  ),
  // Producer function
  [&](int e){ os << e << endl; }
);
```

(B) Implementation.

FIGURE 4.3: Example of Pipeline-Stream-Iterator-Pipeline composition in GRPPI.

Additionally, to demonstrate the composabilty of the stream operators with basic and advanced patterns, we introduce two simple application examples coming from the data stream processing (DaSP) domain. The first application example reads the integers stored in a file, processes them until a threshold is reached and outputs the results according to a guard condition (see Figure 4.3(a)). To express this application in GRPPI terms, we make use of Pipeline-Stream-Iterator-Pipeline composition. As shown in Listing 4.3(b), the consumer and producer functions are part of the outer Pipeline ends, while the computation itself is performed by means of the Stream-Iterator pattern. As stated in the previous section, the parallelism of the Stream-Iterator pattern is only obtained when it is composed with a parallel construct, e.g.,

Farm or Pipeline. In this case, we have leveraged a Pipeline as for the kernel computation, where two different threads simultaneously execute both stages. Recall that the termination function controls the number of times that an item is processed through the inner Pipeline, while the guard function determines which results should be forwarded to the producer function.



(A) Composition schema.

```
using namespace grppi;
...
// FM Radio
pipeline(e,
  // Input antenna
  [&]() -> optional<float> {...},
  window(count_window<float>(64,4)),
  // Low-pass filter
  [&coeff](auto win) {...},
  window(count_window<float>(2,1)),
  // FM demodulator
  [&mGain](auto win) {...},
  window(count_window<float>(64,1)),
  // Equalizer (4 bands)
  split_join(duplicate{},
   pipeline( // Band equalizer 1
     // Band-pass filter
     split_join(duplicate{},
       // Low-pass filter 1
       [&coeffs](auto win) {...},
        // Low-pass filter 2
       [&coeffs](auto win) {...}
     ),
     window(count_window<float>(2,2)
          ),
     // Subtracter
     [](auto w) {...},
     // Amplifier
     [&eqGain](float win) {...}
   ),
   ... // remaining band equalizers
  ),
  window(count_window<float>(4,4)),
  // Bands collapser
  [](auto win) {...},
   // To output speaker
  [&](float f) {...}
);
```

(B) Implementation.

FIGURE 4.4: Pattern composition and implementation of the FM-Radio in GRPPI.

To extend the GRPPI pattern composability demonstration, we leverage the FM-Radio as a real study case of streaming application inspired by an example part of the StreamIt programming model [86] (see Figure 4.4(a)). Concretely, we implement the FM-Radio software taking advantage of the stream operators and basic GRPPI patterns. Basically, this application receives, as for the input stream, the signal from an external antenna and produces a new processed signal that is connected to a speaker. As shown in Listing 4.4(b), the program is structured as a main Pipeline whose first two stages are: a band-pass filter to tune in the desired frequency and a demodulator. Note that these sequential stages are followed by Window operators in charge of conforming count-based windows. Next, the main Pipeline is continued with an equalizer expressed with a Split-Join operator, where each branch fine-tunes the gain of a specific frequency range. The last Pipeline stages collapse all bands and emit the processed signal to an external speaker.

In a nutshell, thanks to the presented stream operators and advanced patterns we are able to implement complex data-flow graphs with relatively small efforts. Simultaneously, the GRPPI interface enhances expressiveness and simplicity due to

the hierarchical block-level abstraction offered by the parallel patterns. All in all, our goal for designing these constructs in a composable way is to improve both programmability and readability of complex streaming applications.

Regarding the data-data pattern composability, Listing 4.24 shows a construct where a Map pattern is composed of a Reduce operation. In this case, the input matrix in the Map pattern is divided into equal partitions among the worker threads. Next, for each row in a partition, the nested Reduce pattern sums up its values and stores the result in the corresponding position of the output vector, passed as an argument in the Map function call. Note that the parallel execution policy for the Map pattern is OpenMP while the nested Reduce pattern uses C++ threads, each of them using 6 worker threads. We denote this composition as `m(r)`, being `m` and `r` the Map and Reduce patterns, respectively.

LISTING 4.24: Example of the Map-Reduce composition.

```
map( parallel_execution_omp{6},
   // Input matrix
   mat_in.begin(), mat_in.end(),
   // Vector of accumulated values from matrix rows
   vec_out.begin(),
   // Map kernel: divide matrix into rows
   [&]( auto row_in, auto sum ) {
      // Reduce kernel: Sum up the values in a matrix row
      reduce( parallel_execution_thr{6},
         row_in.begin(), row_in.end(),
         &sum,
         std::plus<double> );
      );
   }
);
```

As mentioned, we can also compose stream with data patterns. This is a feasible composition, given that the items coming from a stream can be processed themselves using a data parallel pattern. The opposite is however not feasible since the results generated in a data pattern cannot be transformed into streams and, therefore, processed using a stream processing approach. To illustrate a stream-data pattern composition, Listing 4.25 shows an example where a Farm stream parallel pattern is composed of a Divide&Conquer data one. In this particular case, the Farm pattern steadily reads values stored in a file and computes, for each of them, their corresponding i-th Fibonacci number using the Divide&Conquer pattern. Finally, the Fibonacci numbers are printed to the end user. As shown, the parallelization of the Farm is performed using 6 OpenMP threads, while the nested Divide&Conquer pattern uses 6 C++ threads. Since each of the Farm-related threads creates 6 C++ nested ones, the total number of threads computing this composition is 36. This composition is denoted as `f(d)`, being `f` and `d`, Farm and Divide&Conquer patterns, respectively.

In general, Table 4.1 summarizes pattern compositions grouped by the three possible combinations of computational paradigms supported by GRPPI interface: stream-stream, data-data and stream-data compositions. Note that rows and columns in the tables represent the outer and inner patterns involved in a given composition, respectively. We classify each specific pattern composition with one of the following four categories, from less to more restrictive:

Infeasible This category represents a composition that is not supported by GRPPI.

Feasible This category denotes a composition that can be implemented in GRPPI.

Irreducible This category is a feasible composition providing a useful parallel pattern that cannot be simplified any further. Note that pattern compositions falling in this category are natively supported by GRPPI.

LISTING 4.25: Example of the Farm-Divide&Conquer composition.

```
pipeline(parallel_execution_omp,
   [&]() -> optional<int> { // Read values from an input file
      auto value = read_value(is);
      return ( value > 0 ) ? value : {};
   },
   farm( 6, [&]( int value ) { // Compute the fibonacci number using a D&C pattern
      int fibonacci = 0;
      divide_conquer(parallel_execution_thr(6), value, &fibonacci,
         [&](auto &value){
            std::vector< int > subproblem;
            if( v < 2 ) subproblem.push_back(value);
            else subproblem.insert(subproblem.end(), { value-1, value-2 });
            return subproblem;
         },
         [&](auto &problem, auto &partial){
            partial = ( problem == 0 ) ? 0 : 1;
         },
         [&](auto & partial, auto & out){
            out += partial;
         }
      );
      return fibonacci;
   }),
   [&]( int fibonacci ) { // Print the fibonacci values
      cout << fibonacci << endl;
   }
);
```

Useful-Reducible This category is a feasible composition implementing a pattern composition that can be simplified further but that, in some cases, provides a clearer and a more readable code than its simpler equivalent.

As shown in Table 4.1a, the stream-stream pattern compositions involving a Pipeline and other pattern are classified as Irreducible (except those with an outer Reduce pattern), given that it is not possible to obtain the same parallel construction using any simpler pattern. These types of compositions are natively supported in GRPPI, as shown in Listing 4.23. Any other composition is considered as Feasible since they can be simplified using the outer or inner pattern with an increased parallelism degree. However, these compositions do not provide any major advantage compared to the simpler construction. On the other hand, compositions containing an outer Reduce pattern are Infeasible, as this pattern does not receive any user function to be executed in parallel. Focusing on the stream operators, since they are intended to work as part of a bigger composition with more than one pattern or function it can only be used as a stage of a Pipeline pattern. However, both operators can be used before any other pattern inside the Pipeline. With respect to the advanced patterns, the Windowed-Farm can be composed in its Farm phase in the same way as for the Farm pattern, since the internal implementation of this advanced pattern is a Pipeline composed with a Window operator and a Farm. As for the Stream-Iterator can be only composed with a Pipeline and a Farm. For instance, a Stream-Iterator composed with a Filter pattern will result on applying multiple times the same filter over the same item. Similarly, the reduction will result in reducing the same set of elements multiple times. Finally, the Stream-Pool cannot be composed with any other pattern since this pattern models a particular parallel algorithm that cannot further compose nor represented by other pattern composition.

Focusing on data-data compositions, as shown in Table 4.1b, constructions whose outer pattern is Map-like (Map and Stencil) are categorized as Useful-Reducible. This is because there exists a simpler equivalent using only the outer Map-like pattern. Regarding the Reduce pattern, it cannot be combined with any other inner one. The reasons are the same as those for the Reduce pattern in stream-stream compositions.

TABLE 4.1: Parallel patterns compositions in GRPPI.

(A) Stream-stream compositions.

Outer pattern \ Inner pattern	Pipeline	Farm	Filter	Reduce
Pipeline	✓ (Feasible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)
Farm	✓ (Irreducible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)
Filter	✓ (Irreducible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)
Reduce	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)
Stream-Iterator	✓ (Irreducible)	✓ (Irreducible)	✗ (Infeasible)	✗ (Infeasible)
Windowed-Farm	✓ (Irreducible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)
Split-Join	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)

Outer pattern \ Inner pattern	Split-Join	Window	Stream-Iterator	Windowed-Farm
Pipeline	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)
Farm	✗ (Infeasible)	✗ (Infeasible)	✓ (Irreducible)	✓ (Feasible)
Filter	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	✓ (Feasible)
Reduce	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)
Stream-Iterator	✗ (Infeasible)	✗ (Infeasible)	✓ (Feasible)	✓ (Irreducible)
Windowed-Farm	✗ (Infeasible)	✗ (Infeasible)	✓ (Feasible)	✓ (Feasible)
Split-Join	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)

(B) Data-data compositions.

Outer pattern \ Inner pattern	Map	Reduce	Stencil	MapReduce	Divide&Conquer
Map	✓ (Useful-Reducible)	✓ (Useful-Reducible)	✓ (Useful-Reducible)	✓ (Useful-Reducible)	✓ (Useful-Reducible)
Reduce	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)
Stencil	✓ (Useful-Reducible)	✓ (Useful-Reducible)	✓ (Useful-Reducible)	✓ (Useful-Reducible)	✓ (Useful-Reducible)
MapReduce	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)
Divide&Conquer	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)

(C) Stream-data compositions.

Outer pattern \ Inner pattern	Map	Reduce	Stencil	MapReduce	Divide&Conquer
Pipeline	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)
Farm	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)
Filter	✓ (Feasible)	✓ (Feasible)*	✓ (Feasible)	✓ (Feasible)	✓ (Feasible)*
Reduce	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)	✗ (Infeasible)
Stream-Iterator	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)
Windowed-Farm	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)
Split-Join	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)

Other compositions whose outer pattern is MapReduce or Divide&Conquer are classified as Feasible, as they can be implemented in GRPPI although do not bring any major advantage.

Finally, stream-data compositions are summarized in Table 4.1c. Compositions whose outer pattern is Pipeline or Farm are denoted as Irreducible. The combination of two distinct parallel paradigms (stream-data) makes these compositions unique and precludes them to be simplified any further. As for compositions with an outer Filter pattern, the output cardinality of its inner pattern dictates whether the composition is Feasible or Useful-Reducible. This is because the output of the Filter function is a boolean. For instance, in a Filter-Map composition, the output cardinality of the Map pattern is equal to the input cardinality. So that, although the predicate of the Filter pattern can be implemented by transforming the output data set into a boolean, this case does not reflect a common practice. Therefore, we classify these compositions only as Feasible. On the other hand, the output cardinality of the Reduce pattern in a Filter-Reduce composition is a sole element. Thus, the Filter predicate can be easily implemented by transforming such element into a boolean. For this reason, we categorize this construct as a special case of Feasible composition. The Filter-Divide&Conquer combination is also a special case of Feasible composition because the output cardinality of the Divide&Conquer pattern depends on the algorithm. Finally, the Reduce pattern is not composable and, hence, classified as Infeasible. With respect to the stream operators, as stated before, they cannot be composed since they are not a pattern itself and can be only part of a Pipeline pattern. Focusing on the advanced patterns, the Stream-Iterator pattern can be composed with data parallel pattern if its transforming operation matches with such patterns. In these case, the computation of the inner pattern will be executed until reaching the condition stated as for the Stream-Iterator predicate. As for the Windowed-Farm, the composition of this pattern is given by the Farm pattern, so the same possibilities stated for the Farm also applies to this advanced pattern.

4.6 Evaluation

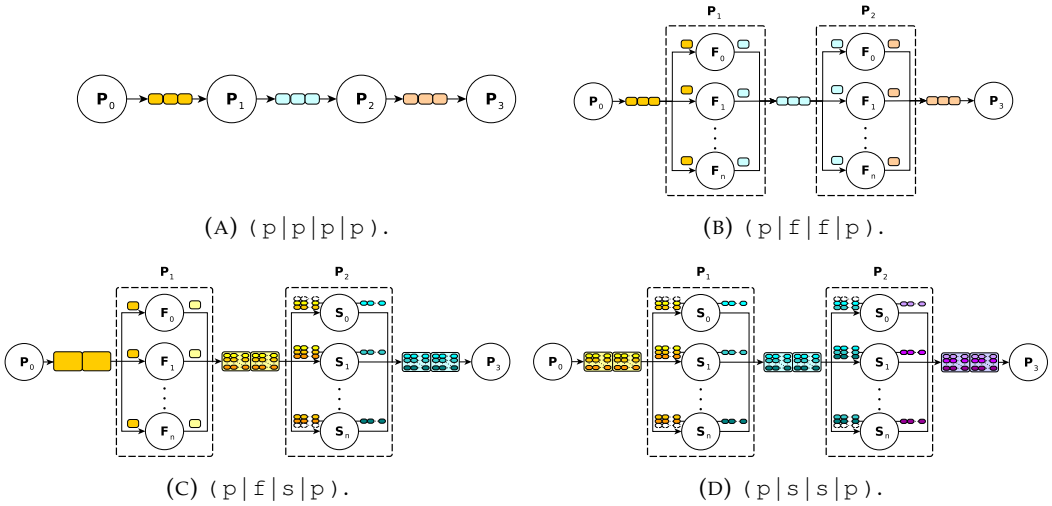
In this section, we evaluate the proposed parallel pattern interface. First, to evaluate the basic parallel patterns, we used a video stream-processing application composed of two filters, the Gaussian Blur and Sobel operators [64, 70]. These filters are applied to an input video in order to detect edges appearing in the frames[2]. Specifically, this application matches the parallel Pipeline pattern, in which the first stage reads the frames from a video file passed as input; the second and third stages apply the Gaussian Blur and Sobel filters, respectively; and the last stage dumps the processed frames to an output video file. Note that both filters use a kernel size of 3×3. Note as well that, while the Gaussian Blur filter only performs arithmetic operations, the Sobel operator also performs a square root operation for each frame pixel processed.

To carry out the experimental evaluation of this set of patterns, we first parallelize this video application using GRPPI with the different supported execution frameworks, i.e. C++ Threads, OpenMP and IntelTBB. Afterwards, we compare both performance and the number of lines of code required to implement such parallel versions with respect to the sequential one. To further experiment with our interface, we implemented different versions of the video application using the execution frameworks for CPUs and distinct compositions of patterns in its main pipeline.

[2]This benchmark has been inspired by an OpenCV edge detection example from http://docs.opencv.org/3.1.0/d3/d63/edge_8cpp-example.html.

Specifically, we use the following Pipeline compositions: *i)* a non-composed Pipeline `(p|p|p|p)`; *ii)* a Pipeline composed of a Farm in its second stage `(p|f|p|p)`; *iii)* a Pipeline composed of a Farm in its third stage `(p|p|f|p)`; and *iv)* a Pipeline composed of two Farm patterns in the second and third stages `(p|f|f|p)`.

Next, we also evaluate and compare the performance when using different stream-stream and stream-data Pipeline compositions and heterogeneous configurations (CPU+GPU). Since, both Gaussian Blur and Sobel filters applied in a given pixel depends on its neightbours, we use the Stencil as for the data pattern in stream-data Pipeline compositons. Specifically, we leverage different Pipeline constructions composed of: *i)* two Farm patterns `(p|f|f|p)`; *ii)* a Farm and a Stencil in its second and third stages `(p|f|s|p)`; *iii)* a Stencil and a Farm in its second and third stages `(p|s|f|p)`; and *iv)* two Stencil patterns `(p|s|s|p)`. Note that when using a Pipeline-Farm composition, each worker thread from the Farm pattern processes individual video frames. However, when leveraging a Pipeline-Stencil construction, each thread in the Stencil pattern is in charge of computing a distinct partition of a single video frame. Figure 4.5 illustrates some of the compositions used in these studies.



(A) `(p|p|p|p)`. (B) `(p|f|f|p)`.

(C) `(p|f|s|p)`. (D) `(p|s|s|p)`.

FIGURE 4.5: Pipeline compositions of the video application.

Finally, to evaluate both stream operators and advanced parallel patterns from the usability and performance points of view, we use the following benchmark applications:

FM-Radio This study case is a signal processing application emulating the software of an FM radio and it is employed to evaluate the basic patterns and the stream operators. To better analyze the behavior of these patterns, we have implemented five different versions of the FM-Radio application with varying number of equalizer bands, from 1 to 5.

TSP This benchmark solves the *traveling salesman problem* (TSP) using a regular evolutionary algorithm and it is intended to analyze the behavior of the StreamPool pattern. Specifically, this NP-problem computes the shortest possible route among different cities, visiting them only once and returning to the origin city.

Sensor This streaming application computes average window values of the readings from an emulated sensor and it is used to evaluate the performance of the Windowed-Farm pattern.

Image This stream-oriented benchmark reduces the resolution of images appearing in the input stream and delivers images with concrete resolutions to the output stream. Basically, this benchmark is used to evaluate the performance of the Stream-Iterator pattern.

4.6.1 Reference platform

The evaluation has been carried out on a server platform comprised of 2× Intel Xeon Ivy Bridge E5-2695 v2 with a total of 24 cores running at 2.40 GHz, 30 MB of L3 cache and 128 GB of DDR3 RAM. The OS is a Linux Ubuntu 14.04.2 LTS with the kernel 3.13.0-57. This platform also incorporates a NVIDIA Tesla K40c with 12 GB and GeForce GTX680 GPUs with 2 GB of DDR5 RAM. These GPUs are denoted as `GPU0` and `GPU1`, respectively. The OS is a Linux Ubuntu 14.04.5 LTS with kernel 3.13.0-85.

4.6.2 Analysis of the usability

In this section, we analyze the usability and flexibility of the developed interface. To analyze these aspects, we first compare the number of lines required to implement the parallel version of the video application leveraging the interface, with respect to using directly the parallel execution frameworks. Table 4.2 summarizes the percentage of additional lines introduced into the sequential source code in order to implement such parallel versions for the above-mentioned pattern compositions. As shown, implementing more complex compositions via C++ threads or OpenMP leads to larger source codes, while for Intel TBB the number of required additional lines remains constant. Focusing on GrPPI, we observe that the effort of parallelizing an application is almost negligible: even the most complex composition increases nearly 4.4 % the number of lines of code. This behavior is in contrast to C++ threads or OpenMP frameworks, which require roughly twice of lines of code. Additionally, switching GrPPI to use a particular execution framework just needs changing a single argument in the pattern function calls.

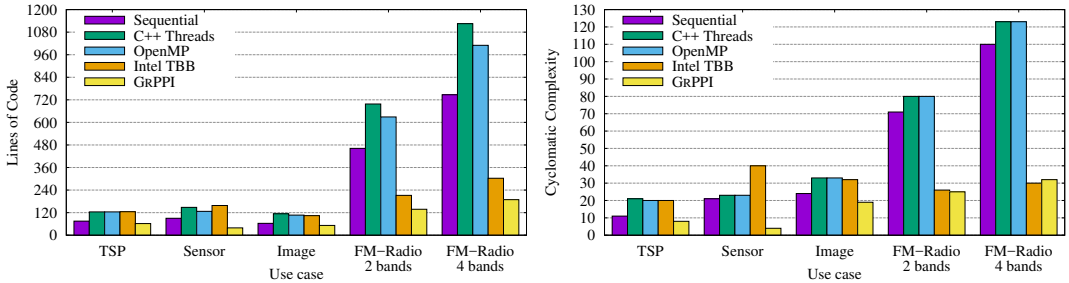
TABLE 4.2: Percentage of increase of lines of code w.r.t. the sequential version for the video application.

Pipeline composition	% of increase of lines of code			
	C++ Threads	**OpenMP**	**Intel TBB**	**GrPPI**
(p|p|p|p)	+8.8 %	+13.0 %	+25.9 %	+1.8 %
(p|f|p|p)	+59.4 %	+62.6 %	+25.9 %	+3.1 %
(p|p|f|p)	+60.0 %	+63.9 %	+25.9 %	+3.1 %
(p|f|f|p)	+106.9 %	+109.4 %	+25.9 %	+4.4 %

To gain insights in the usability and flexibility of the stream operators and advanced patterns, we make use of the Lizard analyzer tool [85] to perform an additional analysis based on two well-known metrics: Lines of Code (LOCs) and the McCabe's Cyclomatic Complexity Number (CCN) [55]. Basically, we leverage these metrics to analyze the different stream operator and advanced pattern use case versions, i.e., with and without using our GrPPI interface. Figure 4.6(a) shows the LOCs required to implement the parallel versions of the use case algorithms directly using the execution frameworks and the GrPPI interface. As observed, the TSP,

Sensor and Image are simple use cases whose sequential versions require about 100 LOCs, while the FM-Radio is a more complex application with roughly 500 LOCs. Focusing on the simple use cases, we detect that the codes directly parallelized with the supported frameworks require almost twice the LOCs of the sequential versions, as none of them intrinsically implement the proposed high-level advanced patterns. On the contrary, the LOCs using GrPPI are significantly reduced with respect to the sequential code. Looking at the FM-Radio benchmark, we find out that for C++ threads and OpenMP, the LOCs increase by almost 50 % compared with the sequential version. This is given by the fact that the synchronization and management control mechanisms are not natively provided by these frameworks and have to be implemented by the user. Also, using a higher number of equalizer bands entails an increase of the LOCs, since each of the bands differs among them and the concurrency mechanisms should be explicitly implemented for each of them. Differently, the high-level Intel TBB and GrPPI abstractions inherently incorporate these mechanisms and lead to smaller and more maintainable codes.

Looking at the cyclomatic complexity, detailed in Figure 4.6(b), the CCNs related to TSP and Image use cases are roughly proportional to their LOCs. Nevertheless, the Image case does not follow this behavior as the Intel TBB framework requires the user to explicitly implement the windowing management mechanisms, entailing an increase in its complexity. Focusing on the FM-Radio with both 2 and 4 equalizer bands, we also detect that LOCs and CCNs keep approximately the same proportions. Another observation is that both Intel TBB and GrPPI frameworks present the same CCNs, while other frameworks outperform their CCNs by a factor of four-fold.



(A) Lines of code.

(B) Cyclomatic complexity.

FIGURE 4.6: Lines of code and cyclomatic complexity of the use cases w.r.t different programming models.

Finally, we perform a side-by-side comparison between the GrPPI and Intel TBB interfaces implementing the FM-Radio use case. As shown in Listing 4.7(a), the GrPPI code follows a comprehensible and readable structure, which clearly shows the compositions among basic patterns and stream operators. On the contrary, the Intel TBB code, shown in Listing 4.7(b), is not as structured and easy to read as the GrPPI implementation. Thus, in order to properly understand the application behavior, users need to carefully analyze the data flow graph implicitly declared in the Intel TBB code. In a nutshell, although both interfaces provide high-level parallel interfaces, we conclude that GrPPI leads to more structured and readable codes, and thus, improves both usability and maintainability.

