Domain-Specific Language & Support Tools for High-Level Stream Parallelism

Thesis Submitted for the Degree Requirements of:
Doctor of Philosophy in Computer Science

Dalvan Griebler

Advisor:
Prof. Dr. Marco Danelutto
Co-advisor:
Prof. Dr. Luiz Gustavo Fernandes

Computer Science Department, Ph.D. Program in Computer Science
University of Pisa, Pisa, Italy

Printed: April 28, 2016

Assessment Committee

Assoc. Prof. Marco Aldinucci

Computer Science Department, University of Torino, Torino - Italy

Assoc. Prof. José Daniel Garcia

Computer Science and Engineering Department, University Carlos III of Madrid, Madrid - Spain

Assoc. Prof. Cesar A. F De Rose

Faculty of Informatics, Computer Science Graduate Program, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre - Brazil

Abstract

Stream-based systems are representative of several application domains including video, audio, networking, graphic processing, etc. Stream programs may run on different kinds of parallel architectures (desktop, servers, cell phones, and supercomputers) and represent significant workloads on our current computing systems. Nevertheless, most of them are still not parallelized. Moreover, when new software has to be developed, programmers often face a trade-off between coding productivity, code portability, and performance. To solve this problem, we provide a new Domain-Specific Language (DSL) that naturally/on-the-fly captures and represents parallelism for stream-based applications. The aim is to offer a set of attributes (through annotations) that preserves the program's source code and is not architecture-dependent for annotating parallelism. We used the C++ attribute mechanism to design a "*de-facto*" standard C++ embedded DSL named SPar. However, the implementation of DSLs using compiler-based tools is difficult, complicated, and usually requires a significant learning curve. This is even harder for those who are not familiar with compiler technology. Therefore, our motivation is to simplify this path for other researchers (experts in their domain) with support tools (our tool is CINCLE) to create high-level and productive DSLs through powerful and aggressive source-to-source transformations. In fact, parallel programmers can use their expertise without having to design and implement low-level code. The main goal of this thesis was to create a DSL and support tools for high-level stream parallelism in the context of a programming framework that is compiler-based and domain-oriented. Thus, we implemented SPar using CINCLE. SPar supports the software developer with productivity, performance, and code portability while CINCLE provides sufficient support to generate new DSLs. Also, SPar targets source-to-source transformation producing parallel pattern code built on top of FastFlow and MPI. Finally, we provide a full set of experiments showing that SPar provides better coding productivity without significant performance degradation in multi-core systems as well as transformation rules that are able to achieve code portability (for cluster architectures) through its generalized attributes.

Sommario

I sistemi che elaborano stream di dati vengono utilizzati in svariati domini applicativi che includono, per esempio, quelli per il trattamento di video, audio, per la gestione delle reti e per la grafica. I programmi che elaborano stream di dati possono essere utilizzati e fatti girare su diversi tipi di architetture (dai telefoni cellulari, ai sistemi desktop e server, ai super computer) e di solito hanno unpeso computazionale significativo. Molti di questi programmi sono ancora sequenziali. Inoltre, nel caso di sviluppo di nuove applicazioni su stream, i programmatori devono impegnarsi a fondo per trovare un buon compromesso fra programmabilità, produttività, portabilità del codice e prestazioni. Per risolvere i problemi relativi alla parallelizzazione e allo sviluppo di applicazioni su stream, abbiamo messo a disposizione un linguaggio di programmazione domain-specific (DSL) che cattura e rappresenta gli aspetti legati al parallelismo in applicazioni su stream. Lo scopo è quello di offrire una serie di attributi (annotazioni) che, modellando gli aspetti relativi al calcolo parallelo dei dati sugli stream, permettano di preservare il codice originale e non dipendano dall'architettura considerata. Abbiamo utilizzato gli attributi C++11 per mettere a disposizione un DSL "interno" chiamato SPar pienamente conforme allo standard C++. L'implementazione di un DSL mediante compilatori è un processo complicato e che normalmente richiede un lungo processo di apprendimento relativo agli strumenti utilizzati; il processo è tanto più lungo quanto meno familiari si è rispetto alla tecnologia degli strumenti di compilazione. La motivazione che ci ha spinto a questo lavoro è dunque quella di semplificare la vita ad altri ricercatori (esperti del loro specifico dominio applicativo) mettendo a disposizione strumenti (CINCLE) che permettono la realizzazione di DSL di alto livello attraverso trasformazioni di codice source-to-source efficaci e potenti. Tramite gli strumenti messi a disposizione i programmatori di applicazioni parallele possono utilizzare la loro esperienza senza dover progettare e implementare codice di basso livello. Lo scopo principale di questa tesi è quello di creare un DSL ad alto livello di astrazione per computazioni parallele su stream e di metterne a disposizione gli strumenti di supporto in un framework basato su compilatori e orientato al dominio delle applicazioni stream parallel. Si è arrivati così alla realizzazione di SPar basata su CINCLE. SPar mette a disposizione dello sviluppatore di applicazioni un ambiente ad alta produttività, alte prestazioni e che garantisce la portabilità del codice, mentre CINCLE mette a disposizione il supporto necessario a generare nuovi DSL. SPar mette a disposizione trasformazioni source-to-source che producono codice parallelo basato su pattern in FastFlow e MPI. Alla fine della tesi presentiamo una serie completa di esperimenti che mostrano sia come SPar fornisca una buona produttività nella progettazione e realizzazione delle applicazioni parallele su stream senza al contempo portare a un degrado nelle prestazioni su sistemi multi core, sia come le regole di trasformazione utilizzate per la generazione del codice FastFlow o MPI permettano di realizzare laiv portabilità del codice basato su attributi su architetture di tipo diverso.

Resumo

Sistemas baseados em fluxo contínuo de dados representam diversos domínios de aplicações, por exemplo, video, áudio, processamento gráfico e de rede, etc. Os programas que processam um fluxo contínuo de dados podem executar em diferentes tipos de arquiteturas paralelas (estações de trabalho, servidores, celulares e supercomputadores) e representam cargas de trabalho significantes em nossos sistemas computacionais atuais. Mesmo assim, a maioria deles ainda não é paralelizado. Além disso, quando um novo software precisa ser desenvolvido, os programadores necessitam lidar com soluções que oferecem pouca produtividade de código, portabilidade de código e desempenho. Para resolver este problema, estamos oferecendo uma nova linguagem específica de domínio (DSL), que naturalmente captura e representa o paralelismo para aplicações baseadas em fluxo contínuo de dados. O objetivo é oferecer um conjunto de atributos (através de anotações) que preservam o código fonte do programa e não é dependente de arquitetura para anotar o paralelismo. Neste estudo foi usado o mecanismo de atributos do C++ para projetar uma DSL embarcada e padronizada com a linguagem hospedeira, que foi nomeada como SPar. No entanto, a implementação de DSLs usando ferramentas baseadas em compiladores é difícil, complicado e geralmente requer uma curva de aprendizagem significativa. Isto é ainda mais difícil para aqueles que não são familiarizados com uma tecnologia de compiladores. Portanto, a motivação é simplificar este caminho para outros pesquisadores (sabedores do seu domínio) com ferramentas de apoio (a ferramenta é chamada de CINCLE) para implementar DSLs produtivas e de alto nível através de poderosas e agressivas transformações de fonte para fonte. Na verdade, desenvolvedores que criam programas com paralelismo podem usar suas habilidades sem ter que projetar e implementar o código de baixo nível. O principal objetivo desta tese foi criar uma DSL e ferramentas de apoio para paralelismo de fluxo contínuo de alto nível no contexto de um *framework* de programação que é baseado em compilador e orientado a domínio. Assim, SPar foi criado usando CINCLE. SPar oferece apoio ao desenvolvedor de software com produtividade, desempenho e portabilidade de código, enquanto CINCLE oferece o apoio necessário para gerar novas DSLs. Também, SPar mira transformação de fonte para fonte produzindo código de padrões paralelos no topo de FastFlow e MPI. Por fim, temos um conjunto completo de experimentos demonstrando que SPar oferece melhor produtividade de código sem degradar significativamente o desempenho em sistemas *multi-core* bem como regras de transformações que são capazes de atingir a portabilidade de código (para arquiteturas multi-computador) através dos seus atributos genéricos.

Acknowledgements

First of all, I would like to thank God for this opportunity and support given to me during my whole life. Second, all the people that were around me, that were part of this project. Their patience and help were fundamental. I especially thank my advisors for supporting and guiding this research as well as their availability to discuss and teach.

Preface

Research in parallel computing has been necessary aiming to achieve high-performance and exploit parallelism in Cluster, Multi-Core, and General-Purpose Graphic Processing Unit (GPGPU) architectures. We have had the pleasure of seeing the growth of new programming models, methodologies, and programming interfaces for the efficient use of such a great amount of computational power. These contributions have supported discoveries in many scientific research fields including molecular dynamic simulations, data analysis, weather forecasting, and aerodynamics simulations.

Even though several new interesting results have been achieved in parallel computing over the last two decades, more work still needs to be done to achieve higher level programming abstractions, coding productivity, better performance, etc. At the moment, exploiting parallelism is still a challenging task that requires significant expertise in parallel programming. For example, a software engineer must have a great deal of knowledge of one or more of the following aspects along with their respective challenges:

1. **Hardware infrastructure:** Optimization is not always abstracted by using programming frameworks. Thus, memory locality, cache misses, network communication, thread/process affinity, storage implementation, and energy consumption will significantly impact the application's performance and are heavily dependent on hardware optimized features.

2. **Programming models:** May be used and carefully optimized in different ways for synchronizing and communicating threads or processes. For instance, in shared memory, one must pay attention to race conditions and deadlocks. In message passing, deadlocks and process synchronizations are the most important. For heterogeneous programming, there are thread synchronization and memory copy between CPU and GPU. Finally, hybrid programming is a challenge for the efficient use of message passing and shared memory models.

3. **Problem decomposition:** Is the computational mapping in the processors. One must identify concurrent works and decompose them by using task, data, DataFlow, and stream parallelism.

4. **Parallelism strategies:** Are algorithmic skeletons and design patterns for helping programmers to express parallelism. Usually strategies like farm, pipeline, and MapReduce are already provided in programming frameworks through a template library such as FastFlow or TBB.

5. **Load balancing:** Refers to workloads being partitioned across CPUs. In general, a static and dynamic approach may be adopted. Some frameworks implement these primitives in their runtime system, such as OpenMP. However, it is up to the user to define appropriate chunk sizes so that the work will be balanced.

6. **Scheduling policies:** These are used to efficiently distribute jobs among CPUs. Most of the frameworks do not offer the user the freedom to implement their own scheduler, instead they must use one of the pre-defined scheduling policies such as round robin, fixed priority, FIFO or EDF. In some applications, if the runtime scheduler is not efficient enough, an ad-hoc scheduling policy must be implemented.

7. **Different Programming frameworks:** These may be used to help developers express parallelism. There are several available such as OpenMP, TBB, MPI, OpenCL, X10, CUDA, and FastFlow. They are not able to abstract all the previous aspects and most of them are designed for a particular programming model.

These items are only a partial list of the problems and possibilities regarding parallel programming. We could also highlight many challenges related to each one of these items that are still being researched to improve performance and abstraction. Moreover, some researchers prefer to focus on a specific set of aspects, concerning a specific architecture. Focusing on a specific domain helps people to achieve better and more efficient solutions. Thus, when looking at the current state-of-the-art, only experts are able to efficiently parallelize their applications. It is clear that abstractions are also needed for software engineers and application developers because they already have to face the complexities of their domain.

This research problem initially prompted my Master's thesis, which proposed an external Domain-Specific Language for Pattern-Oriented Parallel Programming (named DSL-POPP) [Gri12]. The first version provided building block annotations for implementing master/slave-like computations on multi-core platforms. The results demonstrated significant programming effort reduction without performance losses [GAF14]. During the Ph.D., a second version was released, which supported pipeline-like parallelization [GF13]. Even though the results also demonstrated good performance, some abstraction limitations were discovered that considerably changed the subsequent domain and interface [GDTF15]. In general, many other problems arose from this initial research which made the work more advanced, primarily regarding high-level parallelism and DSL design.

Another related domain-specific language for MapReduce-like computations was proposed in [AGLF15a], which had the same principles as DSL-POPP. However, a completely new and unified programming language was created. The goal was to avoid MapReduce application developers from having to implement their code twice in order to run in distributed and shared memory environments. The performance results based on the transformation rules were efficient and the DSL significantly reduced the programming effort. The results of this collaboration further reinforced the importance of having a high-level abstraction to avoid architecture details (Chapter 6).

Additionally, the experience with the external DSL also demonstrated many advantages for coding productivity, as presented before in DSL-POPP [AGLF15b].

These past experiences were fundamental to the planning and development of the proposed programming framework (Chapter 3). DSL-POPP was initially an external interface and a cross-compiler was manually developed. Since the goal was always to preserve the sequential source code by only adding annotations, developers still had to learn another language when using it. Consequently, the drawback was that it did not preserve the syntax of the C language. Also, as the compiler did not follow any standardization and due to its internal customization for a specific pattern, there was no modularity for adding new annotations and features. In the literature there are only a small set of tools for creating internal DSLs in C/C++ (Clang and GCC plugins). Most of them only extend language capabilities and their internal implementations vary. Thus, there is a significant learning curve to implement a solution, which nonetheless has more or less the same limitations found previously in DSL-POPP.

One of the design goals of our framework is modularity and simplicity for building high-level C/C++ embedded DSLs. The challenge is to allow parallel programming experts to provide custom annotations by using the standard C++ mechanism so that parallel code can be automatically generated. Although this mechanism (C++11 attributes) was released in 2011, GCC plugins (compilation time callbacks for the AST) and Clang (compiler front-end on top of LLVM) are still difficult to customize and create new attribute annotations.

Another limitation is that these tools do not allow for transformations in the Abstract Syntax Tree (AST). Thus, there are only two options: use pretty print parallel code or build another AST when parsing the code so that modifications will be performed in the new one. GCC plugin constraints and complexities are justified by the C/C++ flexibility and its design goal is not for source-to-source code transformations/generations. On the other hand, Clang provides better modularity with functionalities for parsing AST and creating another one, but it still requires significant expertise in its front-end library and compilers.

The major difference between the annotation-based proposal is that instead of using pragma annotations, C++11 attributes are not preprocessing annotations and are fully represented in the AST with other C/C++ statements. Consequently, they are parsed in the AST, giving more power to the language developer to perform transformations. All of these aspects demonstrate that a Compiler Infrastructure for building New C/C++ Language Extensions was necessary, which is referred to in the text as CINCLE[i] (Chapter 4). This contribution allows one to achieve high-level parallelism abstractions, as will be presented in Chapter 5 by implementing a DSL for stream parallelism (SPar).

[i]This is also the name of the bird that lives on shallow streams in Italy, France and Germany

Stream domain was chosen as the annotation interface because it is interesting, widely used, simple enough, general, and suitable for teaching purposes. Moreover, it helps us to address another new perspective and allows application-level DSL designers to integrate automatic parallelization through SPar. Vertical validation of the framework was done through a DSL for geospatial data visualization targeting multi-core parallelism [LGMF15] [Led16]. When compared with TBB, SPar was able to increase coding productivity by 30% [ii] without significant performance losses.

This research is also a collaboration with the University of Pisa, providing inspiration for this work (*e.g.* such as the adoption of the stream parallelism domain and the use of the C++ annotation mechanism). The major ideas come from EU (European Union) projects such as REPARA[iii] (annotation mechanism) and the open source project at UNIPI such as FastFlow[iv] (stream parallelism). Both projects have justified and inspired some of our work.

In this thesis, we will provide a new programming framework perspective for high-level stream parallelism. Our motivation is to contribute to the scientific community with a high-level parallelism design with support tools for generating new embedded C++ DSLs. It primarily supports users to build custom and standardized annotation-based interfaces to perform powerful source-to-source transformations.

Another contribution is to enable productive stream parallelism by preserving the sequential source code of the application. In this case, we prototyped a new DSL with suitable attributes at the stream domain level by using our designed infrastructure, which also became the use case to illustrate its capabilities and robustness. Moreover, as a consequence of the generalized transformation rules created, our DSL seeks to support code portability by performing automatic parallel code generation for multi-cores and clusters.

[ii] Measuring the source lines of code.
[iii] http://repara-project.eu/
[iv] http://calvados.di.unipi.it/

List of Papers

1. **Towards a Domain-Specific Language for Patterns-Oriented Parallel Programming.** *Programming Languages - 17th Brazilian Symposium - SBLP* [GF13].

2. **Performance and Usability Evaluation of a Pattern-Oriented Parallel Programming Interface for Multi-Core Architectures.** *The 26th International Conference on Software Engineering & Knowledge Engineering* [GAF14].

3. **An Embedded C++ Domain-Specific Language for Stream Parallelism.** *International Conference on Parallel Computing (ParCo 2015)* [GDTF15].

4. **A Unified MapReduce Domain-Specific Language for Distributed and Shared Memory Architectures.** *The 27th International Conference on Software Engineering & Knowledge Engineering* [AGLF15a].

5. **Coding Productivity in MapReduce Applications for Distributed and Shared Memory Architectures.** *International Journal of Software Engineering and Knowledge Engineering* [AGLF15b].

6. **Towards a Domain-Specific Language for Geospatial Data Visualization Maps with Big Data Sets.** *ACS/IEEE International Conference on Computer Systems and Applications* [LGMF15].

List of abbreviations

SPar Stream Parallelism

CINCLE Compiler Infrastructure for New C/C++ Language Extensions

AST Abstract Syntax Tree

GCC GNU C Compiler

GPGPU General-Purpose Graphics Processing Unit

GPU General-Purpose Graphics Processing Unit

CPU Central Processing Unit

FIFO First In, First Out

EDF Earliest Deadline First

OpenMP Open MultiProcessing

TBB Threading Building Blocks

MPI Message Passing Interface

OpenCL Open Computing Language

CUDA Compute Unified Device Architecture

UNIPI University of Pisa

REPARA Reengineering and Enabling Performance and poweR of Applications

DSL Domain-Specific Language

IoT Internet of Things

DAG Directed Acyclic Graph

PPL Pervasive Parallelism Laboratory

IR Internal Representation

API Application Program Interface

PIPS Parallelization Infrastructure for Parallel Systems

RTL Register Transfer Language

LLVM Low-Level Virtual Machine

EDG Edison Design Group

ANTRL Another Tool For Language Recognition

IDE Integrated Development Environment

FPGA Field-Programmable Gate Array

DSP Digital Signal Processor

MIT Massachusetts Institute of Technology

APGAS Asynchronous Partitioned Global Address Space

SLOC Source Lines of Code

Contents

List of Figures

List of Tables

Part I

Scenario

1

Introduction

This chapter will introduce and contextualize the dissertation research.

Contents

1.1 Contextualization

In order to contextualize the research problems and challenges, the first section will present the central perspectives regarding high-level parallelism. The goal is to present the main challenges and issues of providing high-level parallelism in respect to the state-of-the-art alternatives, and show how our research provides solutions to these concerns. The subsequent section will introduce the stream parallelism domain and highlight its central difficulties when parallelizing with current state-of-the-art tools. Additionally, we believe that stream parallelism properties are generic enough to increase the level of abstraction. Thus, we will show their advantages and how we plan to implement them using a standard C++ mechanism.

1.1.1 Perspectives on High-Level Parallelism

For many years parallel computing has been mostly considered in specialized supercomputer centers, but this has dramatically changed in the last decade. Currently, there are different degrees of parallelism from embedded systems to high-performance servers, due to the availability of multiprocessing architectures such as multi-core, accelerators and clusters [RJ15, RR10]. Also, technology improvements have contributed to increasing the capabilities of hardware resources such as memory, network and storage, and supporting complex software on different kinds of devices. This heterogeneity raises many challenges for software developers regarding performance portability[i], code portability[ii] and coding productivity[iii].

In the software industry, many general-purpose programming languages are making progress on higher-level abstractions, supporting software engineers in the building process of complex applications with better code productivity. However, these applications have many challenges to achieve performance and code portability on parallel architectures while preserving their coding productivity. Unfortunately, compilers such as GCC are not able to automatically parallelize code from these high-level language abstractions. In fact, from a compiler's point of view only vectorized code is automatically parallelized, while other high-level abstractions (viewed as coarse-grained code regions) do not provide the necessary semantic information for the compiler to perform code parallelization. Consequently, developers are forced to restructure their application by using low-level and architecture-dependent programming interfaces

[i] It means the code achieves the same performance on different platforms and architectures.
[ii] Allows a code to run in different architectures and platforms without any changes.
[iii] Reduces the amount of code and programming effort.

such as MPI [GHLL+98], CUDA [KmWH10] and Pthreads [NBF96] if they hope to exploit parallel hardware efficiently.

The current software development process consists of prototyping an efficient program in a high-level language and then implementing some kind of high-performance code. This two-step process is very time-consuming, especially if we take into account that the code eventually produced is strongly architecture-dependent. For instance, it requires the programmer to have expert knowledge in hardware and parallel programming to produce high-performance code when targeting different architectures. In big team projects, some programmers will work on the high-level part while others will have to implement the lowest-level version of the application. The problem is that different versions of the parallelized application will have to be implemented in order to target different architectures. Thus, a project will usually have many versions that differ slightly from the original and any update could require rethinking the parallelization strategy in order to better exploit the architecture's resources.

To solve this problem, the Domain-Specific Language (DSL) approach has proven to be a suitable alternative to high-level parallelism abstractions in recent years [Gri12, GAF14, GF13, AGLF15a, AGLF15b]. Also, DSLs have proven to be effective when targeting code productivity, code portability, and performance in different general-purpose programming languages [SLB+11, DJP+11, HSWO14, Suj14]. Though these solutions lack generality, they increase optimization and abstraction to enable the user to focus on better ideas and algorithms rather than on general problems related to parallelism exploitation.

A DSL can be implemented using different techniques. An "External" DSL implementation is a language completely distinct from the host language. Thus, it is necessary to create a custom compiler [Fow10, Gho11]. Usually, external DSLs are more flexible and easier to model for the domain-specific scenario. Yet, depending on the environment, they require much more expertise in compiler design even using compiler frameworks like LLVM, because the high-level interface must be translated into the low-level intermediate representation. On the other hand, "Internal" DSL implementations are fully integrated with the host language syntax and semantics. They are provided as a library or by using specific host language mechanisms [VBD+13]. Our DSL uses the embedded C++11 attributes [ISO11b, ISO14] already present in the host language grammar and therefore it is "de facto" an internal/embedded DSL.

According to the literature, an internal DSL should be easier to implement because the host language should provide an alternative to suitably integrate custom language constructions. However, in C++, the context is entirely different when one intends to use its standard annotation mechanism. It requires a profound knowledge of compiler design. We solved this problem by providing a new compiler infrastructure so that we can provide a suitable alternative because the literature does not enable higher abstraction level and aggressive code transformations. In addition, our programming

framework perspective aims to facilitate other kinds of language extensions or compiler-based tools to benefit from the infrastructure. This and other new perspectives will be discussed and proven throughout this dissertation.

In spite of the DSL benefits, designing an embedded C++ DSL for high-level parallelism abstractions is still a challenging task. It requires expertise in multiple areas such as parallel programming, programming languages, compiler architecture, etc. Recently, researchers from Stanford University have been working on the same subject to create high-performance and high-level embedded DSLs for the Scala language [OSV10]. They developed the Delite compiler framework to enable DSL designers to quickly and efficiently target parallel heterogeneous hardware [SBL+14]. Overall, their research builds on a stack of domain-specific solutions that in principle share similar goals with this work. In contrast, our idea is to contribute to the C/C++ [Str14] community, which is widely used in several market and real world applications.

Delite is logically between a high-level application DSL and low-level hardware and runtime libraries. Integrated with Scala, its framework provides parallel patterns that can be instantiated by the application's DSL designer without worrying about the parallelism aspect of heterogeneous hardware. Unlike C++, Scala is more recent and was created along with the DSL support implementation, which requires parallelism abstractions. In our proposed research, in addition to providing high-level parallelism to an application's DSL designers, we have also proposed a compiler infrastructure as an alternative for experts in parallelism exploitation to quickly prototype DSLs based on annotations. This contribution can significantly improve the abstraction level of parallelism in C++ programs. Similar to Delite, we propose an embedded C++ domain-specific language, yet for stream-oriented parallelism. Our goal is that the programmer will not be required to instantiate patterns or methods as in Delite. Instead we aim to preserve the application's source code as much as possible, only requiring the programmer to insert proper annotations for annotating the stream parallelism features.

In the C++ community, a research closest to ours is Re-engineering and Enabling Performance and poweR of Applications (REPARA) [REP16]. Its vision is to help develop new solutions for parallel heterogeneous computing [GSFS15] in order to strike balance between source code maintainability, energy efficiency, and performance [DTK15]. In general, REPARA differs from our work in many ways, but shares the idea of maintaining the source code by introducing abstract representations of parallelism through annotations [DGS+16]. Thus, a standard C++11 attribute mechanism is used as skeleton-like code annotations (farm, pipe, map, for, reduce, among others). Attributes are preprocessed and exploited within a methodology, which eventually produces code targeting heterogeneous architectures.

From the high-level parallelism perspective of REPARA, attributes are interpreted by a refactoring tool that is on top of eclipse IDE (Integrated Develop-

ment Environment), which is responsible for the source-to-source transformations to C++/FastFlow. As a consequence of refactoring methodologies, theses transformations occur in place and produce code that is transparent to users. Like REPARA, we aim to be standard C++ compliant. However, in our programming framework, attributes are used at compiler level and source-to-source code transformations are hidden from the users. Moreover, our goal is to be domain-specific for stream parallelism, targeting multi-core and clusters to support an application's DSL designer as in the Delite framework's vision.

There are also other programming interfaces that provide high-level parallelism that are not DSLs. Examples are MapReduce [MS12, DG08, CCA+10, Had16], Charm++ [AGJ+14, Cha16], X10 [Mil15, X1016], Spark [KKWZ15], Storm [And13], and Intel Cilk Plus [BJK+95, Cil16]. While offering many suitable features for different applications, these approaches force programmers to deal with different programming models that are not natural to the application domain. This negatively impacts coding productivity. Moreover, it is difficult for them to provide good performance when targeting different parallel architectures, because their embedded interface is still too low-level. Therefore, they require different implementation versions of the source code to target different hardware.

Our perspective on high-level parallel programming relies on language attributes that annotate abstract representations of potential parallelism. In our vision, other annotation-based models such as OpenMP [Qui03] are conceptually much lower-level, because users have to express the parallelism and deal with low-level details relative to high-performance coding. These interfaces achieve coding productivity only in specific cases such as independent loop parallelization. In addition, code portability is still strongly architecture-dependent, which requires the production of different versions of the application in order to target other architectures.

1.1.2 Stream Parallelism Domain

Stream processing is one of the most commonly used paradigms in computer systems [TA10, AGT14]. We can find it in our daily software applications and computational hardware. All personal computer processors run a sequence of instructions in a streaming fashion to achieve throughput, latency, and quality of service at the user level. On the software side, the increase of the Internet of Things (IoT) field and the big data explosion has made stream processing a trending topic in computer science. There are millions of data sources on the Internet that are collecting and exchanging information through devices and social media. Therefore, the need for high throughput and low latency is also crucial for software that deals with image, video, networking, and real time data analysis.

Different variants of the stream programming paradigm have emerged over the years (reactive, DataFlow and stream parallelism) [TA10, AGT14, TKA02a, ADKT14, ADK+11]. They characterize the stream as a set of continuous data/instructions/tasks that flows naturally through a sequence of operations. Each operation consumes input and produces output like an assembly line that is also called as a pipeline. In general, stream processing has a continuous flow and unbounded stream behavior. However, some of today's application scenarios are irregular and have bounded streams. Consequently, it is a challenge for stream-based systems to control the end of the stream and maintain good performance even if there is not infinite flow.

Stream processing variants share similar motivations, goals, and characteristics that make it difficult for a layperson to differentiate among them. Some scientists simply say that they are equivalent in many aspects. In fact, they all share the same principles. However, the resulting systems have subtle distinctions. It is possible to highlight that, for example, a reactive system is more concerned with latency than throughput. A typical example is a spreadsheet, where a person is usually applying several mathematical equations (stream operations) in the data cells (stream sources). Subsequently, the system reacts to all data updated in the cell, ensuring that latency is small when compared to human perception [Fur14].

A representation of reactive environmental characteristics can be found in Figure 1.1. Each actor in the system propagates operation results and reacts when they receives a new input event. Many web services use reactive programming for different types of events. For instance, the most common is to answer over click events such as subscriptions to a social network and to purchase goods. The system may have to deal with different click sources and instantaneously react to the event. Thus, parallelism is especially designed for each application to achieve latency, attend many events, react over data scaling, and recover when failures happen in a timely manner.

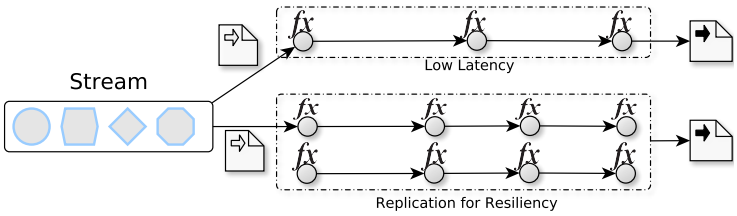


Figure 1.1: Reactive systems representation.

Listing 1.1 illustrates an example of a simple reactive stream code from the programming perspective. It is a program that reads events from the keyboard and can print the multiplication table. Therefore, the stream source is a digit or a set of digits that will be processed to return a multiplication table. Through this example it is possible to see the following challenges: Providing scalability and low response time when the user enters a set of digits to be computed; and preserving resiliency when

a non-digit character is inserted so that the application does not stop, crash or lose information.

```
1  void proc(){
2    int stream_source;
3    while(1){
4      std::cout << "Enter a digit: ";
5      std::cin >> stream_source;              //gets event
6      for (int i = 1; i < 11; ++i){           //computes the event
7        std::cout << stream_source*i << std::endl;  //prints the result
8      }
9      std::cout << "\n----------\n";
10   }
11 }
```

Listing 1.1: Reactive stream code example.

DataFlow programming is another stream-based paradigm, which is also called DataStream by some scientists to avoid being confused with the architectural term DataFlow [AGT14]. It is a way for runtime systems to automatically extract parallelism from an application. In order to do so, the programmer explicitly indicates how data will flow in the program in such a way that the system may build a Directed Acyclic Graph (DAG). Then, when there are independent operations, that have all of their input data available, they are designed to execute in parallel.

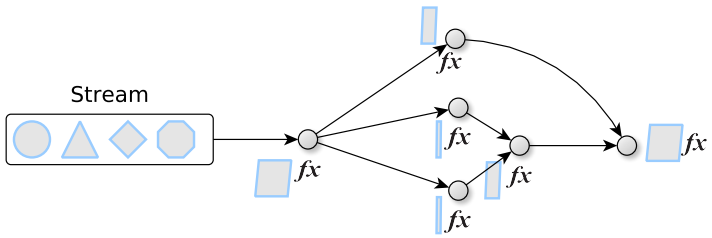


Figure 1.2: DataFlow/DataStream systems representation.

A representative illustration can be found in Figure 1.2, where the spheres are operators and arrows are dependencies. Usually, DataFlow computations are represented through dependency graphs in the main memory. Operators are processed by threads or processes as soon as all their input data items become available as shown in Figure 1.2. Hence, each data operator thread will know when its work can be done and go ahead. When input dependencies are used it means that an operator can only perform its computation after the input data is available. Similarly, an output dependency is a specification for subsequent operators stating where they will obtain their results. Consequently, the connections and synchronizations between operators are represented and ensured by input and output dependencies. This behavior is quite similar to reactive programming. However, DataFlow is more closely related to data parallelism than event driven parallelism (reactive systems).

When looking at the literature regarding parallel programming interfaces for multi-core architectures, we can point out different solutions that target the DataFlow paradigm [CJvdP07, PC11, Omp16]. OpenMP was originally designed for data parallelism in FORTRAN and C, but has also introduced task parallelism and some clauses (`depend, in, out, inout`) targeting a kind of DataFlow parallelism in task regions since its 4.0 version. A simple code example is given in Listing 1.2 to illustrate the programmer's point of view, presenting a program that performs a sequence of operations over a contiguous bounded data stream.

```
1  void proc(){
2    #pragma omp parallel
3    {
4      #pragma omp single
5      {
6        for (int i = 0; i < NREGION; ++i){
7          int *persons = new int[NPERSON];
8          #pragma omp task depend(out: persons[i])       //task group 1
9          {
10           load(persons);
11         }
12         #pragma omp task depend(in: persons[i])         //task group 2
13         {
14           regions_min[i] = min(persons);
15         }
16         #pragma omp task depend(in: persons[i])       //task group 3
17         {
18           regions_max[i] = max(persons);
19         }
20       }
21     }
22   }
23 }
```

Listing 1.2: OpenMP DataFlow code example.

The program performs a data analysis to identify the maximum and minimum age of people in each region. Therefore, the loop iterates for each region, loading everyone's age into a vector to find and store the maximum and minimum age. In order for OpenMP to exploit DataFlow parallelism, we have to describe the data dependencies in such a way that OpenMP can run the `max` and `min` operations in parallel, as presented in Listing 1.2 in the *pragma task* annotations. Nonetheless, its usage is strongly dependent on the OpenMP programming model. For instance, it is not enough to describe the data dependency. One must be aware of task parallelism, where task pragmas are a group of tasks that will run concurrently, and the output and input dependency will synchronize the data flow. In our OpenMP example, the first task group will update the input data of the next two task groups (`max` and `min` operators) so that they execute in parallel. Finally, note that a DataFlow region must also be encapsulated by *parallel* and *single* directives.

Stream parallelism programming inherits many of the capabilities of previous paradigms. In contrast to DataFlow programming, stream operations' dependencies are not determined by input and output data specifications. In stream parallelism, the operation sequence is structured in such a way that dependencies are evidenced and input and output describes what will be consumed and produced by a kernel as illustrated in Figure 1.3. A kernel is composed of an operator or a set of operations performed at each element of the stream.

Inside kernels, operations are expected to be sequential and they are performed locally during the computation. The specifications of input and output also help the system to prevent global data manipulation, whereas local operations can improve memory performance through data locality. The stream parallelism paradigm also simplifies the implementation of the task scheduler since the flow is completely deterministic. In DataFlow for example, the model is highly data dependent, resulting in a non-deterministic flow and complex scheduler implementation because the flow of tokens within the graph may vary depending on the input token values during the execution of the program. Depending on the token flow, the scheduler should make different scheduling decisions.

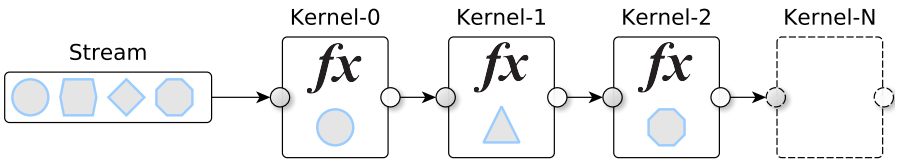


Figure 1.3: Stream systems representation.

Stream parallelism applications work over intensive and unbounded streams, focusing on high throughput rates and low latency. The undefined end of a stream is a difficult for DataFlow systems. On the other hand, stream systems have DataFlow semantics. Therefore, not all the graphs we can express with DataFlow may be expressed with stream parallelism. For example, similar undefined behavior is also present in reactive applications, but there is an elastic stream (a set of events) frequency that must be addressed in timely manner, while stream applications usually have bigger streams with a constant frequency where the throughput may be a priority instead of latency. Examples of real world applications are face tracking, video streaming, network packet routing, image processing, among others. Listing 1.3 outlines the structure of a typical stream parallel application.

The code example implements a stream application, where each element of the stream is a string. Therefore, the stream application starts on the "while" loop. At each iteration, the loop reads a stream element, computes a result using the element, and writes the result on a given output stream. The end of the stream is monitored by a condition checked after reading one element. Note that it is common that the number of computations of the stream elements (also called as filter) vary among the

applications, but a stream application will usually have a sequence for these three operations: read, filter, and write [TA10].

```
void proc_seq(){
  std::string stream_element;
  while(1){
    read_in(stream_element);  //reads each element of a given stream
    if(stream_in.eof()) break;  //test if stream is empty
    compute(stream_element);  //apply some computation to the stream
    write_out(stream_element);  //write the results into a given output
     stream
  }
}
```

Listing 1.3: Stream application code example.