```
using namespace grppi;
...
// FM Radio
pipeline(e,
  // Input antenna
  [&]() -> optional<float> {...},
  window(count_window<float>(64,4)),
  // Low-pass filter
  [&coeff](auto win) {...},
  window(count_window<float>(2,1)),
  // FM demodulator
  [&mGain](auto win) {...},
  window(count_window<float>(64,1)),
  // Equalizer (4 bands)
  split_join(duplicate{},
    pipeline( // Band equalizer 1
      // Band-pass filter
      split_join(duplicate{},
        // Low-pass filter 1
        [&coeffs](auto win) {...},
        // Low-pass filter 2
        [&coeffs](auto win) {...}
      ),
      window(count_window<float>(2,2)),
      // Subtracter
      [](auto w) {...},
      // Amplifier
      [&eqGain](float win) {...}
    ),
    ... // remaining band equalizers
  ),
  window(count_window<float>(4,4)),
  // Bands collapser
  [](auto win) {...},
  // To output speaker
  [&](float f) {...}
);
```

(A) GRPPI implementation.

```
// FM Radio
struct split1 {void operator()(const buf_t &i,
  multi_node::output_ports_type &op) {...}};
struct window1{void operator()(const float &v,
  window_node::output_ports_type &op) {...}};
...
// Input antenna node
source_node<float> antenna(g,[&](float &v)
  -> bool {...} );
// Window node
window_node wind1(g,serial,window1());
// Window queue node
queue_node<buf_t> win_q(g);
// Split node
multi_node spl1(g,unlimited,split1());
// Split queue node
queue_node<buf_t> queue_band(g);
...
make_edge(antenna,win1); // Window edge
// Window-Queue edge
make_edge(output_port<0>(win1),win_q1);
// Low-pass filter edge
make_edge(win_q1,lowp1);
...
make_edge(output_port<0>(win3), win_q3);
// Band equalizer splitter edge
make_edge(win_q3,spl1);
make_edge(output_port<0>(spl1), band_q);
...
make_edge(sub1,amp1); // Amplifier edge
// Joiner edge
make_edge(amp1,input_port<0>(j1));
...
// Band collapser edge
make_edge(j1,adder);
// Output speaker edge
make_edge(adder,speaker);
```

(B) Intel TBB implementation.

FIGURE 4.7: GRPPI and Intel TBB implementations of the FM-Radio.

4.6.3 Performance analysis of pattern compositions

Next, we analyze the performance with and without GRPPI using the different execution frameworks and Pipeline compositions for the video application. Concretely, we employ the frames per second (FPS) metric to analyze the behavior of the particular versions using the same input video with diverse resolutions. Also, we set the Farm stage(s) in all Pipeline compositions to be executed in parallel by 6 threads for all the execution policies. Figure 4.8 depicts the FPS obtained for the different compositions in this experiment. A first observation is that the Pipeline combined with two Farm patterns for the filtering stages, in comparison to the non-composed Pipeline and compositions with only one Farm, improves substantially the FPS for all parallel frameworks. It is also remarkable that compositions using only one Farm do not bring significant improvements since they lead to imbalanced Pipeline stages. Note that the stage running sequentially dictates the Pipeline performance, as it is the slowest one. An additional inspection into the plots reveals that the best case of Pipeline composition, which uses two Farm patterns, both C++11 and OpenMP deliver similar performance figures, while TBB obtains better FPS for all video resolutions. This is due to the ordering algorithm of the output stream is better optimized than those used by the other frameworks and the load balance performed by the Intel TBB runtime scheduler. Finally, we observe that the usage of GRPPI does not lead to significant overheads: it is less than 2 %, on average, for all the execution frameworks and compositions.

4.6.4 Performance analysis of stream vs data patterns

Our next analysis compares the performance of different Pipeline compositions that combine stream and data parallel patterns. Figure 4.9 shows the FPS obtained for

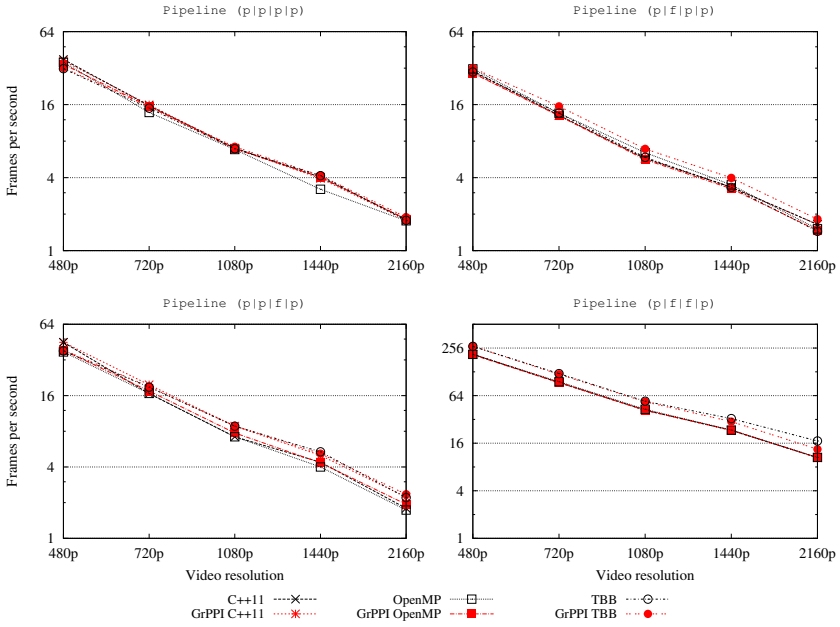


FIGURE 4.8: FPS w/ and w/o using GRPPI along with the different frameworks and Pipeline compositions.

different video resolutions and parallel frameworks using GRPPI in different Pipeline compositions containing both stream and data patterns, Farm and Stencil, respectively. In this case, both C++ Threads and Intel TBB frameworks deliver similar performance results for all compositions. A more detailed inspection of these plots unveils an inflection point where the data-stream compositions start attaining better performance. This occurs from 1080p on for C++ Threads and from 1440p on for TBB. Note as well, the slight difference using only the Stencil for computing the first or second filter. The reason behind this behavior is the higher computational load of the Sobel with respect to the Gaussian Blur filter. Regarding the OpenMP framework, it can be clearly seen that the stream-stream `(p|f|f|p)` composition delivers better results than using stream-data constructs. This is mainly because of the worker threads in the GRPPI-Farm pattern leverage OpenMP tasks that are active during the whole video processing, while the Stencil implementation creates and destroys a task each time a video frame is processed. We figured out that the GCC-OpenMP implementation does not make use of a thread pool and, therefore, the threads in each Stencil computation are recurrently created and destroyed. Consequently, stream-data compositions in OpenMP suffer from considerable performance degradations.

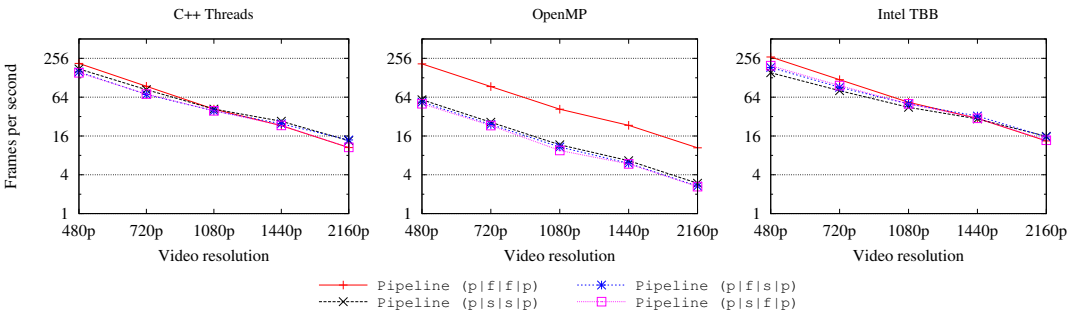


FIGURE 4.9: FPS for different frameworks and Pipeline compositions with stream and data patterns.

4.6.5 Performance analysis on heterogeneous configurations

Our last experiment using the video application analyzes the performance of a stream-data Pipeline composition with different heterogeneous configurations (CPU + GPU). To do so, we have implemented a prototype execution policy that leverages Cuda Thrust for implementing some data patterns using the `transform` and `reduce` algorithms. Figure 4.10 illustrates the FPS delivered by the Pipeline composed of two Stencil stages that are mapped in different ways to the devices available on the platform. As a first observation, the mapping configuration that attains the best performance is when using indistinguishably both GPUs for the Stencil stages of the Pipeline. This performance difference is due to the higher computational capacity of the GPUs, overtaking the CPUs in terms of the number of cores and SIMD capabilities. Note that, in this specific use case, both arrangements of the `GPU0` and `GPU1` executing the Gaussian Blur and Sobel filters attain comparable FPS. Our next observation focuses on the configurations in which, at least, one Stencil stage is mapped to the CPU cores. In these cases, when the slowest Pipeline stage (i.e. the Sobel operator) runs on a GPU, the total Pipeline throughput improves, as it contributes to having more balanced stages. Nevertheless, this advantage only applies to large frame resolutions starting from 1080p, where the data transfers overheads (host-device) pays off the amount of computation performed by the GPUs. On the other hand, if the Gaussian Blur is mapped to one of the GPUs, the throughput is limited by the Sobel operator that, executed on the CPU cores, acts as a bottleneck. Also, in this specific case, host-device data transfers do not pay off the performance improvements with respect to mapping the Gaussian Blur filter on the CPU cores.

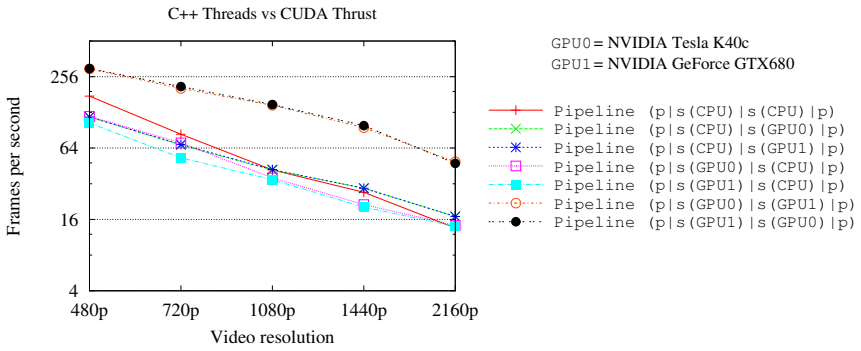


FIGURE 4.10: FPS for the Pipeline composed of two Stencil patterns on different heterogeneous configurations.

4.6.6 Performance analysis of the FM-Radio

In this section, we evaluate the presented stream GRPPI operators using the FM-Radio use case. Figure 4.11 depicts the speedup of the versions of the application with varying number of equalizer bands and using the available backends. In this experiment, it is important to consider that the performance attained by the FM-Radio versions cannot be compared among them, as the applications are intrinsically different and produce distinct results. With this in mind, we observe that both C++ threads and OpenMP approximately attained the same speedups. However, the performance obtained by Intel TBB is much lower and, given that the windowing management techniques are not natively supported by the framework, it is required to use additional graph nodes to accomplish the same business logic. A final inspection on the results using Intel TBB reveals that the versions using a higher number of equalizer bands attain better performance, as the proportion between the number

of Window operators and the basic patterns is lower. Therefore, overheads related to the Window operators are counteracted with effective computations performed by the parallel patterns.

From this experiment, we can conclude that the stream operators composed with regular patterns greatly aid in developing complex constructs, always at the expense of evaluating the best execution environment. The best choice for the backend basically depends on the application nature and the target platform.

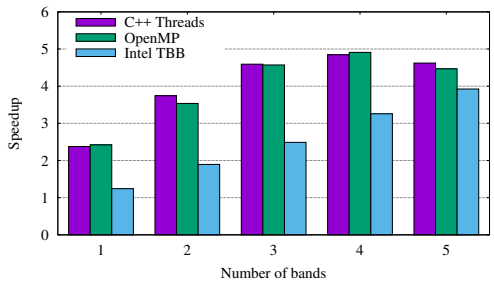


FIGURE 4.11: Speedup of the FM-Radio with varying number of bands.

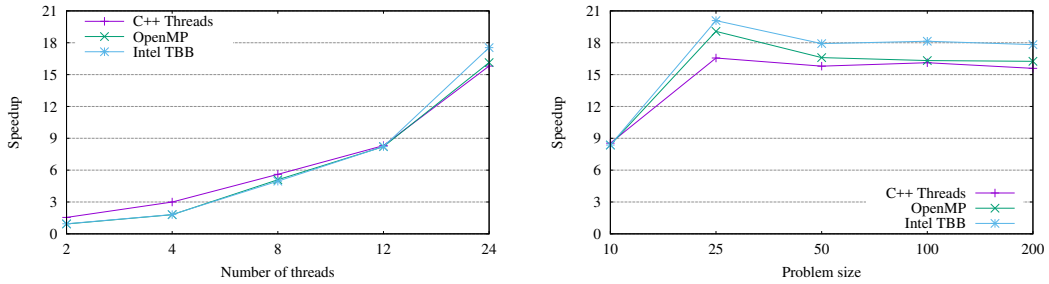
4.6.7 Performance analysis of the Stream-Pool pattern

Next, we evaluate the Stream-Pool pattern on a benchmark that solves the TSP problem using an initial population of 50 individuals representing feasible routes. We also set the benchmark to perform a total of 200 iterations, each of them making 200 selections. Figure 4.12(a) shows the performance gains when varying the number of threads, from 2 to 24, and using the three available GRPPI backends, C++ threads, OpenMP and Intel TBB, with respect to the sequential version. As observed, the speedup roughly increases at a linear rate when using a higher number of threads for all frameworks. Concretely, we observe that between 2 and 12 threads, the efficiency is sustained in the range of 75 %–80 %. On the other hand, Intel TBB with 24 threads delivers an efficiency of 80 %, while the same in C++ threads and OpenMP lead to a slightly decreased efficiency of 70 %. This is mainly due to the better resource usage made by the Intel TBB runtime scheduler.

As a complimentary evaluation, we set the number of threads to 24 and vary the number of cities from 10 and 200. According to the results shown in Figure 4.12(b), the parallelization overheads using 24 threads does not pay off for small workloads, i.e., setting only 10 cities as for the problem size. However, when the problem size grows, the threads perform more effective computations and lead to a better application scalability. Also, we observe that above 50 cities the application becomes memory-bound due to the impact on handling the data structures representing feasible routes. Finally, it is also important to remark that those runtime-based frameworks (OpenMP and Intel TBB) provide better efficiency compared with the C++ threads execution environment.

4.6.8 Performance analysis of the Windowed-Farm pattern

In this section, we evaluate the performance of the Windowed-Farm pattern using a synthetic benchmark that computes average window values from a sensor reading samples at 1 kHz. To carry out this evaluation, the following four subsections



(A) Speedup vs. number of threads.

(B) Speedup vs. problem size.

FIGURE 4.12: Stream-Pool speedup varying with varying number of threads and problem size.

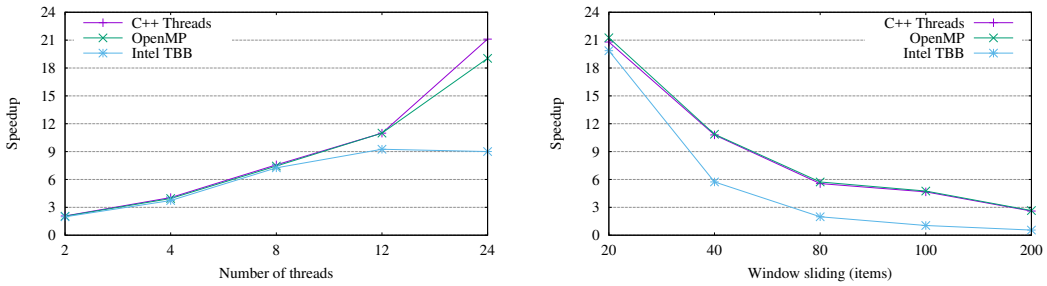
present different scenarios of the Windowed-Farm pattern using the proposed windowing management policies for the Window operator: *count-based*, *delta-based*, *time-based* and *punctuation-based*.

Analysis of the *count-based* windowing policy

As for the Windowed-Farm using the *count-based* policy, we set the window size to 100 elements with an overlap factor among windows of 90 %. Figure 4.13(a) shows the speedup when the number of threads increases from 2 to 24. The main observation is that both C++ threads and OpenMP frameworks scale with the increasing number of threads and behave similarly, given that the OpenMP runtime scheduler does not provide any major advantage over the C++ threads implementation in this concrete use case. This is because the internal Farm pattern leads, by nature, to well-balanced workloads among threads. Note that a Farm is comprised of a pool of threads that constantly poll for items from the input stream and apply the same operation to them. On the other hand, we also observe an almost linear speedup scaling for increasing number of threads. This is mainly caused because the Farm pattern can theoretically scale up to $\frac{T_f}{T_a}$, being T_f the computation time of the window average value and T_a the interarrival time of windows in the input stream. To demonstrate this strong scaling, we experimentally measured the computation time of the average function, which was, on average, 220 ms and the interarrival window time that was 10 ms. Therefore, applying the aforementioned formula, we get 22 as for the maximum theoretical speedup. In contrast, focusing on the Intel TBB backend, we observe that the application stops scaling from 12 threads on. The reason for this behavior is the same as discussed in Section 4.6.6, i.e., high-level pattern interfaces provided by Intel TBB do not intrinsically support any kind of windowing management policies. Therefore, performance overheads caused by the implementation of these policies in such an interface are non-negligible compared with those induced by using low-level frameworks, e.g., C++ threads and OpenMP. Concretely, the time required by the Window operator to generate a window is defined by $(T_i + T_o) \times window\_size$, where T_i and T_o are the item interarrival time and the window management overheads per item, respectively. With this respect, the windowing overheads for Intel TBB are higher than for other frameworks and cause a general throughput decrease due to the use of such Window operator.

As an additional experiment using *count-based* policy, we evaluate the behavior of the same benchmark with varying window sizes and using 24 threads. As can be

observed in Figure 4.13(b), the speedup decreases for increasing window sizes, as the number of non-overlapping items among windows also increases. This basically occurs because the window interarrival time T_a increases, restricting proportionally the maximum parallelism degree.



(A) Speedup vs. number of threads.

(B) Speedup vs. window size.

FIGURE 4.13: Windowed-Farm speedup with varying number of threads and window size using *count-based* windows.

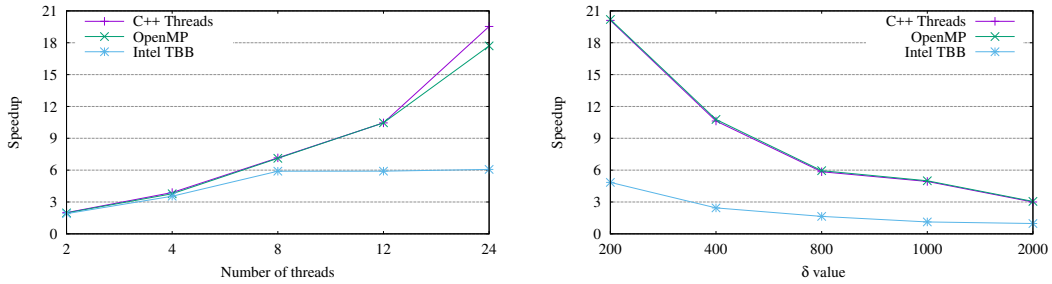
Analysis of the *delta-based* windowing policy

Regarding the evaluation of the Windowed-Farm pattern leveraging the *delta-based* windowing policy, we set the δ threshold to 1000 and the δ sliding to 150. Considering that the sensor of this synthetic application does not produce, by nature, monotonically increasing values, we have slightly modified the input stream to inject tuples containing the sensor reading and the Δ attribute to the Windowed-Farm pattern. Specifically, the Δ attribute equals to the sum of the previous samples. Figure 4.14(a) depicts the speedup of the different GRPPI backends using a different number of threads. As observed, the behavior of the *delta-based* policy is similar to that observed for *count-based*. In this sense, the speedups delivered by the respective C++ threads and OpenMP backends go hand in hand and reach values of 18 and 20, respectively. This is mainly due to the similarities of both *count-based* and *delta-based* policies, where the windowing management overheads are mostly the same. Again, the lack of the Window operator in Intel TBB, produces notorious performance degradations from 8 threads on.

To extend this analysis, we perform additional experiment fixing the number of threads to 24 and varying the δ threshold value from 200 to 2000. Furthermore, we equal the δ sliding to the δ threshold value. Again, both C++ threads and OpenMP deliver similar speedups, with a decrease for higher δ threshold values. This behavior is produced by the fact that a higher δ value tends to generate a lower number of windows with more items in each. Having in mind that processing entities compute at window-level, the concurrency degree at some point might not be fully exploited if the number of windows available to be processed is lower than the number of threads. The Intel TBB backend, however, suffers from considerable performance degradations, as the windows have to be internally handled by our Windowed-Farm pattern.

Analysis of the *time-based* windowing policy

For the evaluation of the Windowed-Farm using the *time-based* policy, we set the window time to 0.1 s with a sliding value of 0.01 s. In this concrete case, we measure



(A) Speedup vs. number of threads.

(B) Speedup vs. window size.

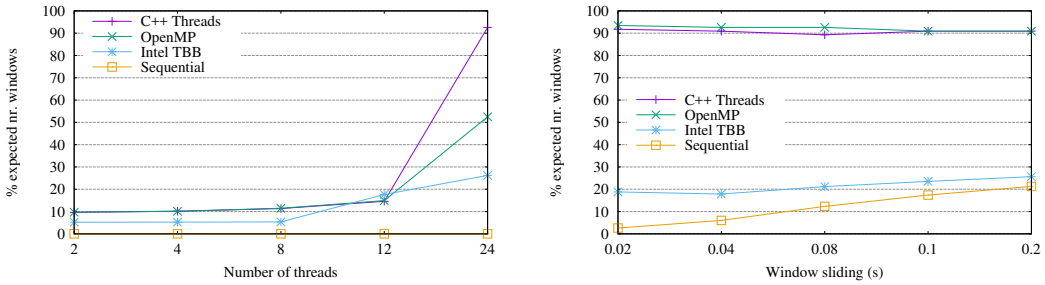
FIGURE 4.14: Windowed-Farm speedup with varying number of threads and window size using *delta-based* windows.

the number of windows that the application generates with respect to the expected ones, instead of the regular speedup metric. This is because the performance of the *time-based* windowing policy is directly related to the window time and sliding parameters, which for the same input stream determine the theoretical number of resulting windows. Any overhead related to the kernel function or to the window management would make the application to deliver a higher number of windows (with fewer items per window) and to a general worse behavior. Considering our use case with a sensor producing 1000 samples/s, a window time of 0.1 s and a window sliding of 0.01 s, we would expect 100 windows/s with 100 elements per window. Figure 4.15(a) shows the percentage of the number of windows with respect to the expected for varying number of threads. As observed, the sequential backend produces the worst results, as the percentages drawn are close to 1%. Regarding the C++ threads, OpenMP and Intel TBB we detect as well that the behavior between 2 and 12 threads is far from the expected and ranges from 5% to 17%. This is due to the inherent congestions caused by the internal communication queues, as the worker threads are not able to process windows at the same pace that they arrive. Finally, focusing on the results for these backends using 24 threads, we observe an in improved behavior for the C++ threads, OpenMP and Intel TBB, reaching an efficiency percentage of about 92%, 52%, and 28%, respectively.

To gain more insights into the behavior of the *time-based* policy, we perform an additional experiment in which we fix the number of threads to 24 and vary the window sliding parameter from 0.02 to 0.2 seconds. As shown, when the sliding time increases, the behavior of the sequential version improves, as fewer windows per second have to be delivered. However, the Intel TBB version does not improve much and stays between 17% and 26%. In contrast, the C++ threads and OpenMP backends deliver much better accuracy figures, around 92%, when the window sliding time increases. The reason for that is the overheads related to window management are almost negligible and affect, to a lesser extent, the production of windows in due time.

Analysis of the *punctuation-based* windowing policy

Finally, we evaluate the Windowed-Farm using the *punctuation-based* policy setting the delimiter value to 0. Specifically, we modified the sensor to randomly generate the value 0 with a probability of 0.01. Figure 4.16 shows the speedup attained by the



(A) Speedup vs. number of threads.

(B) Speedup vs. window size.

FIGURE 4.15: Windowed-Farm speedup with varying number of threads and window size using *time-based* windows.

benchmark when varying the number of threads from 2 to 24. Similar to the *count-based* and *delta-based* analyses, the speedups of both C++ threads and OpenMP using the *punctuation-based* windowing policy proportionally scales with the number of threads. This occurs because the delimiter value is generated enough for producing sufficient windows and feeding the worker entities. Conversely, Intel TBB has to deal with the same drawbacks detected during the evaluation of the previous policies, thus limiting the global speedup scaling beyond 8 threads.

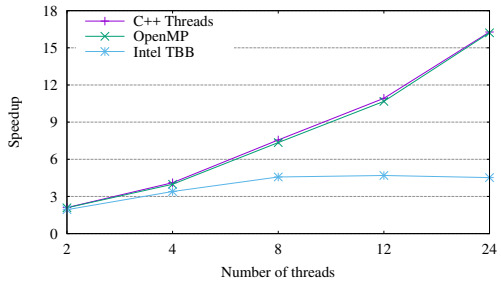


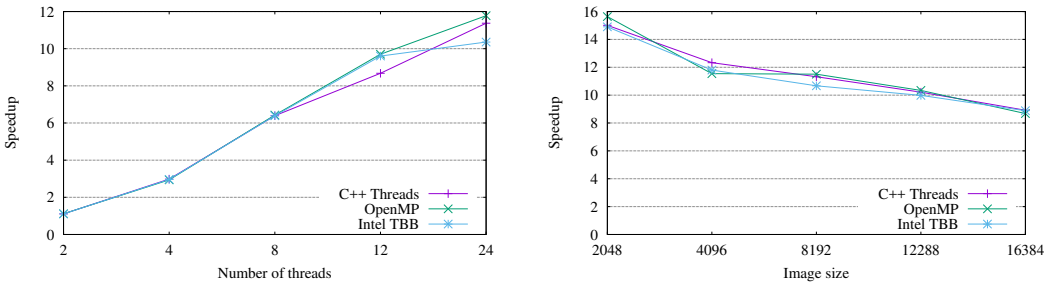
FIGURE 4.16: Windowed-Farm speedup with varying number of threads using *punctuation-based* windows

In a nutshell, the different windowing policies presented for the Windowed-Farm, allows users to model a wide range of scenarios that can appear in Data Stream Processing applications. However, the user needs to be aware of the advantages and drawbacks inherent to each policy and GRPPI backend.

4.6.9 Performance analysis of the Stream-Iterator pattern

Finally, we analyze the performance of the GRPPI Stream-Iterator pattern using the above-mentioned benchmark, in charge of processing square images and halving their sizes on each iteration until reaching concrete resolutions. Specifically, the size of the input images is fixed to 8,192 pixels, and the output images, for each input, have sizes of 128, 512 and 1,024. Figure 4.17(a) illustrates the benchmark speedup when varying the number of threads from 2 to 24 for the different GRPPI backends. In this case, when the number of threads ranges between 2 and 12, the efficiency attained is roughly 75 %, while for 24 this is degraded to 48 % for all programming frameworks. This effect is mainly caused by the fact that each of the threads involved

in the Farm pattern, used in the Stream-Iterator, are simultaneously accessing to different input images. Therefore, these memory accesses become a bottleneck due to constant cache misses when the threads perform the computation of the `task` function of the pattern. In general, these results suggest a memory bandwidth limitation in this particular benchmark.



(A) Speedup vs. number of threads.

(B) Speedup vs. image size.

FIGURE 4.17: Stream-Iterator speedup with varying number of threads and image size.

To gain insights into the performance degradation detected in the previous analysis, we perform an additional experiment in which we set the number of threads to 24 and vary the input image sizes from 2,048 to 16,384. Figure 4.17(b) depicts the performance gains for the different execution frameworks when varying the image size in the preceding range. Again, we observe a slight speedup decrease for increasing image sizes, which confirms our prior impressions. As an example, if we assume 22 worker threads in the internal Farm pattern, individually processing images with a resolution of 2,048×2,048 pixels (represented with matrices of integers), these require about 352 MiB of memory. Therefore, not fitting in any of the available cache levels and leading to an increased L2/L3 cache miss rate when they are simultaneously accessed. All in all, this issue is mainly due to the inherent memory-bound nature of this specific use case.

In a nutshell, in the light of the evaluation performed in these sections, we can conclude that GrPPI greatly aid developers to design and implement parallel applications. Thanks to its unified and generic interface, GrPPI reduces the efforts and the required knowledge and expertise on parallel programming without incurring significant overheads. Additionally, its capability of composing parallel patterns allows arranging complex constructions able to represent a wider range of algorithms in parallel applications.

4.7 Summary

In this chapter, we have proposed a generic and reusable parallel pattern interface that acts as a switch between existing frameworks. This way, developers can benefit from this interface to diminish the required efforts to select the most suitable framework and to migrate parallel code from one framework to another. Thanks to this interface in conjunction with the parallel pattern analyzer tool ease transforming sequential code into parallel code. However, the transformation is still a time-consuming task that should be made manually by the developers. Additionally, optimize and tune the resulting parallel applications for the target platform requires additional efforts. In the following chapter, we propose a framework that

is able to transform the annotated parallel patterns by the PPAT tool into optimized GRPPI parallel pattern calls.

Chapter 5

Automated Patern-based Refactoring

As shown in previous chapters, we provide a generic parallel pattern interface and a tool, able to find parallel candidates in sequential source codes. However, at this point, transforming the annotated code into parallel is manually performed by the developers. Furthermore, the detection may lead to unbalanced patterns that require being optimized to better exploit the available resources. In this Chapter, we describe the Parallel Pattern Refactoring Framework, namely PPRF, that, using PPAT and GRPPI, is able to generate optimized parallel codes. Basically, this framework detects Pipeline and Farm candidates in sequential C++ codes using PPAT, balances and optimizes the pipelines found and generates parallel code using the GRPPI pattern interface.

5.1 Parallel Pattern Refactoring Tool

As mentioned, this tool leverages PPAT to identify parallel patterns and GRPPI as a target interface for the transformed code. Figure 5.1 shows the general workflow of this framework. In general, this framework takes the sequential code and uses PPAT in order to detect parallel pipelines and farms and the resulting annotated code feds the refactoring module. This module extracts the sequential code from the annotated regions and generates GRPPI code with some instrumentation. Afterward, the information collected during the execution is used to evaluate the current parallelization on the target platform and iteratively improve Pipeline load-balance. Finally, after the whole optimization process is finished, the framework generates the GRPPI code using the obtained pattern configuration. Specifically, this framework performs the following steps:

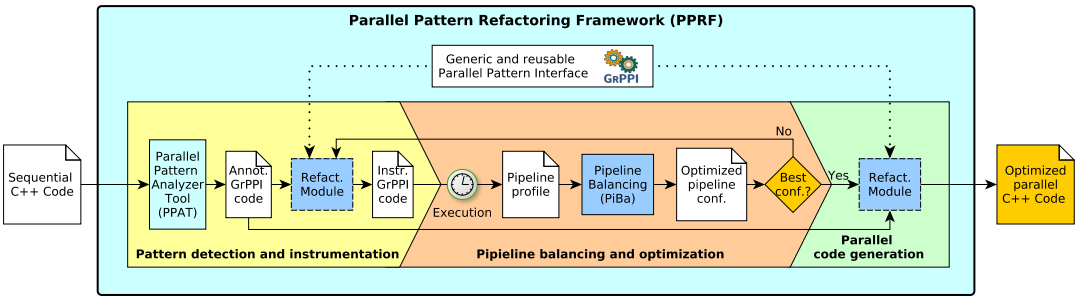


FIGURE 5.1: Workflow of the Parallel Pattern Refactoring Framework.

Pattern detection and instrumentation. During the first stage, the Parallel Pattern Analyzer Tool (PPAT) receives the sequential C++ code to be analyzed. Internally, PPAT detects code constructions that match with any of the supported patterns (Pipeline and Farm). Afterward, the new refactoring module introduces the GRPPI pattern interfaces accordingly on those constructions. Additionally, it instruments the Pipeline stages for measuring their execution time.

Pipeline balancing and optimization. In this phase, the application is run to collect average execution times of the Pipeline stages. Using this execution time data, we feed our Pipeline Balancing Algorithm (PIBA) for generating its optimal configuration. This algorithm merges and replicates the Pipeline stages according to the available CPU cores. Additionally, PIBA is able to determine the optimal concurrency degree by refining the Pipeline arrangement iteratively until finding the configuration that delivers an optimal performance. Note that this framework uses profiling techniques to measure Pipeline stage execution time; therefore, the input data used during the profiling phase should be representative enough to perform the optimizations.

Parallel code generation. Finally, the refactoring module receives the Pipeline configuration and generates the parallel code version using the GRPPI interface. Specifically, the refactoring module is rerun to generate the final Pipeline arrangement according to the configuration determined by PIBA in the previous step.

Listings 5.1-5.3 depicts a worked example of the aforementioned PPRF steps. Listing 5.1 shows an excerpt of sequential code containing potential parallel patterns. Once PPAT has been executed and parallel patterns have been found, the original code is transformed using the GRPPI interface and instrumented for measuring the execution time of the detected Pipeline stages (see Listing 5.2). Note that this instrumentation is provided internally by the GRPPI execution policy. With the Pipeline profile data, PIBA is able to determine the best configuration, merging the first and second Pipeline stages and using 2 threads for executing the third stage using a Farm construction. Finally, Listing 5.3 illustrates the transformed code taking into account the Pipeline arrangement stated by PIBA in the previous step. Thanks to this framework, a sequential code matching with a Pipeline pattern can be automatically transformed into parallel and optimized for the target platform.

5.2 Pipeline Stage Balancing Algorithm

In this section, we describe in detail the Pipeline Balancing Algorithm, namely PIBA, as part of the PPRF framework. Particularly, this algorithm tries to compute the optimal, or near optimal, arrangement of the Pipeline stages to optimize the use of all the available resources, i.e., CPU cores, for the target application.

We make first the following assumptions about the Pipeline construction. Consider a Pipeline $\mathcal{P}$ as the following list of stages

$$\mathcal{P} = (\lambda_1, \lambda_2, \ldots, \lambda_n),$$

where the i-th stage λ_i is the tuple (γ_i, t_i, r_i), being γ_i the function kind (pure or impure), t_i the execution time, and r_i the number of replicas or worker entities that execute the stage. Note that if the function related to a stage λ_i is pure, it can be

LISTING 5.1: Sequential code.