Due to the fact that OpenMP is not designed for naturally annotating these kinds of applications, the most efficient way to explore stream parallelism is to use FastFlow [Fas16, ADKT14] or TBB [Rei07, TBB16] frameworks. Both frameworks also support DataFlow parallelism and present a similar programming interface because they leverage the same meta-programming features. However, the runtime systems are very different as are the design goals, which will be explained in detail later in this dissertation (Chapter 2). To exemplify the expressiveness of stream parallelism when using such frameworks, a pseudo code in FastFlow is given in Listing 1.4, extending the example of Listing 1.3. From this code, we can point out the main drawbacks of these frameworks are source code rewriting and restructuring.

```
struct firstStage: ff_node_t<std::string> {
  std::string stream_element;
    std::string *svc(std::string *) {
    while(1){
      read_in(stream_element);
      if(stream_in.eof()) break;
      ff_send_out(new std::string(stream_element));
    }
    return EOS;
    }
};
struct secondStage: ff_node_t<std::string> {
    std::string *svc(std::string *stream_element) {
        compute(*stream_element);
    return stream_element;
    }
};
struct thirdStage: ff_node_t<std::string> {
    std::string *svc(std::string *stream_element) {
    write_out(*stream_element);
        delete stream_element;
    return GO_ON;
    }
```

```
};
void proc_ff(){
  ff_Pipe<> pipe(make_unique<firstStage>(),
    make_unique<secondStage>(),
    make_unique<thirdStage>());
    if (pipe.run_and_wait_end()<0) error("running pipe");
}
```

Listing 1.4: FastFlow stream code example.

In contrast to these frameworks, the scope of this research is limited to stream parallelism and does not address similar approaches such as DataFlow or Reactive parallelization. The main expected state-of-the-art contribution is to provide a high-level DSL for expressing stream parallelism. Even though C++ template libraries can provide interesting coding productivity and abstractions, this work aims to raise the abstraction level without significantly affecting performance. The proposal is to use the standard C++ attribute mechanism [MW08, ISO14], maintaining on-the-fly stream parallelism such as the example in Listing 1.5, where the sequential source code (Listing 1.3) is not restructured. Chapter 5 will present and discuss the proposed DSL in detail to address stream parallelism by using standard C++ attributes.

```
void proc_spar(){
  std::string stream_element;
  [[spar::ToStream,spar::Input(stream_element)]] while(1){
    read_in(stream_element);
    if(stream_in.eof()) break;
    [[spar::Stage,spar::Input(stream_element),spar::Output(
   stream_element)]]
    { compute(stream_element); }
    [[spar::Stage,spar::Input(stream_element)]]
    { write_out(stream_element); }
  }
}
```

Listing 1.5: Proposal interface exemplification for stream parallelism.

1.2 Goals

The main objectives of this dissertation are the following:

- **G1**: The first goal is to provide support tools that enable parallel programming experts to create standard C++ embedded DSLs.
- **G2**: The second goal is to provide high-level stream parallelism and coding productivity without significant performance degradation in multi-core systems.

- **G3**: The third goal is to introduce code portability in multi-core and cluster systems.

1.3 Contributions

This work contributes to a programming framework for high-level parallelism abstractions that includes the following specific contributions:

- **C1: A compiler infrastructure for generating new language extensions in C/C++.**
- **C2: A domain-specific language for stream parallelism.**
- **C3: Generalized transformation rules for source-to-source code generation that exploits parallelism in multi-core and cluster.**
- **C4: Experimental validation through the implementation of several different use cases to access features of the framework and design choices.**

1.4 Outline

This dissertation is organized in five parts:

- **Scenario**: This part of the dissertation introduces the context, problems, challenges, and motivations in Chapter 1. Chapter 2 presents the related works to highlight the differences and similarities to our research.
- **Contributions**: This is the most important part of the dissertation. Chapter 3 gives an overview of the contributions, first presenting the programming framework. Then, it informally discusses each one of the contributions related to the framework. Chapter 4 details this contribution **C1** (see Section 1.3), presenting the Compiler Infrastructure for New C/C++ Language Extensions (CINCLE). Then, Chapter 5 details contribution **C2**, presenting an Embedded C++ DSL for Stream Parallelism (SPar). Chapter 6 details contribution **C3**, introducing code portability for multi-core and cluster systems.

- **Experiments**: This corresponds to Chapter 7 and contribution **C4**. First we present our experimental methodology. Secondly we discuss the experiments that were performed with a set of applications for evaluating productivity and performance in a multi-core environment, comparing it with other frameworks. Finally, we evaluate the coding productivity along with performance in the cluster environment.

- **Discussions**: In Chapter 8 we conclude the dissertation. Chapter 9 then describes future works and research perspectives.

- **Complements**: This is the complementary part of the work. Chapter 10 is composed of the bibliographies and Chapter A has the appendixes that support our discussions.

Readers may prefer to navigate the document in different ways. Figure 1.4 illustrates a flowchart of the dissertation as a reading guideline. The preface is an extended abstract to give more details about the dissertation and origins of the subject. The introduction discusses the problem, motivations, and the expected contributions in respect to the state-of-the-art. Consequently, if the reader is comfortable with the scenario, it is possible to go directly to the overview of the contribution chapter and then read related work for accurate information.

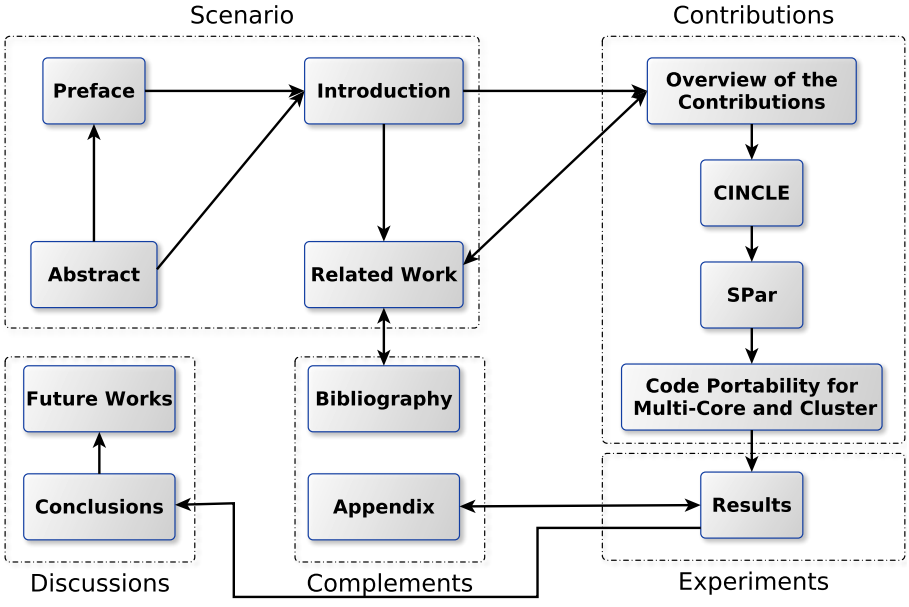


Figure 1.4: Thesis flowchart.

In the overview of the contribution chapter, we illustrate the thesis picture and informally summarize the contributions. As in the flowchart, this part should be read in sequence. First, the CINCLE chapter describes the technical aspects of the infrastructure used to subsequently implement SPar, which is a high-level

domain-specific language for annotating stream parallelism. Then, we introduce code portability with transformation rules based on the SPar attributes.

After the contributions, the experiments portion presents the results of the productivity, performance, and portability evaluation. Here one may refer to the appendix chapter to see complementary materials for the assessments. Having become aware of the achieved results, one can then go to the discussion portion, where they will find the conclusions and future research perspectives.

2

Related Work

This chapter presents an overview of the state-of-the-art works related to this research.

Contents

2.1 High-Level Parallelism

In this section we aim to present two research projects that inspired our work. First, we introduce the REPARA research project which shares similar ideas regarding C++ programming and tools. Then we present the research framework from the pervasive parallelism laboratory at Stanford University which shares the domain-specific perspective. Finally, we will discuss both works and compare them with ours.

2.1.1 REPARA Research Project

Re-engineering and Enabling Performance and poweR of Applications (REPARA) is a research project that started in September 2013 [REP16], founded by the Seven Framework Programme (FP7-ICT). Its vision is to help develop new solutions for parallel heterogeneous computing [GSFS15], balancing source code maintainability, energy efficiency, and performance [DTK15]. Figure 2.1 presents the workflow proposed to meet their goals (detailed in [REP16]), starting from the original source code and moving to parallel the heterogeneous platforms (GPU and FPGA accelerators and multi-cores).

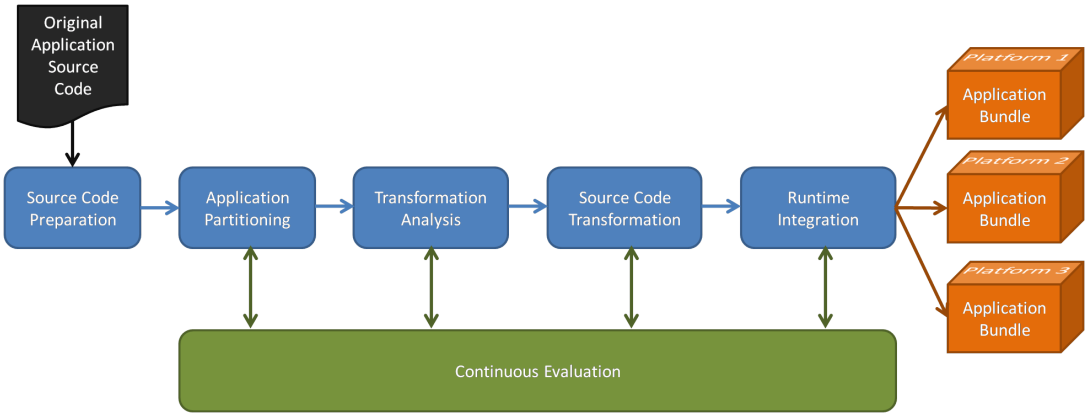


Figure 2.1: REPARA' workflow. Extracted from [REP16]

The first step is to prepare the source code. This is necessary to provide explicit rules to express algorithms, tools for statical analysis and interactive refactoring, and techniques for annotating source codes. The second step is application partitioning in order to help application transformation to be mapped onto different devices. This is based on the source code description for dynamic (run-time) and static (compile-time)

partitioning. The next step (transformation analysis) creates an abstract representation of the opportunities for transformation. Consequently, the source code transformation step takes into account this analysis to implement parallel code refactoring. The goal is to work with interactive refactoring (tools integrated into IDE) to offer specific transformations and non-interactive modes to re-apply the changes after refactoring has been done.

REPARA also has a runtime integration step because transformations are not enough to integrate different platforms. Primarily this is done to target different frameworks with distinct libraries and tools. This can be managed inside application partitioning tools, where the main goal is to change the application configuration dynamically to achieve better balancing of power efficiency and performance. Moreover, the central point of continuous evaluation goal is to also provide: I) qualitative estimations to evaluate the effects of partitioning; II) quantitative predictions to list opportunities in the transformation analysis; III) estimations of performance and energy efficiency during transformations; IV) software maintainability addressing the application source code; and V) the integrated application to be monitored to improve predictions.

All of these projects aim to develop different solutions. They have implemented algorithmic skeletons that are introduced by using the C++11 attribute mechanism for language representation. REPARA has created a set of partitioning tools that target parallel patterns including pipe, farm, kernels, map, reduce, and others [DGS+16, Pro14]. Also, some of the attributes have been integrated into Eclipse IDE plugin to source-to-source transformations with FastFlow targeting multi-core and DSP/FPGA [Pro15]. Because the project is still being developed, other tools are being created that will validate REPARA's approach regarding the workflow of ideas and goals.

2.1.2 Stanford Pervasive Parallelism Research

The Stanford Pervasive Parallelism Laboratory (PPL) has a related research interest in the field of domain-specific languages. The research project has been active since 2011 [BSL+11, CSB+11] and aims to develop a parallel computing platform by the year 2020 [PPL16]. The primary goal is to make new DSLs for software developers (domain experts) available, while taking advantage of heterogeneous parallelism for those who are not experts in parallel programming. To achieve such a challenging task, they are building a framework to support tools that are fully domain-oriented, as shown in Figure 2.2. It provides a stack of multiple layers that goes from major scientific applications to heterogeneous hardware devices.

This approach aims to create a consolidated environment for the Scala language community and DSL designers with a parallel compiler and runtime infrastructure.

Their research efforts are mainly related to the development of efficient mechanisms for communication, synchronization, and performance monitoring. As a result, new domain-specific application DSLs will arise where parallel programming should be completely transparent to the users [BSL+11, CSB+11]. Moreover, unlike the REPARA project, the research is supported through the collaboration of open industrial affiliates.

PPL's first layer consists of embedding a DSL with a high-level host language (Scala). Thus, it can be integrated within the compiler, which is called *Staging* and known as *Lightweight Modular Staging* for generating code by using Scala facilities [RO10]. Along with this support, there are also domain-specific optimizations and polymorphic embedding provided by the parallel runtime. In principle, they are APIs implemented by the Delite compiler architecture that integrate several techniques for source-to-source code transformations and data structures [BSL+11, CSB+11].

Delite was primarily created to support a large number of IR node types to create a common IR to represent the source code for different targets [SBL+14]. It also includes data structures, parallel operators, built-in optimizations (matrix multiplication and vector plus support), traversals and transformers (IR level rewriting rules using pattern matching), and code generators for different platforms (from IR to optionally Scala, C++, CUDA, and OpenCL). In fact, parallel operators available through API's library may extend a pattern operator to enable parallelism. They provide these pattern operators in a simple way to target task and data parallelism and other domain-specific optimizations, such as scheduling and data locality.

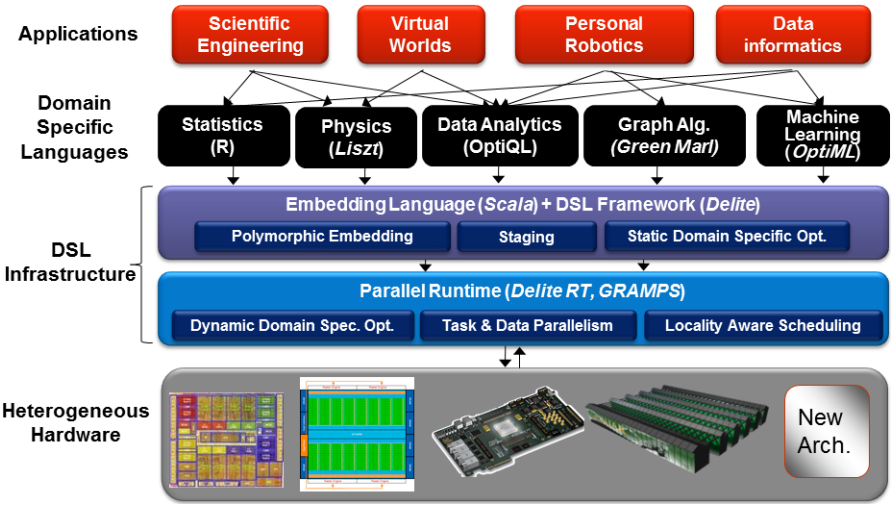


Figure 2.2: Stanford pervasive parallelism research framework. Extracted from [PPL16].

Although the target date is 2020, they have provided many contributions in all the framework layers. In addition to Delite, many DSLs were created on top of the framework to support major important scientific applications [SLB+11, DJP+11, HSWO14, Suj14].

2.1.3 Discussion

In this section, we presented two distinct research projects that inspired our programming framework for research (Section 3.2) and underlying DSL with support tools for high-level stream parallelism. Although there are many differences in their design goals and methods, these researchers share similar perspectives related to high-level parallelism. We are aiming for an appropriate solution to achieve and provide simpler programming environments as well as to productively and efficiently exploit parallelism in a different manner.

In contrast to REPARA, whose annotations target general purpose parallelism with closed semantics to support stream parallelism, we follow a domain-oriented and compiler-based approach to introduce parallelism. In fact, REPARA is more concerned with performance and energy efficiency through static and dynamic partitioning, which are implemented based on the information of the annotated source code. Thus, the attribute must be generic and capable of providing and retaining enough information for the IR to enable sophisticated analysis and transformations. Also, their annotations are integrated into a refactoring tool (IDE plugin), where the attributes are generated by the partitioning tools (or, for the initial part of the project, by programmers), then the IDE plugin tool re-factors the attributes to run-time calls (FastFlow Calls).

We can observe that REPARA and PPL projects seek to create runtime support for heterogeneous devices such as reconfigurable and GPU accelerators. In contrast, we aim to reuse these runtimes to elevate the abstraction level, providing better code portability and productivity. Another similarity of these two projects is that they both provide abstraction at parallel pattern level. Yet, we generate parallel code by using generalized transformation rules that target parallel patterns.

Unlike from PPL, we are contributing to the C++ community by allowing non-expert parallel programmers to take advantage of parallelism in different architectures. Similarly to Delite, we create support tools to support C++ DSL designers with a compiler-based infrastructure that can offer simpler abstractions and aggressive source-to-source code transformations. Yet, even though PPL benefits from a high-level productive language like Scala, parallelism is not delivered by using annotations as we are providing for the DSL designers.

2.2 C/C++ DSL Design Space

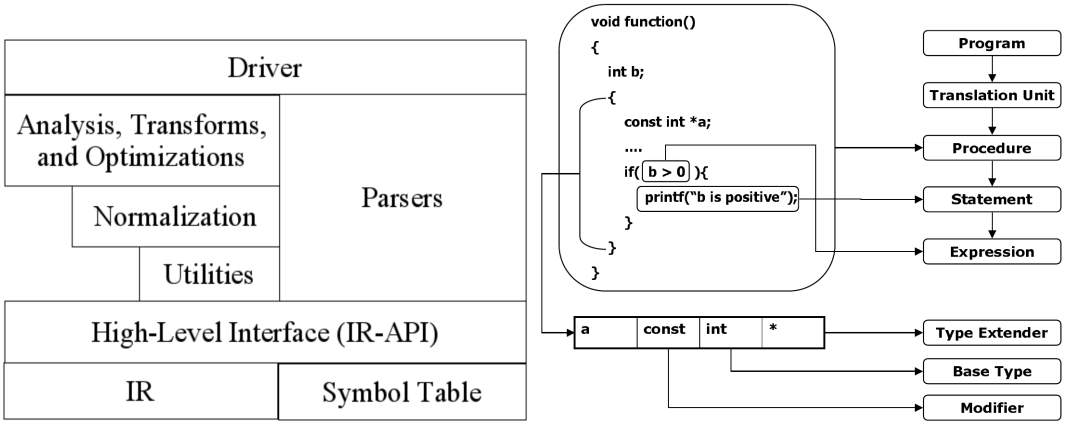
The DSL design space for C/C++ requires a compiler infrastructure or a framework able to provide suitable features to extend languages and support an abstract representation of the source, as presented in the compiler design [ALSU07]. Among several tools available in the literature, we considered only the tools implementing the characteristics related to the needs of our research. Also, we took into account open source projects since proprietary tools are not compatible with our design goals. In this section, we will discuss the aspects related to automatic source-to-source transformation as well as compiler-based tools' infrastructure.

2.2.1 Cetus

Cetus is a source-to-source compiler infrastructure for multi-core systems [DBM+09]. It targets automatic parallelization through C pragmas to support source-to-source code transformations in C programs. The infrastructure is written in Java and uses ANTRL [Par13] to provide an internal C parser. Currently, this project is maintained by Purdue University and supported by the National Science Foundation [BML+12].

In general, Cetus provides features to implement parallelization techniques focusing on parallelism extraction though OpenMP directives. Cetus users may perform analysis and transformations of data dependency, variable recognition/substitution, reduction recognition/transformation, scalar/array privatization, and loop parallelization. Moreover, Cetus' Internal Representation (IR) allows users to work with high-level general passes such as symbolic analysis, alias analysis, and function inlining. However, Cetus does not provide other general purpose tools for restructuring transformations. Also, unlike traditional compilers, it provides an IR instead of AST, and users have limited access to IR facilities to add new representations.

Figure 2.3 illustrates an overview of Cetus, presenting its architecture and IR. In Figure 2.3(a), we identify how the infrastructure is designed in a stack of layers. In fact, they use ANTRL to provide a symbol table to Cetus so that its IR is created from standard C sources (see Figure 2.3(b)). There is also a API layer available to the users to perform different kinds of activities such as parsing the IR, transformations, analysis, and optimizations. The driver layer is the module where the user has to implement source-to-source transformations, which is a Java class where the programmer implements its transformations for Cetus to compile it with other internal implementations.



(a) Cetus architecture. (b) Cetus IR of a program.

Figure 2.3: Cetus overview. Extracted from [JLF+05].

An example of Cetus' IR is given in Figure 2.3(b). Their tree abstracts several standard C grammar constructions. There is a *translation unit* for each program to represent the content of a source file and *procedures* which represent a individual functions. The procedures include a list of simple or compound statements, while the *expressions* represent operations over variables and variable assignments [JLF+05]. Also, data types are divided into basic types, extender, and modifiers. Therefore, providing this high-level abstraction, the developer may have less flexibility and have to learn a new way to represent/parse the standard grammar.

To modify the program, Cetus has an annotation system (implemented through a class) to simplify the insertion of comments, pragmas, and raw text as well as low-level code insertion of new constructors in their IR tree. In fact, we can observe that Cetus was primarily designed for simple source-to-source transformation, such as the insertion of OpenMP directives based on compile-time analysis to automatically exploit parallelism in multi-core systems. In the past, some experimental tests have been performed targeting code generation from OpenMP to MPI code [BE05, BME07], but this feature seems to no longer be supported in the latest versions. Recently, Cetus has also been used to exploit parallelism targeting GPU architectures by inserting annotations [SLJ+14].

2.2.2 PIPS

The research activities related to the PIPS (Parallelization Infrastructure for Parallel Systems) compiler infrastructure started in 1988 at MINES Paris Tech. Initially, PIPS provided automatic parallelization of Fortran code and since 1991 it began supporting C language [IJT91]. Its key features are inter-procedural analysis and

abstract interpretation on polyhedral lattices. The front-end includes Flex/Bison tools to parse C and Fortran codes and implement the intermediate code representation (IR). Moreover, PIPS only supports string transformations. Thus, the user has to: (1) parse the IR, (2) find the regions in the IR to perform a transformation, (3) pretty print code in a separate file, and (4) gather all the files to compile the code (assembling source codes into machine code) [AAC+11].

Figure 2.4 illustrates the PIPS infrastructure that is represented as a stack of layers. On top, there are compiler tools that derive from the PIPS infrastructure. They are user end tools to make automatic parallelization for vectorized instructions and code optimizations. For transformations, the pass manager layer provides a set of APIs useful to interact with the internal representation, which is a custom AST from the source code [Gue11]. Therefore, we can highlight that PIPS is a lower level compiler source-to-source transformation when compared with Cetus. Although PIPS claims to be used as a source-to-source compiler, it is also possible to generate machine code. Moreover, most source-to-source compilers implement automatic code parallelization using empirical methods, while PIPS uses a polyhedric approach [AI91]. This method consists of five compilation phases: (1) static checker; (2) building an array DataFlow Graph (DFG), which is a kind of dependency graph; (3) creating a scheduling function based on DFG; (4) computing a map function that places the code identifying physical parallel machine resources; and (5) generating the parallel code.

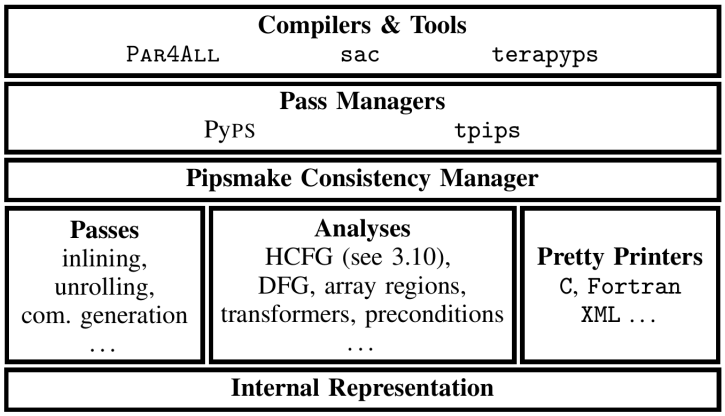


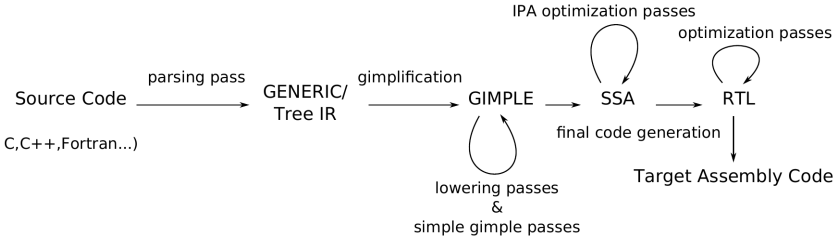
Figure 2.4: PIPS infrastructure. Extracted from [AAC+11].

PIPS's research perspective aims to become an extensible workbench for analysis and source-to-source transformations. Inter-procedural and polyhedral analysis are mathematical approaches that allow the compiler to perform sophisticated analysis to identify array privatization, control flow, dependencies, reduction detection, among other techniques [AAC+11]. Transformations (*e.g.*, loop reductions, loop distribution, coarse grain parallelization, and many others) and reconstructions (*e.g.*, dead code elimination, declaration cleaning, and many others) can eventually output Fortran/C code annotated with OpenMP directives or MPI calls. In fact, PIPS generates MPI

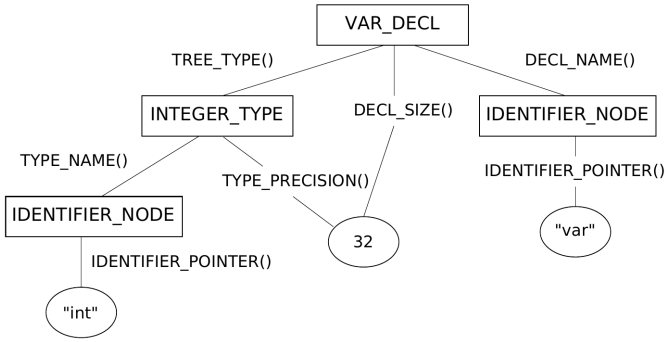
coarse grain parallelism from OpenMP annotated code [MMPSC08]. Nonetheless, recent PIPS efforts have been targeting heterogeneous machines mainly using the inter-procedural technique to perform source-to-source transformations [Ami12].

2.2.3 GCC-Plugins

The GNU GCC compiler [GCC16a] is the default compiler for many open source systems and is widely used by C/C++ communities [vH06]. Since the GCC 4.5 version, there is a new feature called GCC-Plugins [GCC16b]. This allows developers to implement research experiments and code analysis tools on the compiler without changing the core of the compiler source code. Thus, programmers may extend automatic optimizations and languages based on C pragmas and C/C++ attributes, which are registered/executed in compile time when loading with the input code.



(a) GCC passes.



(b) GCC generic tree example.

Figure 2.5: GCC internals overview. Extracted from [Ló14].

To better understand this technique, Figure 2.5 presents a general view of the GCC internal representation, including the main passes (2.5(a)) and an example of the AST (2.5(b)). As can be noted in Figure 2.5(a), GCC performs several modifications (organized in passes) when compiling a program. When creating a new plugin it is necessary to specify the passes that GCC will give access to the AST. At the front-end, the parser is able to recognize different language grammars and build a generic tree (known as AST) for each. During this process, many passes occur that are specific

to the language to prepare the AST for the "gimpler". Most of the passes that GCC allows to access the plugin accessing are related to the GIMPLE (machine-independent intermediate representation). Consequently, the only code generation is for `gimple` intermediate language or SSA (Static Single Assignment) format.

In GCC-plugins, the programmer may parse the AST of a given C/C++ code. Figure 2.5(b) illustrates an example where there is a tree for a variable declaration (`int var`). This is a generic representation of the source code that is preserved in different passes to be optimized until generating the RTL (Register Transfer Language) code. GCC represents the code in a different way than the standard grammar. Other drawbacks are poor documentation and constant changes for every new stable released version, which will require a new version of the developed plugin. Although it is possible to parse the AST, there is no support for performing modifications and attach new nodes at the plugin level.

2.2.4 Clang

Clang is under the umbrella of the LLVM (Low-Level Virtual Machine) project. LLVM is one of the most modern compiler infrastructures and it has been used by several researchers to prototype new ideas in compiler design. Also, it has been successfully used in industry by several companies, including Apple Inc. and Intel to implement C-based language and commercial compilers [LA14]. Clang is the LLVM's front-end compiler that has been proven to be more efficient than the GNU GCC compiler when compiling sequential programs.

Figure 2.6 illustrates the workflow of LLVM. Note that LLVM only interprets intermediate representation language (LLVM IR) as a sort of GIMPLE of GNU GCC. As such, LLVM is implemented as a virtual machine to support source-to-binary code generation in other programming languages, while GCC is not modular. Moreover, its infrastructure provides a set of tools to perform optimizations in the passes and handle IR's back and forth, from memory to disk [LA14].

At the front-end, Clang is constantly being improved to support users to create standalone C++ tools by using its **LibTooling** library [Cla16]. Among the alternatives is the static analyzer, which is a set of bug checks aimed to provide accurate warning messages to find errors during software development. Another particularly interesting characteristic of Clang (with respect to our research), is the possibility to create source-to-source transformation and refactoring tools, which are loaded at the time of compilation. This library facilitates implementation and AST navigation. However, its AST represents the code in a more abstract way than the standard grammar describes and it is not possible to perform code transformation directly on the AST.

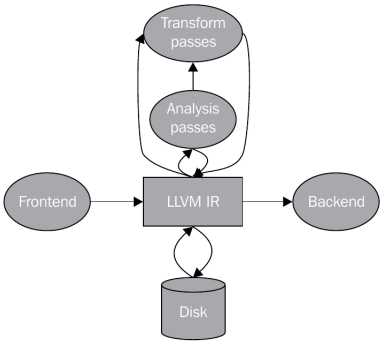


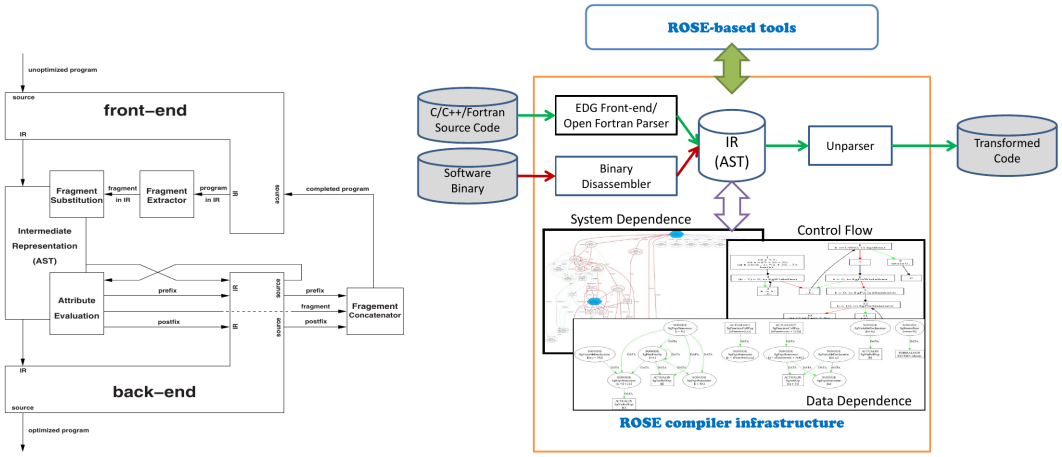
Figure 2.6: LLVM infrastructure. Extracted from [LA14].

2.2.5 ROSE

ROSE [ROS16] integrates a set of tools to simplify research on compiler design, code analysis, optimizations, and source-to-source transformations. Since 2003, [SQ03] when its architecture was proposed, many tools have been integrated into its core framework. Figure 2.7 provides an overview of the architecture's ideas and the current infrastructure. The architecture is classified at the front-end, middle-end, and back-end. The front-end was initially built using Flex and Bison tools and now integrates another tool called EDG[i] to parse C/C++ code [QSYS04]. At the middle-end, ROSE creates a high-level and generic IR to represent the source code with little specific language syntax. This AST also stores attributes and annotations that may be used to perform advanced analysis and optimizations. All transformations are made directly in the IR, like the AST level to transform either into low-level IR LLVM code or into original C/C++ code. Then, a vendor compiler (GCC or Clang) is reused to generate the machine code.

In contrast to traditional compiler infrastructures, ROSE allows one to disassemble binary code and represent it in their generic AST. In fact, it can perform an analysis of a system that was previously compiled by another compiler. However, to perform AST transformation for code optimizations and parallelization of vectorized code, the AST needs to be from a source code. A fragmentation technique is used to successfully perform transformations, which is explained in detail in [SQ03]. ROSE seems more suitable for supporting internal compiler research for information extraction (*e.g.*, data flow analysis) from source and binary code as well as easy loop parallelization by inserting OpenMP directives into the source code. Moreover, for building new language extensions, ROSE has proven to be quite hard because programmers need to learn its customized IR as well as a set of non-standard tools.

[i] https://www.edg.com/



(a) ROSE architecture. (b) ROSE infrastructure.

Figure 2.7: ROSE overview. Extracted from [SQ03, ROS16].

2.2.6 Comparison

This section compares the tools previously discussed with our proposed support tool named as CINCLE (Compiler Infrastructure for New C/C++ Language Extensions). In order to highlight the differences and similarities, only the most important features regarding the DSL design space are detailed in Table 2.1. Those are:

- **C++11 Attr. Integrated**: This characteristic reveals if the infrastructure supports the implementation of the C++11 attribute mechanism.
- **AST Transformations**: This feature reveals if the infrastructure supports AST to AST transformations.
- **C++ Support**: Demonstrates if the infrastructure is able to parse C++ programs.
- **Documentation**: Determines the amount of documentation available to learn about the internal representation as well as language design.
- **Source-to-Source**: Reveals if the infrastructure can support source-to-source code transformations.
- **Recover Original File**: Indicates if the tool is able to produce the original source code again from its internal representation .
- **Purpose**: Describes the primary goal of the tools.

Tools	C++11 Attr. Integrated	AST Transformations	C++ Support	Documentation	Source-to-Source	Recover Original File	Purpose
Cetus	No	No	No	Poor	Yes	Yes	Compiler infrastructure for automatic parallelization and analysis
PIPS	No	No	No	Poor	Yes	Yes	Compiler framework and support tools for automatic parallelization and analysis
GCC-Plugins	No	No	No	Poor	No	No	GCC-Plugins for implementing optimizations and extending custom pragmas and attributes
Clang	Yes	No	Yes	Good	Yes	Yes	Compiler Front-End for LLVM and support tools for source-to-source transformations and analysis
ROSE	No	Yes	Yes	Poor	Yes	Yes	Compiler framework and support tools for automatic parallelization and analysis
CINCLE	**Yes**	**Yes**	**Yes**	**Ongoing**	**Yes**	**Yes**	**Compiler infrastructure for generating C/C++ language extensions**

Table 2.1: Related works for C/C++ DSL design space.

In addition to the previous features, it is important to highlight that all tools face the same problem: their internal code representation does not follow the standardization for C/C++ sources. This is a drawback in learning and generality of the source-to-source transformation algorithms. Most tools require users to learn the IR and compiler passes. Our perspective of high-level DSLs relies on simplicity and standardization of the source code representation. We believe that a generic standardized IR should be provided. In CINCLE we proposed and implemented an AST that follows the C and C++ standard grammar. Consequently, our compiler algorithms aim to be generalized (based on the standard) as we expect the growth of similar tools like/derived from CINCLE.

In Table 2.1, we can observe that none of the tools fulfill all the characteristics implemented in CINCLE to support DSL developers in high-level parallelism design. Unfortunately, the tools lack support for C++11 mechanisms and AST transformations. While Clang may recognize custom attributes, ROSE provides a set of libraries to navigate and transform their IR. In Clang, the user can modify a code by parsing the AST and pretty printing the code into a new file. This technique is also present in the other tools, but they do not support C++ (Cetus and PIPS). Although GCC-Plugins can be loaded during C++ program compilation, there is no support. When the respective passes are accessed now, the internal GCC AST representation loses C++ grammar precision. Moreover, the C++ attributes are translated to GNU C attributes.

Another drawback of GCC-Plugins is that users have to generate GIMPLE code instead of C++ code again. This requires learning a new language which is lower level code. On the other hand, all others are able to produce the source code from their AST again. However, this does make them suitable for our purpose, since we already highlighted the learning curve drawback and in the table. Moreover, their aims are different. Most of them are designed for static analysis, code optimization, and automatic parallelization by inserting OpenMP directives. Consequently, they contribute with compile-time techniques, methods, and algorithms to provide executable code that is faster and more effective. In contrast, our purpose is to focus on higher abstraction level concerns such as language extension design, high-level parallelism ideas for productive programming, and aggressive source-to-source code transformation.

2.3 Parallel Programming Frameworks

In this section, we intend to provide an overview of the parallel programming frameworks that were built on top of message passing and shared memory programming models. We split them into three groups (stream-based, annotation-based, and

general-purpose frameworks) to differentiate their primary characteristics. A detailed description and discussion will be presented in the last section through a comparison. These are the most important for this dissertation, because the focus is to provide high-level abstraction for parallel programming.

2.3.1 Stream-Based

This section will only discuss stream-based parallel programming solutions that are related to our approach.

2.3.1.1 FastFlow

FastFlow is an emergent parallel programming framework created in 2009 by researchers at the University of Pisa and University of Turin in Italy [ADM+09, AMT10, Fas16]. It provides stream parallel abstractions from an algorithmic skeleton perspective. The implementation is built on top of efficient fine grain lock-free communication queues [ADK+11, ADK+12]. During the last three years new features were integrated for high-level parallel programming for data parallel patterns (parallel "for", Macro DataFlow, stencil and pool evolution) [DT14, ADA+12, APD+15]. Also, other architectures have been supported such as clusters and hardware accelerators (GPU, FPGA and DSPs) [SUPT14].

The FastFlow programming interface provides a C++ template library whose classes can be viewed as a set of building blocks. Although built for general-purpose parallel programming, it provides suitable building blocks to exploit stream-oriented parallelism in streaming applications that other frameworks do not. For instance, it gives more freedom to the programmer to compose different parallel patterns and build complex communication topologies in shared memory systems. Also, the runtime support can operate in the blocking and non-blocking mode and enables the programmer to attach their customized task scheduler [Fas16, ADKT14, DT15].

The runtime support has been tested in different applications and has been shown to achieve a good trade-off between time-to-market, portability, efficiency, and performance portability in various platforms [TDM+14, ATD+13, BGP14, DDST14, Mis14, ADP+14, BDLT13, DK14]. Due to this, it has been quite a good C++ programming environment that provides stable runtimes for several research projects and researchers from different countries such as ParaPhrase, REPARA (both EU FP7), and RePhrase (EU H2020) [Fas16]. More details about FastFlow will be given

later (Section 6.4), since we have adopted it as our runtime for the proposed DSL for high-level stream parallelism.

2.3.1.2 StreamIt

StreamIt is a programming language for streaming applications [Str16]. It has been developed for more than ten years at the Massachusetts Institute of Technology (MIT). They have implemented different applications, targeting implicit parallelism in cluster and multi-core systems [TA10]. StreamIt also provides a compiler infrastructure to implement the programming language and to optimize code generation through the implementation of DataFlow analysis techniques [Gor10].

The programming language is unique and independent of the target architecture. The compiler can transform the code automatically for distinct hardware devices [ZLRA08, SGA+13]. StreamIt has been tested for many years and provided interesting insights in stream parallelism research as well as excellent performance in streaming applications [Thi09, Won12]. StreamIt is not considered a robust language, but it is much more productive for exposing parallelism and communication than traditional C/C++ libraries. It provides a straightforward and flexible structure that can be composed to create complex graphs without requiring significant modifications to the source code to change program behavior.

StreamIt applications are typically modeled as a composition of filter modules. Each filter function must implement an initializer and worker function. Inside the worker function, the filter can communicate through input and output channels, which are FIFO queues. Also, users are supported by pop and push routines to obtain and insert stream elements into the channels. StreamIt natively implements stream parallelism without underlying parallel patterns. However, these applications' structures are similar those in FastFlow. They mainly diverge only in name. For instance, StreamIt offers three filter interconnection modes: pipeline, splitjoin, and feedback loop [TKA02b, Thi09]. In contrast, FastFlow offers pipeline, farm, and feedback parallel patterns through C++ templates. To the best of our knowledge, activities on StreamIt stopped in 2013.

2.3.2 Annotation-Based

In this section we present the annotation-based APIs for parallel programming. OpenMP is considered the "*de-facto*" standard for parallel programming targeting shared memory systems. We also present other variants that have extended the

OpenMP runtime. In fact, all of these annotation-based environments use C pragmas as the main mechanism to express parallelism.

2.3.2.1 OpenMP

The first version of OpenMP was released by the OpenMP Architecture Review Board (ARB) in 1997 as a Fortran API. In 1998, they released the open specification for C and C++ languages. Since 2000, all compilers started to integrate these specifications. The first proposal was aimed to provide a new way to denote and express loop parallelism. Later, they also started to introduce task parallelism, which officially happened with the OpenMP 3.0 version (2007)[CJvdP07]. Recently, OpenMP released its 4.5 version, which offers rich support for heterogeneous systems like GPU, FPGA, DSP, and SIMD constructions [Ope16]. Also, performance for data and task parallelism exploitation has been improved. The newest feature is the possibility for specific task dependencies, which may be used as "a kind of DataFlow" specification. Complete documentation and an example of the programming interface is available at [Ope16].

Although OpenMP has been well accepted in the high-performance community, it lacks suitable directives to naturally support stream parallelism exploitation. Consequently, the programmer is obliged to deal with low-level implementation details related to thread synchronization. In contrast, our work solves this problem by using a different approach and annotation mechanism. The next section will present the research work that has proposed to extend OpenMP directives to target different issues. Our goal is to highlight what has been integrated in the OpenMP runtime, as well as the challenges and limitations.

2.3.2.2 OpenMP Extensions: OpenStream and ompSs

In the scientific community, there are also researches making progress and experiments to extend the standard features of OpenMP. OpenStream is one of these extensions designed to better exploit DataFlow parallelism. Authors [PC13] add other directives in a GCC-based compiler targeting dynamically streaming programs that can exploit pipeline, task, and data parallelism. They support these features along with pragma annotations through a runtime library which provides two distinct mechanisms: a pure data-flow and FIFO queue behaviors. Among their experiments, they have shown significant performance improvement compared to other OpenMP extension (ompSs). However, the limitations are from an older compiler version prototype and the lack of support for C++ sources. In addition to the fact that only a limited set of small

benchmarks were tested, there has been no software update or new experiments since 2012 [Pop12].

On the other hand, the ompSs studies are more active, releasing new versions each year [Omp16]. Most of the proposals are integrated with the standard OpenMP. For example, the new feature in task parallelism including dependencies was first proposed by ompSs. They have their own GCC-based compiler infrastructure to perform code transformations and prototype pragma primitives. The group associated with ompSs has developed different tools based on DataFlow parallelism for many years in the Barcelona Supercomputer Center (BSC). Also, they have successfully translated task (only) directives into MPI targeting cluster and GPU-Cluster [BMD+11, BPD+12] environments as well as supported data and task parallelism for hardware accelerators [PBAL13]. However, implicit calls regarding the programming model still has to be given by the user in the code.

2.3.3 General-Purpose Frameworks

This section will present other general-purpose frameworks that target different programming models and approaches to parallel programming. We provide a brief discussion to highlight the most salient points.

2.3.3.1 Cilk

Cilk is the result of research developed at MIT beginning in 1994 [BJK+95]. It was initially released as a C language extension with the possibility to spawn a function call/execution as well as a sync with executing thread terminations. Cilk is one of the first parallel programming runtimes based on work stealing scheduling [Lei09]. Cilk is known for its simplicity in extending C with only three keywords (`cilk_spawn`, `cilk_sync` and `cilk_for`), which allows one to exploit recursive parallelism. In addition, to avoid race conditions caused by global variables, it has implemented the reducer hyperobjects method in a lock-free manner [FHLLB09]. Later, Intel Inc. bought it and made it an open source project. Today, it is already integrated in the standard grammar of C and C++ as well as in their compilers [Cil16].

During the past few years, Cilk has been tested and proven to be efficient in a variety of parallel applications for multi-core systems. It has improved the support by including features to detect race conditions and analyze program scalability. With respect to the language, there is a new support tool to give the compiler permission for vectorizing loops. Moreover, new experimental research is being conducted to support

users when implementing pipe-while loops [LLS+13]. However, its interface has not been updated since the publication appeared. In fact, there is a raw environment using macros to benefit from the compiler preprocessor. It claims that the runtime is for the parallel pipeline construction of the TBB library (see Section 2.3.3.2).

2.3.3.2 TBB

TBB (Threading Building Blocks) is another Intel tool for parallel programming [TBB16]. TBB is a library for implementing high-performance applications in standard C++ without requiring a special compiler for shared memory systems. It emphasizes scalable and data parallel programming. The benefit is to completely abstract the concept of threads by using tasks. TBB builds on C++ templates to offer common parallel patterns (map, scan, parallel_for, among others), equipped with a work stealing scheduler similar to that in Cilk, which dynamically dequeues a stack of tasks implemented in a FIFO-like order [Rei07].

As previously mentioned, TBB and FastFlow are quite similar in many aspects, but the runtime and programming interface approaches are different regarding the design patterns and algorithmic skeleton. In fact, the pipeline pattern is supported in both of them which allows TBB to support stream parallelism exploitation. However, it has presented some limitations when there is a complex scenario where one needs to compose new patterns (*e.g.*, introducing back communications by using feedback patterns) [RCJ11]. Although its scheduler has been proven to achieve good performance in several applications, TBB's runtime does not allow one to attach a customized scheduler. Another drawback is that it only targets multi-core systems.

2.3.4 Comparison

We could have mentioned many other parallel programming solutions, but they do not provide significant contributions to our discussion. For instance, Charm++ [Cha16] is an extension of the C++ language that implements parallelism on top of a message passing programming model. Unlike other frameworks, the language is based on object migration and programmers interact through asynchronous object invocations. Another example is X10 [X1016], it is a completely new language based on Java on top of an APGAS (Asynchronous Partitioned Global Address Space) programming model. It offers a set of parallel constructions to deal with a single memory space. These are examples of frameworks that follow completely different and unrelated approaches to increase productivity in high-performance computing that are loosely related to our work.

In this section, we aim to compare important characteristics of our DSL with the state-of-the-art frameworks. Therefore, we will discuss the following topics in detail:

- **Programming Interface**: It reveals the mechanism used to implement and provide a programming interface for the users.
- **Stream Parallelism**: This item reveals if the framework supports stream parallelism exploitation.
- **Code Portability**: This is when the programmer needs to recompile the program to target other architectures.
- **Support C++**: This is about to support of the standard C++ language.
- **Target Architecture**: Describes the target architectures considered.
- **Programming Model**: Describes if the programming model's shared memory and message passing is abstract, explicit, or implicit to the users.
- **Purpose**: Describes the design goal of the framework.

Before comparing the frameworks, it is important to highlight that MPI and OpenMP are the most well-known and widely used frameworks in parallel programming. MPI provides a low-level network message communication library in cluster environments. On the other hand, OpenMP provides higher level abstractions through annotation. The compiler generates multi-threaded code for either multi-core or accelerators hardware. In our case, because OpenMP is not suitable for the stream parallelism exploitation, the FastFlow runtime library provides the appropriate functionality, flexibility, efficiency, and facilities. This is similar to that offered by MPI in distributed memory architectures.

In Table 2.2, we can observe that none of the frameworks fulfill all of the characteristics implemented in SPar to provide high-level stream parallelism. Only OpenMP and its extensions are annotation-based. Also, only TBB, FastFlow, and StreamIt are able to support stream parallelism. Although it is called OpenStream, it is designed to improve DataFlow parallelism in task regions. In OpenMP 4.5, this is already supported.

Another important aspect in the table is code portability. Only StreamIt and Spar can provide such features simply through recompilation. However, StreamIt is a completely new language while SPar makes it possible to work with standard C++ sources. The drawback of using StreamIt is that C++ programmers need to learn a new language and re-implement the source code.

We have already emphasized the importance of being compliant with standard C++, where C pragma annotations are actually preprocessing directives. In fact, C++ compilers now support OpenMP directives, but new ones will require a dedicated infrastructure similar to what researches from ompSs and OpenStream have used.

Tools	Programming Interface	Stream Parallelism	Code Portability	Standard C++	Target Architecture	Programming Model	Purpose
FastFlow	C++ Templates	Yes	No	Yes	Multi-Core, Accelerators and Clusters	Explicit	Algorithmic skeleton library for general-purpose and efficient parallelism exploitation
StreamIt	New Language	Yes	Yes	No	Multi-Core and Cluster	Abstract	Efficient stream Parallelism exploitation
OpenMP	C Pragma	No	No	Yes	Multi-Core and Accelerators	Explicit	Efficient data and task parallelism exploitation
OpenStream	C Pragma	No	No	No	Multi-Core	Explicit	OpenMP extension to efficient DataFlow parallelism exploitation using FIFO queues
ompSS	C Pragma	No	No	Yes	Multi-Core, Accelerators and Cluster	Explicit	OpenMP-based research to collaborate with the standard as well as with DataFlow parallelism
Cilk	C/C++ language extension	No	No	Yes	Multi-Core	Explicit	Efficient and general-purpose parallelism exploitation
TBB	C++ Templates	Yes	No	Yes	Multi-Core	Explicit	Design patterns library for efficient and general-purpose parallelism exploitation
SPar	**C++11 Attributes**	**Yes**	**Yes**	**Yes**	**Multi-Core and Clusters**	**Abstract**	**Annotation-based and stream-oriented targeting code portability and productivity**

Table 2.2: Related parallel programming frameworks.

Because C++ attributes are part of the C++ grammar, the compiler is able to recognize custom annotations, which makes this mechanism more suitable to create new embedded DSLs.

Turning to programming model features, we can observe that in most of the frameworks, the user must explicitly parallelize the application code taking into account the target architecture. Again, only StreamIt and Spar are able to abstract while taking advantage of the parallelism in different parallel architectures, mainly in multi-core and clusters that require very different programming models. We can also note that other frameworks are still strongly dependent on hardware to achieve high-performance code.

All of these contrasts may be justified by the distinct purpose of our work, which is the design goals of each solution. We can see that the state-of-the-art frameworks are still aiming to extract the maximum performance of the parallel architecture while our work is focusing on raising the abstraction level to provide code portability and productivity. Thus, the current frameworks are seen as potential runtimes for our purpose.

2.4 Concluding Remarks

In this chapter, we provided a discussion on the state-of-the-art research for high-level parallelism, C/C++ DSL design space, and parallel programming frameworks. As we observed, our research is well positioned in the literature. We are adding interesting contributions to the current high-level parallelism research, and this may open new perspectives to achieve better code portability and productivity, mainly in streaming applications. We also are starting with a higher level approach to design DSLs in C++ as well as to perform source-to-source code transformations. Moreover, our solution to support stream parallelism tries to naturally integrate it into the standard language to not be dependent on the programming model or architecture.

Part II

Contributions

3

Overview of the Contributions

This chapter presents the main thesis contributions in a nutshell.

Contents

3.1 Introduction

This chapter is included before the technical section to introduce the scope and informally explain our contributions. The simplest description of this work is that it is about supporting the development of many streaming applications such as image, video, audio, and networking simpler and at the same time taking advantage of different parallel architectures without worrying about their complexities to achieve performance.

Two tools have also been created. CINCLE was designed to help create abstractions to make it easier to develop an application in other domains. Using CINCLE, we successfully created our second tool that is a programming interface (named as SPar) for streaming applications. Therefore, this chapter presents the programming framework that defines our scope and the subsequent sections will informally present the contributions.

3.2 The Programming Framework

The programming framework provides the main contributions of this dissertation. Unlike what have been proposed in the literature, we aim to provide a framework that is fully compiler-based and domain-oriented to support simple, high-level, productive, portable, and modular programming interfaces for C/C++ applications. The challenge is to enable high-performance code through high-level abstractions, as previously discussed in the Preface and Introduction. This proposal builds on the background acquired over the last four years of intensive studies in high-level parallel programming and DSLs. Our idea is to shorten the path providing compiler-based abstractions in different domain levels, where other researchers can benefit from what we have learned and eventually included in this framework.

Figure 3.1 illustrates how the programming framework is designed. Our research work can be divided into five domain-specific research fields that are specified on the left side of the figure. The right side of the figure divides the research fields in two groups to separate our target challenge and highlight the scope. Our original contributions aim to empower the Domain-Specific Language Design (DSLD) group. Moreover, the stack of blocks represents specific elements that are created and reused for a given application domain. The picture also organizes the elements in the stack from the lowest (bottom) to the highest (top) level. Starting from the low-level, the

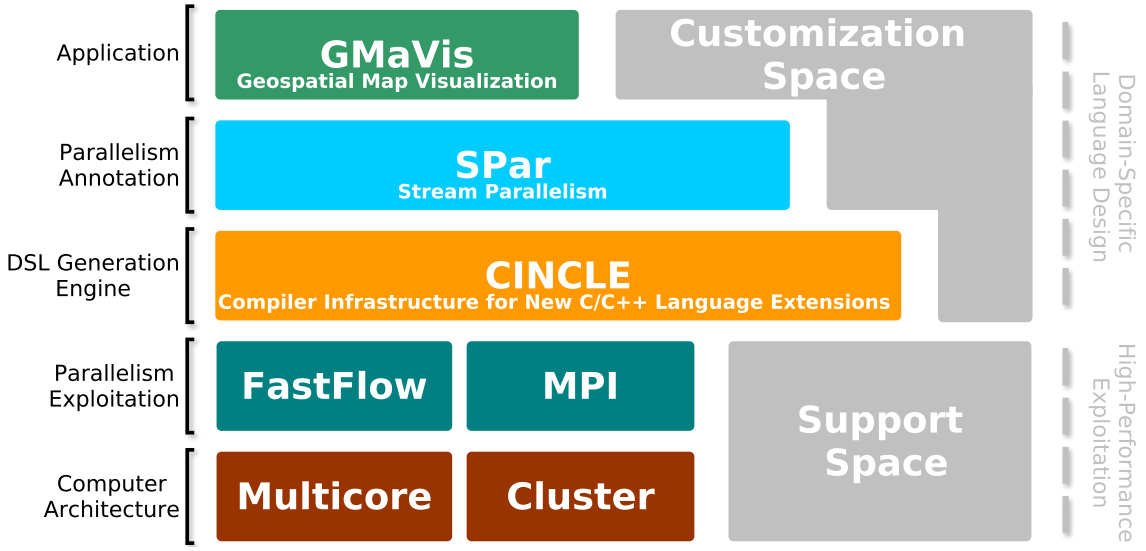


Figure 3.1: The programming framework picture.

following topics will provide more details regarding the concepts and goals of each one of the fields:

- *Computer Architecture:* Is the domain of the different parallel architectures available in different devices. Currently, our targets are multi-cores (server and workstation machines) and clusters (an agglomeration of machines connected through a network). In the future, we plan to expand the support space to include different types of architectures such as heterogeneous and hybrid. Note that we do not intend to create new architectures, but instead to make the software benefit from resources and capabilities of these different architectures.

- *Parallelism Exploitation:* Is the domain of current parallel programming frameworks. Some of them provide higher level abstractions to exploit parallelism (this is the case of FastFlow) and others are low-level programming models (this is the case of MPI). They are suitable runtime and performance accelerators for parallel computing. The idea is to reuse these tools to allow researchers to continuously focus on better programming interfaces and performance exploitation. Firstly, we aim to support code transformation targeting FastFlow on top of multi-core and MPI on top of clusters.

- *DSL Generation Engine:* Is the domain supporting the generation of new compiler-based C/C++ embedded DSLs. In this dissertation we proposed the CINCLE infrastructure. It is the layer where we start to differentiate our programming framework from the literature. Our goal is to generate a high-level parallelism DSL (SPar) as well as an application level DSL (an example is GMaVis [Led16]) that completely abstracts parallelism aspects by instantiating SPar attributes. We contributed by designing CINCLE as our engine to support DSL designers with simple and powerful mechanisms at the host language level.

Moreover, CINCLE was modularly structured so that different consolidated state-of-the-art tools can be easily integrated to support more complex and generalized code transformations.

- *Parallelism Annotation:* Is the domain providing the user with high-level parallelism abstractions. Current compilers are still not able to automatically parallelize instructions (that not vectorized) without any user intervention in C++ programs. This is primarily due to the fact that at the compilation time it is not possible to know whether a library call or a piece of code can run in parallel. Our proposal is to make it easier for compiler and application developers, using annotations (C++ attributes) to provide an equilibrium between abstraction for the compiler and user. Therefore, while the annotation properties can enable efficient parallel code transformations, the annotation language seeks to support the user with code productivity and portability. As a consequence, we initially make a contribution in this layer through the C++ embedded DSL implementation for stream parallelism (SPar). This is an example of how we simply annotate parallelism rather than exploiting it, where the responsibility for exploitation is withf the DSL compiler targeting the multi-core and cluster architectures.

- *Application:* Is the domain that intends to improve abstractions for user applications. We envision a description language friendly to the domain and designated for a particular purpose. As the parallelism annotation layer is more general purpose supporting stream parallelism, we expect that an application's DSL compiler will generate robust C++ codes along with SPar annotations to enable high-performance. Thus, an example and a vertical validation of our perspective were created through GMaVis DSL [Led16]. Our initial goal was to support users in geospatial data visualizations while taking advantage of the multi-core architectures for fast processing and visualization of information.

With respect to Figure 3.1, this dissertation will make direct contributions to the DSL generation engine and parallelism annotation domains. However, indirect contributions will also be made by the use case in the application domain, where GMaVis instantiates SPar to annotate the generated C++ code to take advantage of multi-core parallelism. It is expected that new contributions similar to GMaVis can be developed by improving current developed solutions (SPar and CINCLE) as well as expanding the support and customization space. The following sections will informally describe our desired contributions.

3.3 A Compiler-Based Infrastructure

First of all, "Why are you providing a new compiler-based infrastructure in C++?" Simply put, compiler-based tools are not new in the literature. The problem is that they lack simplicity and do not support the implementation of aggressive source-to-source transformations.

"And what do you mean by simplicity?" To make it easy to develop new DSLs that require compiler-based techniques to provide a high-level and productive interface. The current solutions do not offer a simple infrastructure to enable rapid prototyping and require a significant learning curve to understand internal compiler implementation. Another difficulty is that parsing the code is not easy because internal ASTs usually are not standardized.

"What do you mean by not supporting aggressive source-to-source transformation?" Not being able to use state-of-the-art tools to perform transformation directly on the AST. This feature is crucial when you are designing a tool that aims to perform sophisticated source-to-source transformations. For example, when using a string based technique (such as in Clang), you may have to re-parse your code several times to come to the final transformation, while in AST you may implement it directly in the tree.

"Why did you decide to build CINCLE?" This idea arose when we faced many difficulties trying to prototype the DSL. During the last five years, we have been intensively researching to provide high-level and productive parallelism. Our first DSL compiler was manually implemented because it was simpler and faster to prototype rather than using standard tools like Flex and Bison. The problem was that it lacked modularity, the ability to add new functionalities to perform sophisticated analysis and code generations. Later, we started to look for alternatives to integrate a C++ DSL directly into the GCC compiler. We found the same problems when we discovered GCC plugins. Also, there was poor documentation, no support for aggressive code transformations, no standard AST syntax, and other issues. We then tried to use Clang, but the only advantage with respect to the GCC plugin was better documentation and support to integrate DSLs. Consequently, we decided to design CINCLE to create a simpler environment for creating C++ DSLs, provide more modularity, and support AST to AST transformations.

"How do you describe CINCLE?" We see CINCLE as its abbreviation states: A Compiler Infrastructure for New C/C++ Language Extensions. Of course, we cannot compare it with the state-of-the-art compiler tools because it does not go into machine code. Our goal is to focus on language design and source-to-source transformation. Thus, CINCLE can be understood as a set of tools that are suitable for building C++

embedded DSLs mainly because this dissertation is focused on enabling high-level and productive stream parallelism.

"How does CINCLE target modularity and other goals?" Figure 3.2 sketches the compiler-based infrastructure. We separated it into three domains: Front-End, Middle-End, and Back-End. Modularity isolates this compiler to let other people collaborate with the improvement of CINCLE and generate tools to support the implementation of new features. Also, the programmer does not need to deal with low-level compiler aspects relative to the design when building an embedded DSL. Therefore, we created a tool for source-to-source transformation so that the programmer can concentrate only on the aspects that correspond to their domain. For instance, Front-End generates a sophisticated parser to build an AST, while Middle-End and Back-End only deal with AST visiting and transforming.

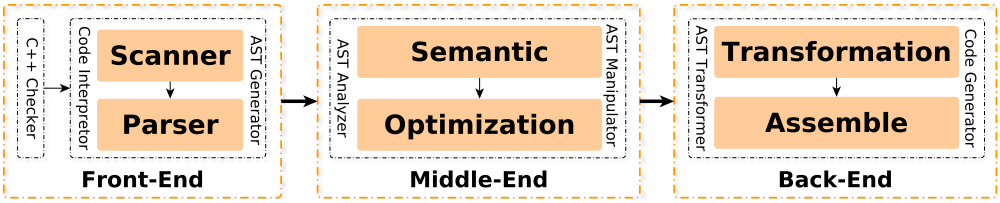


Figure 3.2: The CINCLE Infrastructure

"Is CINCLE ready for robust systems?" Not yet. We recommend CINCLE for research. It can be used to validate the algorithms for source-to-source transformation as well as generating an experimental DSL compiler. Unfortunately, we do not have a team/group working on the code. We still need to test and make complex validations on the Front-End part that implements the latest standard C and C++ grammars. Currently, CINCLE has been demonstrated to be sufficient to build a DSL for stream parallelism.

3.4 High-Level and Productive Stream Parallelism

"What is high-level and productive stream parallelism?" High-level is a general term and has been used for different levels of abstractions. In this dissertation, high-level refers to parallelism abstractions that are only related to domain terms and prevent users from being aware of the parallel architecture details. It allows users to avoid rewriting the source code of the application and reduces the programming effort required to support parallelism.

"What did we use to make it possible?" We believe that parallelism should be annotated in C++ programs rather than exploited or expressed as has been done during recent years. Pragma-based annotations have been well accepted in the

high-performance computing community through OpenMP, which is the "*de-facto*" standard parallel programming interface for exploring parallelism in shared memory systems. However, when we look at the level of language design, they are not in the standard grammar because they are preprocessing directives. In turn, they are strongly compiler-dependent when developing a DSL. In contrast, we adopted the C++11 attributes mechanism that is part of the language grammar and more familiar to the C++ community, because the "*de-facto*" C++ standard. Also, it provides us more freedom to be compliant with the language syntax since it can be customized, placed almost anywhere in C++ sentences, and associated with the AST. Finally, we concentrated on a particular domain to elevate abstraction and cover the lack of productivity in stream parallelism.

"What is lacking in the literature for stream parallelism?" Many things concerning high-level abstraction and coding productivity are missing. For instance, current parallel programming interfaces that support the parallelization of stream-oriented computations are still architecture-dependent and lack code productivity (`e.g.`, TBB and FastFlow). Programmers have to modify their source code to exploit parallelism. On the other hand, general-purpose programming interfaces like OpenMP are made for implementing low-level parallelism exploitation. Yet, they are only productive when there is an embarrassingly parallel computation.

"How do you address this?" We proposed a DSL that is called SPar for solving this problem in the stream domain. It seeks to keep maintain the original source code by only introducing annotations. The DSL also targets code portability through recompilation of the program.

3.5 Introducing Code Portability for Multi-Core and Clusters

"Why is code portability important?" It allows us to be more productive since you do not need to rethink or rewrite the SPar code that is running on a particular parallel architecture to port it to another one.

"What is the problem?" The problem is that state-of-the-art parallel programming tools are still too low-level and are designed to support high-performance code that is strongly architecture-dependent. Consequently, code must be rewritten and parallelism strategies have to be rethought in order to continue providing high-performance code in case the application needs more performance or vice versa.

"Does code portability make sense in our current scenario?" It does to us. It may sound a little bit strange as we are proposing this for multi-core and clusters, which sometimes are associated with dedicated supercomputer centers. However, there

are other parallel architectures and in the future will be more heterogeneous parallel architectures (more domain-specific) that will certainly demand studies to provide code portability with high-performance code. Also, it makes sense because many stream applications may need an elastic performance to address different workloads. If code portability is possible in a simple way, you may use it to save energy and money when switching between the cluster and multi-core environments.

"How do we intend to introduce parallel code portability?" Now we need to be realistic. We lack portability because it is a very complex goal. Therefore, we introduce the challenge through generalized transformation rules along with a stream-oriented and annotation-based interface. The decoupling "*de-facto*" implementing the portability is mainly the level of the patterns. Our contributions to the state-of-the-art are the transformation rules translating SPar annotations to parallel patterns. We have made them independent of the target architecture so that the same code can be compiled again to produce another code for another architecture.

4

CINCLE: A Compiler Infrastructure for New C/C++ Language Extensions

This chapter presents a compiler infrastructure for generating new C/C++ embedded DSLs. It is not a compiler, but a support tool that provides basic features and a simple interface to enable AST transformations, semantic analysis and source-to-source code generation. The main goal is to simplify the processes of creating high-level parallelism abstractions by using the standard C++11 attribute mechanism.

Contents

4.1 Introduction

C++ language extensions and DSL design and implementation are a challenge for a single person or a typical research group due to the amount of work and knowledge necessary to prototype compiler-based abstractions with the current alternatives. From our experience in the last five years as well as the opinion of other scientists (as presented in the related work), the implementation of DSLs in compiler-based tools is difficult, complicated and usually requires a significant learning curve, which is even more difficult for those who are not familiar with this area.

The motivation is therefore to simplify this path for other researchers (experts in their domain) to implement high-level and productive interfaces with powerful and aggressive source-to-source transformations. Our idea is that they can use their expertise without having to enter low-level code and still provide an abstraction to their domain, mainly for the parallel computing area. Despite the fact that the activities needed for efficient parallelism exploitation also require significant expertise and represent a notable challenge for people from other areas of computer science, our infrastructure is an initiative to support experts in parallel programming while providing high-level parallelism. Moreover, from the perspective of the programming framework, current parallel programming environments are still on the level of parallelism exploitation, which are good runtimes for high-level abstractions.

In order to move towards a mechanism to provide parallelism abstractions, the C++11 attributes already present in the standard grammar have been proven to be suitable for supporting powerful code transformations [ISO11b, ISO14]. Attribute recognition along with the other sentences in an AST (Abstract Syntax Tree) are one of the central motivations for their usage. Moreover, the syntax is also integrated into the language, unlike other annotation alternatives such as *C pragmas* that are rather than preprocessing directives. Thus, to benefit from *C pragma* features for implementing parallelism abstractions, the runtimes have been integrated directly into the compiler system. As a consequence, to create new features and conduct experiments, one has to fully understand the compiler internals.

For instance, the implementation of new *pragmas* in GCC is done by using GCC plugins. Although this approach abstracts many compiler complexities, it still requires users to go into compiler source code. The same difficulty arises when using GCC plugins for registering C++ attributes. In addition, C++ attributes are placed on GCC AST as C attributes and transformations on the AST are not allowed during the plugin call back. Another problem of GCC is that the provided AST is modified during the callback, which loses parts of the original semantics of the C and C++

grammar[i]. Such limitations make it even harder to benefit from C++ annotations to provide a higher level programming interface directly on the GCC, reinforcing the need for a tool such as CINCLE.

Although Clang, Cetus, and ROSE share common characteristics with CINCLE, they were built for different purposes and do not support particular features necessary for the programming framework developed within this thesis. In particular, we need AST to AST transformations, C++11 attributes in the AST as specified in the ISO standard as well as the possibility for generating new language extension to support embedded DSLs. This chapter will present our original contributions, implementation design goals, CINCLE infrastructure and simple algorithms as well as examples to start a new project. Lastly, we will present some performance results and use cases.

4.2 Original Contribution

One of our contributions with respect to the state-of-the-art compiler-based tools consists in a parser of C++ standard grammar (ISO/IEC 14882:2014 and 9899:2011) [ISO11b, ISO14]. Although in principle there is no scientific innovation when using standard tools like Bison and Flex, it nonetheless provides a simple interface for supporting the implementation of new embedded C++ DSLs.

The parser implementation was particularly designed to support us in the process of creating AST from actual C++ code. While building a full C++ parser is not a trivial task, the standard compliance and the simple and handy AST representation we produced, explicitly aimed at supporting AST transformations. Therefore, enabling source-to-source transformations eventually results in an useful and practical support for the development of different kind of language extensions, such as the one based on the standard attributes we designed in this thesis.

In contrast to the state-of-the-art C++ tools, our infrastructure is created to provide full access to AST. It gives more power to the user for transforming and visiting ASTs. The advantage is that there is no need to go inside the source code to make string transformations that can require re-parsing source code several times and a huge amount of programming. CINCLE also provides several useful methods that make it easier to many different activities such as: generating a new AST from a string; check AST transformations; tree operations (insert, delete and replace) and visualization.

[i] GCC makes such modifications more friendly with the intermediate language which is known as GIMPLE.

Finally, CINCLE can generate again C++ code from the AST. This contribution is fundamental because it proves to be effective for source-to-source translation.

4.3 Implementation Design Goals

There are various important non functional concerns when creating a new solution for the scientific community such as performance, efficiency and portability. For the first version of our compiler infrastructure, they are not considered first class requirements to simplify tests and validations. However, this does not mean that during CINCLE's implementation necessary attention was not given to these concerns or that some internal design choices cannot be improved in the future. The main design goals considered are the following (and they will clarify some of these aspects):

- *Modularity:* Is an important issue for continuing to research and develop more sophisticated capabilities and it is necessary to be compliant with domain-oriented concerns of the thesis framework. Therefore, CINCLE is divided in three modules: Front-End, Middle-End and Back-End. Also, each one has its own submodules (see Figure 3.1). For example, this structure allows domain experts on a low-level compiler (language recognition and parser algorithms) to work on the Front-End and scientist experts of fast prototype algorithms for source-to-source transformations can concentrate their efforts on the Back-End. On the other hand, the Middle-End is a conversation bridge between the front and Back-End modules as well as a research space for testing semantic and tree optimization techniques. Since the idea is to support new embedded DSLs, this modularity lets software designers concentrate only on the semantic analysis and transformations while the Front-End provides the necessary features such as full AST access needed to implement new C++ DSLs.

- *Extensibility:* CINCLE offers a basic infrastructure for creating new compiler-based tools. For example, in addition to extending DSLs, it enables the creation of pattern matching, code auto-tuning, code analysis, tracing, and other tools that can be a solution for C/C++ language. To achieve such extensibility, CINCLE provides full access to AST and all modules of the infrastructure. The environment must also be modular so that the extension of new capabilities on a given designed tool will not affect the performance and system operation on the rest of the system. Moreover, compliance with the standard language grammar prevents misunderstanding and supports contributions from other research to continue empowering and extending the tool's capabilities.

- *Standard Grammar Compliance:* Is an important aspect because it affects other design goals. It impacts simplicity because people may have to deal with

non-standard terms and concepts, which requires learning new terminology. It increases the system extensibility by covering a wider community that can contribute to improvements for the tool. One may have difficulties in the fast prototyping of some code transformation rules if the AST does not address the standard grammar. Similar to previous drawbacks, the lack of standard compliance makes it difficult for other researchers to reuse algorithms and parts of their software experiments. As a consequence, CINCLE's infrastructure is designed to preserve the representation of the standard grammar in the AST as much as possible and changes are only made when there are conflicts between C and C++ grammars.

- *Simplicity:* CINCLE aims to make the creation of annotation-based DSLs easier. Achieving other main design goals will provide a simpler programming environment for the infrastructure. However, CINCLE simplifies aims by supporting direct AST transformations, the storage of relevant code information on AST nodes, easy recursive top-down and bottom-up tree navigation, and a set of API functions for recurrent actions in the AST. Thus, the rapid prototyping design goal can benefit from simplicity because it reduces the amount of code needed.

- *Reusability:* Is more than providing reusable features such as API and the infrastructure code for generating new language extensions. The idea is to reuse other consolidated software to simplify CINCLE's implementation. For example, reusing the GCC compiler for performing source code syntax and semantic analysis. CINCLE avoids complex semantic and syntax analysis implementations. Moreover, consolidated libraries and tools are integrated to provide more suitable features, *e.g.*, AST visualization, which helps users to learn more about the AST.

- *Rapid Prototyping Support:* Is very difficult to achieve in other related tools. This was an important goal when we began initial and experimental research on language design and source-to-source code transformation. The central point is to be simple enough to test new algorithms before starting to actually implement the final solution. CINCLE's infrastructure intends to automatically integrate custom C++11 attributes, placing them along with other C++ sentences on the AST. Since one is creating a new DSL, they will be aware of the semantics of its annotations. Thus, for the fast prototyping of the transformation rules, one only needs to concentrate on implementing the transformation rules by parsing the AST, instead of doing other pre-implementations or adaptations which must be done when using other tools.