```
while( (i+=1) < MAX ){

  auto it = doStuff(i);

  it = doStuff(it);

  it = pureFunction(it);

  cout << it << endl;

}
```

LISTING 5.2: Instrumented code.

```
sequential_execution seq_exec{};
seq_exec.enable_instrumentation();
grppi::pipeline(seq_exec,
  [&]() -> optional<int> {
    return ( (i+=1) < MAX ) ? i : {};
  },
  [](int it){ return doStuff(it); },
  [](int it){ return doStuff(it); },
  grppi::farm(1,
    [](int it){ return pureFunction(it); }
  ),
  [&](int it){ cout << it << endl; }
);
```

LISTING 5.3: Optimized parallel code.

```
// GrPPI parallel pipeline + farm
// pipeline with 1 thread per stage
grppi::pipeline(parallel_execution_native{},
  [&]() -> optional<int> {
    return ( (i+=1) < MAX ) ? i : {};
  }, // Fusioned stages 1+2
  [](int it){ it = doStuff(it);
              return doStuff(it); },
  grppi::farm(2,
    [](int it){ return pureFunction(it); }
  ),
  [&](int it){ cout << it << endl; }
);
```

executed in parallel by multiple replicas using a Farm pattern. Otherwise, if the function is impure, it can only be executed in series by one replica.

We next define the *stage service time* as the division of its execution time between the number of its assigned replicas (see Equation 5.1). We also define the *pipeline service time* as the maximum service time of its stages (see Equation 5.2). Note that the *service time* is the inverse of the throughput, i.e., units of time per processed item.

$$ST_{\lambda_i} = \frac{t_i}{r_i} \tag{5.1}$$

$$ST_{\mathcal{P}} = \max(ST_{\lambda_1}, ST_{\lambda_2}, \ldots, ST_{\lambda_n}) \tag{5.2}$$

Therefore, the goal of the PIBA algorithm is to find the Pipeline equivalent to the original one with the minimum *service time* using all the available CPU cores for the target application. For that, PIBA receives two input parameters: the Pipeline profile containing the execution time related to its stages, and the number of available CPU cores. Next, it leverages the following two techniques to find the optimal, or near optimal, Pipeline stage arrangement:

- *Stage replication*: this technique increases the parallelism degree of those stages that follow the Farm pattern. Basically, given the definition of the Farm pattern, the parallelism degree of a Farm stage can be increased by introducing a new concurrent entity, i.e., $r_i = r_i + 1$. With this, the stage throughput is improved.

- *Stage merging*: this technique merges two consecutive stages in order to reduce the total number of stages. However, this leads to an increased execution time of the resulting merged stage that is the sum of both stage execution time.It is important to remark that if the stages are pure functions, the new merged stage is also pure and can be executed in parallel. Since this strategy

is intended to free resources to be used in the slowest stage or to use as many threads as cores, the number of replicas if both merged stages are pure will be the sum of both stages replicas minus one. This way, at least, the number of cores used in the pipeline will be one less than the previous pipeline arrangement. In any other case, i.e. if one of the merged stages is impure, the resulting stage is also impure and the number of replicas will be only one since it cannot be replicated. This technique is detailed in Algorithm 1.

Algorithm 1: Stage merging technique.

```
Function mergeStages(stageA, stageB, mergedStage)
    if stageA.kind = Pure & stageB.kind = Pure then
        mergedStage.kind ← Pure
        mergedStage.time ← stageA.time + stageB.time
        mergedStage.replicas ←
            stageA.replicas + stageB.replicas − 1
    else
        mergedStage.kind ← Impure
        mergedStage.time ← stageA.time + stageB.time
        mergedStage.replicas ← 1
    end
```

Using these techniques, in the following sections we present the three different alternatives implementing the PIBA algorithm using *i)* brute-force search; *ii)* heuristic search; and *iii)* an hybrid solution combining *i* and *ii*.

5.2.1 The brute-force search

Regarding the first alternative, we present the naive version of the PIBA algorithm based on brute-force search. As can be seen in Algorithm 2, this procedure uses the aforementioned *stage merging* technique (`genMerges` function) in order to compute all possible merging combinations for different number of stages, i.e., from 2 to the minimum between the number of cores and the number of stages. Then, for each of these combinations, the function `genReplicas` leverages the *stage replication* technique to calculate all feasible replications of the Farm stages. It is important to remark that these replications are made until the number of replicas equals the cores. Finally, the algorithm returns the Pipeline configuration with the minimum *service time*.

However, generating all possible merging combinations of stages and replicas has a non-negligible computational cost of $\Omega(n^3)$ and $O(c^n)$ for the best and worst cases, respectively. In this case, n stands for the combination of the number of stages and threads, while c represents a constant. Precisely, the best case corresponds with a full sequential Pipeline, as the function `generateReplicas` is not used. On the contrary, the worst case is related to a Pipeline whose stages are all Farm constructions.

5.2.2 The heuristic approach

As noted in the previous section, the computational cost of the brute-force search is prohibitive, therefore it is necessary to design an heuristic so as to provide a solution within a reasonable time frame. Algorithm 3 presents the heuristic search of PIBA. This is implemented as an iterative procedure, where the *pipeline service time* is improved in each step. First, the heuristic calculates: *i)* the slowest

Algorithm 2: PiBa brute-force

```
Function PiBaBruteForce(
        pipeline[],                                  // Stages of the pipeline
        numCores)                                    // Number of cores

    for i ← 2 to min(numCores, pipeline[].numStages) do
        // Generate all the possible combinations with i stages
        pipeMerged[] ← genMerges(pipeline[], i)
        for j ← 0 to numCores − i do
            // Generate all the possible pipelines with j replicas
            pipeCombs[] ← genReplicas(pipeMerged[], j)
        end
    end
    pipeline[] ← minServiceTimePipeline(pipeCombs[])
```

sequential and Farm stages (`getMaxSeqStage` and `getMaxFarmStage`, respectively); and *ii)* the merged stage of two consecutive stages with the minimum *service time* (`mergeConsecutiveStages` and `getMinStage`), using the *stage merging* technique. Note that the function `mergeConsecutiveStages` considers all possible mergings among consecutive Pipeline stages on a given iteration. Afterwards, depending on the current Pipeline state, the heuristic performs iteratively one of the subsequent actions in the following order:

A1 If there are Farm stages and the total number of current replicas is less than the number of cores, the Farm stage with the maximum *service time* is granted with an additional replica.. Note that the purpose of this action is to use all available CPU cores for the target application, though in extreme cases the Pipeline throughput improvement might be marginal.

A2 If the total number of replicas is greater than the number of cores, the merge of two consecutive stages with the minimum *service time* is incorporated in the resulting Pipeline.

A3 If the slowest stage is a Farm and it is slower than the merge of two consecutive stages with the minimum *service time*, the algorithm adds a replica to the slowest Farm stage and incorporates the merged stage in the resulting Pipeline.

A4 If none of the previous conditions are met, the procedure finishes, as it cannot reduce the Pipeline *service time* anymore.

To illustrate the workings of the PIBA heuristic, Table 5.1 shows an example of Pipeline that is steadily improved using this procedure. We start from a Pipeline comprised of three sequential and three Farm stages which are intended to run on four cores. Note that $\mathtt{s}_i(t_i, r_i)$ stands for sequential stages and $\mathtt{f}_i(t_i, r_i)$ for Farm stages, executed by r_i replicas. In the first iteration, the PIBA heuristic performs the action **A2**, since the total number of replicas is greater than the number of cores. Hence, $\mathtt{f}_5$ and $\mathtt{s}_6$ are merged into the single stage $\mathtt{s}_{5,6}$. In the second iteration, action **A2** is taken again so as to merge $\mathtt{s}_4$ and $\mathtt{s}_{5,6}$ in $\mathtt{s}_{4,5,6}$. In the third iteration, actions **A1** and **A2** are discarded, given that the number of replicas is equal to the number of cores. Therefore, since the slowest stage corresponds to a Farm and its *service time* is greater ($t_3 = 7$) than the fastest merged stage ($t_{2,3} = {}^{12}/_{2} = 6$), action **A3** merges $\mathtt{f}_2$ and $\mathtt{f}_3$. Finally, as the Pipeline *service time* cannot be reduced anymore and the Pipeline replicas equal to the number of cores, the procedure finishes its execution.

Algorithm 3: PIBA heuristic

```
Function pibaHeuristic(
        pipeline[],                                  // Stages of the pipeline
        numThreads)                                       // Number of threads

    Function getServiceTime(stage)
        return getTime(stage)/getReplicas(stage)

    while true do
        maxFarm ← getMaxFarmStage(pipeline[])
        maxSeq ← getMaxSeqStage(pipeline[])
        sumPipe[] ← mergeConsecutiveStages(pipeline[])
        minMerge ← getMinStage(sumPipe[])
        if countFarms(pipeline[]) > 0 &  countReplicas(pipeline[]) < numThreads
         then
            maxFarm.replicas ← maxFarm.replicas + 1
        else if countReplicas(pipeline[]) > numThreads then
            applyMerging(pipeline[], minMerge)
        else if countFarms(pipeline[]) > 0
              & getServiceTime(maxFarm) >      getServiceTime(maxSeq)
              & getServiceTime(maxFarm) >
                getServiceTime(minMerge) then
            maxFarm.replicas ← maxFarm.replicas + 1
            applyMerging(pipeline[], minMerge)
        else
            break
        end
    end
```

Analyzing the computational cost of the PIBA heuristic shown in Algorithm 3, we determine its complexity as $\Theta(n^2)$ for all cases.

TABLE 5.1: PIBA working example.

It.	Pipeline	Max. Stage	Replicas	Min. Merge	Action
1	$(s_1(3,1), f_2(5,1), f_3(7,1), s_4(2,1), f_5(1,1), s_6(1,1))$	$f_3(7,1)$	6	$(f_5 + s_6) \rightarrow s_{5,6}(2,1)$	**A2**
2	$(s_1(3,1), f_2(5,1), f_3(7,1), s_4(2,1), s_{5,6}(2,1))$	$f_3(7,1)$	5	$(s_4 + s_{5,6}) \rightarrow s_{4,5,6}(4,1)$	**A2**
3	$(s_1(3,1), f_2(5,1), f_3(7,1), s_{4,5,6}(4,1))$	$f_3(7,1)$	4	$(f_2 + f_3) \rightarrow f_{2,3}(12,2)$	**A3**
4	$(s_1(3,1), f_{2,3}(12,2), s_{4,5,6}(4,1))$	$f_{2,3}(12,2)$	4	$(s_1 + f_{2,3})(15,1)$	**A4**

5.2.3 The hybrid approach

The hybrid approach combines the benefits from both brute-force and heuristic search approaches. This variant leverages the heuristic search presented in Algorithm 3, reducing the number of replicas until reaching the total number of available cores. From that point on, it continues improving the Pipeline *service time* using the brute-force search, as presented in Algorithm 2. Thereby, it reduces the computational best-case cost of the brute-force algorithm to $\Omega(n^2)$, providing more accurate *service time* optimizations than the single heuristic search approach. However, the worst-case cost is still $O(c^n)$, as the brute-force search.

5.2.4 Finding the optimal concurrency degree

According to the previous section, the PIBA algorithm transforms the Pipeline stages arrangement to have the same number of replicas as CPU cores used by the target

application. This algorithm also balances, as much as possible, the stages workload. However, in real scenarios, the resulting Pipelines cannot be perfectly balanced in most of the cases. These situations cause bottlenecks due to imbalanced stages, which leads to underused resources (cores). A way to improve the resource usage, and thus the Pipeline performance, is to increase the number of replicas (threads) above the total number of cores. Hence, the additional threads can leverage the partially idle resources, overlapping threads contention with useful computation. In this context, we refer to oversubscription when an application uses more threads than available CPU cores and relies on the OS scheduler to keep them all busy.

To motivate this issue, we have implemented a synthetic benchmark consisting of two Pipeline collections, using CPU- and memory-intensive stages, respectively. These collections were comprised of 1,500 randomly-generated Pipelines constituted by a number of stages ranging from 4 to 12 and the percentage of Farm stages varying between 30 % and 90 %. Afterward, these Pipelines were processed using the PIBA algorithm to adjust the number of stages/replicas in the range of 4 to 24 threads. Next, we executed these Pipelines on an 8-core platform to find out their optimal concurrency degree. Figure 5.2 shows the performance attained by six representative CPU-/memory-intensive Pipelines for different number of threads.[1] Note that the lines extending vertically from the points represent the confidence intervals as a variability metric over 10 Pipeline runs. As can be observed, all Pipelines (P1–P6) improve their speedup up to the number of cores. Nevertheless, while some of them stop improving after this point, others continue boosting their performance beyond that. These results confirm our previous impressions where a concurrency degree higher than the total number of cores may improve the Pipeline performance in some cases.

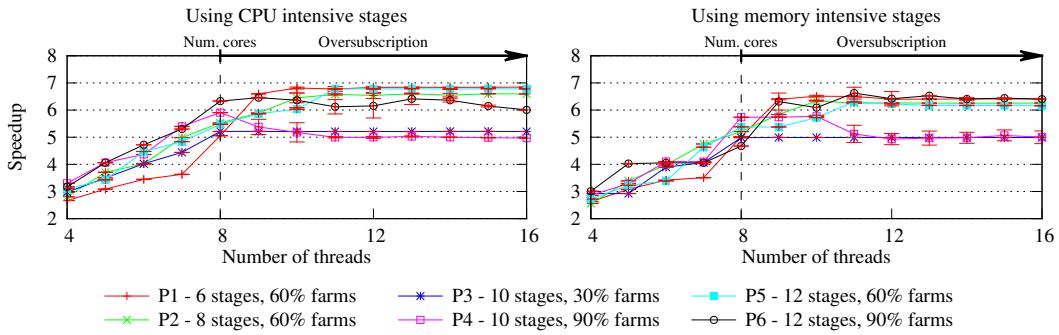


FIGURE 5.2: Analysis of the maximum concurrency degree using representative pipelines.

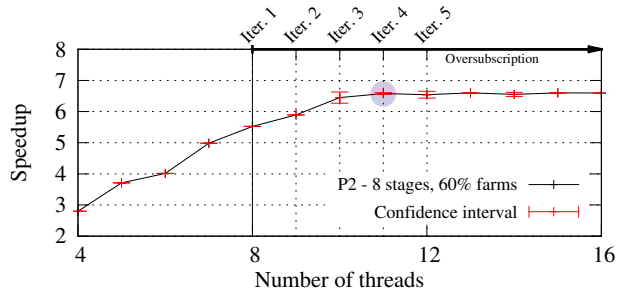
Iterative search

A naive search approach to obtain the optimal concurrency degree is to execute multiple times the PIBA algorithm for a different number of final replicas (instead of using the number of cores) and check which Pipeline arrangement delivers the best performance. However, this method is very time-consuming as the framework has to execute each time all feasible Pipeline combinations.

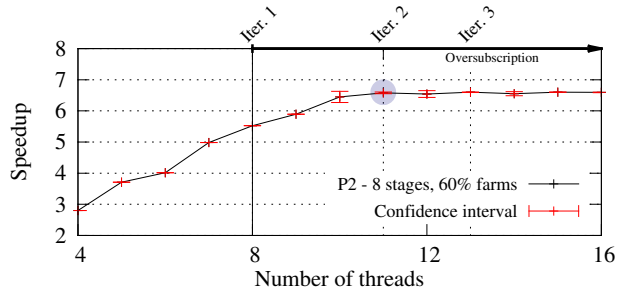
[1] Note that the CPU-intensive stages were mimicked by means of performing an arbitrary number of floating-point operations, while the memory-intensive compute was emulated accessing consecutively to 2 million elements in a floating-point array.

To optimize this naive search approach, we rely on the benchmark results that show, in many cases, strictly increasing speedup curves followed by monotonic decreasing ones. Note that this occurs in more than 90 % of the synthetic Pipelines. With this assertion, we can refine the previous approach with an iterative variant that applies PIBA to optimize the Pipeline configurations using a different number of threads, starting from the number of cores until the performance stops improving. Note that, to deal with the inherent variability of oversubscribed scenarios, we execute each configuration multiple times to calculate averaged reliable values. With the averaged speedups in hand, this iterative search algorithm checks, in each iteration, if the speedup of a configuration B using $n + 1$ threads is higher than A using only n threads. If this is true, it checks for the next configuration using one more thread. This search stops when the configuration B using $n + 1$ threads delivers worse or equal performance than A using n threads.

To illustrate the workings of this iterative search, Figure 5.3(a) depicts the case of the Pipeline P2 using CPU-intensive stages from Figure 5.2. In this scenario, the approach starts iterating from 8 cores on until the 4th iteration, given that in each step the configuration using $n + 1$ threads delivers higher speedups than using n threads. On the 5th iteration, i.e., with 12 threads, the performance stops improving. Then, the algorithm determines 11 threads as for the optimal concurrency degree. As a result, the Pipeline is oversubscribed with 3 threads, which overlap contention of the first 8 with useful computations.



(A) Iterative search.



(B) Greedy iterative search.

FIGURE 5.3: Automatic approaches for finding the optimal concurrency degree of PIBA balanced Pipelines.

Greedy iterative search

The iterative search approach reduces the Pipeline configurations that have to be tested using a trial and error method. However, this variant can be very slow when the optimal concurrency degree is far ahead of the number of cores. To improve this

algorithm, we propose a technique to obtain an estimation of the ideal degree. We start from the fact that the Pipelines are not perfectly balanced, and therefore, available resources are underused due to their inherent congestion. With this assertion, we estimate the Pipeline resource usage rate with

$$usage\_rate = \frac{eff\_usage}{real\_usage} = \frac{n\_items \cdot \sum_{i=0}^{n\_stages} t_i}{n\_cores \cdot exec\_time}, \tag{5.3}$$

where eff_usage and $real\_usage$ are estimations of the effective and real usage of the platform resources, respectively. In the equation, the effective usage is computed as the sequential execution time of the Pipeline stages multiplied by the number of items processed in a given test. Alternatively, the real usage is calculated as the execution time of the previous test multiplied by the total computational resources (cores). Using this technique, we propose a heuristic that pursues a greedy iterative approach and estimates in advance the ideal concurrency degree, instead of testing all configurations. This procedure, shown in Algorithm 4, iteratively applies the PIBA heuristic of Algorithm 3 with a number of threads that depends on the usage rate. This number is calculated on-the-fly dividing the current threads by the usage rate obtained with Equation 5.3 and rounding the result up. The procedure stops iterating when the usage rate using additional threads does not improve any more.

Algorithm 4: PIBA heuristic with oversubscription.

```
Function pibaOversubs(
        pipeline[],                                  // Stages of the pipeline
        numCores,                                    // Number of cores
        numItems)                                    // Number of items

    pibaPipeline[] ← pipeline[]
    pibaHeuristic(pibaPipeline[], numCores)
    execTimeCur[] ← exec(pipeline[], numItems)
    numThreads ← numCores
    repeat
        usageRate ← usageRate(pipeline[], numItems, numCores, execTimeCur[])
        execTimeOld[] ← execTimeCur[]
        numThreads ← ceil(numThreads / usageRate)
        pibaHeuristic(pibaPipeline[], numThreads)
        execTimeCur[] ← exec(pipeline[], numItems)
    until stopCondition(execTimeOld[], execTimeCur[])
```

To demonstrate the benefits of this algorithm over the iterative version, Figure 5.3(b) shows the steps taken for the same Pipeline P2. As can be seen, three steps are required to find an optimal concurrency degree. In the first iteration (using 8 threads), the resource usage rate is roughly 69 %; thus the algorithm determines that 3 additional threads are required. With it, during the second iteration (using 11 threads), the usage rate increases to 82.5 %. Hence, the algorithm determines that 13 threads are needed to exploit idle resources. After the execution with 13 threads, the procedure stops, as it detects that the usage rate does not improve. In this case, the optimal concurrency degree determined by the greedy iterative search is 11, i.e., coming to the same conclusion as the iterative algorithm but with fewer iterations.

5.3 Evaluation

In this section, we evaluate the pattern-based refactoring tool and the proposed balancing PiBA algorithms. To do so, we used the following benchmarks:

Pipe-Bench: A benchmark collection of 1,500 randomly-generated Pipelines, as described in Section 5.2.4, with a number of stages ranging between 4 and 12. These Pipelines were combined with both sequential and parallel stages, where the degree of the parallel (Farm) stages varied between 30 % and 90 %. To generate the Pipelines source codes we used a Python script to emit the Pipeline and Farm GrPPI patterns leveraging the C++ back end. It is important to remark that the workload type for the stages were CPU-bound computations by means of performing double-precision operations.

Video-Bench: A sequential video stream-processing synthetic benchmark composed of two types of filters (Gaussian Blur and Sobel operators) in order to detect edges appearing in the video frames.[2] Specifically, the application core works in a Pipeline fashion, in which the first and last stages read and write the video frames, respectively. Consequently, the intermediate stages apply the aforementioned filters in different ways. Regarding the workload type, the Gaussian Blur filter only performs arithmetic operations, while the Sobel operator also executes square root operations, both using double-precision numbers.

Lane-Detection: A real-world computer vision application for detecting road lane lines in autonomous driving systems. This application is composed of a Pipeline in charge of processing individual video frames using a series of filter algorithms, such as the Canny edge detector and the Hough transform [88]. This application is vital to steer vehicles, as lanes represent a constant reference to the road. Indeed, identifying lane lines on the road is one of the most fundamental vision tasks required by autonomous cruise controls, lane change assist, lane centering, etc.

The evaluation methodology of this section consists of the following parts. First, we analyze the presented Pipeline balancing algorithms and compare their time-to-solution and the execution time of the Pipelines in Pipe-Bench using the PPRF framework. Next, we evaluate the different strategies to find the optimal concurrency degree in terms of speedup gains, number of steps taken and accuracy. Afterward, we test PPRF with Video-Bench, in order to transform the sequential code into parallel and evaluate different configurations of the PiBA algorithm and GrPPI back ends. Additionally, we complement the study using Video-Bench with a fine-grained analysis of different Pipeline configurations via execution traces. Finally, we employ Lane-Detection to demonstrate the benefits of PPRF in a real-world application.

5.3.1 Reference platform

The evaluation has been carried out on a server platform equipped with 2× Intel Xeon Ivy Bridge E5-2630 v3 with a total of 16 cores running at 2.40 GHz, 20 MB of L3 cache and 256 GB of DDR3 RAM. The OS is a Linux Ubuntu 14.04.5 LTS with kernel 3.13.0-85.

[2] This benchmark has been inspired by an OpenCV edge detection example from http://docs.opencv.org/3.1.0/d3/d63/edge_8cpp-example.html.

5.3.2 Analysis of the Pipeline stage balancing algorithm

To evaluate the different versions of the PIBA algorithm, we leverage the collection of 1,500 synthetic Pipelines of PIPE-BENCH. Afterward, we balanced its Pipelines using the three variants of PIBA, i.e. brute-force, heuristic and hybrid searches and configured them to run on platforms having from 2 to 16 cores. Figure 5.4 depicts the averaged time-to-solution of the three PIBA variants for the different number of Pipeline stages and cores. As observed, the execution time of PIBA algorithms increases in general with the number of stages and cores. However, the growth rate of the brute-force is extremely large than that for the heuristic version, which confirms the exponential algorithm complexity stated in Section 5.2.1. A final observation is that the time-to-solution of the hybrid version is equal to the brute-force search for a number of stages lower or equal than the number of cores. Therefore, in those cases, the brute-force alternative is preferred one.

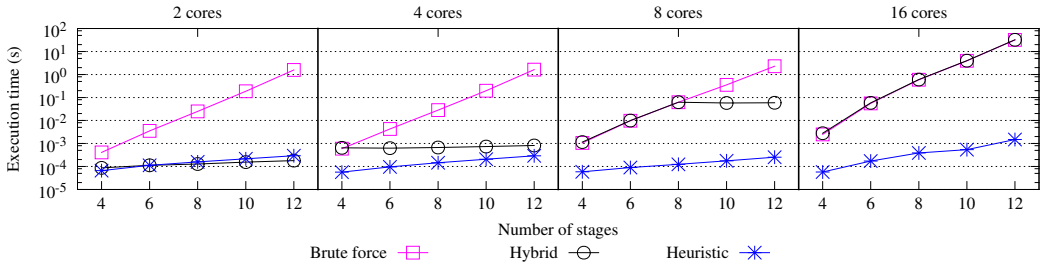


FIGURE 5.4: Time-to-solution of the basic PIBA algorithms.

Our next study analyzes the execution times of the same collection of Pipelines after balancing them using the three different PIBA versions. To emulate platforms with a different number of cores, we leveraged the Linux utility `taskset` for restricting the cores that can be used by the Pipeline threads.[3] As can be observed in Figure 5.5, the Pipelines processed using the heuristic and hybrid PIBA variants attain a similar performance in all cases. On the contrary, the brute-force search provides better speedup only when the number of cores is much lower than the stage count. Note that the observed speedup slowdown of the Pipelines balanced with PIBA is caused by the fact that the presented algorithms use as many threads as available cores. In contrast, the unbalanced Pipelines, always use at least as many threads as stages, leading to oversubscribed scenarios and, in some cases, to higher speedups with respect to the balanced Pipelines.

All in all, it can be concluded that the PIBA heuristic variant can provide acceptable well-balanced Pipelines in reasonable time frames, while the brute-force and hybrid searches are able to provide slightly better stage arrangements at the expense of prolonged time-to-solutions in the worst cases. Therefore, in the subsequent experiments of this paper, we select the heuristic variant as the default balancing PIBA algorithm.

5.3.3 Analysis of the optimal concurrency degree search algorithms

In this section, we analyze the extended PIBA algorithms proposed for finding the optimal concurrency degree with the basic PIBA variant. First, we compare the basic procedure with the two new strategies, the iterative and greedy iterative searches, able to obtain the ideal number of threads above the total number of cores. Next, we

[3]The `taskset` utility is used to set or retrieve the CPU affinity of a running process in Linux. http://linuxcommand.org/man_pages/taskset1.html

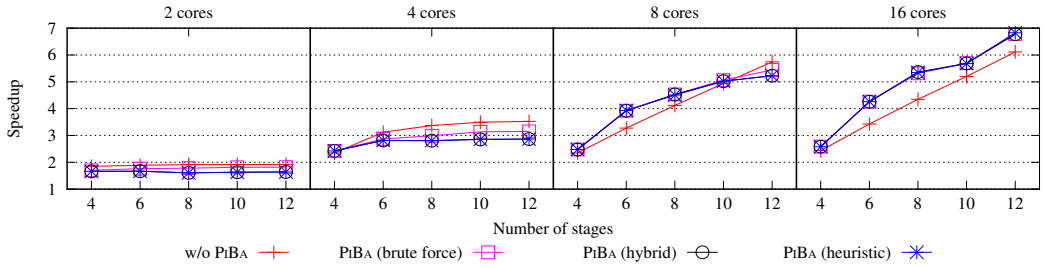


FIGURE 5.5: Speedup of the of Pipelines in PIPE-BENCH balanced with the basic PIBA algorithm variants.

examine the performance of the Pipelines in PIPE-BENCH on oversubscribed scenarios.

Figure 5.6 compares these algorithms regarding *i)* speedup, with respect to the sequential execution of the Pipeline; *ii)* the number of iterations needed until finding the optimal concurrency degree; and *iii)* the accuracy rate with regard to the best solution obtained with the brute-force approach. Note that these metrics were averaged over the 1,500 Pipelines comprised by the tested benchmark and using only eight cores. Focusing on the speedup results, it can be clearly seen that the extended PIBA algorithms for oversubscription are able to deliver better performance figures than using only the basic PIBA variant. This is mainly because the inherent threads contention is overlapped with useful computations of the exceeding threads. A detailed inspection of the executions revealed that this contention was caused by the threads suspended on an internal blocking queue when no items could be (de)queued.

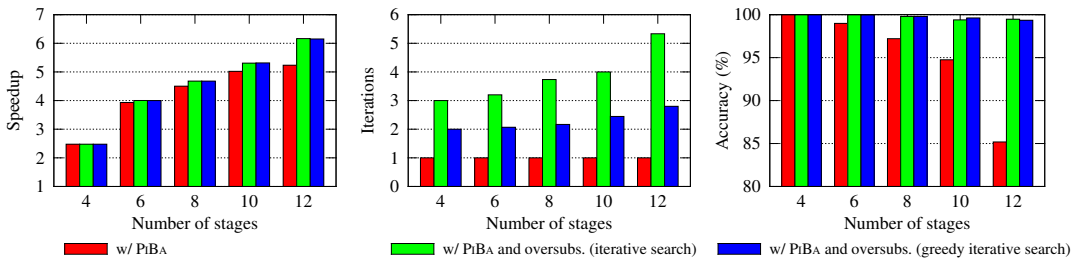


FIGURE 5.6: Comparative of the speedup, number of iterations and accuracy of the basic PIBA w.r.t the extended versions for finding the optimal concurrency degree.

Looking at the number of iterations taken by both PIBA algorithms including the oversubscription strategies, we observe that the iterative search requires more iterations with respect to the greedy approach. It is remarkable that this difference increases with the number of Pipeline stages since these Pipelines usually deliver better performance when employing oversubscribed threads. Finally, focusing on the accuracy, we notice that Pipelines comprising several stages attain lower accuracy than using only the basic PIBA algorithm. This is given by the fact the PIBA heuristic working alone does not always lead to optimal Pipeline arrangements when they contain many stages. For instance, from 10 stages on, the known error might be higher than 5 %. In contrast, with the extended PIBA algorithms, the accuracy is sustained regardless of the number of Pipeline stages. This demonstrates that the inherent flaws of the PIBA heuristic algorithm can be bypassed by increasing the concurrency degree using the extended PIBA algorithms.

In the light of the results, while both extended PIBA versions deliver similar performance figures, the iterative search requires more iterations than the greedy approach. Therefore, the experiments carried out hereafter are only performed using the greedy iterative search for finding the optimal concurrency degree.

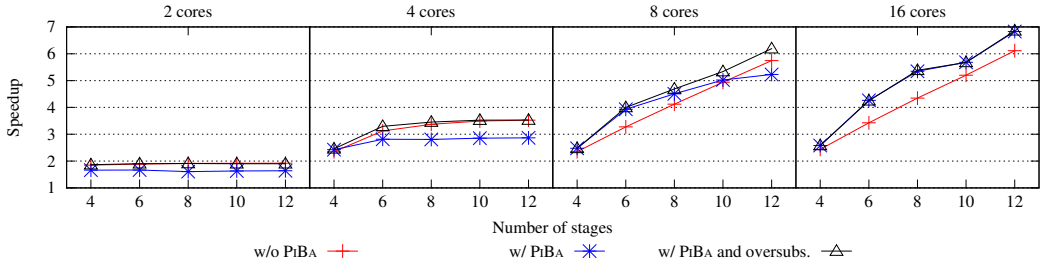


FIGURE 5.7: Speedup of the of Pipelines in PIPE-BENCH balanced with the extended PIBA algorithms for oversubscribed scenarios.

As a complementary study, Figure 5.7 analyzes the speedups attained by the Pipelines in PIPE-BENCH with a different number of cores when *i)* PIBA is not applied; *ii)* the basic PIBA algorithm is used; and *iii)* the extended PIBA algorithm along with the iterative greedy search are leveraged. As observed, when the number of stages is higher than the total cores, the fact of not balancing the Pipelines leads to oversubscribed scenarios, providing better performance than if the basic PIBA algorithm shortens the Pipelines. On the other hand, if the available cores are greater than the number of Pipeline stages, the balanced versions exploit better the resources. Looking at the results of the extended PIBA algorithm, the speedups are always higher or equal to the best case. The reason is that the oversubscribed threads in these Pipelines can effectively overlap the potential bottlenecks generated by the basic PIBA algorithm.

5.3.4 Evaluation with VIDEO-BENCH, a video streaming application

To evaluate the PPRF framework along with the Pipeline balancing algorithms in a real-world scenario, we leverage VIDEO-BENCH, a video stream-processing application able to detect edges appearing on the incoming frames. First, we feed the PPAT module with the sequential code in order to obtain an annotated version of the code with the potential parallel patterns detected. It is important to remark that, in this case, the main parallel pattern detected is a Pipeline where some of its stages, corresponding with the image filtering operations, are potential Farms. Next, we use the PPRF refactoring module to generate the parallel version of the code using the C++ threads and the Intel TBB GRPPI backends. Finally, we leverage the basic and extended PIBA algorithms to improve the performance of VIDEO-BENCH.

Given that both filters Gaussian Blur and Sobel are pure functions by nature, we have slightly modified VIDEO-BENCH in order to include impure versions of these filters. This way, the Pipeline stages can be detected as sequential or potential Farms, depending on the filter version encountered. Thus, for the subsequent experiments, we have developed three different versions of the VIDEO-BENCHPipeline, composed by 10 intermediate filtering operations, using both pure and impure versions. The Pipeline stages of these three VIDEO-BENCH variants have been arranged in the following way: *i)* fully sequential `(s|s|s|s|s|s|s|s|s|s|s|s)`; *ii)* combined `(s|f|f|s|f|f|f|s|f|f|s|s)`; and *iii)* fully parallel `(s|f|f|f|f|f|f|f|f|f|f|s)`. As a mere example of the steps taken by PPRF, Listing 5.4 shows the combined

Pipeline variant of the VIDEO-BENCH original code. Next, Listing 5.5 shows the result after executing PPAT and generating the instrumented code. Finally, Listing 5.6 shows an optimized version of the VIDEO-BENCH parallel code according to the arrangement proposed by PIBA using the profile execution data.

- fully sequential `(s|s|s|s|s|s|s|s|s|s|s|s).`
- combined `(s|f|f|s|f|f|f|s|f|f|s|s).`
- fully parallel `(s|f|f|f|f|f|f|f|f|f|f|s).`

As a mere example of the steps taken by PPRF, Listing 5.4 shows the combined Pipeline variant of the VIDEO-BENCH original code. Next, Listing 5.5 shows the result after executing PPAT and generating the instrumented code. Finally, Listing 5.6 shows an optimized version of the VIDEO-BENCH parallel code according to the arrangement proposed by PIBA using the profile execution data obtained with the previously instrumented code.

LISTING 5.4: Sequential code.