4.4 The CINCLE Infrastructure

CINCLE attempts to provide the most modular environment for creating new language extensions as possible. Figure 4.1 demonstrates how the infrastructure is organized, where the Front-End is split in two separate modules. The engine Front-End is where CINCLE implements language recognition and parsing, and creates a representation of the source on the AST. On the other hand, the Front-End interface provides a set of capabilities for the DSL creator to perform transformations on the AST and customize additional features.

Middle-end and Back-End are merged in a single group as they are provided to the DSL creators, supported through a set of template modules. Each one has a CINCLE internal routine that will be called during the compilation according to the sequence illustrated in Figure 3.2. The engine subsequently loads the interface Front-End, Middle-End and eventually the Back-End modules.

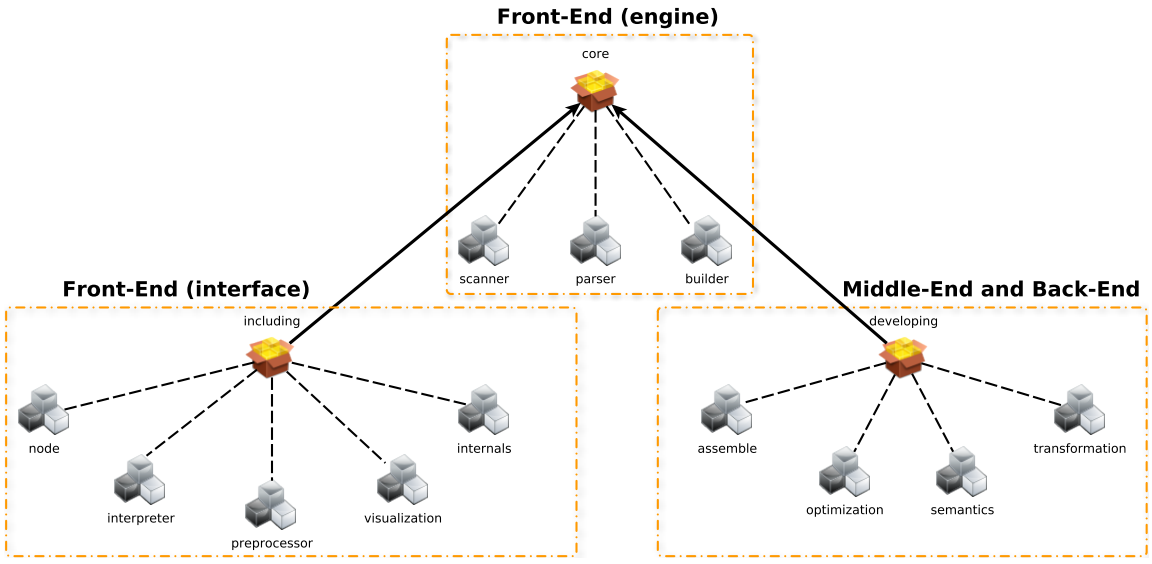


Figure 4.1: The environment of CINCLE infrastructure.

As it can be noted, CINCLE provides an entire infrastructure dedicated for creating embedded C++ annotation-based DSLs. The only parts that have to be implemented by the DSL creator are the modules inside the Middle-End and Back-End group. The following sections will describe how the system was designed, demonstrating how to deal with AST and code transformations through basic examples.

4.5 CINCLE Front-End

The frond-end engine was implemented using Flex and Bison tools [Lev09]. They were used to generate the parser and build the CINCLE AST. To deal with the grammar ambiguities of the C++ language, the Generalized Left Righ (GLR) algorithm was implemented by Bison. The tokens were recognized in the Flex runtime and integrated with Bison. CINCLE also stores preprocessing directives on the AST as a single token and comments are simply ignored.

On the other hand, the Front-End interface provides already implemented modules to deal with node representation, internals AST operations, code interpretation, preprocessing and AST visualization. These modules are instantiated by the engine modules and can also be used when developing Middle-End and Back-End modules. The interface simplifies AST manipulation and the implementation of source-to-source code transformation.

The AST plays an important role in source-to-source transformations. It is created by the parser at compile time, representing all tokens according to the standard grammar specifications [ISO11b, ISO14]. In CINCLE, a node is fulfilled with the information illustrated in Figure 4.2. Therefore, each node of the tree will have its type identified through a constant, which is also the name used in the standard grammar. When the token is a literal or identifier, its content will be stored on the AST node. Information about the token location are also stored, such as the coordinates of its position in the source code. Finally, there is a pointer to its father (**node_up**) and to a list of child nodes (**node_down**), also storing the number of children (**childs_n**).

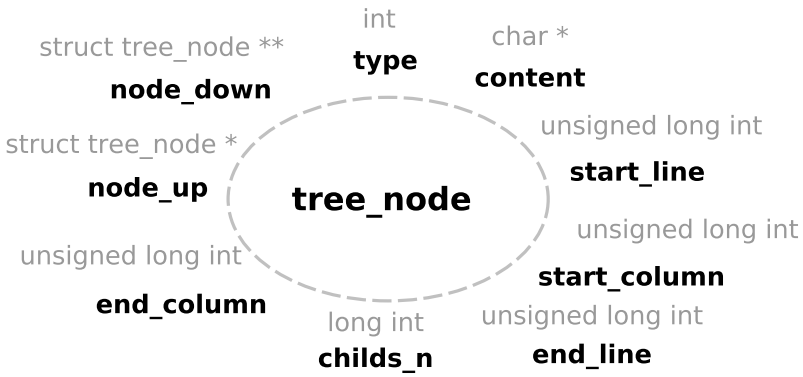


Figure 4.2: CINCLE AST node.

Figure 4.3 illustrates how tree nodes are connected on the AST. Each node can visit its father and child nodes through the dedicated pointer, simplifying the tree

navigation and access to information. Also, the tree is created from left to right and there is no limit to the number of child nodes.

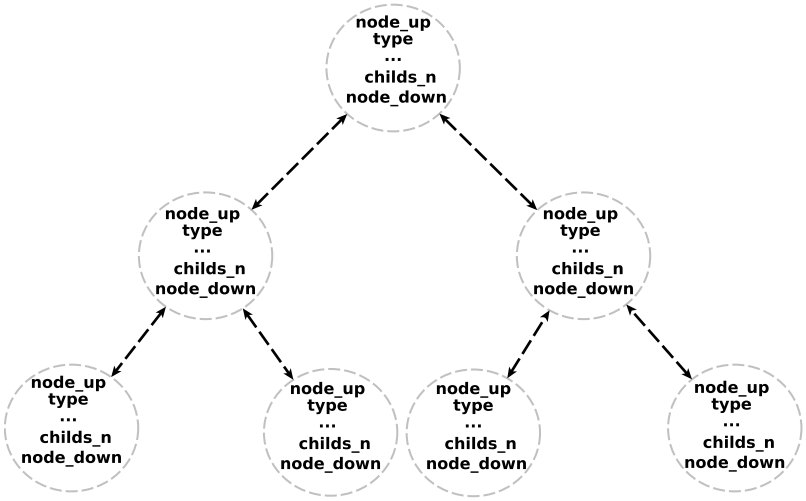


Figure 4.3: CINCLE AST representation.

4.6 CINCLE Middle-End

To implement a new language extension, users may have to implement the CINCLE Middle-End modules. Semantic analysis and AST optimizations are not necessary to perform source-to-source transformations, therefore this is an optional implementation. The top-down tree navigation can be implemented using recursive functions (see Algorithm 1) based on how the AST is built in the CINCLE environment. Searching for a node type is a recurrent operation on all Middle-End and Back-End modules when creating a DSL on the infrastructure's environment (Figure 4.1). All algorithms presented in this section are used to demonstrate how to manipulate the AST.

In principle, the top-down search algorithm should not be difficult to implement through a recursive function. One way to implement it is an Algorithm 1, where it first checks if the node is the type that intends to be searched. Second, it uses a loop for navigating into the node child list, making a recursive call to each one of the children and testing whether the returned node is the type that intends to be searched. Finally, if the node was unsearchable, the function will return an empty value.

Another way to navigate on the tree is bottom-up. Again, it is possible to implement this using a recursive function such as presented in Algorithm 2. Such an algorithm needs to first check if it is an empty node because it could be the case the root node has no father. Second, we can test if it is the node type that intends to

Algorithm 1: Recursive top-down navigation to search for a node type.

```
1 Function TopDownSearch (node,type)
2     if node is type then
3         return node;
4     for all node child i do
5         nodex ← TopDownSearch(node child i,type);
6         if nodex is type then
7             return nodex;
8     return empty;
```

be found. Finally, we return the function call passing as an argument to the current node's father, since the node has a pointer to its father.

Algorithm 2: Recursive bottom-up navigation to search for a node type.

```
1 Function BottomUpSearch (node,type)
2     if node is empty then
3         return empty;
4     else if node is type then
5         return node;
6     return BottomUpSearch(node father,type);
```

When semantic analysis is needed, one has to traverse the whole AST. Algorithm 3 provides an example for traversing the CINCLE AST recursively to perform an analysis. Assuming that a function will receive the tree root node as an argument, the algorithm can implement a top-down navigation. Thus, inside the function, first we have to check if the node is one of those intended to be analyzed. Then, if it is the node type, we can apply another recursive function, using bottom-up and top-down navigation. Second, we have to call the function for each one of the child nodes and test whether it is a semantic error. Finally, if no error was found during this process, the function will return true.

4.7 CINCLE Back-End

In the CINCLE Back-End, there are two modules: transformation and assemble. By default, the assemble module calls the GCC compiler to generate source-to-binary. However, one can simply integrate their favorite compiler or manually assemble transformed source-to-source code. To illustrate how the transformation can be done directly on CINCLE's AST, a basic example of pattern matching is given in Algorithm

Algorithm 3: Traversing recursively for performing semantic analysis.

```
1 Function TraverseSemantic (root,type)
2     if root is type then
3         if analyze(root) is false then
4             return false;
5     for i ← 0 to number of root child do
6         if TraverseSemantic(root child i,type) is false then
7             return false;
8     return true;
```

4 and its corresponding real implementation for the transformation module in Listing 4.1.

This function follows the same recursive logical implementation to traverse the entire AST. In the example, the goal is to transform all integer tokens into character tokens. The implementation is also very simple for the real code implementation. We only need a recursive call, node type checking and a token replacement.

Algorithm 4: A simple example for pattern matching transformation.

```
1 Function ConvertInteger (root,type)
2     if root is integer then
3         root ← type;
4     for i ← 0 to number of root child do
5         ConvertInteger(root child i,type);
```

```
1 void transform_int_token(cincle::node *root_node, int token){
2   if(root_node->type == NODE_TYPE_int_token){
3     root_node->type = token;
4   }
5   for (int i = 0; i < root_node->childs_n; ++i){
6     transform_int_token(root_node->node_down[i], token);
7   }
8 }
```

Listing 4.1: The real C++ code implemented form Algorithm 4 on CINCLE.

Another alternative for generating source-to-source code on CINCLE's AST is to use a pretty printer. One way for implementing in the transformation module is to use a recursive traverse function such as that presented in Algorithm 5. The printable nodes are tokens, identifiers and literals, which are terminal nodes. Then, during the recursive operation a check has to be done before printing the node content. Finally, as in the previous algorithms, it is simple for one to perform AST or pretty print transformations using the CINCLE infrastructure.

Algorithm 5: Generating code recursively from AST.

```
1 Function TraversePrettyPrint (node)
2     if node is token or identifier or literal then
3         print node content;
4     for i ← 0 to number of node child do
5         TraversePrettyPrint(node child i,type);
```

4.8 Supporting New Language Extensions

In addition to the infrastructure available to build new language extensions, CINCLE also offers important features such as a set of APIs to manipulate the AST and generate tree visualizations. Basic and useful routines are described in Table 4.1. These routines were developed based on the previous algorithm examples.

Routines	Description
`insert_node_before(...)`	inserts a given node before another one
`insert_node_after(...)`	inserts a given node after another one
`replace_node(...)`	replaces a given node by another one
`return_node_string(...)`	returns as a string the content of nodes like tokens, identifier and literals
`return_tree(...)`	returns a AST from a given piece of C/C++ code
`delete_tree(...)`	erases a given tree node
`verify_tree_structure(...)`	checks if the tree is correct accordingly the C/C++ grammar
`generate_visualization(...)`	generates a visualization to a given tree node
`return_decl_identifier(...)`	return the pointer of the declaration node relative to a given identifier node
`return_tree_statement(...)`	return a tree statement type from a string
`return_tree_expression(...)`	return a tree expression type from a string

Table 4.1: Basic API functions.

The `generate_visualization` routine can be applied to specific AST nodes, since it is called inside Middle-End and Back-End modules. It is used to quickly identify how to navigate on the AST for specific sentences as well as where the transformations should be performed. Also, it can be used to debug AST user transformations because after load Front-End modules no more analysis is performed on the tree and the users must manage their actions. The CINCLE APIs avoid incorrect transformations because they check the node types before making an operation. Although AST provides much more power to the DSL creator, one must be very careful when manipulating the tree because these operations are dangerous and may affect the correctness of the produced source code.

Two code examples (Listing 4.2 and 4.3) are used to illustrate how CINCLE produces an AST and the minimal changes performed on the grammar allowing for C and C++ attributes. There are preprocessing directives in the grammar, headers, defines, and pragmas . However, CINCLE was designed to recognize them because when generating source-to-source code they are important to maintain the correctness of the original code. Also, another important modification to the original grammar was to separate what is a C++ and a C attribute. As mentioned in Section 4.1, in the GCC compiler they are treated as the same thing, which makes no sense because C attributes are grammatically and syntactically different.

```
1 #include <stdio.h>
2 int main(){
3
4 }
```

Listing 4.2: Code example for AST visualization.

```
1 #include <stdio.h>
2 [[test]] int main(){
3
4 }
```

Listing 4.3: Code example using C++ attribute.

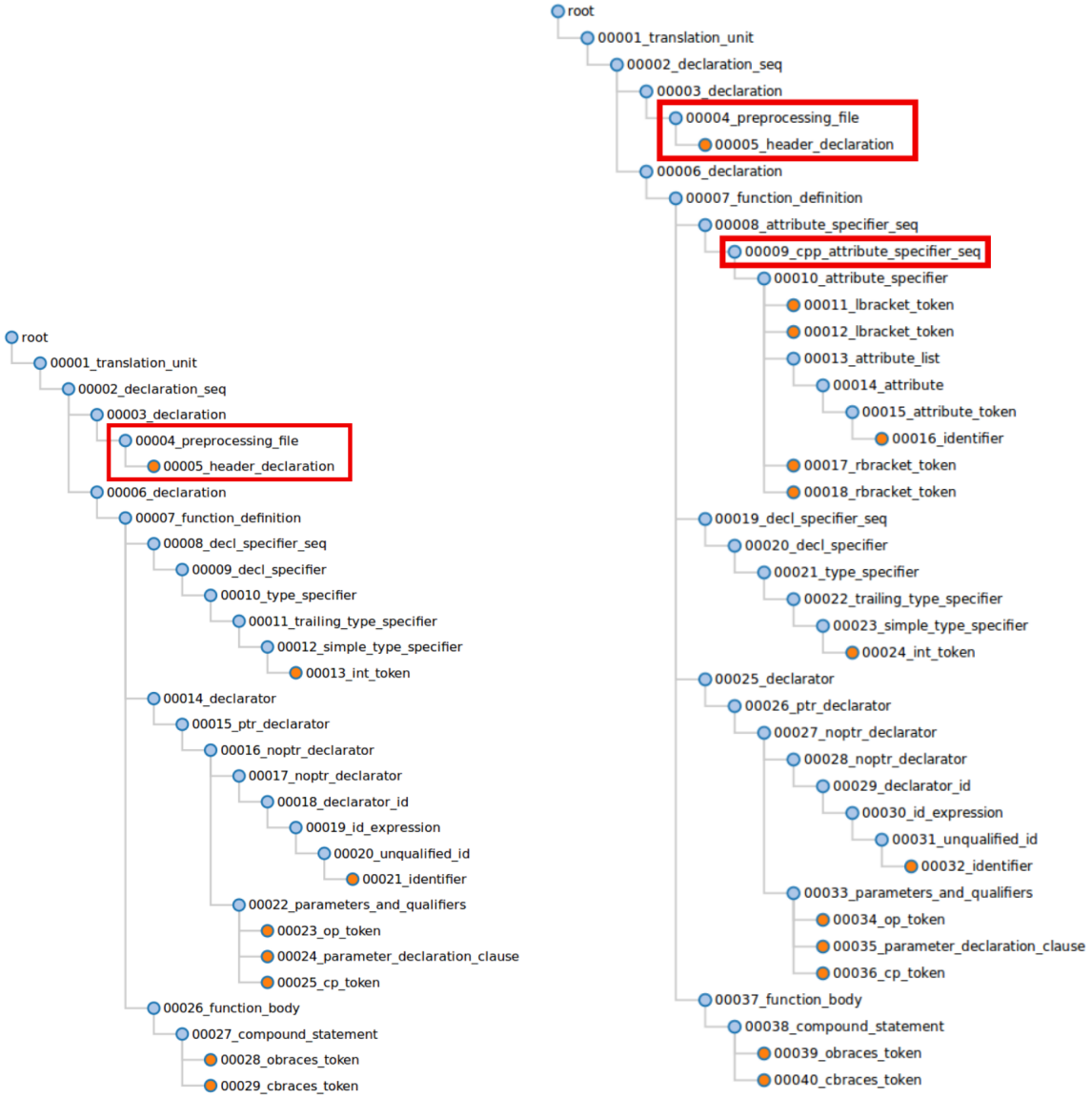
When generating the AST visualization from Listing 4.2 and 4.3, we get representations as demonstrated in Figure 4.4(a) and 4.4(b), respectively. These examples were intentional in order to observe and highlight the pictures. These highlights are in red, whereas terminator nodes are represented in orange and intermediate nodes in blue. Also, each node has a number that identifies its creation order and their names are the same as the standard grammar describes.

This visualization generation uses protovis library[ii] and produces a JSON file. The user can simply "open/close" (expand/collapse) nodes and there is an identification of the visualization tree that is given as an argument in the routine (not present in the pictures), enabling one to navigate between different tree visualizations.

4.9 Real Use Cases

In order to give an idea of the infrastructure's efficiency, we generated an "empty" compiler to perform tests and provide results. Consequently, this compiler only stresses the five compilation phases, which are check C++, code interpretation, AST verification, code generation and code assemble. First, it will call the GNU GCC compiler to check C++ code semantics and syntax (invoke the compiler so that it stops before assembling the code). Second, the source code is parsed and an AST is

[ii] http://mbostock.github.io/protovis/



(a) Tree visualization of Listing 4.2. (b) Tree visualization of Listing 4.3.

Figure 4.4: AST visualizations.

built (it invokes the Front-End implementation that uses Flex and Bison tools). Third, the compiler verifies if the AST was created correctly (calling a function that tests the correctness of the child nodes). Fourth, we generate from the AST C++ code (it is a simple pretty-printer function). Finally, we call the GNU GCC compiler again to assemble the code.

Figure 4.5 presents a performance comparison among GNU GCC, CINCLE and Clang compilers. We benchmarked a set of simple applications that can test distinct C++ constructions and grammar ambiguities [iii]. The graph presents the

[iii] Almost all the source code are taken from http://users.cis.fiu.edu/ weiss/adspc++2/code/, except the SimpleRNG from http://www.johndcook.com/blog/cpp_random_number_generation/

completion time on the Y axis, the standard deviation of 10 executions through error bars, and the application names along with the size in bytes in the X axis. As expected, Clang performed the best, while Clang and GCC achieved a significant completion time differently with respect to CINCLE. The reason is that CINCLE calls the GCC compiler twice. Figure 4.6 explains this more clearly.

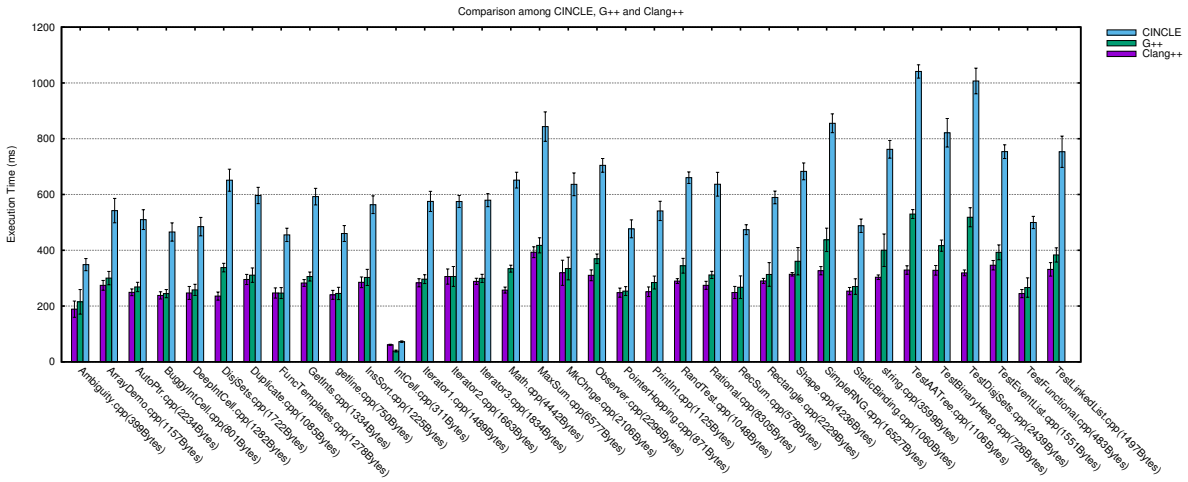


Figure 4.5: Performance comparison (machine with SSD hard drive).

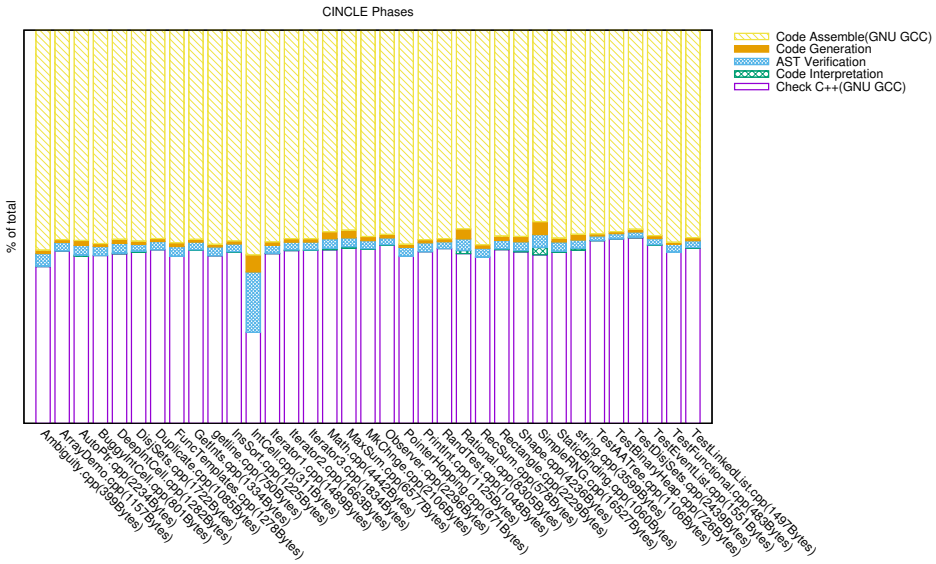


Figure 4.6: Only SPar compiler performance (machine with SSD hard drive).

The graph in Figure 4.6 presents the total percentage relative to the completion time for each one of the CINCLE compilation phases. In general, results of check C++ and code assemble phases are expected since both call the GNU GCC compiler. Among the actual CINCLE phases, the check of AST correctness requires the greatest amount

of time as well as the code generation. Finally, little time is needed for interpreting the code during the AST creation. Therefore, we can conclude that CINCLE does not add significant overhead to the program compilation.

Other tests were made by compiling a set of applications that will be used later for the SPar DSL evaluation, which was built on top of CINCLE infrastructure. We summarized the amount of AST nodes necessary to represent the source code versions on Table 4.2. It is important to highlight that the OpenMP version needs fewer nodes than SPar because pragma annotations are placed on the tree as a single node (they are similarly placed as a header declaration node highlighted in Figure 4.4(a)). Consequently, this is one clear example showing the difference of preprocessing directive compared to attributes that are placed along with the standard C++ grammar. While pragmas are seen as a single string, attributes are represented as a tree.

App.	Seq.	SPar	OpenMP	FastFlow	TBB
Filter Sobel (pipe)	21933	22244 (22533)	21957 (21965)	23417 (24768)	24113 (25511)
Video OpenCV	4808	5151	*n.a.*	6354	6910
Mandelbrot Set	4890	5376	5037	8563	7702
Prime Number (loop)	5391	5676	5394	6782 (5772)	7463 (5868)
K-Means	9901	10073	9910	10100	10045

Table 4.2: Statistics of CINCLE (number of nodes on the AST).

4.10 Summary

In this chapter, we introduced CINCLE, a new compiler-based infrastructure for generating a C++ internal DSL. We demonstrated its contributions to the state-of-the-art tools such as AST to AST transformations and AST compliance with the standard C++ grammar. Also, through small algorithm examples, it was possible to illustrate the simplicity and other essential features of modularity, extensibility and rapid prototyping.

Moreover, we presented a set of features to support DSL designers performing AST transformations by using API functions and AST visualization. During the presentation of CINCLE, we discussed several algorithms for navigation and transformation, where simple pattern matching requires just a few lines of code. Real use cases were also provided to demonstrate the efficiency and representativity of CINCLE. Its robustness will be seen in the next chapters through the implementation of the SPar compiler, which makes a source-to-source transformation to support high-performance code.

5

SPar: an Embedded C++ DSL for Stream Parallelism

This chapter presents an embedded C++ DSL for stream parallelism. SPar was built using the standard C++ annotation mechanism and CINCLE infrastructure. The goal is to provide high-level parallelism abstraction aiming for coding productivity in streaming applications. A secondary goal is to be architecture-independent, using the same interface to provide code portability.

Contents

5.1 Introduction

Stream-based applications represent several programs including video, networking, audio, graphics processing, etc. Such programs may run on different kind of parallel architectures (desktop, servers, cell phones, and supercomputers) and represent significant workloads on our current computing systems. However, most of them are still not parallelized, and when a new one has to be developed, programmers have to face a trade-off between coding productivity, coding portability, and performance. Unfortunately, the only suitable solutions to achieve efficient programs increase programming effort, mainly source code rewriting and porting an application across different architectures without modifying it (for example, a new compilation is needed to take advantage of parallelism). In fact, parallel programming is still too low level and complex, reserved just for experts in high-performance computing.

To solve this trade-off, we are providing a new DSL for stream parallelism aimed to naturally/on-the-fly represent parallelism in stream-based applications that are prevalent on our computing systems. The idea is to offer a set of attributes in an annotation manner that preserves the source code of the program. In general, such applications compute a sequence of distinct activities (stages) over the stream, where each activity consumes (input) a stream element, computes, and produces another one (output). This structure can be viewed as a graph of independent activities with explicit communications and contiguous flow. Representing the computation in such a way enables one to identify situations where it is possible to duplicate (replicate) stateless operations since they can process different stream elements [TA10, ADKT14]. As a consequence, stream programs may fit on coarse- and fine-grained parallelism, which is suitable for multi-core and cluster architectures. Thus, the stream properties motivated us to present parallelism abstractions in a straightforward manner as well as in generalized terms to achieve code portability and coding productivity through annotations.

In this chapter, we will first describe our original contribution in respect to the state-of-the-art. Second, our design goals and fundamental implementation choices will be presented. Third, we will formally describe the attributes of the DSL along with standard C++ grammar. Then, we create a methodology for guiding the developers during the code annotation. After this, how to annotate using our method as well as good practices for achieving efficient programs will be taught through examples. Next we describe the compiler implementation in a nutshell. Then, the internal representation of the attributes to be used for source-to-source code transformation is given. Finally, a statistic of the attributes in real use cases highlights the simplicity of this DSL.

5.2 Original Contributions

One of the original contributions of this thesis is the design of standard attributes that can eventually be adopted in the standard language for annotating stream parallelism. Even though C++11 attributes have been available in the grammar since 2011, to the best of our knowledge, we are the first to introduce stream parallelism by using this mechanism as a DSL[i].

Our second contribution is to provide a methodology to help users to easily find and annotate parallelism. This plays an important role as it gives a set of steps (questions) to guide the programmer. Consequently, supported by such a methodology, developers may simply concentrate on application specific features to annotate the most efficient parallel solution.

Another contribution is that we provide a compiler able to parse these attributes and perform the relative semantic analysis. This allows the user to simply compile the program to produce a parallel code for multi-core or cluster, which is another contribution of this thesis, described in Chapter 6. Moreover, the compiler implementation is also a contribution to prove CINCLE's robustness and efficiency.

Finally, to the best of our knowledge, we are the first in providing a high-level interface for stream parallelism that preserves the sequential source code. We also contribute to provide these attributes without being dependent on actual target architecture features, which is not usually the case when using state-of-the-art tools.

5.3 Design Goals

SPar's design goals are described in these sections to justify the design choices and principles of the present research. In general, they are greater than those possibly achieved by the thesis as they also include plans for the future of the proposed framework and programming interface. Accordingly, the main design goals are:

- *High-Level Parallelism:* SPar targets abstractions that prevent users from dealing with low-level programming models, hardware-level performance optimizations, scheduling policies implementation, load balancing, data and task level problem decomposition, and parallelism strategies. Our goal is to support high-level parallelism to work at the code annotation level rather than at the actual

[i]REPARA uses this kind of mechanism in a slightly different way that is not characterized as a DSL, but as an internal processing of parallelism.

exploitation level. When using annotations to exploit parallelism, the user is simply indicating where there is a potential parallelism. To express parallelism, the users are required to provide the appropriate parallelism strategy, learn different programming models, study efficient ways to optimize performance, and determine scheduling and load balancing implementations. This design goal is a starting point motivation to achieve code portability and coding productivity design goal.

- *Code Portability:* It is still a big challenge in parallel computing because the architectures require the use of different programming models in order to efficiently use hardware resources. Thus, a software that was implemented for exploiting parallelism on multi-core can not be simply used in a cluster architecture (and vice versa) without using a different programming interface or rewriting the code in some way. In order to support code portability in SPar, we propose the creation of a unified stream-oriented interface, believing that it provides properties that are generic enough to perform automatic parallel code transformations for both multi-core and clusters architectures. Thus, once the code has been annotated, no more modifications need to be made for running an application on different parallel architectures, they must only be recompiled.

- *Coding Productivity:* Is related to programming effort, code rewriting/restructuring and amount of code that a given application needs take advantage of the architecture parallelism. We benefit from the standard interface and code portability design goals to provide better coding productivity. For instance, code portability will avoid code rewriting when running a given software on different parallel architectures. However, the main aim here is to provide a small annotation vocabulary, preserving the original sequential source code and supporting on-the-fly stream parallelism.

- *Standard Interface:* Does not require users to learn a new language syntax. Being standard compliant with the host language syntax is the main motivation for using the C++ annotation mechanism (also called C++11 attributes) to build SPar as an internal DSL. At the implementation level, it is compliant with the standard and provides suitable advantages with respect to other mechanism such as pragmas (classified as a preprocessing language) for source-to-source code transformations, which were previously discussed in Chapter 4. Also, the C++ standardization allows the proposed research to target a wider community, since C++ has been used for decades to create robust software infrastructures, high-performance and critical applications.

- *Flexibility:* Is an important aspect for SPar. The idea is to allow different ways for annotating stream parallelism as well as alternatives to orchestrate parallel executions of C++ statements. By default, C++11 attributes are flexible and we will use them in such a way this flexibility is fully preserved. Consequently,

flexibility is in conformity with other goals such as high-level parallelism, coding productivity and code portability.

- *Performance:* Is our last priority in the design goal list as we intend to reuse runtime libraries designed to exploit parallelism that has already been proven efficient. However, performance is no less important than other properties listed above and the main concern is to avoid significant performance degradation while adding high-level abstractions. Therefore, good performance of the DSL will depend on the transformation rules, runtime library, and the appropriate usage of SPar annotation for a given application.

5.4 SPar DSL: Syntax and Semantics

C++ attributes originated from GNU C attributes (`__attribute__((<name>))`). Since C++11 up to the most recent version, a new way to provide annotation was included in the standard C++ language, namely the `[[attr-list]]` style syntax [MW08, ISO11a]. The syntax of the attributes was improved as well as the interface to support C++ features. A great advantage over the pragma-based annotation is the possibility to introduce annotations almost anywhere in a program. However, each attribute implementation will determine where the different attributes may be actually used (*e.g.*, to annotate types, classes, code blocks, etc.).

This section introduces the domain-specific language syntax used to meet our design goals. SPar maintains the standard C++ attributes' syntax [MW08] to introduce code annotations. However, limitations are imposed to ensure correct parallel code transformation. Also, SPar classifies the attribute in identifiers (ID) and auxiliary (AUX). Such a distinction was made to provide the appropriate meaning when annotating the code. In the following, ***ToStream*** and ***Stage*** are ID, while the others will be AUX.

5.4.1 ToStream

The ***ToStream*** attribute is intended to be used to denote that a given C++ program region is going to provide stream parallelism. The DSL grammar for this attribute uses and extends the same syntax used to describe the grammar in the International Standard [ISO14]. When possible, standard names are used. To distinguish from the terms defined and used in the standard, our specific terms will be written in blue from now on.

A *tostream_specifier* is only used to annotate in front of a compound statement or iteration statement. Due to the fact that SPar requires that inside an annotated region must be at least one stage, compound statements and iteration statement grammar productions are re-defined as follows:

tostream_specifier:
tostream_attr tostream_compound_statement
tostream_attr tostream_iteration_statement
tostream_compound_statement:
{ *tostream_statement* ***}***
tostream_iteration_statement:
while ***(*** *condition* ***)*** *tostream_statement*
do *tostream_statement* ***while*** ***(*** *expression* ***) ;***
for ***(*** *for_init_statement* ***;*** ***)*** *tostream_statement*
for ***(*** *for_init_statement* ***;*** *expression* ***)*** *tostream_statement*
for ***(*** *for_init_statement condition* ***;*** ***)*** *tostream_statement*
for ***(*** *for_init_statement condition* ***;*** *expression* ***)*** *tostream_statement*
for ***(*** *for_range_declaration* ***:*** *for_range_initializer* ***)*** *tostream_statement*

Another modification needed was to characterize *tostream_statement* in such a way it is possible to define which are the legal syntax entities in a ***ToStream*** clause.

tostream_statement:
statement_seq stage_specifier_seq
stage_specifier_seq

Finally, the clauses relative to *tostream_attr* may be defined as follows:

tostream_attr:
[[tostream_token]]
[[tostream_token aux_attr_list]]
aux_attr_list:
, input_specifier
, output_specifier
, replicate_specifier
, input_specifier , output_specifier
, output_specifier , input_specifier
, input_specifier , replicate_specifier
, output_specifier , replicate_specifier
, replicate_specifier , input_specifier
, replicate_specifier , output_specifier
, replicate_specifier , input_specifier , output_specifier
, replicate_specifier , output_specifier , input_specifier

, *input_specifier* , *output_specifier* , *replicate_specifier*
, *output_specifier* , *input_specifier* , *replicate_specifier*
, *input_specifier* , *replicate_specifier* , *output_specifier*
, *output_specifier* , *replicate_specifier* , *input_specifier*
tostream_token:
tostream_scoped_token
tostream_scoped_token:
attribute_namespace :: ***ToStream***
attribute_namespace:
spar

NOTE: *as in the standard grammar, the auxiliary attributes (aux_attr_list) are not necessary ordered. Restrictions are only made for ID attributes to identify a region in the stream parallelism.*

5.4.2 Stage

As the name indicates, ***Stage*** is used to annotate a phase where operations are computed over the stream items. If we imagine that we are in an assembly line, ***Stage*** is a workstation in the production line. Inside a ***ToStream*** region, SPar supports any number of ***Stage***. The relative grammar clauses are represented as follows:

stage_specifier_seq:
stage_specifier
stage_specifier stage_specifier_seq
stage_specifier:
stage_attr compound_statement
stage_attr iteration_statement
stage_attr:
[[stage_token]]
[[stage_token aux_attr_list]]
stage_token:
stage_scoped_token
stage_scoped_token:
attribute_namespace :: ***Stage***
attribute_namespace:
spar

NOTE: *by default* ***ToStream*** *and* ***Stage*** *attributes may have arguments, which are not supported in current version of SPar.*

5.4.3 Input

The ***Input*** attribute represents an important property of stream parallelism. In SPar, the programmer should use this keyword to express the input data format of the stream for both ID attribute annotations. Its arguments will be parsed to build the stream of tasks (data items) that will flow inside the ***ToStream*** region. Using the assembly line example, input denotes the items "consumed" by each workstation. The relative grammar may be described as follows:

input_specifier:
    input_attr *attribute_argument_clause*
input_attr:
    input_token
  input_token:
    input_scoped_token
  input_scoped_token:
    attribute_namespace :: ***Input***
attribute_namespace:
    spar

NOTE: *Semantically, when using* ***Input*** *attribute at least one argument should be given. This argument could also be a variable derived from a data type. Literals are not accepted.*

5.4.4 Output

The ***Output*** attribute also represents another important property of stream parallelism: the programmer should use it to express the output data format of the stream for both ID attribute annotations. Its arguments will be used to build the stream that will flow inside the ***ToStream*** region. Using the assembly line example, output is what ID attribute will produce for the next workstation. The ***Output*** grammar clause are therefore described as follows:

output_specifier:
    output_attr *attribute_argument_clause*

output_attr:
    output_token
  output_token:
    output_scoped_token
  output_scoped_token:
    attribute_namespace :: ***Output***
attribute_namespace:
    spar

NOTE: *Semantically, when using* ***Output*** *attribute at least one argument should be given. This argument could also be a variable derived from a data type. Literals are not accepted.*

5.4.5 Replicate

The ***Replicate*** attribute is used to model another important propriety of stream parallelism. Again, drawing from the assembly line example, it is important to balance the load in a single workstation and accelerate the production line by implementing several replicas of the workstation in place of a single one. When adding replicas to a stage, one is replicating the relative region as many times as denoted by the number of worker's parameter.

replicate_specifier:
    replicate_attr attribute_argument_clause
replicate_attr:
    replicate_token
  replicate_token:
    replicate_scoped_token
  replicate_scoped_token:
    attribute_namespace :: ***Replicate***
attribute_namespace:
    spar

NOTE: *Semantically, no more than one argument is accepted to represent the number of workers in a given stage. This argument can be an integer literal or an integer variable. If no argument is passed, SPar gets the number of workers from the* `SPAR_NUM_WORKERS` *environment variable.*

NOTE: *Syntactically,* ***Replicate*** *can be part of the* ***ToStream*** *attribute list, but currently SPar simply ignores it when associated to the* ***ToStream*** *attribute list.*

5.5 Methodology Schema: How to Annotate

This section introduces a methodology to annotate stream parallelism using SPar attributes. Figure 5.1 presents five questions that one should ask themselves to annotate source code. The methodology intends to instruct the programmer on how to annotate by answering these questions. Following the order, the first thing to do is to discover where the stream region is. Usually, a stream region can be associated with the assembly line. In a program, we can identify and visualize the stream region as the most time consuming piece of code.