```
for (;;) {
  Mat fr;

  if (!cap.read(fr)) break;

  Gaussian_pure(fr, ... );

  Sobel_pure(fr, ... );

  Gaussian_impure(fr, ... );
   ...
  Sobel_pure(fr, ... );

  Gaussian_pure(fr, ... );

  Sobel_impure(fr, ... );

  imshow("edges",fr);
}
```

LISTING 5.5: Instrumented parallel code.

```
sequential_execution seq_exeq;
seq_exec.enable_instrumentation();
grppi::pipeline(seq_exec,
  [] () -> optional<Mat> {
    Mat fr;
    if (!cap.read(fr)) return {};
    else return fr;
  },
  grppi::farm(seq_exec,
    [](Mat fr) {
      Gaussian_pure(fr, ... );
      return fr;
    }
  ),
  ...
  [](Mat fr) {
    Sobel_impure(fr, ... );
    return fr;
  },
  [](Mat fr) { imshow("edges",fr); }
}
```

LISTING 5.6: Optimized parallel code.

```
grppi::pipeline(parallel_execution_thr{},
  [] () -> optional<Mat> {
    Mat fr;
    if (!cap.read(fr)) return {};
    else { Gaussian_pure(fr, ... );
           return fr; }
  },
  grppi::farm(parallel_execution_thr{2},
    [](Mat fr) {
      Sobel_pure(fr, ... );
      return fr;
    }
  ),
  ...
  [](Mat fr) {
    Gaussian_pure(fr, ... );
    Sobel_impure(fr, ... );
    return fr;
  },
  [](Mat fr) { imshow("edges",fr); }
}
```

Performance evaluation

Given the foregoing, in this section, we assess the proposed basic and extended PIBA algorithms using the three variants of the VIDEO-BENCH Pipeline. These experiments have been run on 6, 12 and 24 cores using the C++ threads and Intel TBB GRPPI

back ends. Figure 5.8 depicts the frames per second delivered by each of the VIDEO-BENCH versions. Focusing on the fully sequential Pipeline (left-hand side plot), we observe that the fact that all stages are sequential prevents PIBA from replicating stages, and thus, the impact of the potential bottlenecks cannot be reduced. Therefore, the performance either using any of the PIBA algorithms or not is limited by the slowest stage.

Looking at the Pipeline comprised of sequential and parallel Farm stages (center and right-hand side plots), we notice a completely different behavior. Our first observation is that using the basic PIBA algorithm and C++ threads back end, the application can benefit from the stage merging and replication techniques. Basically, when two stages executed by individual threads are merged, a resource (core) becomes available and can be exploited by an additional replica (thread) in the slowest Farm stage. The PIBA benefits are more pronounced when increasing the number of cores compared with the non-balanced Pipelines. On the other hand, the results with the extended PIBA algorithm even outperform the throughput delivered by those processed with the basic PIBA version. Note that the percentages above the bars indicate the oversubscription degrees calculated by the greedy search approach. These improvements are achieved by using more threads than available cores. Therefore additional replicas can accelerate Pipeline bottlenecks by overlapping contention of the fastest stages with useful computations.

On the contrary, setting TBB as the GRPPI back end, the use of the basic PIBA algorithm negatively affects the performance, as TBB does not allow to establish the number of replicas in Pipeline stages. Indeed, TBB uses an entirely different approach for executing stages: it leverages task-parallelism with a pool of worker threads that continuously poll for work from a ready task queue [61]. Therefore, for each item in the Pipeline, a new task is created and picked by a worker thread as soon as its dependencies have been resolved. In this sense, the fact that PIBA uses stage merging leads to longer but fewer Pipeline stages and results in larger congestion. In some cases, we observe that using 24 cores, extended PIBA with the C++ threads GRPPI back end attains better performance results than using TBB. All in all, we can conclude that the extended PIBA algorithm is the recommended option when using the C++ threads GRPPI back end, while for TBB the recommended solution is not to balance the Pipelines.

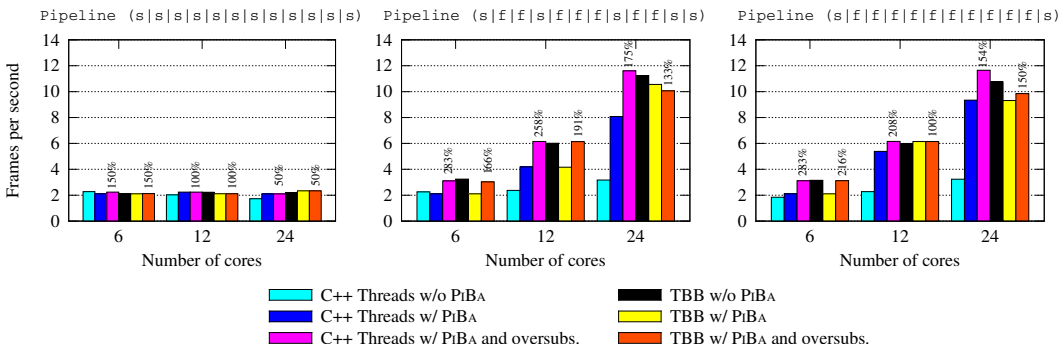


FIGURE 5.8: Frames per second obtained by the different Pipeline variants of VIDEO-BENCH using C++ threads and Intel TBB GRPPI backends and the basic and extended PIBA algorithms.

Fine-grain analysis

In this section, we complement the study with a fine-grained inspection of different Pipeline configurations to analyze their internal behavior. This study has been carried out using the C++ threads GRPPI back end instrumented with the Extrae library [25] to obtain execution traces. Afterward, the traces are visualized using the Paraver tool [69]. We only focus on the combined Pipeline comprising both sequential and parallel stages and using 12 cores. Figure 5.9 depicts a task trace and the number of simultaneously active threads during the execution of VIDEO-BENCH without having used any of the PIBA algorithms. Similarly, the left-hand side plot in Figure 5.11 represents the time percentage spent by the threads for the different states. Note that the colors in the trace and plot represent three different states: *i) in-stage* stands for the effective computation of the stages; and *ii) enqueue* and *dequeue* represent blocking states due to communications between stages via queues. As can be seen, the stages 1, 5 and 8 correspond to the slowest stages (bottlenecks), which dictate the total execution time. Also, the number of simultaneously active threads is always lower than 6, i.e., half of the available cores. Correspondingly, only half of the threads are simultaneously performing useful computations during the execution.

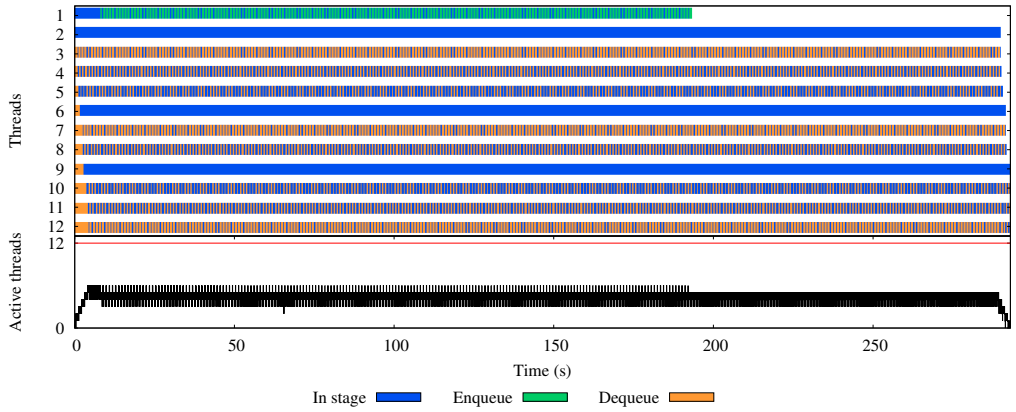


FIGURE 5.9: Task trace and number of active threads during the execution of the application without PIBA.

With the previous results, it can be observed that if the Pipeline were balanced, the resources could be better exploited, and thus, the execution time could also be improved. Given that, we make use of the basic PIBA algorithm to provide a better-balanced stage arrangement. As shown in Figure 5.10, the VIDEO-BENCHPipeline processed by the basic PIBA algorithm generates a trace where the stages (threads) show much lower contention times. This is by the fact that the workload among stages is better balanced than if PIBA is not applied. Focusing on the right-hand side plot in Figure 5.11, we observe that PIBA has assigned additional replicas to those bottlenecks detected in the previous experiment (stages 1, 5 and 8). We also observe that the simultaneous active threads are, in the beginning, close to 12. As the execution progresses, and as soon as the stages complete processing the items, the number of active threads slightly decays until the end. In this case, thanks to the basic PIBA algorithm the execution time has been reduced by a factor of 30 %.

Focusing again on the trace in Figure 5.10, it can be seen that some of the resources are underused. This reveals an opportunity to further improve the VIDEO-BENCH execution time. In consequence, we leverage the extended PIBA algorithm to exploit better available resources. Figure 5.12 depicts the execution task trace where the ideal concurrency degree was 253 %, i.e., 31 threads running on 12 cores,

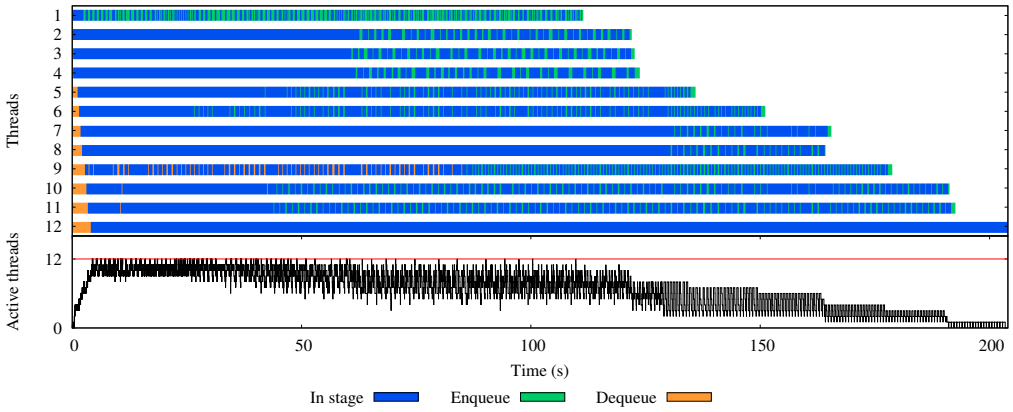


FIGURE 5.10: Task trace and number of active threads during the execution of the application with the basic PIBA algorithm.

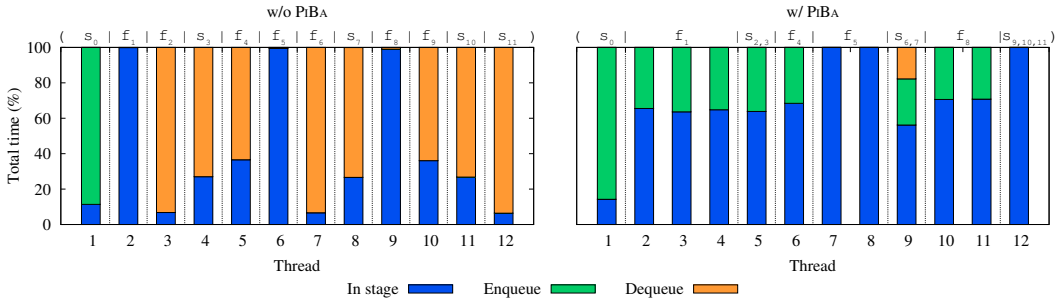


FIGURE 5.11: State time percentage per thread/stage of the VIDEOBENCHPipeline w/o and w/ the basic PIBA algorithm.

according to the results in Figure 5.8. A first inspection of the resulting trace reveals a higher contention ratio due to queue communications. However, as the contention is shared among the fastest stages, it allows exceeding threads to exploit resources freed by such contentions (see Figure 5.13). On the other hand, the active threads are mostly sustained above 12 during the entire execution. Note that having more active threads than available cores results in suspended threads waiting for CPU time. In the end, the extended PIBA algorithm leads to better performance than using only the basic variant. In this concrete case, the execution time has been reduced by a factor of 60 % with respect to the non-balanced Pipeline.

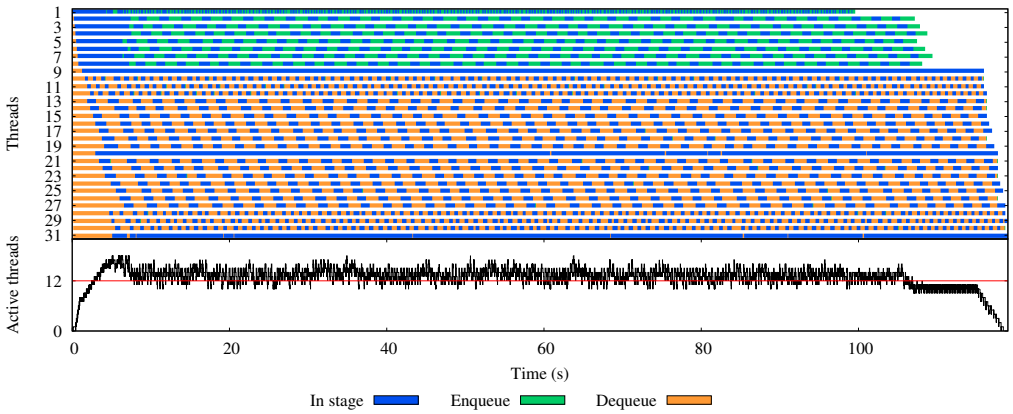


FIGURE 5.12: Task trace and number of active threads during the execution of the application with the extended PIBA algorithm.

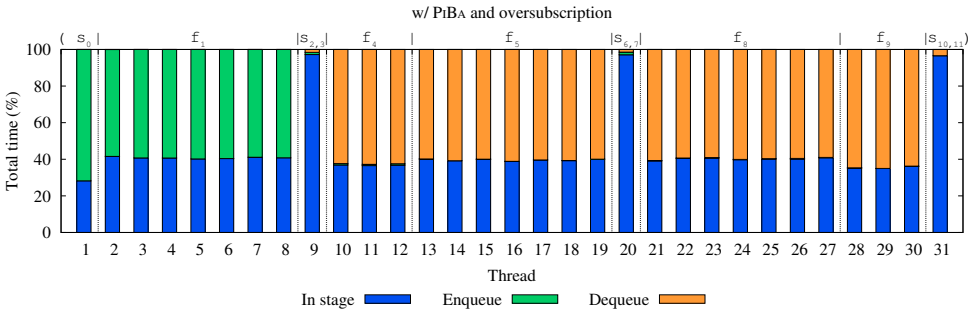


FIGURE 5.13: State time percentage per thread/stage of the VIDEO-BENCHPipeline using the extended PIBA algorithm.

5.3.5 Evaluation of LANE-DETECTION

In this section, we evaluate the proposed PPRF framework using a sequential real-world computer vision application able to detect road lane lines in autonomous driving systems. This stream-processing application processes individual video frames using a series of filter algorithms, such as the Canny edge detector and the Hough transform [88]. Figure 5.14 depicts the application workflow through an 11-staged Pipeline, where the first and last stages are processed in series, and the intermediate ones can be executed in parallel using the Farm pattern.

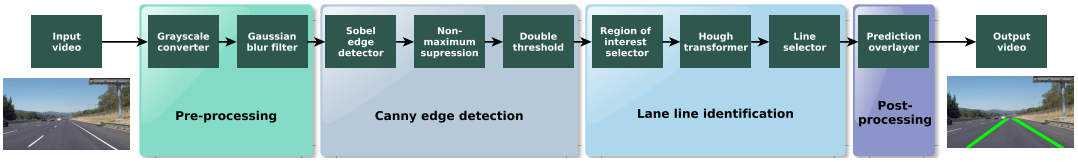


FIGURE 5.14: LANE-DETECTION application workflow.

In a first step, we use PPRF to introduce parallel patterns in the application source code. In this case, the PPAT module detects a Pipeline with the following structure `(s|f|f|f|f|f|f|f|f|s)`, i.e., where all intermediate stages can process individual video frames in parallel. Afterward, we employ the refactoring module along with both basic and extended PIBA algorithms to evaluate the application performance using the C++ threads, and the Intel TBB GRPPI back ends. Figure 5.15 shows the speedup obtained by LANE-DETECTION when using both back ends and PIBA versions running on 6, 12 and 24 cores. As can be seen for the C++ back end, the efficiency obtained when no PIBA algorithm is on average is 22 %, while for TBB is roughly 80 %. These contrasting results are given due to the different nature of both GRPPI back ends: C++ threads back end maps Pipeline stages onto threads, while TBB handles the processing of a stage on a given item as a task which is executed by one of the worker threads in its internal scheduler pool. On the other hand, when the basic PIBA algorithm is leveraged to balance the application Pipeline, the speedup obtained by the C++ threads back end is much closer to that delivered by TBB. Although the PIBA helped in balancing the Pipeline, there remain bottleneck stages which cause congestions in the faster ones. This fact leads the C++ threads back end to slightly reduced speedups compared to TBB. Finally, although the extended PIBA algorithm does not avoid these bottlenecks, the exceeding threads help in overlapping the contention with useful computations. Specifically, the C++ threads back end achieves comparable performance figures with respect to the TBB back end. Further, in some cases (e.g., 24 cores), the oversubscription used by the extended PIBA algorithm can even outperform the TBB performance.

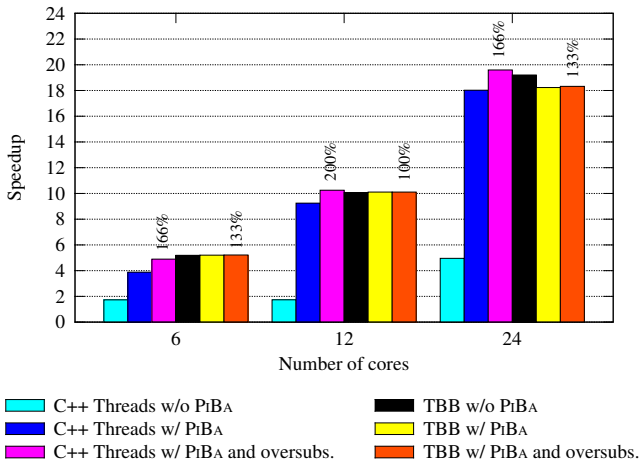


FIGURE 5.15: Speedup obtained by LANE-DETECTION using C++ threads and Intel TBB GRPPI back ends and the basic and extended PIBA algorithms.

In general, throughout this study, we conclude that the extended PIBA algorithm is a more suitable option for balancing Pipelines, as it takes advantage of the inherent contention that this pattern may cause. The benefits of this algorithm can be summarized as follows. Firstly, the extended PIBA includes by design the basic variant, and thus, when the optimal Pipeline arrangement requires less or equal replicas than the number of cores, both approaches arrive at the same solution. Secondly, in some cases, a concurrency degree higher than the total cores might be the preferred option. However, this framework is based on profiling for tuning the parallel source code and, therefore, the input data used for the profiling should be representative enough to correctly tune the application. Otherwise, running the application with real input data might not provide the best performance. Additionally, if the application has an irregular workload, depending on the profiling input data, the algorithm may not find the optimal pipeline arrangement.

5.4 Summary

In this chapter, we have proposed a framework that is able to transform the sequential source code into parallel by leveraging the proposed parallel pattern detection tool and the pattern interface. Additionally, this framework is able to rearrange and tune the pattern constructions in order to better exploit the available resources in the target platform. However, as stated in Chapter 2, some algorithms available in several libraries are already parallelized and optimized. Nonetheless, this situation arises a new challenge to select the most suitable alternative depending on the target architecture and problem size. In the following chapter, we propose a novel mechanism to select the most suitable implementation for a given problem based on previous performance information.

Chapter 6

Automatic implementation selection techniques

So far, we have discussed that programming for parallel and heterogeneous platforms are notoriously difficult since there are many different frameworks and programming interfaces. This situation has led to different implementations of the same algorithm targeted and optimized to different devices, and thus, delivering different performance depending on the platform used. When dealing with these algorithms, the use of highly tuned implementations available in specific libraries is a better option than implementing and parallelizing the code from scratch. In these cases, those implementations usually outperform ad-hoc implementations since they are fully optimized for concrete platforms. This situation reveals a new challenge: to select the most suitable device and routine implementation to solve a given problem. Usually, in order to improve performance, developers need to analyze a priori the target platform and the application, along with its implementation alternatives and available libraries. To achieve this goal, some aspects need to be considered. For instance, some libraries exhibit better behavior than others for a given problem size [78]. Also, devices can have different features (such as the number of cores, processor frequency or memory size), and thus, they may influence, or even restrict, the use of a specific library routine.

An approach to cope with this problem is to manually select the algorithm implementation and map the execution onto the underlying parallel device based on past knowledge. Nevertheless, this procedure becomes complex when dealing with multiple architectures and libraries. A common technique is to define a set of constraints in order to guide a runtime scheduler to select the most suitable implementation. This technique, however, has non-negligible performance overheads, since it is necessary to check such constraints each time a routine is called. An alternative to the aforementioned technique is to shift the decision-making task directly at compile time. Several proposals leveraging this approach and based on analytic models, machine learning and adaptive optimization methods can be found in the literature [18]. However, these works present two major drawbacks: *i)* they are strongly tied to the target platform, and *ii)* they are limited to a concrete set of devices. Given the foregoing, we present a way to express semantic constraints pursuing a more generic approach and targeting heterogeneous platforms.

Specifically, in this chapter, we present an adaptive and offline implementation selector that leverages a profile-guided approach and is able to decide the tuple device-implementation that delivers the best performance. To do so, we have developed two novel features part of the standard C++ language, concepts and attributes, as for the end-user interfaces of the implementation selector. Additionally, we have also implemented two different selection techniques: a full compile-time selection and a hybrid static-dynamic selection.

6.1 Compiletime Implementation Selection

This section describes the proposed hybrid static-dynamic implementation selection framework. This framework provides an interface based on C++ attributes with the objective of generating source-code decision trees with the most suitable combinations of processor-implementation. Particularly, this framework has been designed as a feedback system, i.e., data collected during an execution influences the next ones. The main goal of the framework is to improve the selection task in each compilation. Depending on the constraints used in the attribute-annotated user codes, the selector can work using a full-static or a hybrid static-dynamic approach. On the one hand, the full-static mode replaces the annotated interfaces by a single implementation at compile time. This mode is useful when the user already knows the problem size. On the other hand, the hybrid static-dynamic mode of the selector generates `if-else` decision trees at compile time, which are processed by the user application at run-time. In the following sections, we introduce the mathematical foundations of the selection algorithm and describe in more detail the modules of the selection framework.

6.1.1 Formal definition of the selection algorithm

In order to proceed further with our rationale for selecting the implementation delivering the best performance, we formally describe the theoretical basis of the selection algorithm used in our framework. Consider $\mathcal{V}$ a set of available versions of a same routine, s the problem size, and $t_i(s)$ the execution time of the version i using the problem size s. With this, the equation

$$Best_{point}(\mathcal{V}, s) = A \quad \Leftrightarrow \quad A \in \mathcal{V} \wedge \forall i \in \mathcal{V} : t_A(s) \leq t_i(s) \tag{6.1}$$

determines that the implementation A has an execution time lower than any other version in $\mathcal{V}$ for a certain problem size s. Similarly, the equation

$$Best_{range}(\mathcal{V}, [s_b, s_e]) = A \quad \Leftrightarrow \quad \forall i \in \mathcal{V} : \int_{s_b}^{s_e} t_A(x)\,\mathrm{d}x \leq \int_{s_b}^{s_e} t_i(x)\,\mathrm{d}x \tag{6.2}$$

states that the version A has the smallest area under its function $t_A(x)$ in the range of sizes $[s_b, s_e]$. Therefore, A is the implementation providing the lowest run time when it is executed multiple times on different problem sizes within the range.

With the equations 6.1 and 6.2, it is possible to obtain the best implementation in $\mathcal{V}$ for a fixed size and a range of sizes only if the execution times for any problem size are known in advance. In other words, if the functions $t_i(x)$ are defined accurately in all its domain. However, in a real scenario, this is not the case. To deal with this issue, we approximate the domain of $t_i(x)$ as the union of problem sizes intervals in the set E, i.e.,

$$E = \big\{[s_0, s_1), [s_1, s_2), ..., [s_{n-1}, s_n)\big\} \quad \Leftrightarrow \quad \mathcal{D}omain(t_i(x)) = \bigcup_{I \in E} I.$$

Note that the intervals in E are defined by the problem sizes whose execution time is known in a given point in time, e.g., if the values of $t_i(s_a)$ and $t_i(s_b)$ are known, the interval $[s_a, s_b)$ would be part of E. Furthermore, we approximate the function $t_i(x)$ with the set of functions

$$\{\tau_i^I(x) = mx + c : \forall I \in E\} \tag{6.3}$$

being $\tau_i^I(x)$ a linear function between the endpoints of the interval I in E. With these definitions, we reformulate the function $Best_{point}$ in Eq. 6.1 as

$$Best'_{point}(\mathcal{V}, s) = A \;\Leftrightarrow\; \forall i \in \mathcal{V}, \exists I \in E : s \in I \;\wedge\; \tau_A^I(s) \leq \tau_i^I(s) \tag{6.4}$$

for determining approximately which version provides the best performance with the current knowledge. In this case, A is the version whose function $\tau_A^I(s)$, defined in the interval I where s belongs, has the lowest value than any other version in $\mathcal{V}$. Likewise, we also define an estimation of $Best_{range}$ in Eq. 6.2 as

$$Best'_{range}(\mathcal{V}, [s_b, s_e]) = A \Leftrightarrow \forall i \in \mathcal{V} : area(A, [s_b, s_e]) \leq area(i, [s_b, s_e]) \tag{6.5}$$

where the area under its function $t_A(x)$ is approximated with

$$area(A, [s_b, s_e]) = \int_{s_b}^{I_e^B} \tau_A^{I^B}(x)\,\mathrm{d}x + \sum_{r \in R} \int_{r_b}^{r_e} \tau_A^r(x)\,\mathrm{d}x + \int_{I_b^E}^{s_e} \tau_A^{I^E}(x)\,\mathrm{d}x$$
$$: \exists s_b \in I^B \in E \;\wedge\; \exists s_e \in I^E \in E \;\wedge\; \exists R \in E \Leftrightarrow \forall r \in R, r \in (s_b, s_e).$$

Basically, this formula estimates the area of the version A in the interval $[s_b, s_e]$ by adding the areas under their $\tau_A^I(x)$ defined for each interval I contained in the range. Note that the areas of the intervals containing the endpoints of $[s_b, s_e]$ are only partially included.

6.1.2 Description of the framework

In this section we describe in detail the proposed selection framework (see Fig. 6.1) that using problem size information is able to select the most adequate routine implementation. Basically, this framework performs the following steps in order to determine which is the implementation providing the slowest execution time.

First, the hardware information module extracts the platform information and stores it into a file (`HPP.json`). In the next step, the users should provide the annotated header files with the different implementations available for each interface. In the same way, the user annotates application function calls, candidates to be analyzed and replaced by our framework. With this information, the selector analyzes the function calls annotated in the user source code and replaces them by the most suitable implementation in the header files. Furthermore, the framework instruments these function calls to measure their execution time. Finally, after the application run, the framework stores the performance profiles of the instrumented functions into a file (`PERF.json`). Basically, this file contains, for each implementation and problem size, the average run time and the total number of samples collected. This allows recalculation the averages run times in an incremental way, i.e., each time a new sample arrives. This file is later used to make selections using a profile-guided optimization approach.

Next, we describe the attributes of the framework used for annotating function calls and declarations.

Header attributes. As mentioned, our selection framework requires the user intervention to declare a set of constraints for each interface and routine implementation. These restrictions specify which implementations are associated with each different function interface and the target device. These requirements, in form of attributes under the `rph` *namespace*, are the following:

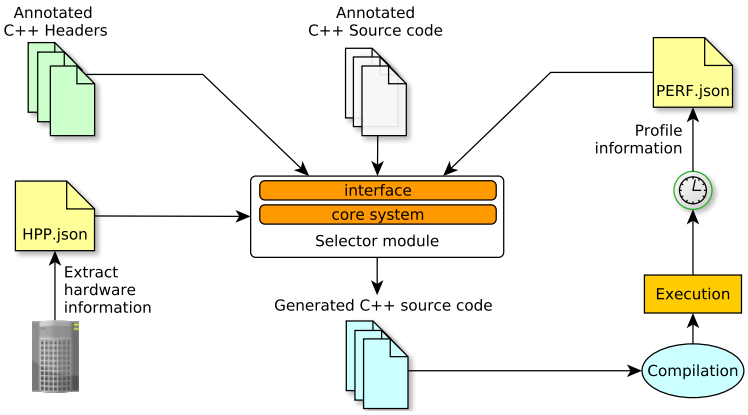


FIGURE 6.1: The hybrid static-dynamic implementation workflow.

- `rph::implements`: This attribute specifies that the code under the attribute is an alternate implementation of a given interface. Basically, it receives, as the sole parameter, the function name to let the selector know which implementations are available for that interface. Consider, for instance, the general matrix-matrix multiplication (`dgemm`). In this case, the attribute contains the generic function name of the `dgemm`, although the routines actually implementing this algorithm might have different names. Note, as well, that all the implementation alternatives for an interface should specify the same name in `rph::implements`.

- `rph::device`: This attribute bounds a given implementation to a specific target device. Supported parameter values for this attribute are: `CPU`, `GPU`, `PHI` (for the Intel Xeon Phi co-processor), etc. [74].

Function call attributes. In this stage, the user is responsible for annotating function calls that are candidates to be analyzed by the selector in the user application code. This set of C++ attributes is the following:

- `rph::interface`: This attribute indicates that the annotated function call is an interface, and should be replaced by the framework with an actual implementation during the selection process.

- `rph::target`: It determines the preferred target device to execute the annotated function call(e.g., `rph::target(CPU)`). Valid arguments for this attribute are those that are accepted by the `rph::device` attribute.

In order to specify static problem constraints and to enable the hybrid static-dynamic mode in an annotated function call, the user should employ one of these options:

- `rph::size`: This attribute is used when the user already knows the problem size during the function call. This attribute receives the problem size as a single parameter. This attribute makes the selector work using the full-static approach, as stated before.

- `rph::minsize` and `rph::maxsize`: Alternatively, when the user is not able to specify the problem size, `rph::minsize` and `rph::maxsize` can be used to establish both lower and upper bounds of the problem size. Similar to the `rph::size` attribute, these attributes enable the full-static selection approach.

- `rph::dynamic`: If this attribute is set, the selector replaces the function interface by a decision tree in the source code, implemented using `if-else` statements. Therefore, the application will be able to select, at run-time, the most suitable implementation. The conditions in the `if-else` statements are evaluated using the problem size, obtained at run-time, and the intersection values where an implementation delivers better performance than other. Additionally, this attribute receives an expression producing an integer which obtains the problem size in the application context.

It is important to remark that function calls annotated by the user should match those provided in the corresponding header files.

6.1.3 The profile-guided selection algorithm

This section describes the internal workings of the selector, as the core module of the framework. Basically, this module analyzes function calls annotated with the `rph::interface` attribute. Then, it starts a selection process that replaces interfaces by actual implementations (using the full-static mode) or by decision trees (using the hybrid mode) complying with the restrictions stated by the user according to available implementations and processors. For that, the selector leverages a profile-guided optimization approach that takes advantage of the information gathered in the performance profile information (introduced in Section 6.1.2). Note that the entire selector has been implemented using the Clang 3.8.0 compiler API that is used to analyze C++ attributes [54]. Specifically, the selector module performs the following steps:

1. The selector analyzes the annotated header files and the implementations provided by each function interface used in the application code.

2. It checks for annotated functions in the application user code using the attribute `rph::interface`. Simultaneously, it examines whether the user has marked the interfaces with the attribute `rph::target` or not, i.e., to use a preferred target processor. In this case, the selector considers only the implementations for such interface that can be executed on the processor specified by the user. Other implementations are automatically discarded. If there are no implementations that can run on the preferred processor, regarding the knowledge about the platform, as specified in the platform description specification, the implementation delivering the best performance on any available processor is taken instead.

3. If the function interface has been annotated either with the `rph::size` or `rph::minsize` and `rph::maxsize`, the selector performs a static decision to determine which implementation, among the candidate ones, offers the best performance. Otherwise, if the `rph::dynamic` attribute has been used, the module will calculate the intersection values where an implementation delivers better performance than other. Next, it will generate an `if-else` decision tree, which is processed by the user application at run-time in order to decide which implementation should be executed. All decisions are made according to the performance profile information.

The full-static selection mode. The full-static selection mode implemented is entirely based on the problem size and boundaries specified by the user. Depending on the attributes used, the algorithm proceeds as follows.

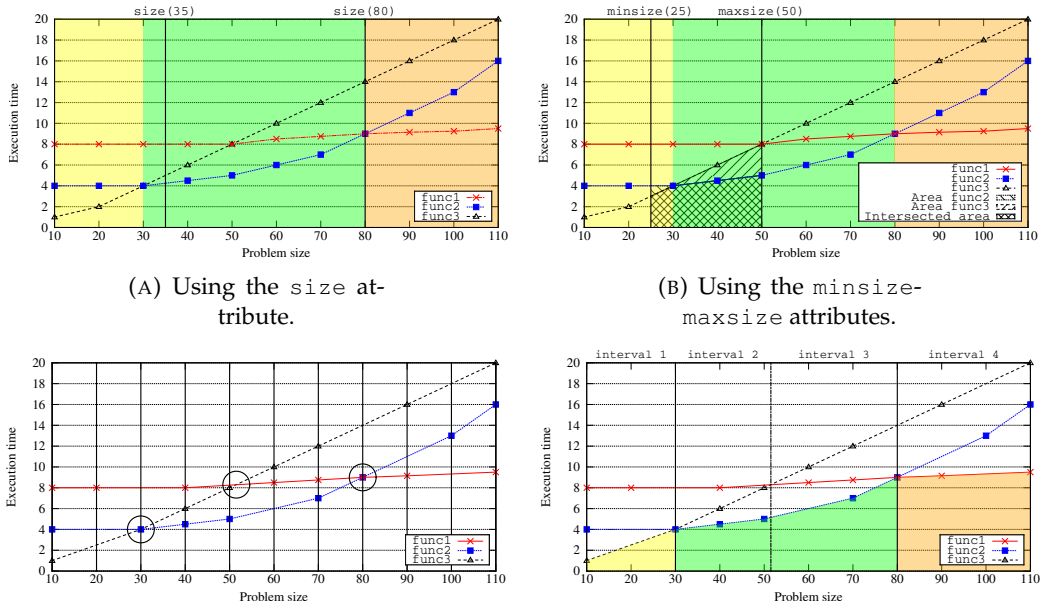
- If the attribute `size` is set, the selector takes the implementation offering the minimum execution time according to the function $Best'_{point}$ in Eq. 6.4. For this purpose, the selector performs a linear interpolation for the requested problem size for all implementations available in such a function interface, in case it is not present in the performance file. (Note that to smooth extreme performance values stored in `PERF.json`, the selector only considers average execution times entries that have been computed with, at least, three samples.) Otherwise, if multiple implementations deliver the same minimum performance, the selector randomly picks one of them. However, this random policy can be eventually replaced by another that takes into account the lower maximum performance in order to avoid extreme behaviors. Consider the scenario in Fig. 6.2(a) that shows the behavior of a given function interface offering three different implementations. For instance, if the user sets the `size` attribute to 35, the selector will consider `func2`, while if the problem size is fixed to 80, the framework will randomly select `func1` or `func2`.

- Otherwise, if the developer has used both `minsize` and `maxsize` attributes to determine a range of possible problem sizes, the selector computes the area under the performance curve (or integral) for the available implementations using the function $Best'_{range}$ in Eq. 6.5. With it, the framework selects the implementation that has the smallest area in the range. As shown in Fig. 6.2(b), if the user selects a range between 25 and 50 as minimum and maximum problem sizes, the selector module will compute the integrals for the three implementations available. Afterward, it will compare the areas below the curves and take that having the smallest one, i.e., `func2`. Note that if there are no performance values in the boundaries of the range, the values that intersect the boundaries are computed via linear interpolation. As in the previous option using the `size` attribute, if there are two or more implementations whose area value is equal, the selector will pick one randomly. Also, only average execution times entries with, at least, three samples are considered.

The hybrid selection mode. This mode generates, at compile time, a decision tree that is based on the performance data collected from previous executions. This tree is generated when the `dynamic` attribute is set with the following algorithm.

First, the selector calculates the intersection points among all the functions estimating the execution time, as defined in Eq. 6.3, for the available implementations. Following the aforementioned scenario, Fig. 6.2(c) shows the problem size intervals and the intersection points highlighted with circles. Next, for every two consecutive intersections, the best version for the interval is obtained using the function $Best'_{range}$ in Eq. 6.5. Fig. 6.2(d) shows the different intersection intervals with their minimum areas denoting the fastest implementation in the range. (Note that the intervals 2 and 3 are merged together in the end, as the best implementation for both is the same.) With this, the selector is able to generate a tree whose decision nodes correspond to the boundaries of all the obtained intervals and the leafs represent the implementations. Therefore, the function responsible for computing the problem size in the application context allows to walk the tree until reaching a leaf node.

It is important to highlight that, at present, the current version of the selector only considers the problem size to select the fastest implementation. In the future, we plan to extend the set of user C++ attributes to allow users to specify other kinds of constraints, such as memory usage or energy consumption.



(A) Using the `size` attribute.

(B) Using the `minsize-maxsize` attributes.

(C) sing the `dynamic` attribute (step 1).

(D) Using the `dynamic` attribute (step 2).

FIGURE 6.2: Example of the behavior of an hypothetical function interface `func` offering three different implementations (`func1`, `func2` and `func3`). Note the cutoffs for problem sizes 30 and 80 between the implementations 1–2 and 2–3, respectively.

6.1.4 Working example

In this section, we illustrate the workings of the framework. Listing 6.1 shows a header file with the attributes set by the users for the function interface `func`.

LISTING 6.1: Annotated header file.