In most cases, the stream computation will be inside a loop, which generates a new stream element per iteration. In all other cases, the stream will come from an external source and the developer should pay attention to identify the relevant code section gathering the stream items and computing results out of them. Once the stream region has been identified and properly annotated, we have to look inside the region searching for what the region consumes and produces. The idea behind this is to fill, when necessary, the ***Input*** and ***Output*** auxiliary attributes for the stream region.

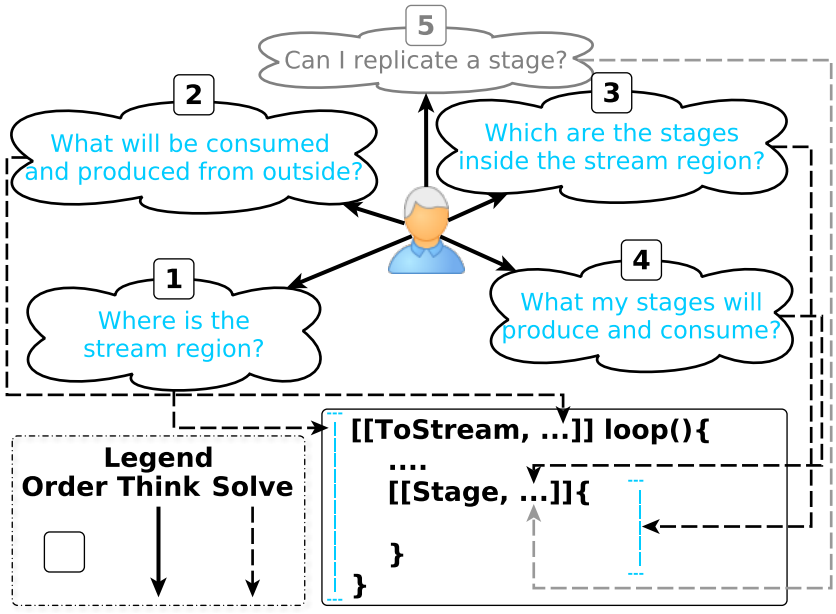


Figure 5.1: Annotation methodology schema.

The third question helps to identify the assembly line's workstations. In the program, they are inside the stream region already annotated in the previous steps of our methodology. To answer the question, the suggestion is to look for the operation sequence and annotate as many stage regions as necessary, respecting the SPar syntax

and semantics. Then, it is important to specify what will be consumed and produced by each one of the stages by using ***Input*** and ***Output*** attributes, answering the fourth question.

In the assembly line, we can only assign more workers in a given workstation when the computations relative to different task are independent. The same rule applies to SPar when answering the fifth question. In the program it means that each worker can get a new stream element and compute independently from other stream elements. To be sure this property holds, the developer may use the ***Replicate*** attribute to improve the performance of the stream region. The next section will demonstrate some code examples and our best practices for speeding up the performance.

5.6 Examples and Good Practices

This section discusses simple examples that can be used in range of real applications. The goal is to demonstrate through real code the usage of SPar attributes guided by the methodology presented previously. First, Figure 5.2 illustrates four activity graphs to represent stream parallelism abstractly. The idea is to highlight one of the graphs when discussing an example. As the methodology implicitly recommends, a good practice is to start with simple and move towards more complex graphs (from left to right in Figure 5.2). In terms of performance, we can not make any assumptions because it depends on the application features (*e.g.*,throughput, latency, memory usage and parallelism degree).

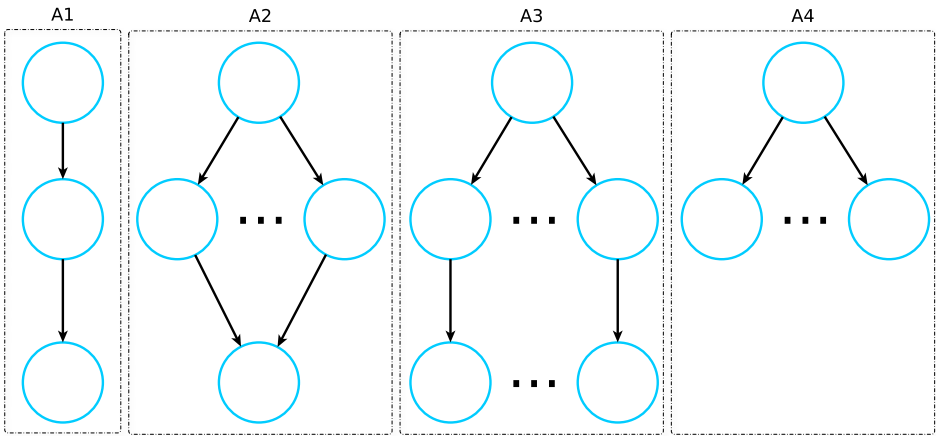


Figure 5.2: Activity graphs on SPar.

Listings 5.1, 5.2, 5.3 and 5.4 are relative to the same application achieving different activity graphs through SPar annotations. The code is an example of typical and recurrent situations in stream parallelism. In this application, we clearly characterized

the stream format, which is a string. The stream region is the loop block and stream comes from an external source that is a file. For each iteration a new stream element is read and a sequence of operations is performed. A similar code may be used if the stream comes from the network or any other external source and the programmer may not know the length of the stream. Consequently, the programmer has to check or decide whether the program should stop or not. This stream operation can be seen on line 4, which is actually checking the end-of-stream condition (the end of the file, in this case). When stream comes from the network there is no end, making it necessary to filter the stream content to create a stop condition.

With these issues in mind, we can start to put the annotations in the code. Following the methodology recommendation, we should start with Listing 5.1. We add a ***ToStream*** annotation in front of the `while` loop because it is the stream region. No input is needed since the stream comes from an external source and produces each stream item inside the stream region. Also, no output specification is required because nothing is produced inside the stream region that will be used outside. Now, we have to find the stream operations and annotate them by using the stage attribute. We identify them as: 1) read stream element (line 3), 2)check end of the stream (line 4), 3) compute the stream element (line 6) and 4) write the result in an output source (line 8).

Note that semantically we cannot annotate the end of a stream checker operation as a stage, because ***ToStream*** performs the initial computation. The problem is that the ***ToStream*** region will never know when to stop because SPar performs on-the-fly (there are no back communications, it always is forward). Therefore, we leave the checker and reader operations for the ***ToStream*** and annotate the compute and write operations as stages. As a consequence, SPar will produce the A1 activity graph from Figure 5.2.

```
1 [[spar::ToStream]] while(1){
2   std::string stream_element;
3   read_in(stream_element);
4   if(stream_in.eof()) break;
5   [[spar::Stage,spar::Input(stream_element),spar::Output(stream_element)
      ]]
6   { compute(stream_element); }
7   [[spar::Stage,spar::Input(stream_element)]]
8   { write_out(stream_element); }
9 }
```

Listing 5.1: Stream computations in SPar producing A1.

The last step recommended in the methodology schema is to find the stages that can be replicated. Before adding the ***Replicate*** attribute, the programmer must be sure that operations can operate independently in different stream elements. Listing 5.2 exemplifies such an implementation for the A2 activity graph of Figure 5.2. Note

that no significant changes to the source code were made with respect to the Listing 5.1 to introduce a different version of stream parallelism.

```
[[spar::ToStream]] while(1){
  std::string stream_element;
  read_in(stream_element);
  if(stream_in.eof()) break;
  [[spar::Stage,spar::Input(stream_element),spar::Output(stream_element)
    ,spar::Replicate(4)]]
  { compute(stream_element); }
  [[spar::Stage,spar::Input(stream_element)]]
  { write_out(stream_element); }
}
```

Listing 5.2: Stream computations in SPar producing A2.

Listing 5.3 demonstrates how to produce the A3 activity graph from Figure 5.2. However, in this application such annotation will produce incorrect results because we can not put it inside a stateful stage. Consequently, another SPar lesson which demonstrates that it is up to the user identify whether replication can be done without side effects. The advantage in SPar is that one can test different combinations without significant effort.

```
[[spar::ToStream]] while(1){
  std::string stream_element;
  read_in(stream_element);
  if(stream_in.eof()) break;
  [[spar::Stage,spar::Input(stream_element),spar::Output(stream_element)
    ,spar::Replicate(2)]]
  { compute(stream_element); }
  [[spar::Stage,spar::Input(stream_element),spar::Replicate(2)]]
  { write_out(stream_element); }
}
```

Listing 5.3: Stream computations in SPar producing A3.

Since we discussed in Listing 5.3 that the write operation can not be done independently, Listing 5.4 will also produce incorrect results. Thus, it only illustrates how to achieve A4 activity graph from Figure 5.2.

```
[[spar::ToStream]] while(1){
  std::string stream_element;
  read_in(stream_element);
  if(stream_in.eof()) break;
  [[spar::Stage,spar::Input(stream_element),spar::Replicate(4)]]
  { compute(stream_element);
    write_out(stream_element); }
}
```

Listing 5.4: Stream computations in SPar producing A4.

In principle, stream parallelism can be used to express other kinds of parallelism. For example, Listing 5.5 lists code relative to a simple reactive computation. The stream comes from the user command line arguments asking to compute the multipliers of a given digit up to 10. SPar is generic enough to support suitable annotation modeling the parallel structure of the code. The stream region is the `while` loop block and the operation in each stream element (which are integers) are made by reading from terminal (line 4) and calculating the multipliers of the stream element (between line 5 and 7). We do not need to specify ***Output*** and ***Input*** for the ***ToStream*** attribute by the same reason of the previous examples. Listing 5.5 is typical way to annotate stream region, where we put one ***Stage*** annotation in front the `for` loop leveraging the possibility provided by the SPar grammar to reuse constructions of the loop statements.

```
1 [[spar::ToStream]] while(1){
2   int stream_source;
3   std::cout << "Enter a digit: ";
4   std::cin >> stream_source;
5   [[spar::Stage,spar::Input(stream_source)]] for (int i = 1; i < 11; i
     ++){
6     std::cout << stream_source*i << std::endl;
7   }
8 }
```

Listing 5.5: Reactive computations in SPar producing A1.

As in previous examples, the stream comes from an external source, but in this case, there is no stop criteria implemented. This means that the stream may never end since the code filters each stream element and does not implement any protocol for finalizing the stream. Also, unlike the previous application, reactive computation has different objectives, namely latency instead of throughput. Thus, it is important to find the most appropriate annotation schema to target latency, which could be difficult without testing different alternatives. SPar makes this process easier, because it supports alternative structure evaluations without requiring the user to modify the sequential code.

In addition to Listing 5.5 and 5.6, we could have annotated the same sequential code in at least five different ways. With respect to Listing 5.5, we could have used the ***Replicate*** attribute (line 5), included a stage for getting the stream elements (line 4) and combined this solution including, or not, the ***Replicate*** attribute on the multiplier stage (line 5). From a part of Listing 5.6, we could have omitted the ***Replicate*** attribute. This demonstrates the flexibility and capability of SPar also for annotating parallelism in other domains derived from the stream paradigm.

To conclude the discussion of Listing 5.5, we can easily translate this pattern into a real world application. As an example, consider a social network service for counting the number of subscriptions. Instead of reading from the user command line,

the application reads requests from the network and instead of making multipliers, the operation will be the sum of the subscriptions. Also, we can follow the methodology suggestion of putting a ***Replicate*** attribute because it is possible to compute each stream element independently. Adding the ***Replicate*** attribute may represent a significant performance improvement in the social network service to guarantee latency and throughput when there are many client requests. Finally, Listing 5.5 produces A1 activity graph, but adding the ***Replicate*** attribute it will produce A4 from Figure 5.2.

```
1 [[spar::ToStream]] while(1){
2   int stream_source;
3   std::cout << "Enter a digit: ";
4   std::cin >> stream_source;
5   for (int i = 1; i < 11; i++){
6     [[spar::Stage,spar::Input(i,stream_source),spar::Replicate(10)]]
7     {std::cout << stream_source*i << std::endl;}
8   }
9 }
```

Listing 5.6: Reactive computations in SPar producing A4.

Another way to annotate this particular application is to reduce the granularity by putting the stage inside `for` loop (line 6) such as in Listing 5.6. Also in this case, we can add a ***Replicate*** attribute leveraging on the fact that for each stream element the multiplier can be computed independently. Moreover, by specifying the number of replicas, we can precisely assign a worker to each one of the multiplier operations. Therefore, this stream region will behave like the A4 activity graph from Figure 5.2.

```
1 [[spar::ToStream]] for (int i = 0; i < NREGION; ++i){
2   int *persons = new int[NPERSON];
3   load(persons);
4   [[spar::Stage,spar::Input(i,persons)]]
5   {regions_min[i] = min(persons);}
6   [[spar::Stage,spar::Input(i,persons)]]
7   {regions_max[i] = max(persons);}
8 }
```

Listing 5.7: DataFlow computations in SPar producing A1.

DataFlow is also a stream-based paradigm, but it represent a different approach with respect to SPar. However, we can use SPar attributes to implement DataFlow computations as well (see Listing 5.7, 5.8, and 5.9). The program in Listing 5.7 calculates the minimum and maximum age for a set of people in a given number of regions. From the SPar point of view, the stream region starts on the "for" loop that iterates for each one of people region's vector, loading the people and finding the minimum and maximum age. Also, the stream source are the people vector and the for index because it will be used for navigating on the vectors that will eventually store the age results.

Listing 5.7 shows one of the alternative ways we have to annotate the code, producing the activity graph A1 from Figure 5.2. Since the stream source is internally produced in the stream region, no input was specified. Also, no output is specified because nothing will be produced to be used outside. We annotated each one of the search operations (min and max) to be a stage, and only the data they will consume is annotated using the ***Input*** attribute. Both operations will process as input the same people's ages vector and the result vector index.

In SPar, we can observe that DataFlow-like parallelism can be annotated, but not exploited due to the fact that the present version only focuses on stream parallelism. By extending the previous example and responding the last question of the methodology, we can add ***Replicate*** attributes on the stages due to the fact the stream operations can act independently. Then, the annotation code will be 5.8, producing A3 from Figure 5.2.

```
1 [[spar::ToStream]] for (int i = 0; i < NREGION; ++i){
2   int *persons = new int[NPERSON];
3   load(persons);
4   [[spar::Stage,spar::Input(i,persons),spar::Replicate(2)]]
5   {regions_min[i] = min(persons);}
6   [[spar::Stage,spar::Input(i,persons),spar::Replicate(2)]]
7   {regions_max[i] = max(persons);}
8 }
```

Listing 5.8: DataFlow computations in SPar producing A3.

Listing 5.9 presents another annotation schema for this DataFlow computation. In this case, we merge both stream operations at a single stage so that another activity graph is produced, which is A4 from Figure 5.2.

```
1 [[spar::ToStream]] for (int i = 0; i < NREGION; ++i){
2   int *persons = new int[NPERSON];
3   load(persons);
4   [[spar::Stage,spar::Input(i,persons),spar::Replicate(4)]]
5   {regions_min[i] = min(persons);
6   regions_max[i] = max(persons);}
7 }
```

Listing 5.9: DataFlow computations in SPar Producing A4 .

Finally, to illustrate the applicability of SPar for data parallel computations, Listing 5.10 sketches matrix multiplication algorithm. In SPar, we can model a stream out of the first `for` loop index that ranges over the matrix lines. Since inside the second `for` loop (line 4) the multiplication of lines by columns is performed, we can annotate it as a stage operation. Also, due to the fact that all the multiplications can be performed independently, we can add ***Replicate*** attribute and produce an activity graph such as A4 on Figure 5.2,

```
[[spar::ToStream]]
for(long int i=0; i<MX; i++){
  [[spar::Stage,spar::Input(i), spar::Replicate(4)]]
  for(long int j=0; j<MX; j++){
    for(long int k=0; k<MX; k++){
      matrix[i][j] += (matrix1[i][k] * matrix2[k][j]);
    }
  }
}
```

Listing 5.10: Data parallel computations in SPar

These examples demonstrated SPar's usage in stream and other domain applications. Different applications will be discussed in Chapter 7, presenting their respective performance results. Moreover, it is worth pointing out that in all of the examples shown, the number of replicated stages may be simply changed varying the replica parameter. As a consequence, the programmer may easily experiment with different degrees of parallelism to find the one that is most suitable for the computation (code and input task) at the hand.

5.7 SPar Compiler

The compiler we designed to handle SPar DSL uses the CINCLE infrastructure previously described in Chapter 4. In order to highlight what was implemented to generate the DSL compiler and what CINCLE already offered, in Figure 5.3 the boxes in orange show the CINCLE related modules and the SPar implementations are in cyan. The picture clearly outlines that by using CINCLE's infrastructure it becomes much more simple to build SPar, since the only missing parts (w.r.t. CINCLE) are actually the semantic analysis and the AST transformations.

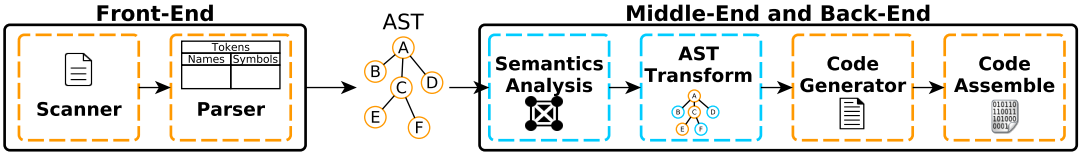


Figure 5.3: SPar Compiler.

When compiling a program using the SPar compiler, the system calls the GCC compiler before invoking the scanner to perform the semantic and syntax analysis of the C++ code. Next, the scanner gets the tokens produced from the original code and delivers them to the parser to create the AST that will be the interface for

implementing the middle-end and back-end. Only then, the semantic analysis of SPar annotations can rely on annotation correctness so that AST transformations can be actually implemented to enable stream parallelism. The final step of the compiler is relative to the generation of the parallel code, directly represented in the AST. Subsequently, the GCC compiler is called again to produce the binary code.

Even though many work has been avoided by using CINCLE, the internal implementation of semantics requires a good understanding of the C++ grammar. Similar knowledge is required to perform AST transformations, because expertise in parallel programming and runtime interface are required. Therefore, SPar was also a case study for CINCLE that demonstrated the simplicity and usability of the designed infrastructure. Also, the CINCLE infrastructure enabled the DSL creator to only concentrate on the parallelism related aspects. The automatic parallel code transformations will be detailed in the following Chapters.

5.8 SPar Internals

After the semantic analysis, we traversed the whole CINCLE AST and built the SPar AST that includes the representation of the annotations in the code. It is used to implement the source-to-source code transformation needed to target multi-core and clusters architectures. Figure 5.4 exemplifies how the attributes are represented in the SPar tree. Each attribute is interpreted as a node of the tree that stores information about the arguments and a pointer to its `attribute_specifier_seq` node. In addition to that, ***ToStream*** (T) have a function definition node (FD) as father, list of auxiliary attributes (AUX) that can be ***Input*** (I), ***Output*** (O) and ***Replicate*** (R) as child nodes and identifier node (ID) of stages. The ***Stage*** node (S) has only one child node that is a list of auxiliary attributes. For this version, we do not allow for explicit nesting so that new constructions of ***ToStream*** are not enabled inside stage regions.

The SPar AST was designed to simplify the applicability of the transformation rules as well as to support the internal CINCLE AST transformation by storing relevant information. Therefore, when prototyping the rules, the system can look separately at the two representations (CINCLE and SPar ASTs) and precisely perform the changes in the CINCLE AST to allow the runtime to exploit the parallelism of the target architecture.

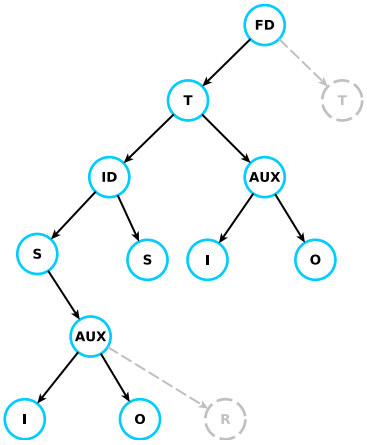


Figure 5.4: SPar AST.

5.9 Annotation Statistics on Real Use Cases

In order to demonstrate the applicability of SPar in real use cases, Table 5.1 provides the annotation statistics relative to the set of applications that will be used later in the result section. These applications were used to investigate code portability, performance and productivity. Therefore, the amount of attributes needed to annotate the parallelism was accounted for in each. Observing the data, we can see that it is possible to solve different problems with a few attributes. Also, auxiliary attributes are always present even if they are not syntactically necessary. Moreover, ***Input*** attribute is the most frequently used attribute in our applications.

Attribute	Filter Sobel	Filter Sobel (pipe)	Video OpenCV	Mandelbrot Set	Prime Number	K-Means
ToStream	1	1	1	1	1	2
Stage	1	2	2	2	2	2 (1/1)
Input	2	3	3	3	3	4 (2/2)
Output	1	2	1	1	2	2 (1/1)
Replicate	1	1	1	1	1	2 (1/1)

Table 5.1: Statistics of SPar annotations on the experiment.

This table demonstrates that our simple examples and good practices mentioned/used in the previous sections actually reflect what will eventually happen when dealing with real use cases. For instance, the ***Replicate*** attribute is needed to increase performance. Thus, it is supposed to be present on all application that aims for high performance. As semantically expected, note that there will be also at least one ***Stage*** per ***ToStream*** and ***Replicate*** should not be necessary present in each one of the stages, but it was at least in one of them.

5.10 Summary

This chapter has provided an overall illustration of SPar and it usage for enabling productive stream parallelism in C++ programs. We can highlight that it demonstrated SPar is a straightforward and high-level interface as well as friendly to the domain and capable of annotating other kinds of parallelism (*e.g.*,data parallelism). Also, through a small set of attributes, it was able to achieve the necessary flexibility when implementing different versions of the applications at hand without requiring any source code rewriting. Such benefit should be evident since the DSL has actually no dependence/relationship with the target architecture features. In the following chapters more evidence will be provided concerning these contributions.

6

Introducing Code Portability for Multi-Core and Cluster

This section will present how code portability is achieved and source-to-source transformations are made for multi-core and cluster.

Contents

6.1 Introduction

Code portability with high-performance code is a challenge in parallel programming. The state-of-the-art frameworks are still too low-level and closely tied to the architecture programming model. Unfortunately, this drawback may result in several different implementations of the same software to meet different application constraints such as scalability, energy, memory, latency, etc. Consequently, code portability significantly impacts productivity as well. To solve this problem, our goal is to create generalized transformation rules from SPar's annotations as a step towards achieving automatic parallel code generation for multi-core and cluster systems. Our scope is transformations targeting parallel patterns that support the stream-oriented paradigm.

In this chapter, we first introduce our original contributions. Second, a brief introduction to parallel patterns, FastFlow and MPI runtime will be given. Next, we present our transformation rules along with their formalization. Lastly, we demonstrate through a real case how the same annotation sentences are transformed automatically by the compiler into FastFlow (targeting multi-cores) and manually generated by using MPI runtime (targeting clusters).

6.2 Original Contribution

Our original contributions are the generalized transformation rules for SPar sentences. These rules can be applied when targeting different parallel architectures as well as different pattern-based runtimes.

In addition, we demonstrate how code portability is achieved by SPar at the annotation level. This is made possible by using the generalized transformation rules and the high-level interface provided by SPar in Chapter 5. Therefore, it is only a matter of compiling again for the program to execute on another parallel architecture.

Since in MPI we have to implement everything including pattern-based constructions, synchronization and scheduling, we also contribute by creating an intermediate interface to allow MPI to exploit stream parallelism with the round robin scheduler.

6.3 Parallel Patterns in a Nutshell

Parallel patterns have a long history from two distinct research communities. After the original definition of the Algorithmic Skeletons as proposed by Murray Cole [Col89], they were proposed as high-order functions modeling common parallelism exploitation patterns and providing parallel building blocks for parallel application programmers. Since then, several researchers from the parallel computing community [Col04, AD07, GVL10] started to create parallel programming frameworks providing skeletons, investigating and designing new ones. Later, design patterns for parallel programming [MSM05] were designed by the software engineering community inspired by design patterns [GHJV02]. Unlike skeletons, they were proposed as a methodology for recurrent patterns to exploit parallelism in applications. Also, the methodology seeks to be more general in targeting different programming models, since each pattern includes a description of its name, problem, solution, context, forces and examples, which are all independent of the target programming model.

In conclusion, both approaches have produced similar results, differing only in the name used to represent the pattern. For instance, the farm can produce a Master/Slave or Master/Worker while MapReduce results from the combination of map and reduce skeletons. Currently, both approaches converge into a single term for structured parallel programming [MRR12]. As an example, TBB is a framework that comes from design patterns and FastFlow from algorithmic skeletons. At the present moment both frameworks use parallel patterns to represent their parallelism strategies and library constructions. As we are only interested in the strategy aspects, during the thesis they will be called parallel patterns.

Theoretically, a pattern either exploits task or data parallelism. Task and data parallel patterns can be composed to produce other more complex patterns or skeletons. A set of well-known patterns are presented in Figure 6.1 to give an overall idea of the amount and diversity of parallel patterns. Almost all of these structures can be used to explore data parallelism which makes these strategies not appropriate for our domain. Most of them are implemented in standard frameworks like TBB (pipeline, map, reduce, scan). Although FastFlow originated from the task-based and stream-oriented perspective through classical stream parallel patterns (pipeline and farms), it also implements data parallel patterns (map, reduce and stencil).

Figure 6.2 presents a set of task-based parallel patterns that are suitable for exploiting stream parallelism. They can also be composed and produce new more complex stream parallel patterns. Essentially, Master/Worker is structured by a master activity with N number of worker activities. Communication with the worker can be synchronous and asynchronous, where the initiative may be from both activity

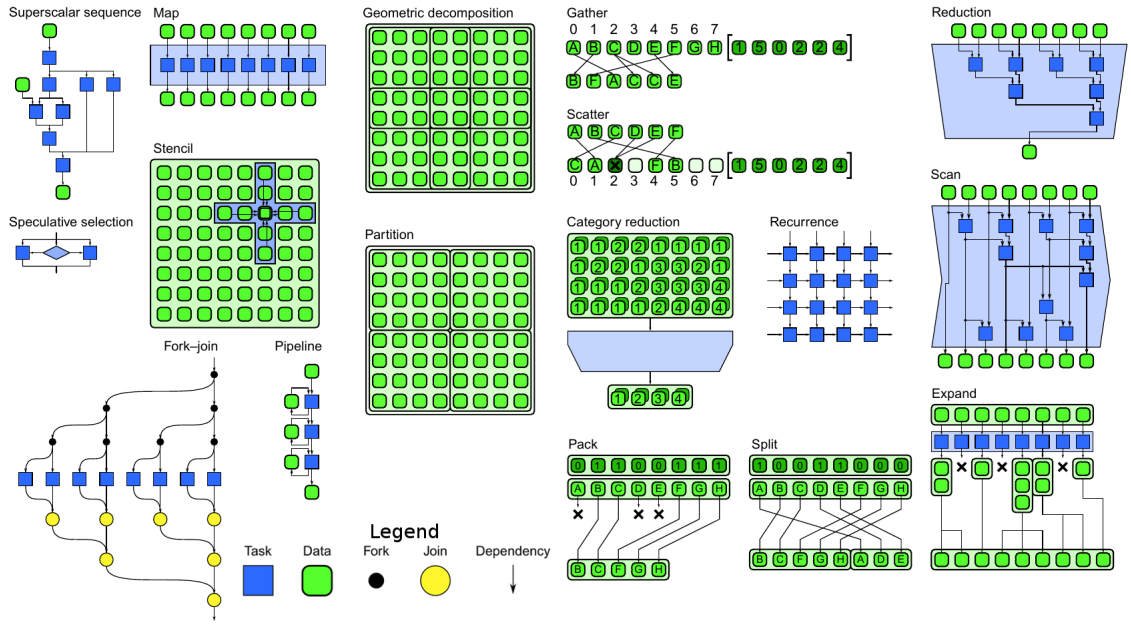


Figure 6.1: Overview of different parallel patterns. Extracted from [MRR12].

entities of the pattern. A feedback pattern may be applied to any stream parallel pattern. A condition node decides whether to route back a result to the pattern input or to deliver it to the output stream. The pipeline consists of a sequence of stage activities that communicates synchronously and each stage computes the results that were computed by predecessor stage. Finally, the farm includes an emitter activity, N worker activities and possibly a collector activity. Semantically, the farm computes a given function in all items of the input stream.

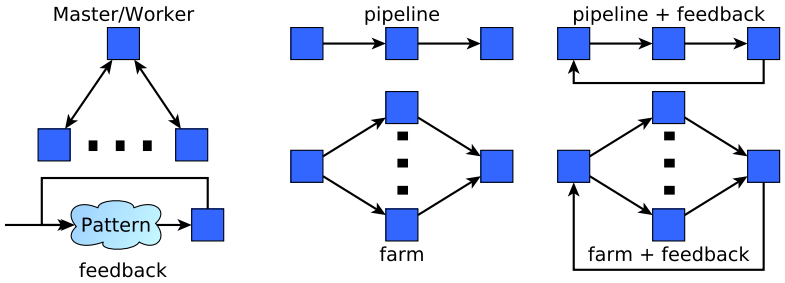


Figure 6.2: A set of task-based parallel patterns for stream parallelism.

In stream parallelism, each one of the activities processes distinct stream elements sequentially (this can also be seen as tasks). For instance, each stage of the pipeline processes a task and sends it to the next one until all tasks are completed. In the farm pattern, the emitter is the entity that generates the stream items and sends it to the workers while the collector gathers all results from the workers. Moreover, people in the algorithmic skeleton community have provided a formalism to represent stream parallelism along with farm and pipeline patterns [AD07], modeling,composition, and

nesting.

6.4 Multi-Core Runtime (FastFlow)

FastFlow is an open source parallel programming framework that originated from algorithmic skeleton and structured programming approaches. It is a research project that has been developed at the University of Pisa and Torino since 2008 and has been used in several research projects. The main goal is to provide an efficient and portable runtime library targeting different kinds of heterogeneous parallel architectures [ADKT14, DT15]. FastFlow builds on top of Pthreads[i], providing a suitable parallel programming abstraction algorithmic skeleton for streaming applications, it can exploit fine-grain parallelism in cache-coherent shared memory platforms and heterogeneous systems.

Conceptually, FastFlow is designed as a stack of layers that looks like those in Figure 6.3. There is a clear access separation of the layers with respect to the application level. The lowest programming levels are the building blocks, supporting different queue implementations, extensible and configurable schedulers, and thread and processes C++-like containers. Internally, the shared memory support implements the runtime using a lock-free mechanism [ADK+11, ADK+12]. The distributed runtime is zero-copy message [SUPT14]. Finally, the GPGPU exploits asynchronous parallelism [ADKT12].

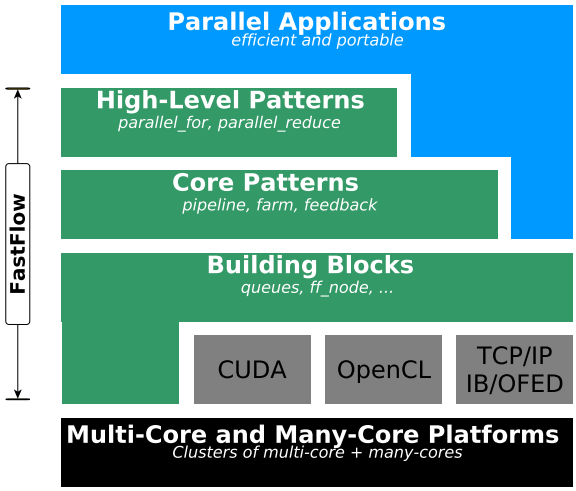


Figure 6.3: FastFlow Architecture. Adapted from [DT15].

At the low-level of FastFlow the queues are the fundamentals for creating new skeleton topologies and compositions in shared memory environments. A representation

[i]Although any other thread library may be used.

of the runtime implementation of typical parallel structures and behavior can be found in Figure 6.4. There is a clear view of the node connections that are made using Single Producer and Single Consumer (SPSC) lock-free queues. For each new node a new queue will be created. When communication patterns such as Multiple Producer and Single Consumer (MPSC) or Single Producer and Multiple Consumer (SPMC) are needed, in FastFlow they are implemented by using SPSC channels and an extra thread to enforce the correct serialization for consumers and producers.

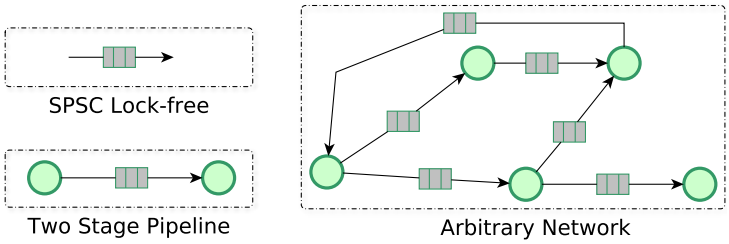


Figure 6.4: FastFlow Queues. Adapted from [Fas16].

FastFlow's intermediate programming level is called the core patterns layer. In this layer, the programmer has access to the other patterns like pipeline and farm that can be possible extended with feedback channels. These patterns are the core used to build and compose other skeletons. Unlike the traditional frameworks, the FastFlow runtime is lock-free, which is important for implementing efficient fine grain streaming applications. Thus, from the implementation' point of view, computations are modeled inside nodes (C++ classes) that will become a process/thread. Also, communications are performed using channels that represent stream data dependency between nodes. The nodes behave as infinite loops to get a task from the input channel (actually a pointer), perform the computation of an internal method provided by the end user, and put the computed results in the output channel [ADKT14, DT15].

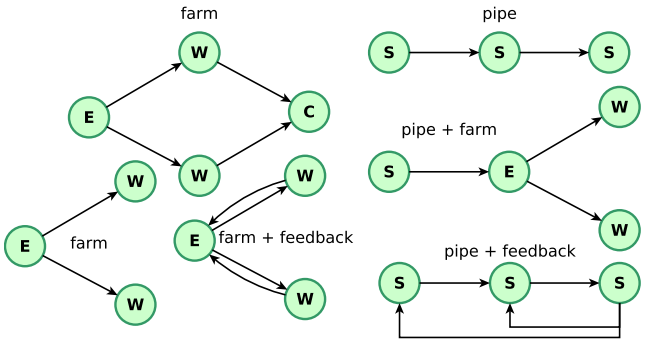


Figure 6.5: FastFlow skeletons from the core pattern layer.

In FastFlow, the farm skeleton is made up of an emitter, one or more workers, and a collector node. The emitter is implemented by the farm scheduler, the workers implement the stream element operations and the collector gathers the stream elements

to be delivered to the output stream. On the other hand, the pipeline skeleton only includes stage nodes. Figures 6.5 illustrate some possible skeleton compositions of pipeline and farm.

At the high-level patterns' layer, FastFlow implements data parallel skeletons such as parallel "for" loops with and without reduce operations [DT14], macro DataFlow [ADA+12], stencil [APD+15] and pool evolution [ACD+15]. These patterns can be used by themselves or together (*e.g.*, as stages of a pipeline or workers in a farm).

Source-to-source code transformation in SPar will benefit from farm and pipeline core patterns. Their level of abstraction provides the flexibility and capabilities necessary to support automatic exploitation of parallelism in multi-cores from SPar annotations. Moreover, FastFlow provides suitable interfaces to tune and optimize performance by supporting customized scheduling implementation, queue access at the building block level, and the possibility of choosing between blocking and non-blocking implementation of communication primitives [ii]. More information about FastFlow usage and parallel programming can be found on the web page tutorial[iii], which is well documented.

6.5 Cluster Runtime (MPI Boost)

We chose MPI Boost instead of FastFlow as our runtime to target clusters because MPI is the "*de-facto*" standard for message passing in parallel programming systems and FastFlow didn't work as well on cluster as on multi-core at that time. However, when using MPI to target the cluster environment, we need to implement the farm and pipeline patterns representing the output of our SPar transformation rules.

Boost is a set of libraries based on the C++ standard aimed to increase productivity [Sch14, Kar05]. We used a subset of Boost to support MPI data serialization. In terms of implementation, it provides us better abstractions and suitable mechanisms to design parallel patterns. In the following sections, we describe how were implemented the pipeline and farm patterns on top of MPI in such a way that they may be used as the output of the SPar transformation rules.

[ii] The blocking mode implement communication channels as non-blocking mode.
[iii] http://calvados.di.unipi.it/storage/tutorial/html/tutorial.html

6.5.1 Farm

Here we discuss how we implemented the farm pattern in MPI. Initially, we implemented the farm conceptually similar to that in FastFlow so that the transformation rules target cluster can be mostly left unchanged in respect to those target multi-cores. However, instead of working with threads and shared memory message queues, we used processes running in distributed machines communicating through message passing.

Figure 6.6 illustrates two typical activity graphs implemented using MPI. As in FastFlow, there will be three kinds of nodes in our MPI farm, which are the Emitter (E), Worker (W), and Collector (C). The emitter is the stream element scheduler that will distribute each element to the workers. Before sending, it serializes each stream item. Then the serialized items are sent in a round robin fashion to workers. When the stream ends, the emitter will broadcast the special end of the stream message used to implement termination.

The worker only receives stream elements and performs sequential operations over them. The results are then sent to the collector, if necessary. Therefore, after receiving a message, the worker will deserialize it and will perform data serialization before sending results to the collector.

The collector is able to gather stream elements from all workers. Consequently, it needs to deserialize stream elements after receiving them in such a way that sequential operations can be computed in the stream items.

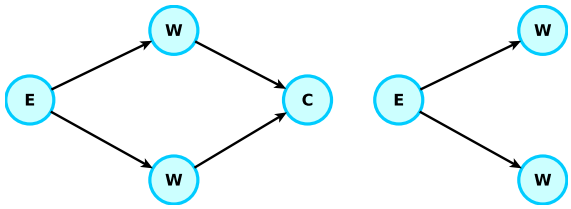


Figure 6.6: MPI farm implementation (circle represents process and arrows represent communications).

Each pattern implementation works with at least one process that is chosen as the root process. To guarantee this occurs, we must control the number of processes from the MPI launcher argument and implement an algorithm to assign the process to different pattern nodes. Therefore, each node will have three "pids" vectors (its, before and after) for implementing the node's connections and control the number of process per node. This is also a general approach for the pipeline and the composition of patterns that we will be referred to later in the text.

6.5.2 Pipeline

In the pipeline, the first stage is responsible for streaming elements to the other stages. The last stage simply receives results of the pipeline computation and the middle stages receive items from previous stages and send intermediate results to the next stages. Therefore, the first stage only marshals the stream items while the last only deserializes, and middle stages have to both deserialize and serialize items and results.

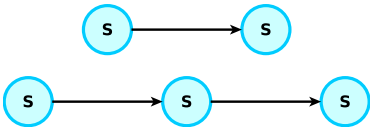


Figure 6.7: MPI pipeline implementation (circle represents process and arrows represent communications).

Unlike farm, in pipeline there will be only one node type, which will be classified as first, middle and last stage. Also, each node can only have a single processes that will communicate with the next one, except in the last stage.

6.5.3 Pattern Compositions

The possibility to compose new parallel patterns is very important in stream parallelism and for the Spar transformations. In FastFlow, the composition is simpler and integrated in the library as previously described in Section 6.4. The option to include feedback channels in a pattern that further improves the possibility to implement more complex patterns. For the current version of SPar, such features are not necessary and will be taken into account in future work. Figure 6.8 illustrates possible parallel patterns obtained by composing pipelines and farms.

The persistent nesting of patterns allows us to build more complex topologies [BC05]. For instance, those on top of Figure 6.8 ($pipe(farm(E,W,C), farm(E,W,C))$) implement the combination of a pipeline with two farm stages. Internally, the arrows (representing the communication in the figure) are implemented through our runtime support based on the "pids" vector described previously. Two other examples are at the bottom of Figures 6.8. The example of $pipe(S, farm(E,W))$ is a pipeline with one sequential stage and a farm stage. $pipe(farm(E,W,C), S)$ is a pipeline where the first stage is a farm and last is a sequential stage. As can be observed, we initially focused mostly on creating pattern

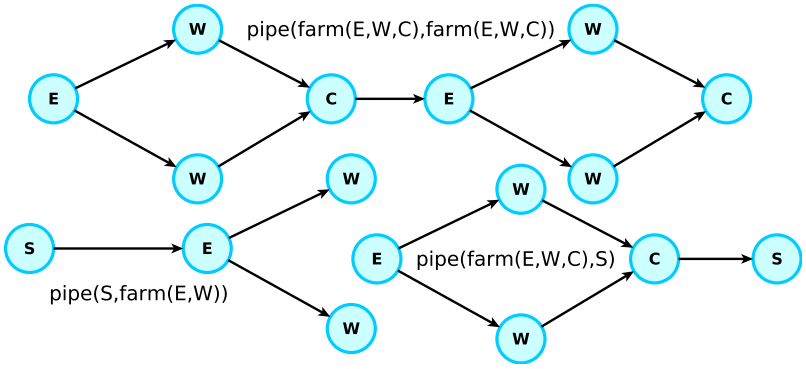


Figure 6.8: MPI skeleton compositions.

variants in such a way that the pipeline is combined with farms. This is a consequence of transformation rules demands that will be discussed on the next section.

6.6 Generalized Transformation Rules

First of all, we introduce some notations that are useful to express SPar semantics:

- T_{id}: is a `ToStream` annotation region associated with an integer variable identifier $(_{id})$.
- S_{id}: is a `Stage` annotation region associated with an integer variable identifier $(_{id})$.
- $\square_{id}$: is a block containing one or more statements, where each block is associated with an integer identifier $(_{id})$.
- I_i: denotes `Input` auxiliary attribute, where $_i$ is an argument list that represents one or more variables with the same or different data types.
- O_i: denotes `Output` auxiliary attribute, where $_i$ is an argument list that represents one or more variables with the same or different data types.
- R_n: denotes `Replicate` auxiliary attribute, where $_n$ represents the number of replicas that correspond to an integer variable.
- [[...]]: denotes an annotation that may have a list of attributes.
- {}: denotes the scope of the sentence.

The transformation rules use farm and pipeline parallel patterns to introduce parallelism as presented in Section 6.3. We can represent these patterns by using a functional style as follows:

farm(): accepts one to three arguments. One argument is the emitter (E), which is the task scheduler. There is also a worker (W) that performs replicas of a given $\square$. Then, it is possible the collector (C), which is a representation of the gather implementation. Each one of the three elements only accepts a single $\square_{id}$ as an argument and only W can be used as a single farm.

pipe(): accepts from two or more arguments, where each argument is $\square_{id}$ or $farm()$ and represents a stage of the pipeline.

The transformation rules from SPar to parallel patterns are based on the following definitions:

Definition-0: when the last $\square$ is annotated with S that contains in its attribute list R_n and O_i, an extra $\square$ is necessary to gather the results. To differentiate from the typical ones, we denoted it as ψ.

Definition-1: when the $\square$ is annotated with S that does not contain in the attribute list R_n, or $\square$ appears alone, the $\square$ can be the argument of *pipe*, E, or C.

Definition-2: when $\square$ is annotated with S that contains in the attribute list R_n, the $\square$ can only be translated to an argument of W. Then, if possible with ***Definition-1***, the predecessor is E and successor is C so that they are arguments of the $farm$ with W.

Definition-3: T is only transformed directly into a $farm$ when the first S annotation in the SPar sentence is the only one of a maximum two S annotations that contains R_n in the attribute list.

Definition-4: T is only transformed directly into a *pipe* when in a sentence of SPar the first S in its attribute list does not have R_n of maximum two S, or when there are more than two Ss.

Definition-5: $farm$ is a stage inside a *pipe* when ***Definition-3*** cannot be applied and $\square$ is annotated with S that contains in the attribute list R_n.

A rule must respect all of the previous definitions. In ascending order of the definitions, transformation rules will be created to translate SPar annotation to parallel patterns. The rules' syntax presents the annotation schema with corresponding transformations.

First, we perform three transformation rules where a T region will be transformed directly into a $farm$ (rules 6.1, 6.2, 6.3). Rule 6.1 is for the $[[T_0]]\{\square_0, [[S_0, R_n]]\{\square_1\}\}$ sentence. We can transform into a $farm$ that has an emitter (which receives $\square_0$) with

worker replicas of $\square_1$ because we induce through definitions 1, 2 and 3. Rule 6.1 can therefore be represented as follows:

$$\begin{array}{c} [[T_0]]\{\square_0, [[S_0, R_n]]\{\square_1\}\} \\ \Downarrow \\ farm(E(\square_0), W(\square_1)); \end{array} \tag{6.1}$$

Rule 6.2 handles an annotation sentence with $[[T_0]]\{\square_0, [[S_0, O_i, R_n]]\{\square_1\}\}$. The relative transformation produces first C that receives ψ (by Definition-0), E have $\square_0$ (inductively by Definition-1), and $\square_1$ assigned to W (inductively by Definition-2). Therefore induced by Definition-3, our transformation can be made a $farm$ as follows:

$$\begin{array}{c} [[T_0]]\{\square_0, [[S_0, O_i, R_n]]\{\square_1\}\} \\ \Downarrow \\ farm(E(\square_0), W(\square_1), C(\psi)); \end{array} \tag{6.2}$$

Another possible sentence of SPar is $[[T_0]]\{\square_0, [[S_0, R_n]]\{\square_1\}, [[S_1]]\{\square_2\}\}$. Rule 6.3 states that we can transform it into a $farm$ (inductively by Definition-3) where the E receives $\square_0$ (inductively by Definition-1), W is $\square_1$ (inductively by Definition-2) and C is the $\square_2$ (inductive by Definition-1). The rule can therefore be written as:

$$\begin{array}{c} [[T_0]]\{\square_0, [[S_0, R_n]]\{\square_1\}, [[S_1]]\{\square_2\}\} \\ \Downarrow \\ farm(E(\square_0), W(\square_1), C(\square_2)); \end{array} \tag{6.3}$$

A sentence like $[[T_0]]\{\square_0, [[S_0]]\{\square_1\}\}$ will be directly transform in a *pipe* inductively by Definition-4 if the first S from the T region does not include any R_n. Rule 6.4 is therefore represented as follows:

$$\begin{array}{c} [[T_0]]\{\square_0, [[S_0]]\{\square_1\}\} \\ \Downarrow \\ pipe(\square_0, \square_1); \end{array} \tag{6.4}$$

The next rules are more complex sentences that we can induce through Definition-4 to become a *pipe* and Definition-5 a $farm$ to become a stage in the pipeline. For example, in Rule 6.5 we know that it will be a pipeline because of S_0 and a $farm$ will become a stage by Definition-5, as S_1 is followed by R_n. As we can induce E by Definition-1 and W by Definition-2, our rule for the $[[T_0]]\{\square_0, [[S_0]]\{\square_1\}, [[S_1, R_n]]\{\square_2\}\}$ SPar sentence is therefore represented as:

$$\begin{gathered}[[T_0]]\{\square_0, [[S_0]]\{\square_1\}, [[S_1, R_n]]\{\square_2\}\} \\ \Downarrow \\ pipe(\square_0, farm(E(\square_1), W(\square_2)));\end{gathered} \tag{6.5}$$

For the $[[T_0]]\{\square_0, [[S_0]]\{\square_1\}, [[S_1, O_i, R_n]]\{\square_2\}\}$ sentence we will use the same definition of Rule 6.5 due to the S_0 and S_1. Yet, we have to generate ψ to make it an argument of C because there is O_i along with R_n in the last S (induction from Definition-0). The resulting transformation is therefore Rule 6.6.

$$\begin{gathered}[[T_0]]\{\square_0, [[S_0]]\{\square_1\}, [[S_1, O_i, R_n]]\{\square_2\}\} \\ \Downarrow \\ pipe(\square_0, farm(E(\square_1), W(\square_2), C(\psi)));\end{gathered} \tag{6.6}$$

Rule 6.7 presents another common sentence in SPar for stream parallelism. The transformation is induced by Definition-4 to become a *pipe* and Definition-5 to make a $farm$ as an argument of *pipe*. Thus, $farm$ is derived from Definition-1 and Definition-2.

$$\begin{gathered}[[T_0]]\{\square_0, [[S_0]]\{\square_1\}, [[S_1, R_n]]\{\square_2\}, [[S_2]]\{\square_3\}\} \\ \Downarrow \\ pipe(\square_0, farm(E(\square_1), W(\square_2), C(\square_3)));\end{gathered} \tag{6.7}$$

Sentence $[[T_0]]\{\square_0, [[S_0, R_n]]\{\square_1\}, [[S_1]]\{\square_2\}, [[S_2, O_i, R_n]]\{\square_3\}\}$ is another complex situation. We have more than two Ss that can be induced by Definition-4 to become a *pipe*. Also, we have to generate ψ due to Definition-0. Consequently, two equivalent rules may be produced by the induction of Definition-5 (Rule 6.8 and 6.9). In Rule 6.8, the $farm$ stages are built from the first R_n looking for the predecessor and successor since it is inductively by Definition-2 and then the second ones such as follows:

$$\begin{gathered}[[T_0]]\{\square_0, [[S_0, R_n]]\{\square_1\}, [[S_1]]\{\square_2\}, [[S_2, O_i, R_n]]\{\square_3\}\} \\ \Downarrow \\ pipe(farm(E(\square_0), W(\square_1), C(\square_2)), farm(W(\square_3), C(\psi)));\end{gathered} \tag{6.8}$$

On the other hand, if we start from the last R_n looking for the predecessor and successor as states in Definition-2, we represent our equivalent transformation rule (Rule 6.9) as follows:

$$\begin{gathered}[[T_0]]\{\square_0, [[S_0, R_n]]\{\square_1\}, [[S_1]]\{\square_2\}, [[S_2, O_i, R_n]]\{\square_3\}\} \\ \Downarrow \\ pipe(farm(E(\square_0), W(\square_1)), farm(E(\square_2), W(\square_3), C(\psi)));\end{gathered} \tag{6.9}$$

Because SPar semantics impose few restrictions, its sentences may be eventually combined in many ways. Even though not all of the possibilities are illustrated, our definitions allow one to implement new and different transformation rules. In this section, we have shown how transformation rules are built from SPar sentences to parallel patterns according to the respective definitions and functional semantics. Therefore, an algorithm that intends to perform new transformation rules must meet our definitions and decide between two equivalents to be applied in the system. In case, the SPar compiler implements such an algorithm to meet all possible transformation rules.

To prototype these rules one must take into account the following notes:

1. *We are assuming that the parallel patterns details like communication and synchronization are already dealt with in the target runtime.*
2. *A stream element is derived from O_i and I_i arguments.*
3. *Optimizations must be implemented at the level of the runtime such as load balancing and scheduling.*

6.7 Source-to-Source Transformations Use Cases

In order to illustrate how source-to-source code is generated and code portability is achieved, this section will present and discuss the aspects related to multi-core and cluster targeting through our transformation rules. The idea is to present a real code example where rules are used to map the SPar code into the parallel code.

6.7.1 Transformations for Multi-Core

During the implementation of the SPar compiler, in addition to the transformation rules, we also have the support of the SPar AST previously described in Section 5.8. Figure 6.9 uses the prime number application to illustrate the transformations in six steps and map where original code is after the code is generated by the SPar compiler. On top of the figure we put the annotated code using SPar attributes with blocks labeled as SPar steps.

First of all, the compiler algorithm starts analyzing the SPar AST, looking for the input and output dependencies so that the data structure inside of the (1) step block can be built. We process the input and output specifications to represent each stream element in a generic way. Then, pieces of the source code and annotation blocks

are transformed according to the transformation rule 6.3. Therefore, we produced the first stage and subsequently the second stage. Note that inside blocks (2) and (3) we must manage data, during this kind of transformation it is necessary to look for the input and output dependencies. Then, we build the block (4) for the stream region that will be used as the emitter. In addition, we must manage when to send the stream elements and control the end-of-stream. Lastly, right after the function definition, the whole structure of the farm is initialized (block (5)). Also, as a result of the transformation in place of the original `ToStream` annotation block, we produced block (6) and update input and output values. We also call the FastFlow runtime actually executing the farm skeleton.

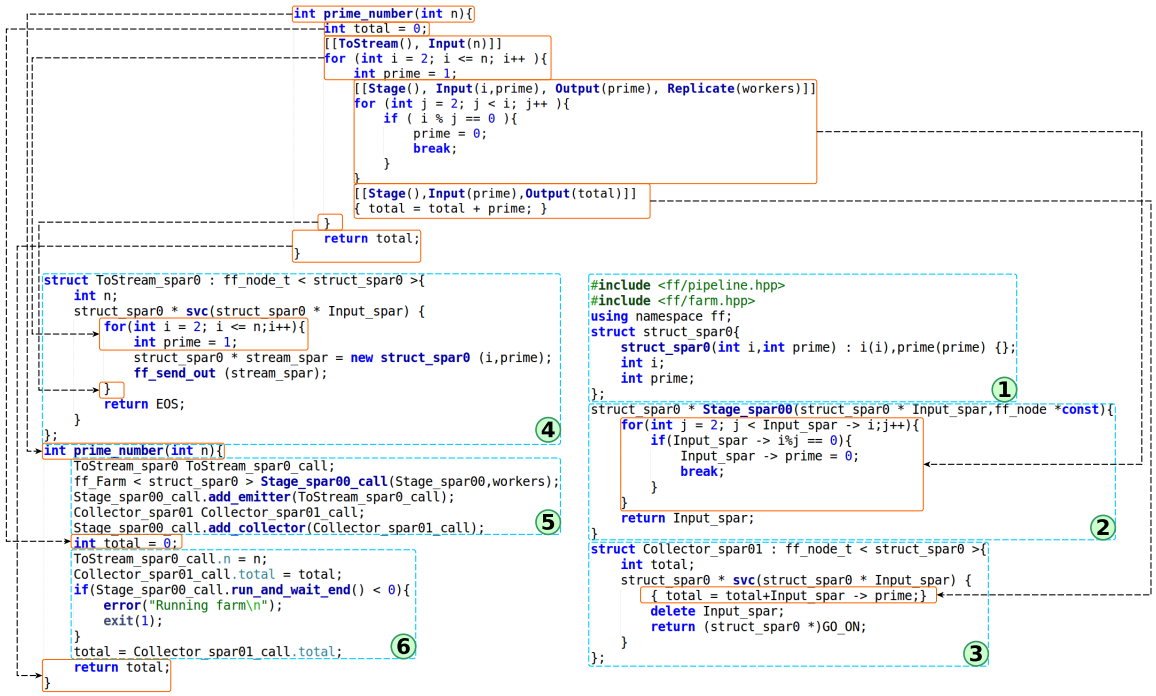


Figure 6.9: Mapping the transformations to FastFlow generated code.

The transformation flow detailed above is the same as other applications, though the transformation rule used may vary. FastFlow supports us with an interface suitable to make stream parallelism possible in SPar, but it does not prevent us from dealing with data management and other C++ low-level aspects such as pointers. On the other hand, it prevents us from creating several different algorithms to allow for different scheduler and stream ordering. For instance, when one sets the `spar_ondemand` optimization flag, we only need to add at the end of the block (5), a routine that sets the on-demand scheduler without changing the rest of the structure. Similarly, the `spar_ordered` does not require changes since the only thing to do is to use a different method to initialize the farm. The implementation of the `spar_blocking` optimization is even simpler, because it is achieved by adding an extra flag when assembling the code with GCC. The meanings of these optimization flags are as follows:

- `spar_ondemand`: used to generate the on-demand scheduler.
- `spar_blocking`: used to activate the FastFlow blocking mode.
- `spar_ordered`: used to say that output stream elements must be delivered respecting the input order.

We implemented these optimization flags along with the default transformations to speed up different kinds of applications. In the future, we expect to have more possibilities to optimize the code during compilation. Later in Chapter 7, we evaluate the impact of these optimization flags on the performance, when used alone or in conjunction with each other.

6.7.2 Transformations for Cluster

Some technical reasons did not allow us to use FastFlow for the runtime in the cluster environment: no stable version, few documentation, and no available launcher. We also had a greater motivation for using a different runtime to target clusters which was an evaluation of the generality of the proposed transformation rules including possible challenges that we would face with a runtime which does not primitively support stream parallelism. Consequently, the first challenge given was to prepare an intermediate interface for the farm and pipeline patterns, implement the scheduler and data serialization.

Unlike the multi-core transformations, we are still in the process of completing the implementation of the cluster support (we plan to enable it simply by using `spar_cluster` compilation flag). The code shown has been hand generated from the rules presented in Section 6.6, as we still have not completed the tools to generate it automatically.

To demonstrate the differences between cluster and multi-core code generation as well as how we eventually achieved code portability, Figure 6.10 uses the same prime number application. The annotated code can be found at the top of the figure and we organized the generated code in blocks labeling the step sequence. Because we wanted to concentrate only on the essential parts, some blocks of code are hidden and represented through white boxes.

Similar to the compiler algorithm for multi-core generation, the first step was to build the stream structure by looking for the data dependencies through the SPar AST. We produced the first block understanding the input and output specification of the stages. Also, since it is necessary to send data through the network, we used the Boost library to implement data serialization. Therefore, only standard C++ types are easily handled by Boost, while pointer and other complex structures require manual

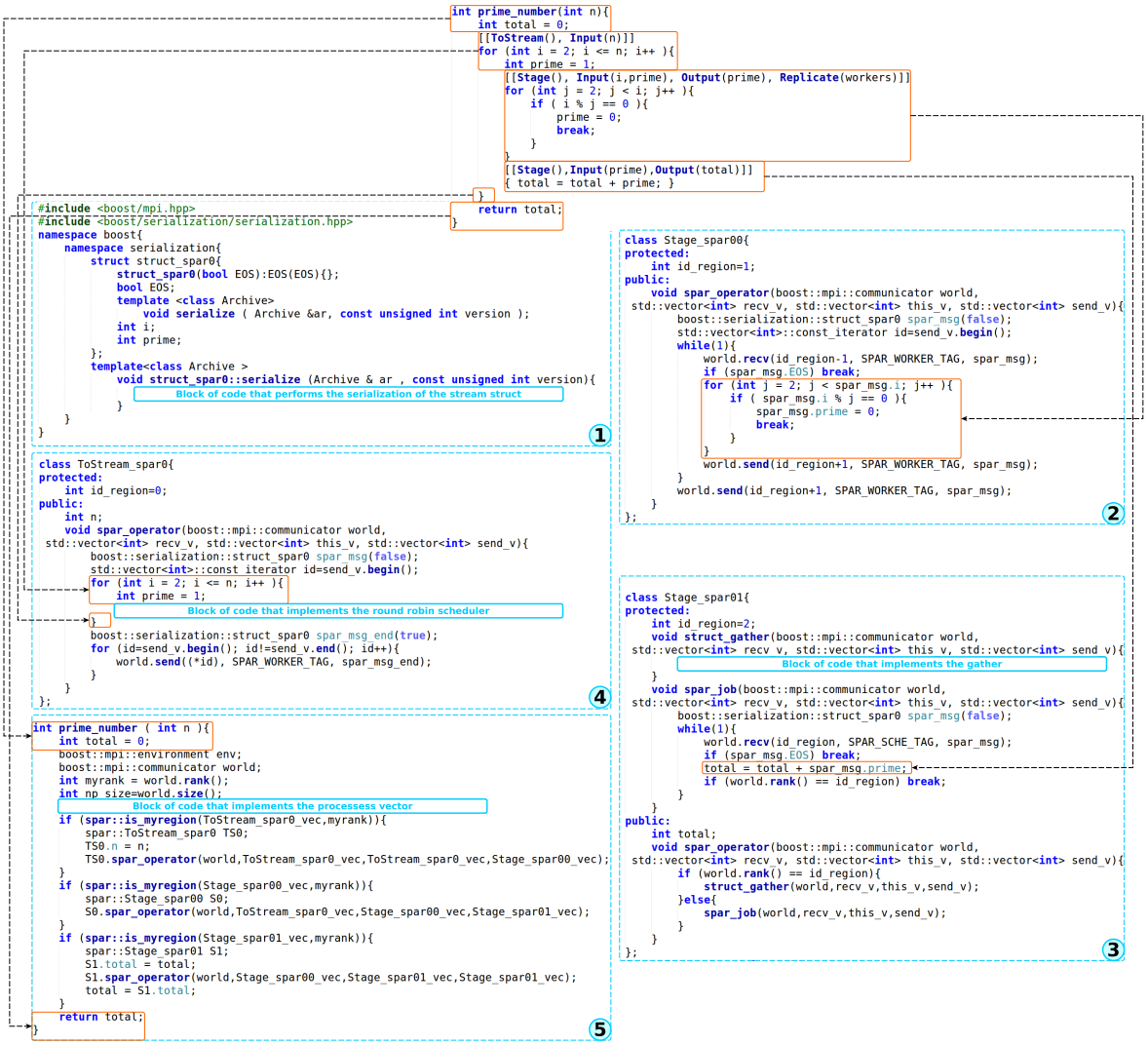


Figure 6.10: Mapping the transformations to MPI generated code.

implementation of the serializations. This type of verification must be done internally when analyzing the data types of the input and output on the AST.

The next step is to apply the corresponding transformation rule (Rule 6.3). As a consequence, we transform the first stage (step (2)) and then the subsequent stage (step (3)). The first stage will be replicated so that it becomes a worker. The last one must generate a gather function and implement a generalized protocol to read from all replicas each one of the stream elements, becoming a collector. In contrast, the first stage must only manage the data like in multi-core and generate the intermediate interface with the support of the MPI Boost library.

The fourth step produced the emitter that originates from the stream region. This generation requires two special tasks: sending the stream elements in a round robin fashion and controlling the end-of-the-stream. The difference with respect to

FastFlow runtime is the generation of the scheduler. The last step is to transform the compound statement of the function definition that was annotated by SPar. It will generate all the MPI code, an algorithm to fill the "pids" vector and call all generated methods. This vector is necessary for indicating which process will run in a given region as well as for implementing the relevant communication protocol.

During the presentation of this illustrative example we can highlight that it requires significant effort to prepare a compatible interface using the MPI library runtime for implementing stream parallelism in cluster environments. However, we were able to generate code based on the transformation rules. In the experiments, we give performance insights addressing functionality and performance. The most important factor is that code portability is possible through recompilation of the program. No modification or additional attributes are needed in the source code.

6.8 Summary

This chapter introduced essential features of the used (FastFlow) and developed runtime (created on top of MPI) as well as parallel patterns. We provided a new contribution with the generalized transformation rules aiming at code portability. They were formulated in such a way we can translate the code by hand, integrating the rules into the compiler algorithm. We demonstrated through the real use cases that the whole process not only works, but it is also straightforward. Moreover, due to the higher abstraction FastFlow presents for developers and the fact that it already provides farm and pipeline patterns, targeting FastFlow has been demonstrated to be much easier than targeting MPI.

Part III

Experiments

7

Results

This chapter present the results of the experiments for evaluating and comparing performance and coding productivity.

Contents

7.1 Introduction

This chapter will introduce a set of experiments relative to five carefully chosen applications (described in Section 7.2.1) aimed at evaluating and comparing productivity, performance and code portability. Firstly, we will describe our methodology for the experiments. Then, performance and productivity are tested in the multi-core environment. We also perform a comparison with state-of-the-art parallel programming frameworks (OpenMP, TBB, FastFlow and eventually Pthreads). Subsequently, we present the results of two applications (already tested on Multi-core) derived from the same annotated source code and targeting MPI runtime.

7.2 Experimental Methodology

This section presents the methodology used to conduct the experiments.

7.2.1 Benchmarking Setup

Our criteria for choosing the applications were diversity and real world applicability. Consequently, the suite of applications we picked are as follows:

- **Sobel Filter**: Applying filters over a set of images is a recurrent operation in image processing applications. It can be used as a representative example of real world applications for evaluating expressiveness, parallelism exploitation and performance. Also, due to its well defined structure, it allows us to reproduce different versions of implementation to test the programming frameworks' flexibility as well as different workload performances.

- **Video OpenCV**: We intentionally implemented this application using OpenCV library because it is widely used in video streaming processing while demonstrates that SPar may be used in conjunction with standard libraries. The program structure is representative of other real world stream applications such as those commonly found in the networking domain. The very same schema used to implement video operation applications, may be used to implement network package analysis.

- **Mandelbrot Set**: This provides an interesting problem to be solved using stream parallelism, because it is originally designed to be a data parallel computation. In fact, it is an scientific application, where iterative screen visualization is used to support the scientist to follow the results. A similar pattern is present in medicine and biology applications, where in case, the scientists could be interested in following the DNA results on the screen while computing more samples.

- **Prime Numbers**: This is a mathematical algorithm used in the scientific community and real world cryptography applications. It is also a well know problem in the parallel programing field for testing the performance of the programming framework, especially as far as load balancing is concerned. Although it is not a stream application, it allows us to evaluate SPar's expressiveness and performance compared to the state-of-the-art tools.

- **K-Means**: One motivation for using K-Means is justified by the significant current research effort spent to achieve quick insights from big data. K-Means algorithm is recurrently applied in data analysis to classify data. Secondly, it is a challenge because it requires SPar to implement stream and data parallelism at the same time.

7.2.2 Tests Environment

In order to evaluate the performance of the applications, we used two different machine architectures. Table 7.1 describes the characteristics of the Pianosau machine environment, used to run the multi-core experiments. On the other hand, Table 7.2 presents Dodge cluster machines' configurations, composed of four identical nodes that has been used to run our "cluster" experiments.

Characteristic	Description
Processor model	Intel(R) Xeon(R) CPU E5-2650.
Processor performance	Two CPU sockets, each one with 8 cores and 16 threads with frequency base of 2.00GHz.
Memory settings	NUMA DDR3 32GB and smart cache of 20MB.
Disk capacity	local hard driver 385GB
Network	Two Intel Corporation I350 Gigabit Network Connection
Operating System	CentOS release 6.3 (64 bits).
Compiler	GCC 5.3.0 with -O3 optimization flag.
Runtime	FastFlow library (version 2.1.0)

Table 7.1: The Pianosau machine configurations.

Characteristic	Description
Processor model	Intel(R) Xeon(R) CPU X5560.
Processor performance	one CPU with 4 cores and 8 threads with frequency base of 2.80GHz.
Memory settings	DDR3 24GB and smart cache of 8MB.
Disk capacity	local hard driver 657GB and more 657G in the NFS
Network	Two Intel Corporation 82574L Gigabit Network Connection.
Operating System	Ubuntu 14.04.3 LTS (64 bits).
Compiler	GCC 5.3.0 with -O3 optimization flag.
Runtime	Boost library (version 1.6.0).

Table 7.2: The Dodge cluster machines' configuration (total of 4 nodes).

7.2.3 Performance Evaluation

The experiments in performance evaluation take into account different metrics that are listed below:

- **Completion time**: Is the time that an application takes to start and finish its computations. It will be used to calculate subsequent metrics. We obtain the time before starting the stream region and after it ends, then we subtract the end by start time to get the completion time.
- **Latency**: Is the response time in milliseconds to process each stream element [Gre14].
- **Throughput**: Is relative to the amount of stream elements that the application is able to process in a given time, which is also called the rate of work [Gre14]. The throughput is the inverse of service time that is based on Little's Law, dividing the completion by the number of elements processed in this period [Gus11].
- **Speedup**: This metric is based on Amdhal's law to represent the scalability of the application [MRR12]. In this case, we measured the throughput speedup relative to the number of replicas (parallelism degree).
- **Efficiency**: This is also based on the Amdhal's law to represent how efficiently the application uses machine resources. In this case, we provide this metric as it is usually presented in HPC, by first calculating the speedup relative to the completion time and then dividing the resultingt speedup by the amount of replicas used.

- **Energy consumption**: This is the energy used by the application in a specific interval of time. It is expressed in Joules and given through the integral of power over time. In this case, we used the Intel RAPL (Running Average Power Limit)[i] driver which read from Linux hardware counters.

- **Memory usage**: This is the amount of memory that an application used during its execution. This metric is collected by reading the process event files (for example, the metric can be extracted from */proc/pid/status*).

- **Cache misses**: These occur when thread or processes try to read on cache and do not find data, which then requires them to go to the main memory. The amount of times this happens during the program execution is called the cache miss rate [HP11]. They are collected through the Linux performance library [ii].

Each application is executed 10 times for each number of replicas (parallelism degree) tested in order to calculate the arithmetic mean of these values. Also, the standard deviation was calculated based on 10 executions.

7.2.4 Coding Productivity Instrumentation

The applications used to test productivity are the same ones used for performance evaluation. There are many measurements we could have used for evaluate productivity [SS96], but we evaluated coding productivity by measuring the physical source lines of code, which in its simplicity already provide an indirect measurement of the amount of time needed to develop the code. Therefore, the applications were carefully implemented, modifying only the parts of the code that had to be parallelized. To ignore comments and count only the real code line, we used SLOCCount tool[iii].

Because the number of lines is only a quantitative metric, we also analyzed the source code implementation and parallelism modeling. This evaluation is a comparative way looking for the programming model and associating if there is low-level programming and code rewriting. The idea is to demonstrate the drawbacks and advantages of SPar with respect to the state-of-the-art parallel programming tools.

[i] https://01.org/rapl-power-meter
[ii] http://man7.org/linux/man-pages/man2/perf_event_open.2.html
[iii] http://www.dwheeler.com/sloccount/

7.3 Multi-Core Environment

This section performs the experiments in the multi-core environment to evaluate and compare of SPar's performance and coding productivity.

7.3.1 Sobel Filter

Applying a filter over images is a recurring task in the image processing field. To represent these types of applications, we benchmark the Sobel filter. Listing 7.1 presents a pseudo-code application. It is possible to observe that the program is reading all files from a given directory and applying the Sobel filter over bitmap images only. Using SPar's methodology, the recommendation is to first identify the stream region. Therefore, we start from the "while loop", obtaining a new file descriptor iteration by iteration until the directory is empty. After this, we can simply identify what will be consumed and produced by this region, specifying input and output attributes with the respective variables as annotated in line 5.

Inside the `ToStream` region, there are three stream operations that are read, filter and write, which configures three stages. Before starting the stream operations, each file is preprocessed to get its name and extension. Then, files that are not bitmap extensions are ignored so that it is possible to count how many images were read. When reading a bitmap file, the program stores image information (*e.g.*, image size, height, and width) and loads all image bytes in the memory. Next it applies the Sobel filter and writes the results on the disk.