```
namespace ns1 { //Implementation 1
  [[rph::implements("func"), rph::device(CPU)]]
  void func1(...);
}
namespace ns2 { //Implementation 2
  [[rph::implements("func"), rph::device(CPU)]]
  void func2(...);
}
//Implementation 3
[[rph::implements("func"), rph::device(GPU)]]
void func3(...);
```

In the same way, Listing 6.2 contains an example of user code with different attribute-annotated functions matching the interface `func` defined in the previous header file. As shown in the header file, three different implementations are interfacing function `func`. In their attributes, the user has defined some restrictions. For example, implementation 3 requires a GPU. Looking at the application code, the user has invoked four times this function using different attribute parameters. Finally, the selector processes and replaces these function calls with the implementations selected for each case. Listing 6.3 shows the code generated by the framework.

LISTING 6.2: Annotated user code.

```
#include <header.hpp>
int main(){
  //function call 1
  [[rph::interface,rph::minsize(25),rph::maxsize(50)]]
  func(...);
  //function call 2
  [[rph::interface,rph::size(20)]]
  func(...);
  //function call 3
  [[rph::interface,rph::size(50),rph::target(GPU)]]
  func(...);
  //function call 4
  [[rph::interface,rph::dynamic(density_func(...))]]
  func(...);
  return 0;
}
```

As observed, the first call has been replaced by `ns2::func2` as it is the most suitable implementation for the attribute parameters given by the user. To make this decision, the selector computes the area for the range given and selects the implementation having the smallest value. Next, the second call is replaced by `ns1::func1` because it has the smallest minimum size for problem size 20. The third call has been substituted by `func3`, as the user specified, via `target`, the GPU as the preferred device, and any other implementation targeted to CPUs has been discarded. Finally, the fourth call has been replaced by the corresponding decision tree, as it was annotated with the `dynamic` attribute. Thus, depending on the expression `density_func(...)` provided by the user through the `dynamic` attribute and evaluated at runtime, different implementations are executed.

LISTING 6.3: Processed user code.

```
#include <header.h>
int main(){
  //function call 1
  ns2::func2(...);
  //function call 2
  ns1::func1(...);
  //function call 3
  func3(...);
  //function call 4
  auto rph_dens = density_func(...);
  if ( rph_dens < 30 ) func3(...);
  else if ( rph_dens >= 30 && rph_dens < 80 )
      ns2::func2(...);
  else ns1::func1(...);
  return 0;
}
```

6.2 Evaluation

In this section, we evaluate the presented implementation selector framework along using two use cases: the general matrix-matrix multiplication (GEMM) and a real medical application that computes a spherical deconvolution algorithm of diffusion MRI data (HARDI) of human brains [31, 30]. First, we perform an evaluation of the accuracy and convergence of the selector algorithm of the framework using the GEMM case. Next, we demonstrate how a real use case (HARDI) can benefit from our framework. Finally, we compare our framework with the runtime-based *versioning* scheduler from the OmpSs programming model.

6.2.1 Reference platform

We evaluate the GEMM and HARDI use cases using the reference machine, namely ARCH1, and another machine, namely ARCH2, respectively. The ARCH1 machine is comprised of of 2× Intel Xeon Ivy Bridge E5-2695 v2 with a total of 24 cores running at 2.40 GHz, 30 MB of L3 cache and 128 GB of DDR3 RAM. Additionally, this platform also incorporates with two AMD Radeon GPUs, R9 290X (AMD1) and R9 285 series (AMD2), and an Intel Xeon Phi 3120 co-processor (MIC). The ARCH2 machine is comprised of two multi-core Intel Xeon E5-2630 v3 processor with a total of 8 physical cores running at 2.40 GHz, equipped with 128 GB of RAM. This machine also has with an NVidia Tesla K40 and a GTX 680 under CUDA version 7.5.

In both platforms, the OS used is Linux Ubuntu 14.04 x64 and the compiler employed is GCC 5.1 with the flag `-O3` set.

6.2.2 Analysis with the GEMM use case

In this section, we analyze the `dgemm` kernel performance and the selector accuracy using the implementations from the clBLAS [15] and GSL [28] libraries on the ARCH1 machine. Fig. 6.3 plots the execution times using square matrix sizes ranging from 4 to 4,096 and double-precision numbers. As observed, depending on the problem size, a kernel implementation delivers better performance than others. For instance, using the size range 4–504, the GSL version is the preferred option, while for the ranges 504–1,990 and 1,990–4,096, the clBLAS implementation running respectively on XEON and AMD1 are the optimal alternatives. Thus, it becomes essential to select the best implementation depending on the matrix input size, in our case using the automatic approach presented in this thesis.

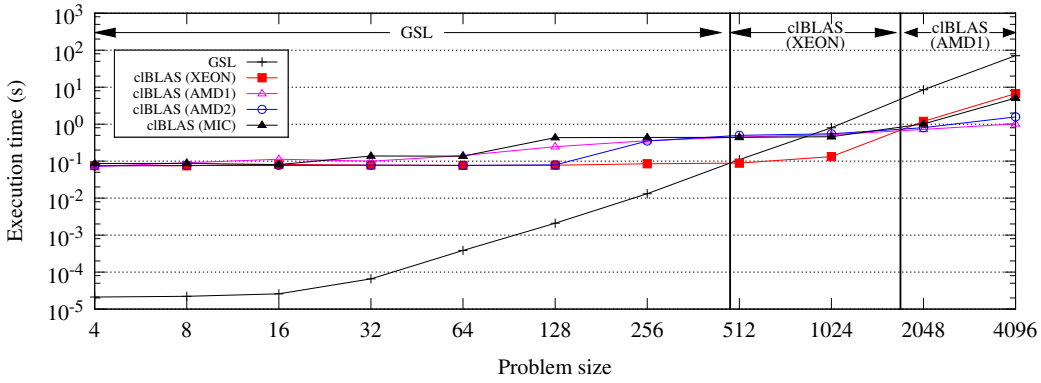
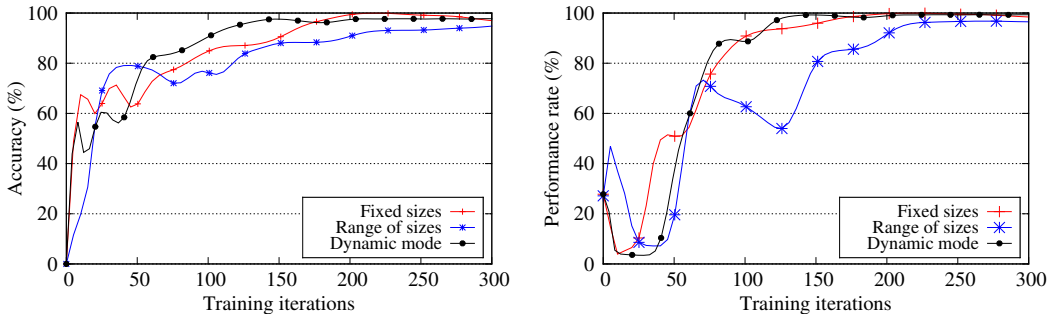


FIGURE 6.3: Execution time of the `dgemm` kernel for different square matrix sizes and implementations.

Additionally, we evaluate the selector accuracy and the `dgemm` kernel performance rates by increasing the number of training iterations, using the attribute `size`, `minsize-maxsize`, and `dynamic`, indistinguishably. Note that the performance rates were obtained dividing the execution time of the fastest between the selected implementation. For each of these iterations, we train the system running an instance of the `dgemm` kernel using matrices of random size. Afterward, we evaluate the knowledge gained by the selector performing 100 runs of the same kernel also with random sizes. In Fig. 6.4(a), we show the accuracy progress when using the static, i.e. using fixed and range of sizes attributes, and the dynamic modes. As can be seen, these percentages increase in a smooth curve until reaching, after 300 training iterations, roughly 97 % of the total accuracy. This behavior is mainly because the selector has already gained enough knowledge about the performance delivered

by the different implementations. Looking at the performance in Fig 6.4(b), using both static and dynamic-related attributes, the performance rates after 300 iterations reach almost 100 %. Therefore, all selections made from that point on will provide a good performance. An interesting remark is that the drops on the accuracy appearing during the first training iterations are not proportionally reflected in the performance progress. This is because a wrong selection has different consequences on the performance, and thus, depending on the implementation chosen and problem size, it might cause a lower or a higher decrease in the performance rate. In general, we find out that both static and dynamic modes provide a similar performance gain. Nevertheless, there are differences between these modes: using the static approach the applications have to be recompiled whenever the problem size changes, while in the dynamic mode the applications are able to adapt themselves without recompiling them.



(A) Accuracy through training iterations.

(B) Performance through training iterations.

FIGURE 6.4: Progress of the accuracy of the selector and `dgemm` performance through training iterations.

6.2.3 Analysis with the HARDI use case

We leverage the HARDI use case, responsible for executing the Robust and Unbiased Model-Based Spherical Deconvolution (RUMBA-SD) method, to demonstrate the benefits of our framework. This algorithm is, up to date, one of the most advanced algorithms for detecting crossing fibers in white matter [14]. For our experiments, we use the parallel RUMBA-SD method part of HARDI. We execute this parallel algorithm using different linear algebra implementations on the multi-core CPU (via Intel MKL) and the GPUs (via ArrayFire) of ARCH2 with single-precision floating-point numbers.

Regarding the HARDI input data, we use a real diffusion MRI dataset acquired from the healthy subject. Specifically, the whole-brain HARDI data was acquired in a 3T Philips Achieva scanner with an 8-channel head coil along 100 different gradient directions on the sphere in q-space with constant $b = 2000\,\mathrm{s/mm^2}$. Additionally, $1b = 0$ volume was acquired with in-plane resolution of $2.0 \times 2.0\,\mathrm{mm^2}$ and slice thickness of 2 mm. The acquisition was carried out without undersampling in the k-space (i.e., $R = 1$). The final dimension of this dataset is $128 \times 128 \times 60 \times 100$ voxels, being 60 the number of slices of 128×128 voxels, each of them comprising 100 directions.

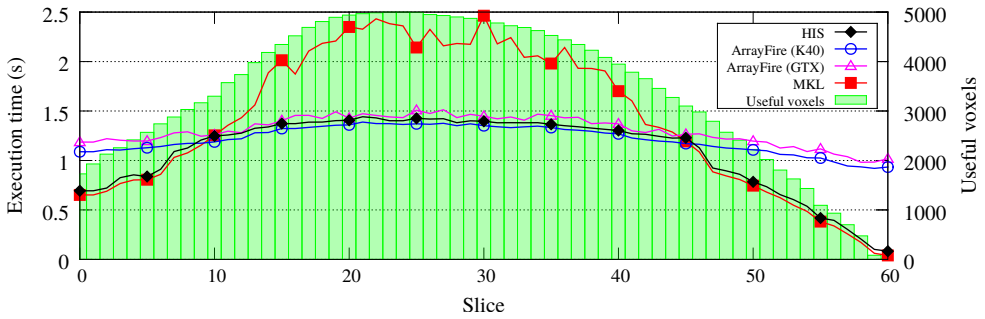


FIGURE 6.5: Progress of the accuracy using a range of sizes.

Fig. 6.5 depicts the execution time and the number of useful voxels for each slice of the dataset, using lines and bars, respectively. As observed, using the MKL library, the execution times are fairly correlated with the number of useful voxels. On the contrary, the ArrayFire versions, using the GPUs (K40 and GTX 780) obtain a flatter curve. In this case, the use of ArrayFire for GPUs is only compensated for slices containing a high number of useful voxels, as the data transfers pay off the computational load. Focusing on our hybrid implementation selector (HIS), the selector takes the MKL and the ArrayFire implementations for slices comprising low and high number of useful voxels, respectively. Specifically, we configured the selector using the dynamic mode, so that, the decisions are made at run-time. We observe some negligible overheads (2 %) using our approach mainly caused by the density function run time. Note that this density function, responsible for calculating the number of useful voxels in each slice, is used each time by the decision tree in order to select the most suitable implementation. Finally, to evaluate the benefits of our approach, we computed the total execution time of HARDI using the aforementioned implementations (including HIS) in order to obtain speedup figures. We find out that using our approach with respect to MKL reduces the execution time by 24 %, while compared with ArrayFire, HIS decreases the execution time about 10 %. Therefore, our approach, in this case, helps improving the overall performance of applications.

6.2.4 Comparison with alternative approaches

In this section, we validate the performance benefits of our hybrid static-dynamic implementation selector (HIS) and compare it with an existing runtime scheduler. Concretely, we compare our approach with the *versioning* runtime scheduler counterpart from the OmpSs programming model [20], as it offers a similar implementation selector to our static solution.

To compare our solution with OmpSs, we developed a microbenchmark composed of two consecutive 30-iteration loops computing the matrix-matrix product (`dgemm` kernel) using square matrices of size 256×256 and random sizes, respectively, in each iteration of the loops. For HIS, we annotate the `dgemm` kernel calls using the attribute `size` for the first loop and with `dynamic` for the second one. In contrast, for OmpSs we define different tasks for the available implementations that are annotated with the `implements` and `target` directives. Take into account that the multiplication is performed using the same `dgemm` implementations as in the previous experiments.

Fig. 6.6 depicts the execution progress of this microbenchmark. As can be seen, HIS starts from the first loop iteration selecting the implementations that perform best for the different matrix sizes. It is important to note that HIS was previously trained performing 100 executions of the `dgemm` kernel with random matrix sizes

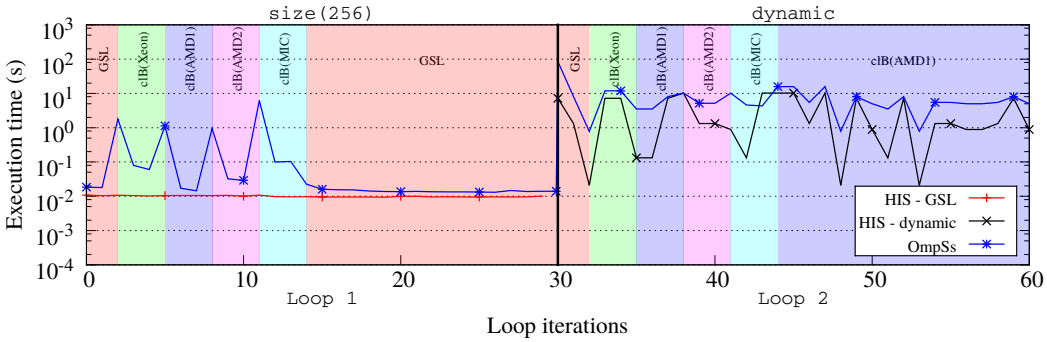


FIGURE 6.6: Execution progress of two 30-iteration loops computing the `dgemm` kernel using HIS and OmpSs.

and the measured profiling overhead was not higher than 1 %. On the contrary, OmpSs cannot be trained offline, so it makes a few trial runs of the different implementations until it finds, at runtime, the fastest one. In these cases, the training phase of HIS pays off the OmpSs trial runs and the runtime scheduler overhead. Specifically, the OmpSs measured overhead ranges between 2 % and 40 % for the large and small matrix sizes, respectively. However, when the matrix size varies among iterations, OmpSs is not able to self-adapt and continues selecting an implementation that is not optimal. In contrast, HIS relies on the problem size to select in each iteration the most suitable implementation, and thus, improving the overall performance. On the other hand, we have calculated the number of application executions that our approach requires (assuming that there is no previous performance data in the `PERF.json` file) in order to improve the execution time of OmpSs. We found out that, using our approach, 40 executions of the user application are necessary to compensate the training phase overheads and overtake the performance of the OmpSs `versioning` scheduler. In general, HIS offers a hybrid implementation selector that is adaptive and learns among executions, while OmpSs is a runtime alternative that has non-negligible overheads but does not require a training phase.

6.3 Summary

In this chapter, we propose a mechanism to select the most suitable implementation for a given problem based on previous performance information. This mechanism can work by performing a unique selection independently on the actual current problem size or to decide the most suitable implementation for a specific problem size at runtime. This way, working this mechanism in conjunction with the refactoring framework, the resulting source code can be improved and attain better performance.

Chapter 7

Conclusions and future work

In this Thesis, we have proposed and developed a series of independent tools that can be used together in order to automatically transform sequential codes into optimized parallel codes. Following the original goal thesis the state of the goal is the following:

- **To automatically detect parallel patterns in sequential codes**. In Chapter 3 we have proposed and implemented a technique to analyze the Abstract Syntax Tree of a given source code in order to detect parallel patterns that can be eventually incorporated.
- **To define an unified parallel pattern interface**. In Chapter 4 we have defined a common interface for parallel patterns in order to act as a switch layer that allows to hide specific implementation details of the different programming frameworks and to easily select the adequate backend for a given application.
- **To define a set of transformations** to optimize parallel source codes. In Chapter 5 and 6 we have proposed different refactoring techniques to provide more optimized parallel applications. On the one hand, we have proposed a mechanism that is able to rearrange pipeline constructions composed of farm patterns in order to determine an optimal concurrency degree and arrangement. On the other hand, we have proposed a technique to select among different versions of the same algorithm coming from different fully-optimized libraries.

7.1 Contributions

During the development of this thesis, the following contributions have been reached:

- **A parallel pattern detection tool** that, leveraging Clang libtooling, is able to analyze sequential source code, detect parallel pattern candidates and generate annotated codes. These annotations may be used by refactoring tools in order to automatically generate parallel source codes.
- **A generic parallel pattern interface** that provide a unified interface for parallel patterns by acting as a switch between different parallel frameworks acting as back-ends. This way, a single application can be easily migrated from one programming framework to another by modifying a single function argument. Furthermore, its composability allows constructing more complex patterns in terms of basic ones.
- **An automatic refactoring module.** This module takes an annotated source code and generates parallel code using GRPPI.

- **A pipeline balancing algorithm** able to generate optimized configurations for pipeline patterns only using the original construction schema and the execution time of the pipeline stages.
- **Mechanisms to select among routine implementations.** These mechanisms allows to define a set of implementation alternatives and select, at compile time or at runtime, the most suitable version for a given algorithm depending on the problem size.

7.2 Disemination

The contributions of this thesis can be found in the following publications:

- Journals
  - Assessing and discovering parallelism in C++ code for heterogeneous platforms, David del Rio Astorga, Rafael Sotomayor, Luis Miguel Sanchez, Javier Garcia Blas, Alejandro Calderon, Javier Fernandez. The Journal of Supercomputing. 2016.
  - Finding parallel patterns through static analysis in C++ applications. David del Rio Astorga, Manuel F. Dolz, Luis Miguel Sanchez, J. Daniel Garcia, Marco Danelutto, Massimo Torquati. International Journal of High Performance Computing Applications. 2017.
  - An adaptive offline implementation selector for heterogeneous parallel platforms. David del Rio Astorga, Manuel F. Dolz, Luis Miguel Sanchez, Javier Fernandez, J. Daniel Garcia. International Journal of High Performance Computing Applications. 2017.
  - Enabling Semantics to Improve Detection of Data Races and Misuses of Lock-Free Data Structures, Manuel F. Dolz, David del Rio Astorga, Javier Fernandez, Massimo Torquati, J. Daniel Garcıa, Felix Garcıa-Carballeira, Marco Danelutto. Concurrency and Computation Practice and Experience. 2017.
  - A generic parallel pattern interface for stream and data processing. David del Rio Astorga, Manuel F. Dolz, Javier Fernandez, J. Daniel Garcıa. Concurrency and Computation: Practice and Experience. 2017.
  - Hybrid static-dynamic selection of implementation alternatives in heterorgeneous environments, David del Rio Astorga, Manuel F. Dolz, Javier Fernandez, Javier Garcia Blas. The Journal of Supercomputing. 2017.
  - Paving the way towards high-level parallel pattern interfaces for data stream processing. David del Rio Astorga, Manuel F. Dolz, Javier Fernandez, J. Daniel Garcıa. Future Generation Computer Systems. 2018.
  - Towards Automatic Parallelization of Stream Processing Applications. Manuel F. Dolz, David del Rio Astorga, Javier Fernandez, J. Daniel Garcıa, Jesus Carretero. IEEE Access. 2018.
- Conferences
  - ACTIS: Automatic Compile-Time Implementation Selector for Heterogeneous Platforms. David del Rio Astorga, Manuel F. Dolz, Luis Miguel Sanchez, J. Daniel Garcıa. Proceedings of the High-Level Programming for Heterogeneous and Hierarchical Parallel Systems Workshop. 2016.

  - Discovering Pipeline Parallel Patterns in Sequential Legacy C++ Codes. David del Rio Astorga, Manuel F. Dolz, Luis Miguel Sanchez, J. Daniel Garcıa. Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores. 2016.
  - Embedding Semantics of the Single-Producer/Single-Consumer Lock-Free Queue into a Race Detection Tool. Manuel F. Dolz, David del Rio Astorga, Javier Fernandez, J. Daniel Garcıa, Felix Garcıa-Carballeira, Marco Danelutto, Massimo Torquati. Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores. 2016.
  - CID: A Compile-time Implementation Decider for Heterogeneous Platforms based on C++ Attributes. Luis Miguel Sanchez, David del Rio Astorga, Manuel F. Dolz and Javier Fernandez. The 2nd International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms. 2016.
  - A C++ Generic Parallel Pattern Interface for Stream Processing. David del Rio Astorga , Manuel F. Dolz, Luis Miguel Sanchez, Javier Garcıa Blas, and J. Daniel Garcıa. 16th International Conference on Algorithms and Architectures for Parallel Processing. 2016.
  - Probabilistic-based selection of alternate implementations for heterogeneous platforms. Javier Fernandez, Andres Sanchez Cuadrado, David del Rio Astorga, Manuel F. Dolz, J. Daniel Garcia. 2nd International Workshop on Ultrascale Computing for Early Researchers. 2017.
  - Supporting Advanced Patterns in GrPPI: a Generic Parallel Pattern Interface. David del Rio Astorga, Manuel F. Dolz, Javier Fernandez, J. Daniel Garcia. International Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing. 2017.
  - Parallelizing and Optimizing LHCb-Kalman for Intel Xeon Phi KNL Processors. Placido Fernandez, David del Rio Astorga, Manuel F. Dolz, Javier Fernandez, Omar Awile, J. Daniel Garcia. Euromicro International Conference on Parallel, Distributed and Network-based Processing. 2018.

- Registered products

  - AKI: Automatic Kernel Identification. Registration of the intellectual property. Code 09-RTPI-02819.5/2016.

- Technical reports

  - REPARA D2.5: Semantic specification for libraries.
  - REPARA D3.4: Automatic kernel identification and assessment.
  - RePhrase D2.3: Report on shaping and pattern discovery for initial patterns.