Even though SPar allows us to annotate this application in different ways, we demonstrate the two most efficient alternatives in Listing 7.1 and 7.2. In the first we annotate the filter and write operation regions as stages and let the stream region to read and produce for the other stages. Consequently, our stream will be different from what `ToStream` was consuming because it is producing another stream inside that are image information and the buffer containing all image bytes. Therefore, we can easily identify and annotate the input and output. Moreover, as the most time-consuming part is to apply the Sobel filter and it can operate independently for each new stream element (image), the `Replicate` attribute can be assigned to speedup performance. Note that we could also put the replicate attribute in the last stage because it could operate independently, but this second version may add extra overhead because the application is also reading from the disk. In contrast, when there is a sophisticated storage architecture, adding the replicate at the last stage may improve performance.

Thus, this is one example that we can use to illustrate the importance of SPar's flexibility for taking advantage of the hardware resources.

```
1  int main(int argc, char *argv[]){
2    //open directory ...
3    DIR *dptr = opendir(...);
4    struct dirent *dfptr;
5    [[spar::ToStream,spar::Input(dptr,dfptr,tot_img,tot_not), spar::Output
       (tot_img,tot_not)]] while((dfptr = readdir(dptr)) != NULL){
6      //preprocessing
7      if (file_extension == "bmp"){
8        //Reads the image ...
9        tot_img++;
10       image = read(name,height,width);
11       [[spar::Stage, spar::Input(height,width,image),spar::Output(
       new_image), spar::Replicate(workers)]]{
12         //Applies the Sobel
13         new_image=sobel(image,height,width);
14       }
15       [[spar::Stage, spar::Input(newname,height,width,new_image)]]{
16         //Writes the image ...
17         write(newname,new_image,height,width);
18       }//end stage
19     }else{
20       tot_not++;
21     }
22   }//end of stream computing
23   //end processing
24   return 0;
25 }
```

Listing 7.1: Sobel Filter using SPar (pipe-like).

Listing 7.2 represents a slightly different version of Listing 7.1. It demonstrates SPar's flexibility ad expressiveness which allows to perform minimal changes in the sequential code to produce different annotation schemes and activity graphs (see on Section 5.6 on Figure 5.2). The only changes consist in commenting out lines 14 and 16 of Listing 7.1 and putting the stage annotation before the read operations. Also, we updated the input and output attributes because `ToStream` now produces only the "name" (original file name) and "newname" (result file name) during the preprocessing. Finally, since nothing needs to be produced inside the stage to survive the stream region, the output attribute is not necessary any more.

In the next sections we will first present the performance experiment results relative to the original application and applications obtained annotating the original code with SPar attributes. Subsequently, producing parallel code targeting multi-cores through our transformation rules described in Chapter 6.6, which were implemented in the SPar compiler. After, we evaluate coding productivity as well as compare performance with state-of-the-art frameworks, where SPar is actually generating

FastFlow code. Therefore, plain FastFlow implementations may benefit from the very same optimizations implemented by SPar flags, which are stressed in Section 7.3.1.1.

```
int main(int argc, char *argv[]){
  //open directory ...
  DIR *dptr = opendir(...);
  struct dirent *dfptr;
  [[spar::ToStream,spar::Input(dptr,dfptr,tot_img,tot_not), spar::Output
   (tot_img,tot_not)]] while((dfptr = readdir(dptr)) != NULL){
    //preprocessing
    if (file_extension == "bmp"){
      tot_img++; //counts the number of images
      [[spar::Stage, spar::Input(name,newname), spar::Replicate(workers)
    ]]{
        //Reads the image ...
        image = read(name,height,width);
        //Applies the Sobel
        new_image=sobel(image,height,width);
        //Writes the image ...
        write(newname,new_image,height,width);
      }//end stage
    }else{
      tot_not++; //count the number of images not read
    }
  }//end of stream computing
  //end processing
  return 0;
}
```

Listing 7.2: Sobel Filter using SPar (farm-like).

7.3.1.1 SPar Performance

We differentiate the previous versions of the Sobel filter application by using the word "pipe" when referring to Listing 7.1 and not using "pipe" term when referring to Listing 7.2. We put in the graphs an abbreviation of the compilation flags used to differentiate from generated source-to-source code such as follows:

- `spar_blocking` (*blk*): used to activate the FastFlow blocking mode.
- `spar_ondemand` (*ond*): used to generate the on-demand scheduler.

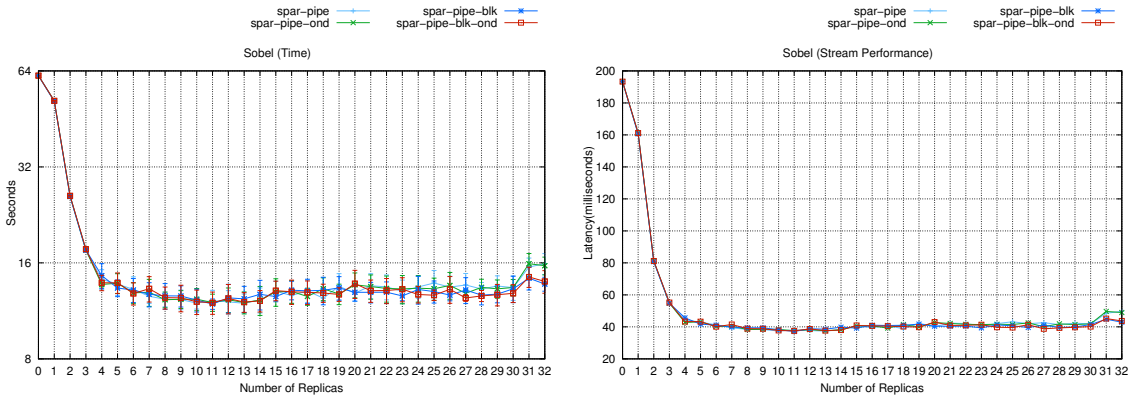
Therefore, the syntax of the legend in the graphs is *spar-[version]-[compiler flag]*. Moreover, the Y axis always is relative to the performance metric and X axis is relative to the number of replicas instantiated for testing the application behavior. Replica 0 represents the source code (sequential). All metrics are exclusively related to the

stream region from its start to its end. The machine used for these experiments was Pianosau, which was previously described in Section 7.2.2.

We setup two kinds of workloads to stress the application. For the balanced workload 320 images with 3000x2250 resolution were used. On the other hand, the unbalanced workload was composed of 1280 images and four different resolutions were selected (800x600, 1024x768, 1600x1200 and 3000x2250).

Figure 7.1 and 7.2 shows the performance results concerning the metric completion time (a) and latency per image (b). Considering completion time, it is not possible to significantly distinguish the best optimization flag used in the both versions due the standard deviation illustrated in the graphs (using error bars), which are overlapping.

When comparing both versions, we can see that adding more stages significantly impacts latency when more replicates are instantiated in this application. This occurs specifically from 16 replicas that coincide with the start of the hyper threading facilities of this machine. However, we can observe that the application performs better if it uses more stages when there are enough resources available. Due to the pipeline, even performing a single replica can significantly reduce latency and completion time with respect to Listing 7.2's version.

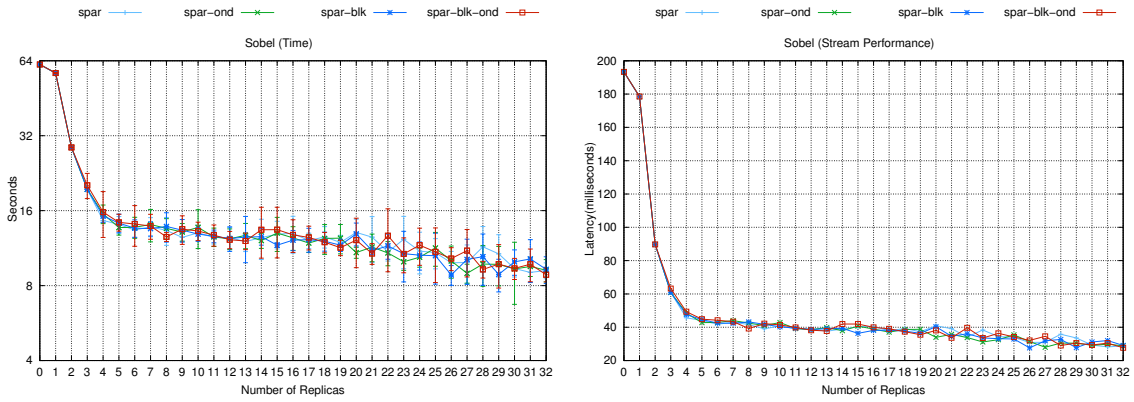


(a) Sobel Filter execution times.

(b) Sobel Filter latency.

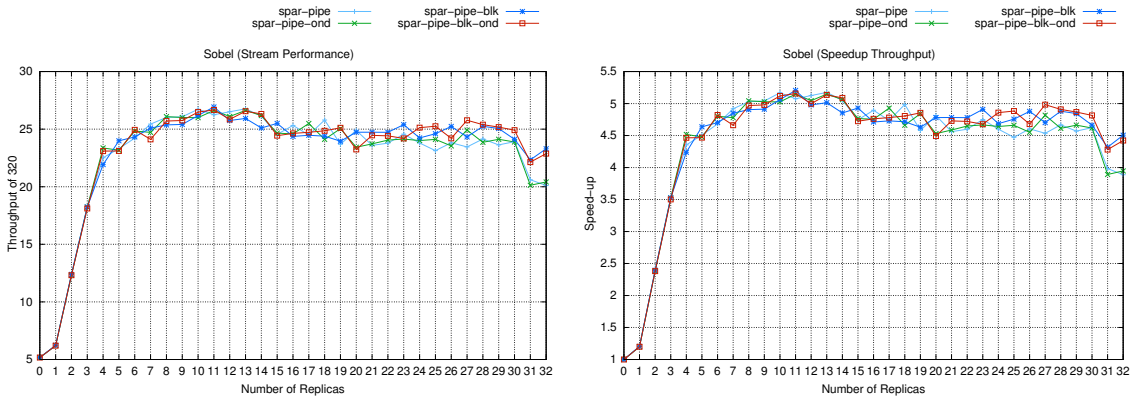
Figure 7.1: Time performance using balanced workload (Listing 7.1)

Another important metric in stream parallelism is the throughput that is presented in Figure 7.3 and 7.4. Graph (a) illustrates the throughput and graph (b) the throughput speedup. These results demonstrate how many images the application is able to process at a given time that can be associated with Figure 7.1 and 7.2. Unlike the time and latency metrics, throughput shows a significant difference among the versions implemented using SPar. For instance, picking up the highest throughput rate of both versions, Listing 7.2's version is able to process 10 more images (with 32 replicas). However, Listing 7.1's version requires less replicas to achieve its highest throughput rate (with 11 replicas).



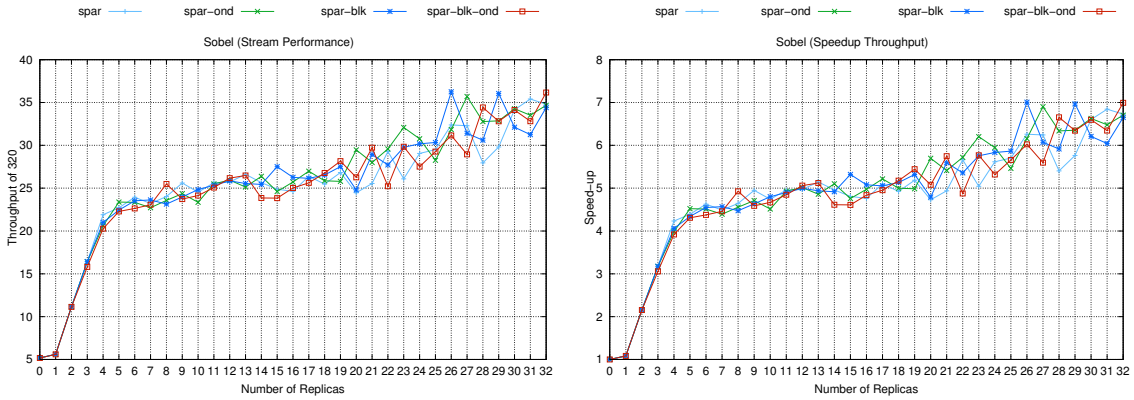
(a) Sobel Filter execution times. (b) Sobel Filter latency.

Figure 7.2: Time performance using balanced workload (Listing 7.2)



(a) Sobel Filter throughput. (b) Sobel Filter throughput speed-up.

Figure 7.3: Stream performance using balanced workload (Listing 7.1)



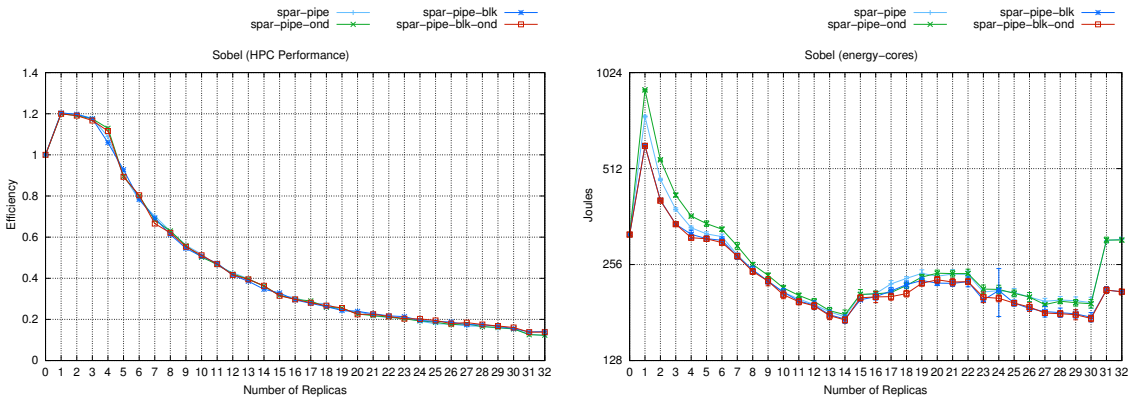
(a) Sobel Filter throughput. (b) Sobel Filter throughput speed-up.

Figure 7.4: Stream performance using balanced workload (Listing 7.2)

High-Performance Computing (HPC) performance concerns more on CPU efficiency and energy consumption of the application. In both versions in Figure 7.5

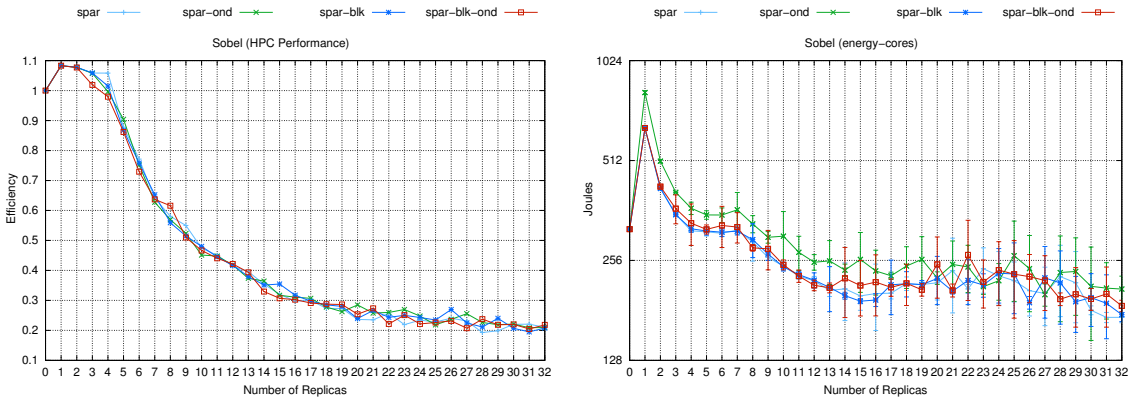
and 7.6, (a) represents efficiency and (b) demonstrates the energy consumption only concerning the CPU cores [iv]. We can observe that this application does not present an efficient usage of the CPU starting from 5 replicas up to 32. In fact, it decreases to 20 percent for the Listing 7.2's version and even more for Listing 7.1's version. These results are due to the fact that the disk is a bottleneck for this application, since two of the three stream operations are performed primarily on the disk (read and write).

When analyzing the energy spent to achieve efficiency, it is again evident that the second version provides the best balance between effectiveness and power consumption. On the other hand, when looking at the optimization flags, the results show that when an `spar_ondemand` flag is added, the application consumes more energy than in case we use standard or different kinds of flags.



(a) Sobel Filter HPC efficiency. (b) Sobel Filter CPU cores energy consumption.

Figure 7.5: HPC performance using balanced workload (Listing 7.1)



(a) Sobel Filter HPC efficiency. (b) Sobel Filter CPU cores energy consumption.

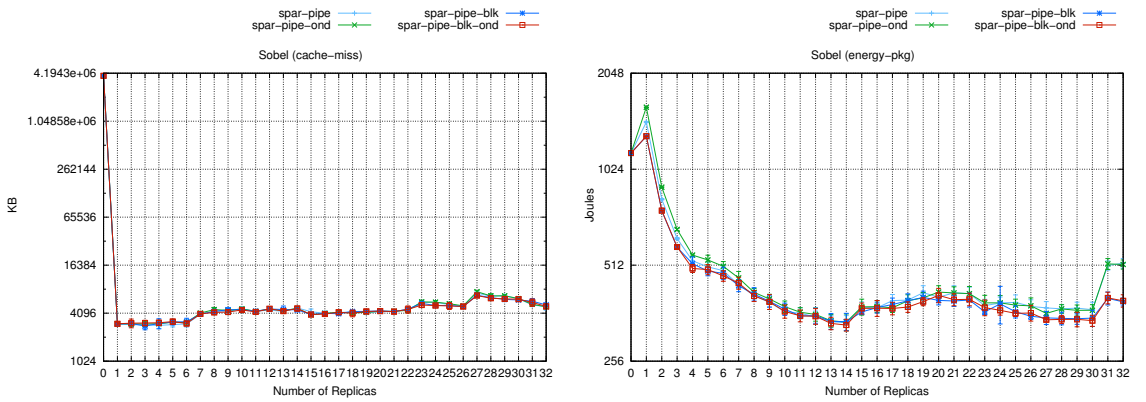
Figure 7.6: HPC performance using balanced workload (Listing 7.2)

Figures 7.7 and 7.8 shows the HPC metrics: cache misses (graph a) and the energy consumption from the entire CPU socket (graph b), which includes the cache

[iv] This metric was collected by using hardware counters.

memory. When comparing the versions, Listing 7.2 had less cache misses, but the energy consumption of the CPU socket was higher because it provides better efficiency (Figure 7.6(a)).

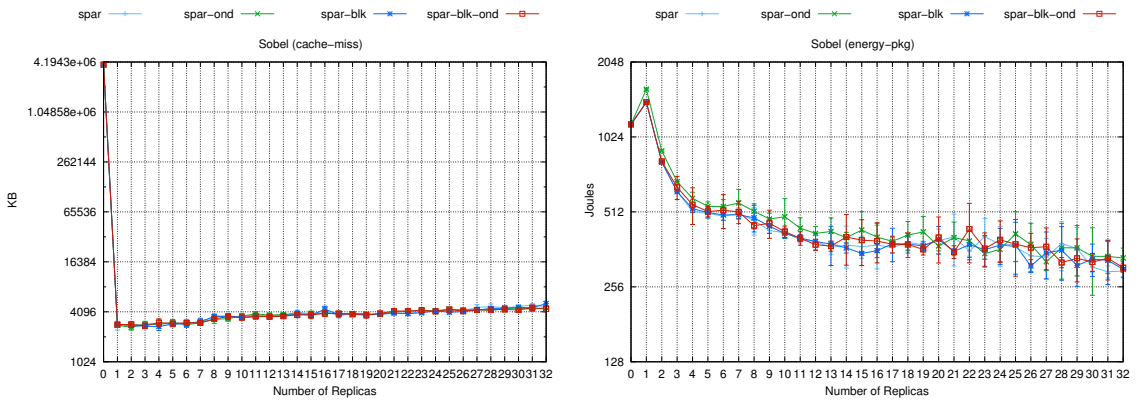
Analyzing the optimization flags, we can observe that cache misses were not significantly different. Also, energy consumption does not present significant differences as before (Figure 7.6(b) and 7.5(b)) when we were only looking at the CPU core consumption. The `spar_ondemand` optimization flag result is a consequence of generating code that stresses the CPU more than other resources from the energy consumption perspective, yet efficiency and cache miss are not significantly affected.



(a) Sobel Filter cache misses.

(b) Sobel Filter socket energy consumption (cache and CPUs).

Figure 7.7: CPU Socket performance using balanced workload (Listing 7.1)



(a) Sobel Filter cache misses.

(b) Sobel Filter socket energy consumption (cache and CPUs).

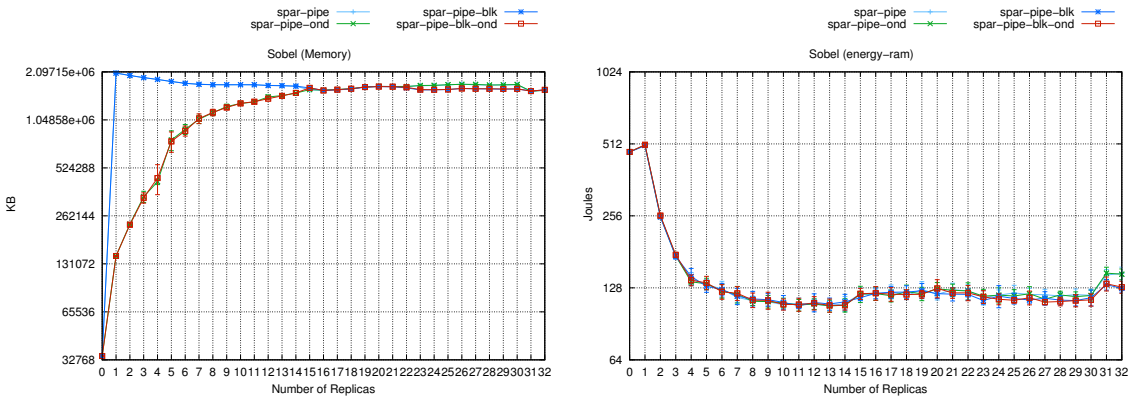
Figure 7.8: CPU Socket performance using balanced workload (Listing 7.2)

The last resource that we analyzed was memory. Memory efficiency is essential for avoiding unnecessary traffic due to the memory swaps that will eventually lead to performance losses. Increasing memory usage is normal since we add replicas to a

code region during parallelization. Figure 7.9 and 7.10 illustrate memory usage (graph a) and energy consumption (graph b).

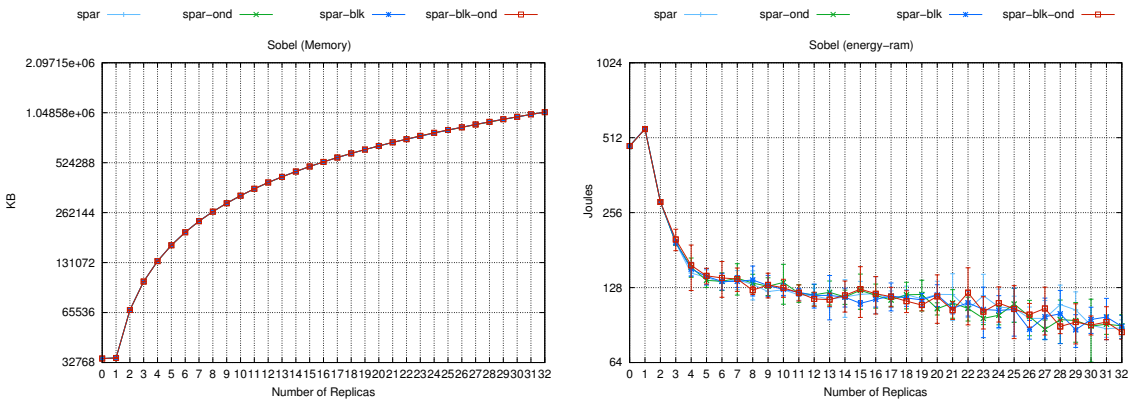
There are significant differences in memory usage when the results of the application versions are compared. While Listing 7.2 version increases linearly and uses less memory to exploit parallelism, Listing 7.1 version starts with much memory and increases until it becomes stable, using more memory to exploit parallelism. This is a consequence of adding one more stage contributing to pipeline parallelization in the application. However, the results of energy consumption do not reflect such results as memory usage does not seem to affect power consumption in this case.

Next we compare the effect of using different optimization flags. Only Figure 7.9(a) the `spar_blocking` flag revealed a significant impact. In this case, from beginning up to 14 replicas much more memory is used. This result is a consequence of the round robin scheduler that combined with the blocking mode that requires the use of much more memory for the FastFlow queues.



(a) Sobel Filter memory usage. (b) Sobel Filter memory energy consumption.

Figure 7.9: Memory performance using balanced workload (Listing 7.1)



(a) Sobel Filter memory usage. (b) Sobel Filter memory energy consumption.

Figure 7.10: Memory performance using balanced workload (Listing 7.2)

The results regarding unbalanced workloads can be found in the Appendix chapter in Section A.1.1. In general, they do not present many contrasts when comparing all of the metrics of the previously balanced workload to an unbalanced workload. Latency and throughput were expected to be lower and higher respectively because smaller image sizes were used. However, the throughput speedup was lower for Listing 7.2 (compare Figure 7.4(b) than the A.13(b)) version as well as the HPC efficiency due to scheduling and disk overhead. With respect to the optimization flags, the `spar_ondemand` demonstrated better speedup and efficiency in the most cases. This result was also expected because on demand scheduling performs better with unbalanced workloads.

7.3.1.2 Productivity Comparison

In this section, we will compare code productivity with other alternatives for parallel programming. One way to look for productivity is to evaluate the physical Source Line of Code (SLOC) that was necessary to support parallelism in the application. Figure 7.11 plots the percentage difference with respect to the sequential version (considering the whole application's code). The implemented versions using SPar (spar), OpenMP (omp), TBB (tbb), and FastFlow (ff). We can observe that programming frameworks designated to stream parallelism (TBB and FastFlow) require much more code intrusion. Even though OpenMP is not particularly suitable to support stream parallelism, this application provides specific characteristics that enable implementation by using task parallelism. Consequently, it produces more code intrusion than when parallelizing data parallel computations. As SPar builds on the standard grammar, it allows us to reuse C++ iteration statements when annotating a stream region or stage. In turn, this enables us to achieve better productivity than OpenMP.

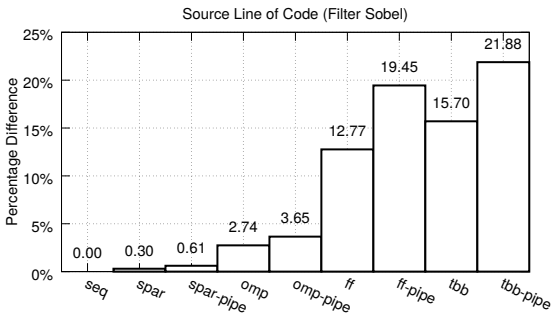


Figure 7.11: Source line of code for filter Sobel application.

In addition to SLOCs, it is important to analyze the conceptual productivity concerning implementation details and particular characteristics from the programming framework that need to be understood. For instance, OpenMP is not designed to

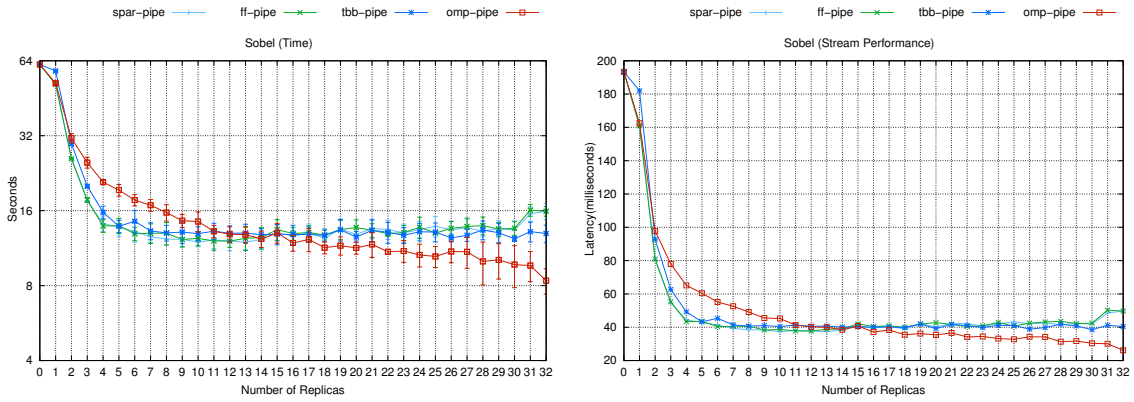
implement stream parallelism. Therefore, it is necessary to understand more about its API and low-level programming model. Listing A.1 implements a comparable pseudo code version of Listing 7.2, where OpenMP uses terms like single, parallel, and task that are considered low-level parallelism exploitation directives and not stream domain friendly terms. It will influence code productivity because developers have to learn terms and programming models that are not friendly to their domain. Moreover, OpenMP was able to produce a kind of pipeline-like implementation because stages can operate in a DataFlow graph mode. The resulting pseudo code version is presented in Listing A.2, which in principle can be compared with Listing 7.1. Thus, users must again deal with low-level terms and explicitly define data dependency.

The comparison between SPar, FastFlow and TBB implementations (Listings A.3, A.4, A.5, and A.6), reveals that two (FastFlow and TBB) provide less productivity because the original code must be restructured and more code is needed. However, unlike OpenMP, FastFlow and TBB terms are more friendly to the domain user as well as their interface.

7.3.1.3 Performance Comparison

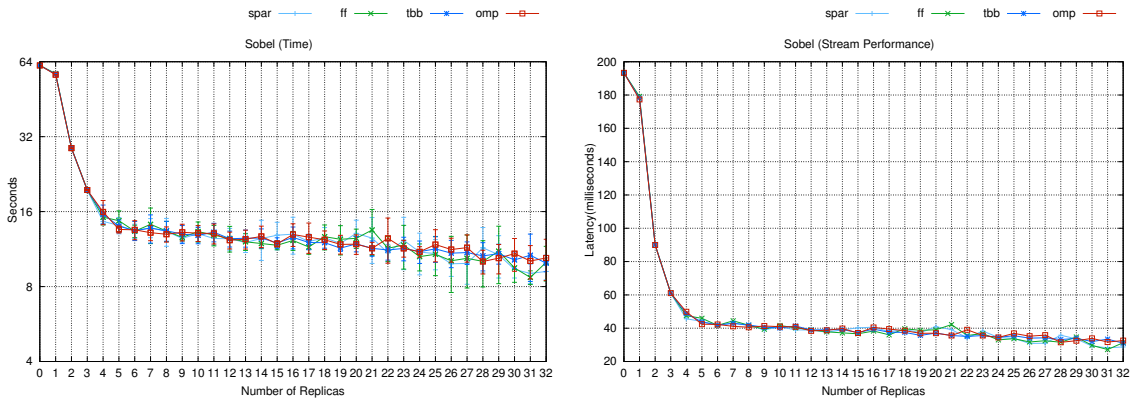
A performance comparison is a way to identify if the rules for source-to-source transformation are efficient with respect to the state-of-the-art tools. Therefore, all other compared versions are implemented in an optimal way, while in SPar we plot the default version without any optimization flag. Experiments used a balanced workload as discussed in the previous section (Section 7.3.1.1). Figures 7.12 and 7.13 present the two implementation versions, comparing the completion time (graph a) and the latency (graph b). In summary, the results of Figure 7.12 demonstrate that SPar achieved similar completion time and latency with respect to state-of-the-art stream parallelism tools (FastFlow and TBB). However, SPar outperforms OpenMP up to the point we start using the hyper threading resources of the machine. After that point, it loses against OpenMP (out against TBB and FastFlow too). Note that the OpenMP error bars present a higher standard deviation when running on hyper threading facilities.

In the second version (Figure 7.13), there is an equilibrium among the programming frameworks in completion time and latency. Note that OpenMP has the same results in Figure 7.12 even when different annotations are used. Consequently, OpenMP would not benefit from a pipeline parallelism such as SPar when there are enough machine resources (2 up to 15 replicas). The disadvantage of OpenMP in this application is that expressiveness is not reflected in performance, because it requires more replicas to achieve better performance than SPar in the pipeline-like implementation version.



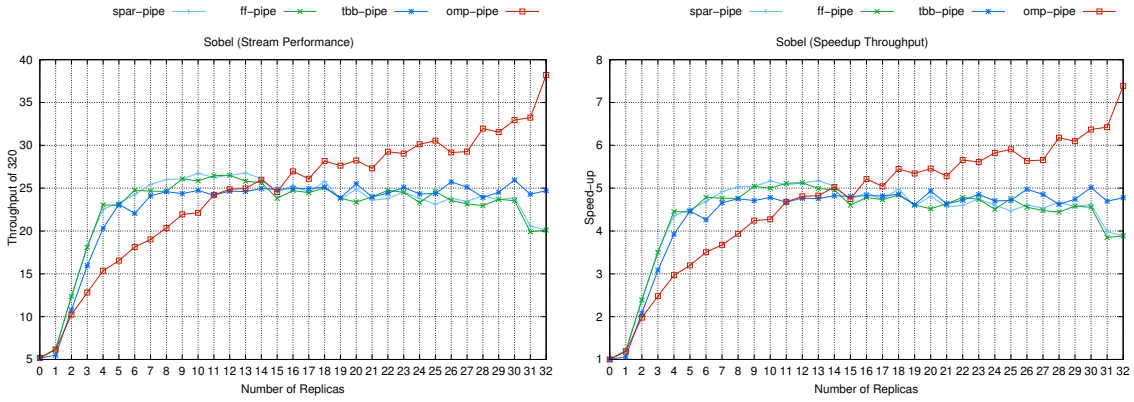
(a) Sobel Filter execution times. (b) Sobel Filter latency.

Figure 7.12: Time performance comparison using balanced workload (Listing 7.1)



(a) Sobel Filter execution times. (b) Sobel Filter latency.

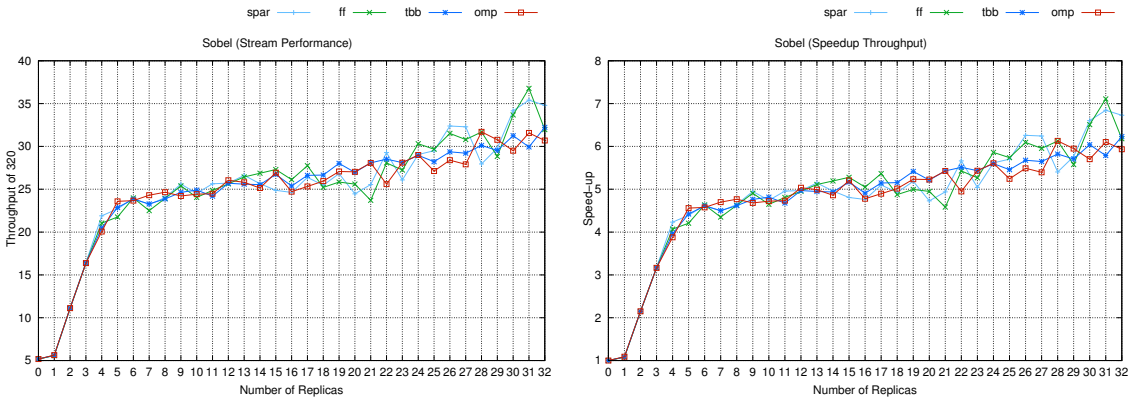
Figure 7.13: Time performance comparison using balanced workload (Listing 7.2)



(a) Sobel Filter throughput. (b) Sobel Filter throughput speed-up.

Figure 7.14: Stream performance comparison using balanced workload (Listing 7.1)

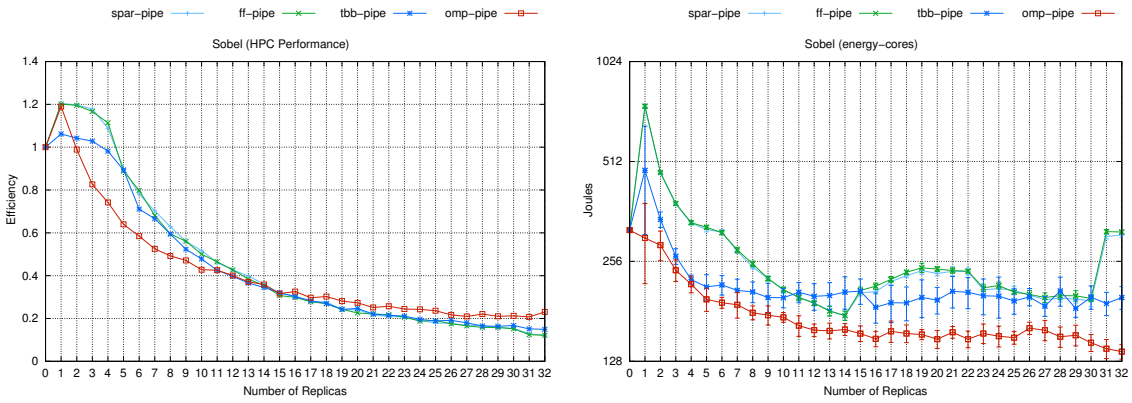
When we look for the stream throughput metric, the difference is more evident in Figure 7.14 and 7.15. We can conclude that DataFlow parallelism (OpenMP) has



(a) Sobel Filter throughput. (b) Sobel Filter throughput speed-up.