  - RePhrase D2.4: Software for implementations of initial patterns.
  - RePhrase D3.2: Combined report describing testing, verification, catastrophic failures detection and properties violation detection for the initial set of patterns.

7.3 Future work

This section presents some extensions of the work presented in this thesis based on the contributions listed in Section 7.1.

First, the parallel pattern detection tool can be extended with new detection modules to increase the set of parallel patterns that can be discovered at compile-time, such as the MapReduce and the Split-Join. Furthermore, since the static analysis is limited in detecting some potential loop-carried dependencies due to, for example, double indexed arrays, the detection can be improved by leveraging dynamic analysis techniques (e.g. symbolic execution).

On the other hand, the generic parallel pattern interface introduced in this thesis can be extended by means of supporting new parallel patterns or stream operators (e.g. Keyed farm, Feedback operator) and by implementing new execution policies for distributed systems implemented with Message Passing communications (e.g. MPI) or internet protocols (e.g. Message Queue Telemetry Transport).

Regarding the refactoring module, this tools can be extended by supporting sequential-to-parallel source code transformations for other parallel patterns such as the Map and Reduce using the GRPPI interface or introducing new modules for refactoring the code with a different framework.

Focusing on the pipeline balancing, the presented algorithm can be extended for distributed systems in which different stages can be offloaded to different computing nodes. To do so, the algorithm should be extended to inform the algorithm about the topology of the network, latencies, etc., in order to rearrange the pipeline and distribute the stages adequately in the available resources.

Finally, the presented mechanism to select among routine implementations can be extended to introduce different restrictions in order to select the most suitable one. In this sense, these restrictions could be energy consumption or both energy and execution time using a muti-objective approach.

Bibliography

[1] M. G. Al-Obeidallah, M. Petridis, and S. Kapetanakis. "A Survey on Design Pattern Detection Approaches". In: *International Journal of Software Engineering* 7.3 (Dec. 2016), pp. 73–90. ISSN: 2180-1320. URL: http://www.cscjournals.org/library/manuscriptinfo.php?mc=IJSE-163.

[2] Marco Aldinucci and Marco Danelutto. "Stream Parallel Skeleton Optimization". In: *in proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems, MIT*. IASTED/ACTA press, 1999, pp. 955–962.

[3] Marco Aldinucci et al. "FastFlow: high-level and efficient streaming on multi-core". In: *in Programming Multi-core and Many-core Computing Systems, ser. Parallel and Distributed Computing, S. Pllana*. 2012, p. 13.

[4] Marco Aldinucci et al. "Pool Evolution: A Parallel Pattern for Evolutionary and Symbolic Computing". In: *International Journal of Parallel Programming* 44.3 (2016), pp. 531–551. ISSN: 1573-7640. DOI: 10.1007/s10766-015-0358-5.

[5] Saman Amarasinghe et al. *ASCR programming challenges for exascale computing*. Tech. rep. U.S. DOE Office of Science (SC), 2011.

[6] Henrique C. M. Andrade, Bugra Gedik, and Deepak S. Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. 1st. New York, NY, USA: Cambridge University Press, 2014. ISBN: 1107015545, 9781107015548.

[7] D. H. Bailey et al. *The NAS parallel benchmarks*. Tech. rep. The International Journal of Supercomputer Applications, 1991.

[8] Robert D. Blumofe et al. "Cilk: An Efficient Multithreaded Runtime System". In: *SIGPLAN Not.* 30.8 (Aug. 1995), pp. 207–216. ISSN: 0362-1340. DOI: 10.1145/209937.209958.

[9] István Bozó et al. "Discovering Parallel Pattern Candidates in Erlang". In: *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*. Erlang '14. Gothenburg, Sweden: ACM, 2014, pp. 13–23. ISBN: 978-1-4503-3038-1.

[10] Christopher Brown, Huiqing Li, and Simon Thompson. "An Expression Processor: A Case Study in Refactoring Haskell Programs". In: *Trends in Functional Programming: 11th International Symposium, TFP 2010, Norman, OK, USA, May 17-19, 2010. Revised Selected Papers*. Ed. by Rex Page, Zoltán Horváth, and Viktória Zsók. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 31–49. ISBN: 978-3-642-22941-1. DOI: 10.1007/978-3-642-22941-1_3. URL: https://doi.org/10.1007/978-3-642-22941-1_3.

[11] Christopher Brown et al. "Paraphrasing: Generating Parallel Programs Using Refactoring". In: *Formal Methods for Components and Objects: 10th International Symposium, FMCO 2011, Turin, Italy, October 3-5, 2011, Revised Selected Papers*. Ed. by Bernhard Beckert et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 237–256. ISBN: 978-3-642-35887-6. DOI: 10.1007/978-3-642-35887-6_13.

[12] Christopher Brown et al. “Paraphrasing: Generating Parallel Programs Using Refactoring”. English. In: *Formal Methods for Components and Objects*. Ed. by Bernhard Beckert et al. Vol. 7542. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 237–256. ISBN: 978-3-642-35886-9.

[13] *C++ compiler support*. 2018.

[14] Erick J Canales-Rodríguez et al. “Spherical deconvolution of multichannel diffusion MRI data with non-Gaussian noise models and spatial regularization”. In: *PloS one* 10.10 (2015), e0138910.

[15] clMathLibraries. *clBLAS*. https://github.com/clMathLibraries/clBLAS. 2015.

[16] Marco Danelutto and Massimo Torquati. “Structured Parallel Programming with “core” FastFlow”. In: *Central European Functional Programming School*. Ed. by Viktória Zsók, Zoltán Horváth, and Lehel Csató. Vol. 8606. LNCS. Springer, 2015, pp. 29–75. ISBN: 978-3-319-15939-3.

[17] Marco Danelutto et al. “Parallelizing High-Frequency Trading Applications by Using C++11 Attributes”. In: *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA - Volume 03*. TRUSTCOM-BIGDATASE-ISPA ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 140–147. ISBN: 978-1-4673-7952-6.

[18] M. I. Daoud and N. Kharma. “Efficient compile-time task scheduling for heterogeneous distributed computing systems”. In: *12th International Conference on Parallel and Distributed Systems - (ICPADS’06)*. Vol. 1. 2006, 9 pp.–.

[19] R. Duncan. “A survey of parallel computer architectures”. In: *Computer* 23.2 (1990), pp. 5–16. ISSN: 0018-9162. DOI: 10.1109/2.44900.

[20] Alejandro Duran et al. “Ompss: a proposal for programming heterogeneous multi-core architectures”. In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193.

[21] Johan Enmyren and Christoph W. Kessler. “SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems”. In: *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*. HLPP ’10. Baltimore, Maryland, USA: ACM, 2010, pp. 5–14. ISBN: 978-1-4503-0254-8. DOI: 10.1145/1863482.1863487.

[22] Steffen Ernsting and Herbert Kuchen. “Data Parallel Algorithmic Skeletons with Accelerator Support”. In: *International Journal of Parallel Programming* 45.2 (2017), pp. 283–299. DOI: 10.1007/s10766-016-0416-7. URL: https://doi.org/10.1007/s10766-016-0416-7.

[23] *EU Project “Reengineering and Enabling Performance And poweR of Applications”*. http://repara-project.eu/. 2016.

[24] *EU Project “RePhraseRefactoring Parallel Heterogeneous Resource-Aware Applications - a Software Engineering Approach”*. http://rephrase.weebly.com/. 2016.

[25] *Extrae: User guide manual for version 2.4.1*. http://www.bsc.es/computer-sciences/extrae.

[26] Message P Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA, 1994.

[27] Edgar Gabriel et al. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Proceedings, 11th European PVM/MPI Users’ Group Meeting*. Budapest, Hungary, 2004, pp. 97–104.

[28] M. Galassi et al. *GNU Scientific Library Reference Manual*. 2009. ISBN: 0954612078.

[29] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[30] Javier Garcia-Blas. *Parallel High Angular Resolution Diffusion Imaging Toolbox*. https://bitbucket.org/fjblas/phardi. 2016.

[31] Javier Garcia-Blas et al. "Porting Matlab Applications to High-Performance C++ Codes: CPU/GPU-Accelerated Spherical Deconvolution of Diffusion MRI Data". In: *Algorithms and Architectures for Parallel Processing - 16th International Conference, ICA3PP 2016, Granada, Spain, December 14-16, 2016, Proceedings*. 2016, pp. 630–643. DOI: 10.1007/978-3-319-49583-5_49.

[32] Horacio González-Vélez and Mario Leyton. "A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers". In: *Softw. Pract. Exper.* 40.12 (Nov. 2010), pp. 1135–1160. ISSN: 0038-0644.

[33] William Gropp. "MPICH2: A New Start for MPI Implementations". In: *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 2002, pp. 7–. ISBN: 3-540-44296-0. URL: http://dl.acm.org/citation.cfm?id=648139.749473.

[34] Tobias Grosser et al. "Polly - Polyhedral optimization in LLVM". In: *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*. Chamonix, France, Apr. 2011.

[35] Kevin Hammond et al. "The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems". In: *Formal Methods for Components and Objects: Intl. Symposium, FMCO 2011, Torino, Italy, October 3-5, 2011, Revised Invited Lectures*. Ed. by Bernhard Beckert et al. Vol. 7542. LNCS. Springer, 2013, pp. 218–236. ISBN: 978-3-642-35886-9.

[36] Intel. *Improve Vectorization Performance with Intel® AVX-512*. 2016.

[37] Intel. *Intel Advisor Roofline*. 2017.

[38] Intel. *Intel AVX State Transitions: Migrating SSE Code to AVX*. 2012.

[39] Intel. *MKL - Math Kernel Library*. https://software.intel.com/en-us/intel-mkl. 2015.

[40] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2012.

[41] ISO/IEC. *Information technology – Programming languages – C++*. International Standard ISO/IEC 14882:20111. Geneva, Switzerland: ISO/IEC, Aug. 2011.

[42] ISO/IEC. *Information technology – Programming languages – C++*. International Standard ISO/IEC 14882:2017. Geneva, Switzerland: ISO/IEC, Dec. 2017.

[43] V. Janjic et al. "RPL: A Domain-Specific Language for Designing and Implementing Parallel C++ Applications". In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. 2016, pp. 288–295. DOI: 10.1109/PDP.2016.122.

[44] Hartmut Kaiser et al. "HPX: A Task Based Programming Model in a Global Address Space". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS '14. Eugene, OR, USA: ACM, 2014, 6:1–6:11. ISBN: 978-1-4503-3247-7.

[45] Philipp Kegel, Maraike Schellmann, and Sergei Gorlatch. "Comparing programming models for medical imaging on multi-core systems". In: *Concurrency and Computation: Practice and Experience* 23.10 (2011), pp. 1051–1065. ISSN: 1532-0634.

[46] Khronos OpenCL Working Group. *SYCL: C++ Single-source Heterogeneous Programming for OpenCL*. https://www.khronos.org/sycl. [Last access May 2015].

[47] V. Kindratenko and P. Trancoso. "Trends in High-Performance Computing". In: *Computing in Science Engineering* 13.3 (2011), pp. 92–95. ISSN: 1521-9615.

[48] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: San Jose, CA, USA, 2004, pp. 75–88.

[49] Xiaoming Li et al. "FreshBreeze: A Data Flow Approach for Meeting DDDAS Challenges". In: *Procedia Computer Science* 51.Complete (2015), pp. 2573–2582.

[50] Zhen Li et al. "DiscoPoP: A Profiling Tool to Identify Parallelization Opportunities". In: *Tools for High Performance Computing 2014*. Springer International Publishing, Aug. 2015. Chap. 3, pp. 37–54. ISBN: 978-3-319-16011-5.

[51] D. Marques et al. "Performance Analysis with Cache-Aware Roofline Model in Intel Advisor". In: *2017 International Conference on High Performance Computing Simulation (HPCS)*. 2017, pp. 898–907. DOI: 10.1109/HPCS.2017.150.

[52] Kiminori Matsuzaki et al. "A Library of Constructive Skeletons for Sequential Style of Parallel Programming". In: *Proceedings of the 1st International Conference on Scalable Information Systems*. InfoScale '06. Hong Kong: ACM, 2006. ISBN: 1-59593-428-6. DOI: 10.1145/1146847.1146860. URL: http://doi.acm.org/10.1145/1146847.1146860.

[53] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. First. Addison-Wesley Professional, 2004. ISBN: 0321228111.

[54] Jens Maurer and Michael Wong. "Towards support for attributes in C++ (Revision 6)". In: *JTC1/SC22/WG21 - The C++ Standards Committee*. N2761=08-0271. 2008.

[55] T. J. McCabe. "A Complexity Measure". In: *IEEE Trans. Softw. Eng.* 2.4 (July 1976), pp. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[56] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123914439, 9780124159938.

[57] Anne Meade, Jim Buckley, and J. J. Collins. "Challenges of Evolving Sequential to Parallel Code: An Exploratory Review". In: *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*. IWPSE-EVOL '11. Szeged, Hungary: ACM, 2011, pp. 1–5. ISBN: 978-1-4503-0848-9.

[58] Maged M. Michael and Michael L. Scott. "Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms". In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 267–275.

[59] Panagiotis D. Michailidis and Konstantinos G. Margaritis. "Scientific computations on multi-core systems using different programming frameworks". In: *Applied Numerical Mathematics* 104 (2016), pp. 62–80. ISSN: 0168-9274.

[60] Korbinian Molitorisz, Tobias Müller, and Walter F. Tichy. "Patty: A Pattern-based Parallelization Tool for the Multicore Age". In: *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM '15. San Francisco, California: ACM, 2015, pp. 153–163. ISBN: 978-1-4503-3404-4.

[61] Angeles Navarro et al. "Load Balancing Using Work-stealing for Pipeline Parallelism in Emerging Applications". In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS '09. Yorktown Heights, NY, USA: ACM, 2009, pp. 517–518. ISBN: 978-1-60558-498-0. DOI: 10.1145/1542275.1542358. URL: http://doi.acm.org/10.1145/1542275.1542358.

[62] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. "Understanding Source Code Evolution Using Abstract Syntax Tree Matching". In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–5. ISSN: 0163-5948.

[63] John Nickolls et al. "Scalable Parallel Programming with CUDA". In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500. URL: http://doi.acm.org/10.1145/1365490.1365500.

[64] Mark S. Nixon and Alberto S. Aguado, eds. *Feature Extraction & Image Processing for Computer Vision (Third edition)*. Third edition. Oxford: Academic Press, 2012. ISBN: 978-0-12-396549-3.

[65] nVidia. *cuBLAS Library User Guide*. v5.0. nVidia. Oct. 2012.

[66] NVIDIA Corporation. *Thrust*. https://thrust.github.io/.

[67] *OpenMP API for parallel programming, version 4.0*. http://openmp.org/.

[68] *Paraver project*. https://www.paraformance.com/.

[69] *Paraver project*. http://www.bsc.es/computer-sciences/performance-tools/paraver.

[70] Srikanta Patnaik and Yeon-Mo Yang, eds. *Soft Computing Techniques in Vision Science*. Vol. 395. Studies in Computational Intelligence. Springer, 2012. ISBN: 978-3-642-25506-9. DOI: 10.1007/978-3-642-25507-6.

[71] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 0124077269, 9780124077263.

[72] Louis-Noël Pouchet et al. *Hybrid Iterative and Model-Driven Optimization in the Polyhedral Model*. Tech. rep. 6962. INRIA Research Report, 2009.

[73] RePhrase EU Project. *D2.5: Advanced patterns*. https://rephrase-eu.weebly.com/uploads/3/1/0/9/31098995/d2-5.pdf.

[74] R. Sotomayor, L. M. Sanchez, J. Garcia Blas, A. Calderon, J. Fernandez. "AKI: Automatic Kernel Identification and Annotation Tool Based on C++ Attributes". In: *Proceedings of the IEEE TrustCom-BigDataSE-ISPA 2015*. Helsinki, Findland, 2015, pp. 148–156. ISBN: 978-1-4673-7952-6/15.

[75] J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007, pp. I–XXV, 1–303. ISBN: 978-0-596-51480-8.

[76] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. "A profile-based tool for finding pipeline parallelism in sequential programs". In: *Parallel Computing* 36.9 (2010), pp. 531 –551. ISSN: 0167-8191.

[77] L. M. Sanchez et al. “A Comparative Study and Evaluation of Parallel Programming Models for Shared-Memory Parallel Architectures”. English. In: *New Generation Computing* 31.3 (2013), pp. 139–161. ISSN: 0288-3635.

[78] LuisMiguel Sanchez et al. “A Comparative Study and Evaluation of Parallel Programming Models for Shared-Memory Parallel Architectures”. English. In: *New Generation Computing* 31.3 (2013), pp. 139–161. ISSN: 0288-3635.

[79] Vivek Sarkar. “Optimized Unrolling of Nested Loops”. In: *International Journal of Parallel Programming* 29.5 (2001), pp. 545–581. ISSN: 1573-7640. DOI: 10.1023/A:1012246031671. URL: https://doi.org/10.1023/A:1012246031671.

[80] Jie Shen, A.L. Varbanescu, and H. Sips. “Look before You Leap: Using the Right Hardware Resources to Accelerate Applications”. In: *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on*. 2014, pp. 383–391.

[81] C. Shuai et al. “Rodinia: A benchmark suite for heterogeneous computing”. In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. 2009, pp. 44–54.

[82] John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *IEEE Des. Test* 12.3 (May 2010), pp. 66–73. ISSN: 0740-7475. DOI: 10.1109/MCSE.2010.69. URL: http://dx.doi.org/10.1109/MCSE.2010.69.

[83] H. Sutter. *Welcome to the Jungle*. http://herbsutter.com/welcome-to-the-jungle/. [Last access 6th May 2015]. 2012.

[84] Wen Jun Tan et al. “A Code Generation Framework for Targeting Optimized Library Calls for Multiple Platforms”. In: *IEEE Transactions on Parallel and Distributed Systems* 26.7 (2015), pp. 1789–1799. ISSN: 1045-9219.

[85] Terry Yin. *Lizard: an Cyclomatic Complexity Analyzer Tool*. https://github.com/terryyin/lizard. Online; accessed 19 October 2017.

[86] William Thies, Michal Karczmarek, and Saman Amarasinghe. “StreamIt: A Language for Streaming Applications”. In: *Compiler Construction: 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings*. Ed. by R. Nigel Horspool. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 179–196. ISBN: 978-3-540-45937-8. DOI: 10.1007/3-540-45937-5_14.

[87] Georgios Tournavitis and Björn Franke. “Semi-automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profiling Information”. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’10. Vienna, Austria: ACM, 2010, pp. 377–388. ISBN: 978-1-4503-0178-7.

[88] Sibel Yenikaya, Gökhan Yenikaya, and Ekrem Düven. “Keeping the Vehicle on the Road: A Survey on On-road Lane Detection Systems”. In: *ACM Comput. Surv.* 46.1 (July 2013), 2:1–2:43. ISSN: 0360-0300. DOI: 10.1145/2522968.2522970. URL: http://doi.acm.org/10.1145/2522968.2522970.

[89] Z. Zhong, V. Rychkov, and A. Lastovetsky. "Data Partitioning on Multicore and Multi-GPU Platforms Using Functional Performance Models". In: *IEEE Transactions on Computers* 64.9 (2015), pp. 2506–2518. ISSN: 0018-9340. DOI: 10.1109/TC.2014.2375202.