Figure 7.15: Stream performance comparison using balanced workload (Listing 7.2)

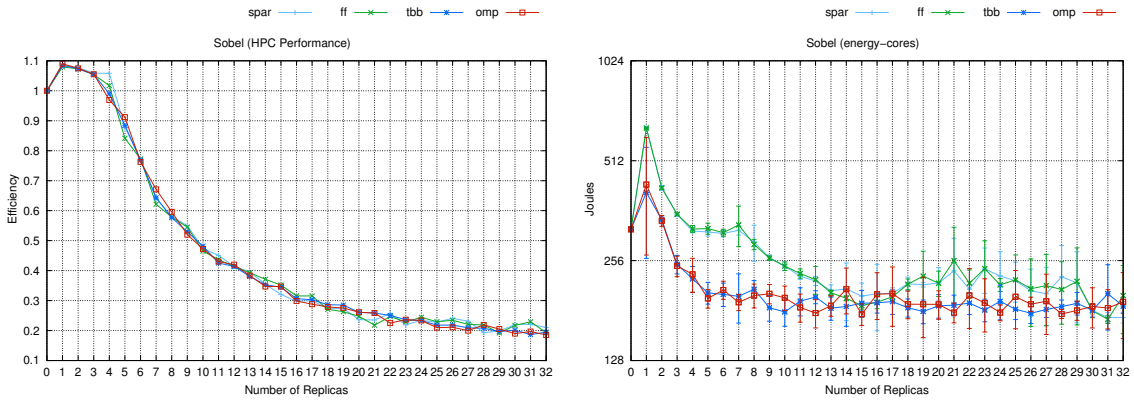
higher throughput rates compared to stream parallelism in Figure 7.14's version. This is primarily due to the fact that the first and last stages are not able to process fast enough to maintain the pipeline full. One way to address this performance behavior would be replicate the last stage, such as OpenMP dues with DataFlow. Hence, with a lower number of replicas the application can not perform better than when using OpenMP, while SPar allows the users to choose the version that fits their design goals regarding performance. For instance, in the version presented in Figure 7.15, SPar achieved the highest speedup and throughput rate as well as similar performance to the highest OpenMP in Figure 7.14.



(a) Sobel Filter HPC efficiency. (b) Sobel Filter CPU cores energy consumption.

Figure 7.16: HPC performance comparison using balanced workload (Listing 7.1)

Figure 7.16 shows the results of completion time on the efficiency for this version, where stream-based parallelism interfaces perform better up to the point hyper threading facilities are used. However, this difference was not that significant if compared with the throughput graph, estimating better efficiency to SPar concerning the number of replicas needed. On the other hand, it consumes much more energy than OpenMP. As presented in Figure 7.5(b), SPar allows users to achieve less energy



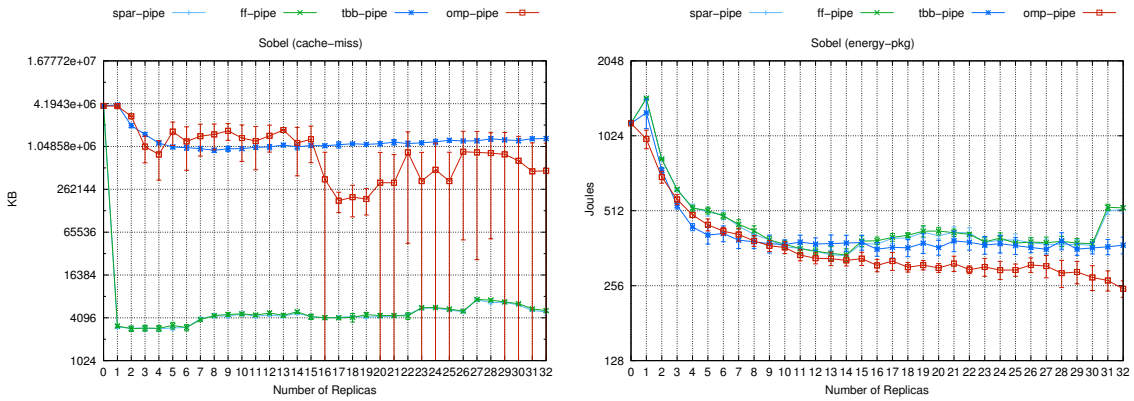
(a) Sobel Filter HPC efficiency.

(b) Sobel Filter CPU cores energy consumption.

Figure 7.17: HPC performance comparison using balanced workload (Listing 7.2)

consumption by adding the `spar_blocking` optimization flag, being competitive with the other frameworks. Regarding Figure 7.17, we can observe that efficiency was not significantly different among the programming framework while energy consumption was worst in FastFlow and SPar. This result could be improved in SPar by adding the `spar_blocking` optimization flag during the compilation of the program.

Another analysis of HPC is relative to the cache misses, which were significantly better in SPar and hand written FastFlow in both implementation versions (see Figures 7.18 and 7.19). Also, even though SPar demonstrates more energy consumption considering the whole CPU socket, it can be competitive if an optimization flag is used.

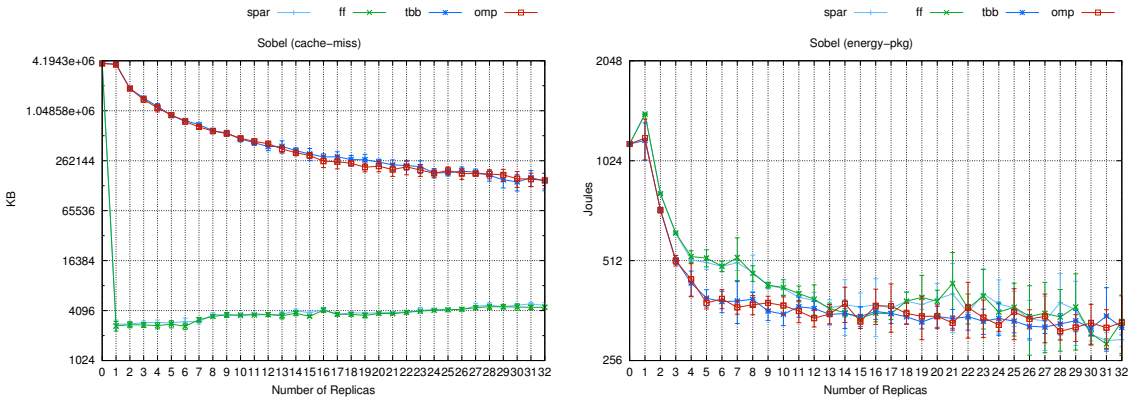


(a) Sobel Filter cache misses.

(b) Sobel Filter socket energy consumption (cache and CPUs).

Figure 7.18: CPU Socket performance comparison using balanced workload (Listing 7.1)

Finally, SPar only uses much more memory when compared to TBB in Figure 7.20. This is because SPar builds on top of FastFlow, which uses queues to communicate and by default it uses round robin scheduling. However, SPar is flexible enough to

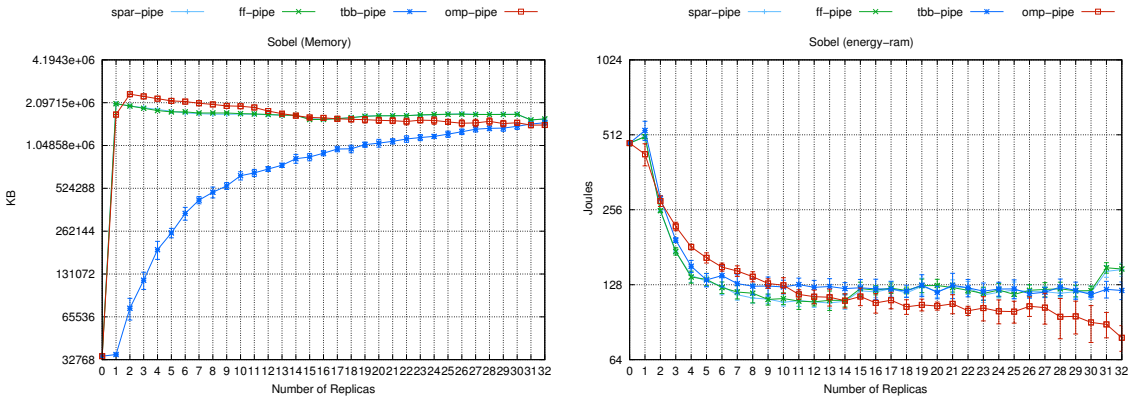


(a) Sobel Filter cache misses.

(b) Sobel Filter socket energy consumption (cache and CPUs).

Figure 7.19: CPU Socket performance comparison using balanced workload (Listing 7.2)

allow the user to improve memory usage by adding a `spar_ondemand` optimization flag (which was presented in Figure 7.9(a)), with a version of the code competitive with TBB.

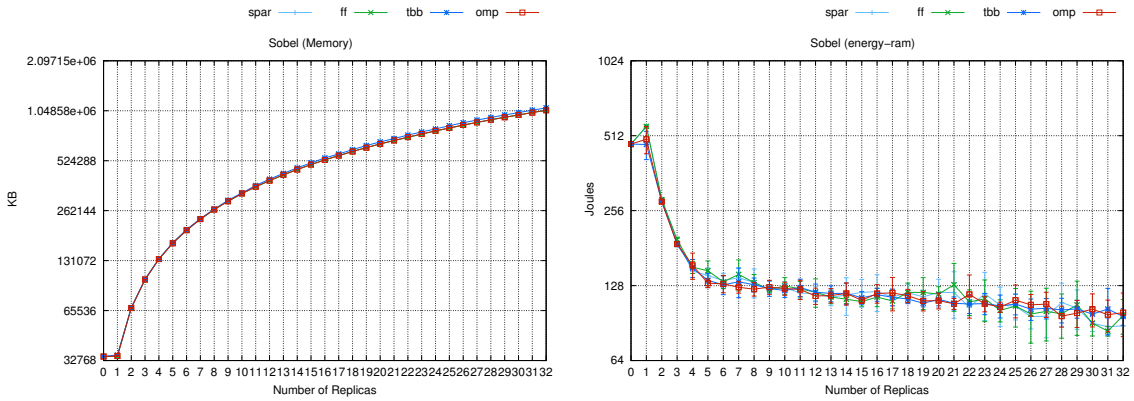


(a) Sobel Filter memory usage.

(b) Sobel Filter memory energy consumption.

Figure 7.20: Memory performance comparison using balanced workload (Listing 7.1)

The results using unbalanced workloads presented only significant differences compared to experiments with balanced workloads in the pipeline-like version. This can be view in the Appendix, specifically in Section A.1.2. Starting from the execution time and latency, they were even better up to and after using hyper threading facilities. The highest rates of SPar and OpenMP throughput were close to each other, as can be seen in Figure A.13. Consequently, the efficiency of the application was also more competitive with the OpenMP results (Figure A.15). However, energy consumption and other metrics do not present significant changes when compared to unbalanced and balanced workload results. Therefore, we can conclude that fine grained and high-frequency streams significantly affect the performance of the OpenMP in the pipeline-like computation, while stream-oriented runtime and SPar maintain the same



(a) Sobel Filter memory usage. (b) Sobel Filter memory energy consumption.

Figure 7.21: Memory performance comparison using balanced workload (Listing 7.2)

performance.

7.3.1.4 Summary

In the Sobel filter application, we can highlight that flexibility does not affect SPar's productivity and provides more opportunities for improving machine resource usage and accomplishing the application constraints. Also, we demonstrated that optimization flag allows us to fine tune performance and provide more options to the user for energy consumption and memory usage constraints.

For this application, code productivity was not significantly affected by using SPar annotations, representing less than one percent on physical SLOCs. Also, SPar keywords proved to be more suitable for the domain than the primitives provided by OpenMP. On the other hand, when compared to TBB and FastFlow, the productivity was significantly better. In general, SPar does not add significant performance degradation and in some cases it outperforms the state-of-the-art tools.

7.3.2 Video OpenCV

Video applications represent a classic example of stream parallelism. Video streams can come from different sources (network, local and camera) and it is hard to determine the end of the stream. Real world video streaming operations have body or face tracking and filtering. One of the most commonly used C++ libraries in this area is OpenCV [KB16]. Therefore, we decided to use it as a benchmark. Listing 7.3 presents only the stream region of the application, which was taken from OpenCV examples. Instead of reading from the camera, we had it read from a video file, applying common video computations on each ovideo frame.

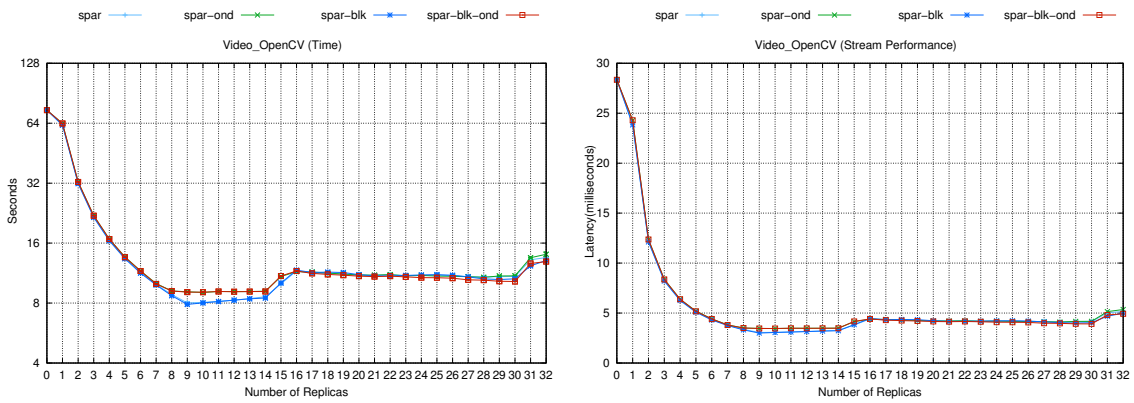
Hence, it aims to extract a specific RGB channel, apply a Gaussian filter, make a Weighted screen operation (commonly used in film production [Wri10]) and apply the Sobel filter on each video frame. Before entering into the stream region, the application opens the input and output video files. Inside the infinite loop, the application reads frame by frame (line 4 in the listing below), tests if it is empty (line 5), performs a sequence of video operations (between line 7 and 17) and writes the results in the output file (line 20). Following SPar's methodology, we can clearly identify the stream region and what it will consume from preprocessing the source code. Also different ways to annotate stages may be used. For example, one can introduce more stages to fragment the stream operations. However, Listing 7.3 provides an annotation schema that will be evaluated later in performance and productivity.

```
1  [[spar::ToStream, spar::Input(res,channel,src,S)]] for(;;){
2    total_frames++;
3    inputVideo >> src;
4    if (src.empty()) break;
5    [[spar::Stage, spar::Input(res,channel,src,S), spar::Output(res), spar
       ::Replicate()]]{
6      vector<Mat> spl;
7      split(src, spl);
8      for (int i =0; i < 3; ++i){
9        if (i != channel){
10         spl[i] = Mat::zeros(S, spl[0].type());
11       }
12     }
13     merge(spl, res);
14     cv::GaussianBlur(res, res, cv::Size(0, 0), 3);
15     cv::addWeighted(res, 1.5, res, -0.5, 0, res);
16     Sobel(res,res,-1,1,0,3);
17   }
18   [[spar::Stage, spar::Input(res)]]
19   { outputVideo << res; }
20 }
```

Listing 7.3: Video OpenCV using SPar.

7.3.2.1 SPar Performance

The experiments for evaluating the performance of this application were conducted on a Pianosau machine, using an AVI video file with a duration of 1.53 minutes, 640x480 resolution and containing 2626 frames. This section will evaluate only SPar's performance when using the optimization flags. Figure 7.22 demonstrates the completion time results (left) and latency (right). We can observe that adding the `spar_ondemand` flag significantly affects the completion time as well as the latency of the application from 7 up to 15 replicas. In Figure 7.23 we can see how much this difference will represent in throughput, were this flag allows the application to process about fifty frames less per replica time.

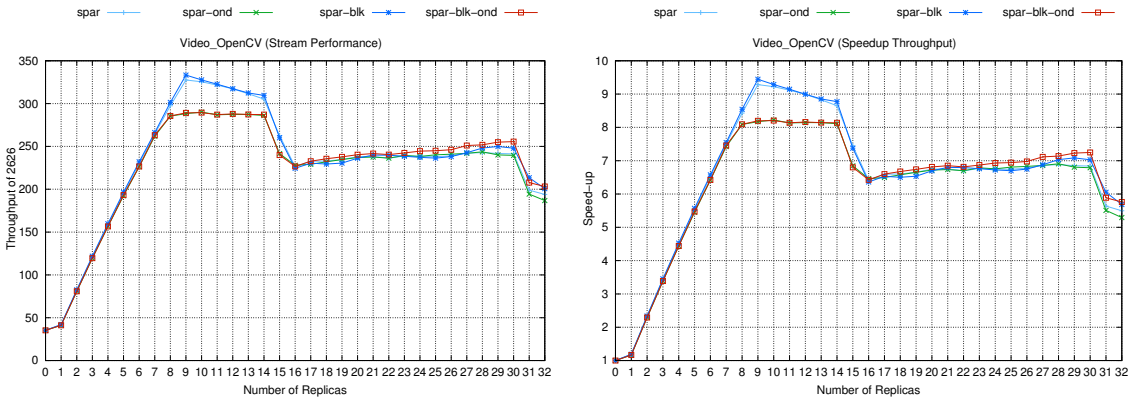


(a) Video OpenCV execution times. (b) Video OpenCV latency.

Figure 7.22: Time performance (Video OpenCV)

Figure 7.23 shows that the application scalability is at max 9.5 replicas. This result is due to the same reason we already verified in the Sobel filter application: the disk becomes a bottleneck. In contrast, this application achieved the highest speedup rates because there are more stream operations not banded by the disk performance. Also, this application cannot work in a farm-like composition because write operations cannot operate independently and require the use of the `spar_ordered` flag to guarantee that elements are processed in order. Thus, although the video and image application share similar characteristics, they also present different constraints. Yet, the flexibility of SPar can address them both and provide good performance.

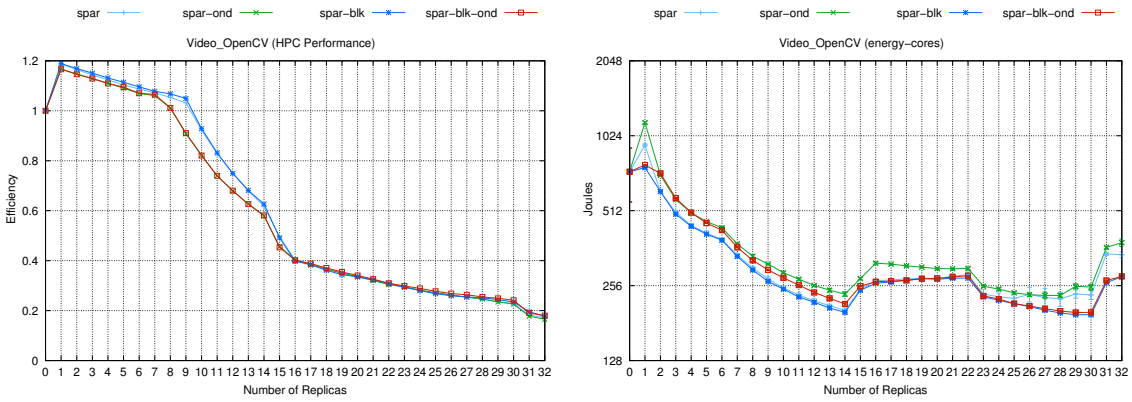
The previous results also reflect more CPU efficiency if the optimization flags are not used and less energy consumption, as presented in Figure 7.24. However, the FastFlow runtime is able to avoid different cache misses as can be seen in Figure 7.25(a). Since the `spar_ondemand` compiler flag creates more energy consumption on CPU cores, the energy consumption will be different when measuring the entire CPU socket (Figure 7.24(b)).



(a) Video OpenCV throughput.

(b) Video OpenCV throughput speed-up.

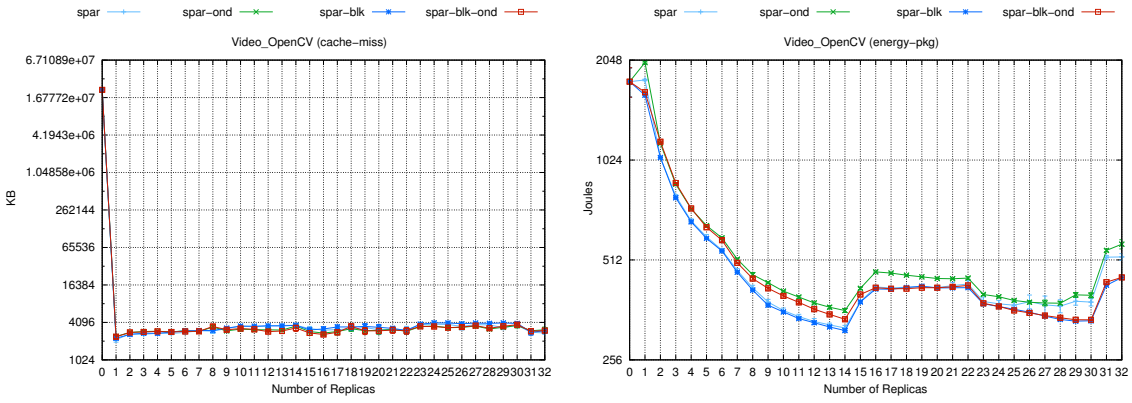
Figure 7.23: Stream performance (Video OpenCV)



(a) Video OpenCV HPC efficiency.

(b) Video OpenCV CPU cores energy consumption.

Figure 7.24: HPC performance (Video OpenCV)

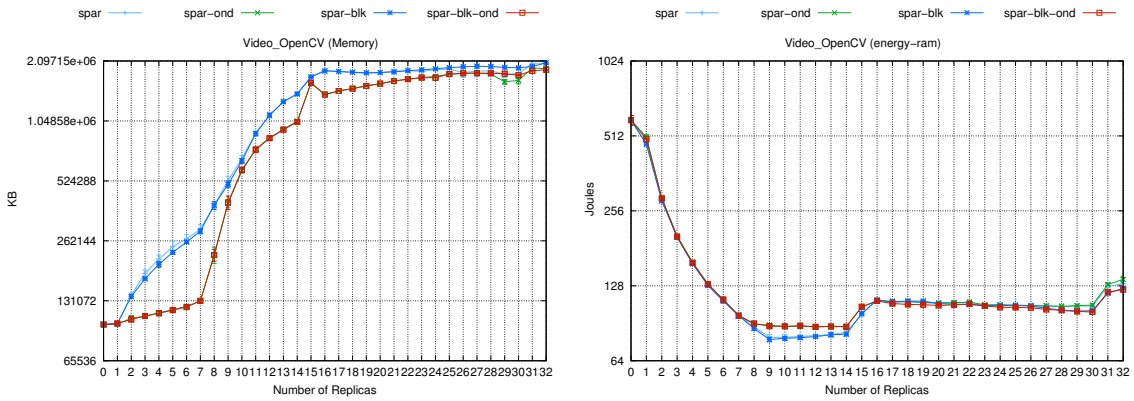


(a) Video OpenCV cache misses.

(b) Video OpenCV socket energy consumption (cache and CPUs).

Figure 7.25: CPU Socket performance (Video OpenCV)

The last metric is memory usage, which is plotted in Figure 7.26. This reveals energy consumption. Once more the usage of the optimization flag makes the difference. It requires less memory because on-demand scheduling stores only one element per FastFlow queue, while round robin by default stores 512 stream elements, which explains these results.



(a) Video OpenCV memory usage. (b) Video OpenCV memory energy consumption.

Figure 7.26: Memory performance (Video OpenCV)

During the discussion of the results of this section, we can highlight that SPar's optimization flag can affect the performance of the application in some cases. The advantage is that the user can easily switch the optimizations on or off without source code intervention.

7.3.2.2 Productivity Comparison

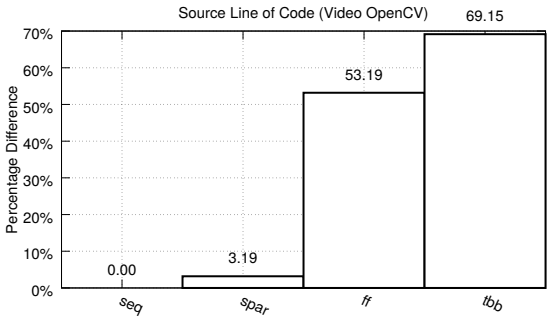


Figure 7.27: Source line of code for Video OpenCV application.

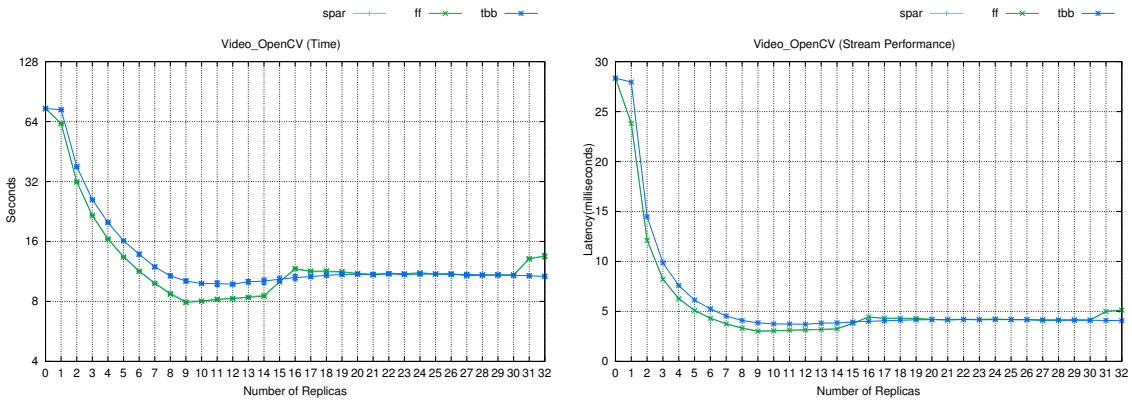
Before discussing coding productivity, it is important to highlight that OpenMP is not able to naturally parallelize this application. Since code refactoring is needed to parallelize in the application with OpenMP as it does not provide suitable annotations

to stream parallelism and therefore we do not have a comparison with it. Figure 7.27 provides a statistic of the physical SLOC needed by the programming frameworks. We observe that SPar is significantly more productive in comparison with state-of-the-art tools, fifty percent more than FastFlow and about sixty-five percent better than TBB.

We can also observe better productivity not only with respect to SLOC, but also regarding source code maintainability. While in SPar only code annotations are needed, FastFlow and TBB require the developer to restructure the application such as demonstrated in their respective pseudo codes in the Appendix Listings A.7 and A.8. In FastFlow, it is necessary to program the emitter, worker, and collector classes by implementing the `svc` method as well as the stream through a data structure [v]. Similarly, the TBB version was implemented to allow parallelism in this application.

7.3.2.3 Performance Comparison

In this section, a performance comparison is made to evaluate if SPar transformations are efficient according to hand-written state-of-the-art tools TBB and FastFlow. We used the same video to stress the application and compare the results for all metrics. Figure 7.28 plots the graphic results of the execution time and latency. We can see that SPar presents the same latency and completion time as FastFlow and better results than TBB.



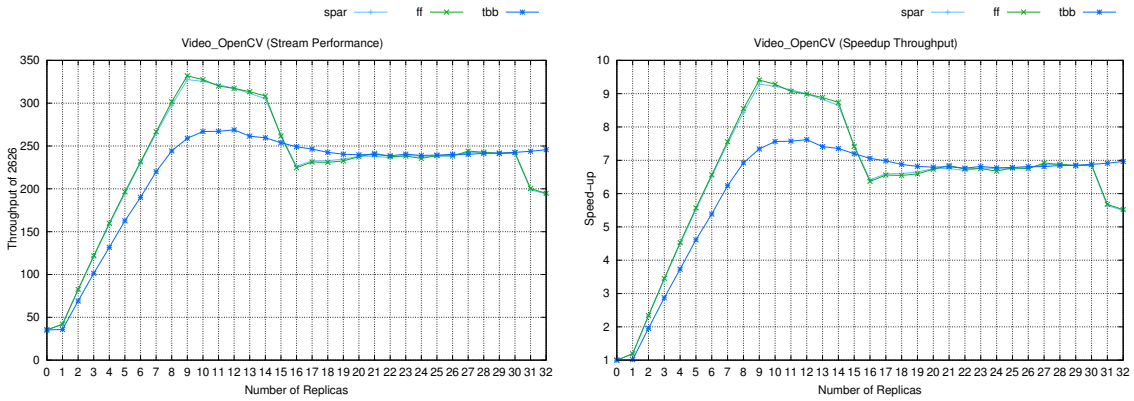
(a) Video OpenCV execution times. (b) Video OpenCV latency.

Figure 7.28: Time performance comparison (Video OpenCV)

In Figure 7.29, we can see how SPar's throughput rate is better than TBB's, enabling about fifty frames more per replica to a given completion time. A similar

[v] The most recent version of FastFlow actually requires less SLOCs leveraging the new features provided by C++11(14). As an example, pipeline stages may be provided as closures, rather than `svn` methods in a `ff-node` object.

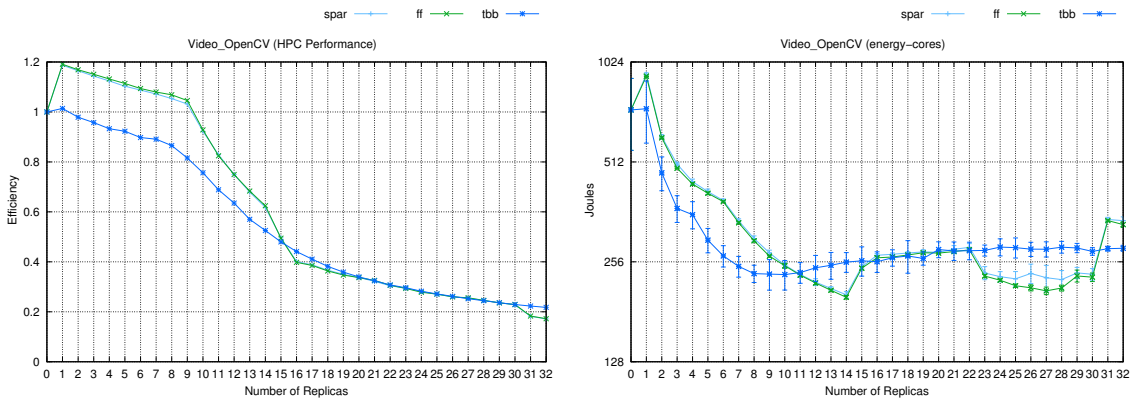
difference was found in Figure 7.23 when comparing SPar's optimization flag. Consequently, we can judge that TBB's work stealing scheduler adds extra overhead to this application. Thus, for each frame a thread will communicate with the scheduler, asking for another job. Since this is how TBB behaves, there is no alternative to improve TBB's throughput rate.



(a) Video OpenCV throughput. (b) Video OpenCV throughput speed-up.

Figure 7.29: Stream performance comparison (Video OpenCV)

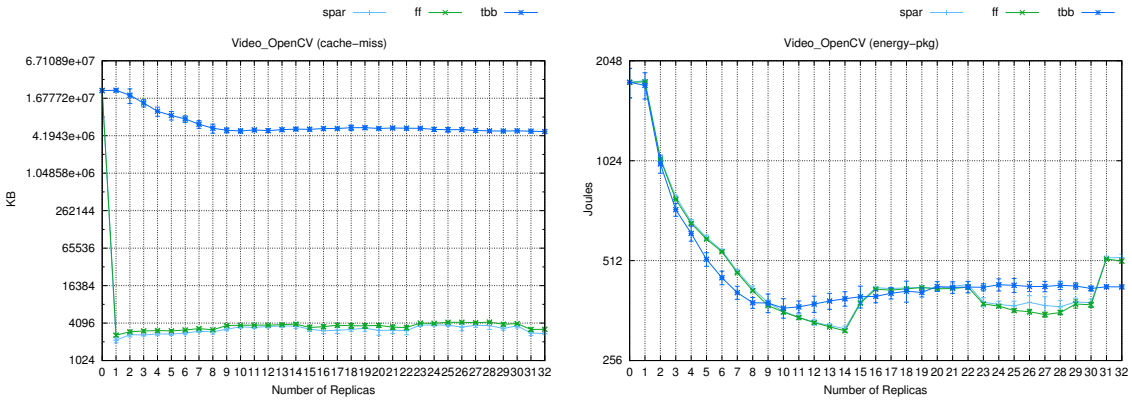
To confirm the previous assumptions and results, Figure 7.30 demonstrates the application's efficiency and energy consumption. SPar and FastFlow are more efficient than TBB. As SPar uses/stresses more the CPU to achieving this level of performance, it is natural that it consumes more energy on the CPU cores (as shown in the graph).



(a) Video OpenCV HPC efficiency. (b) Video OpenCV CPU cores energy consumption.

Figure 7.30: HPC performance comparison (Video OpenCV)

A subtle difference between SPar and FastFlow regarding cache misses can be observed in Figure 7.31(a). Yet, this is not considered significant because of the standard deviation. However, it does indicate that SPar code generation can impact cache misses. Again, SPar and FastFlow significantly outperform TBB.

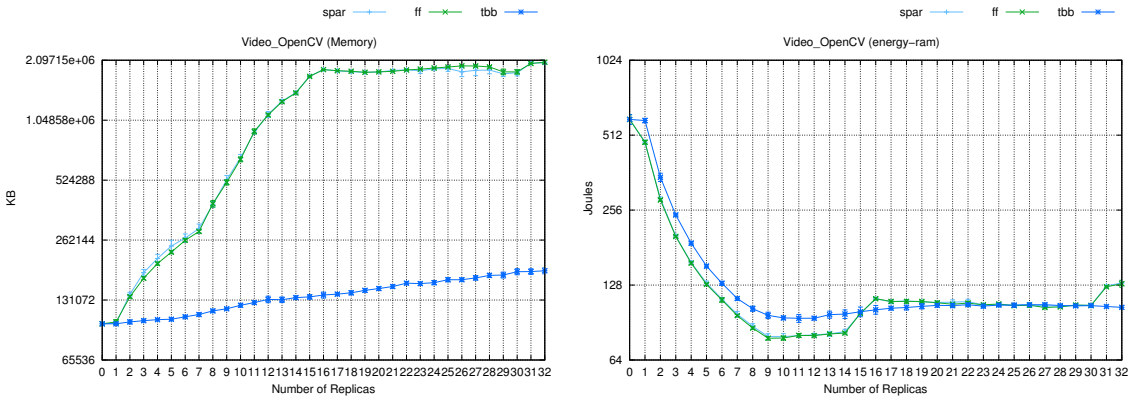


(a) Video OpenCV cache misses.

(b) Video OpenCV socket energy consumption (cache and CPUs).

Figure 7.31: CPU Socket performance comparison (Video OpenCV)

Unfortunately, SPar and FastFlow use much more memory than TBB due to the queue communication. However, in case the user has memory constraints, the use of optimization flags enables SPar to achieve a version that is competitive with TBB in memory usage. For example, specifying the `spar_ondemand` flag.



(a) Video OpenCV memory usage.

(b) Video OpenCV memory energy consumption.

Figure 7.32: Memory performance comparison (Video OpenCV)

7.3.2.4 Summary

After experimenting with our video OpenCV application, we concluded that SPar's annotations were simple, high-level and suitable to provide efficient parallelization. We can highlight that SPar significantly improved productivity on Video OpenCV applications when compared to the state-of-the-art tools. Also, SPar does not add extra overhead when compared to hand-written FastFlow and performed better than

TBB. However, the optimization flags were not synonymous with speeding up the application's performance. Moreover, this experiment revealed the benefit of the stream-oriented runtime (FastFlow) in cache efficiency, where TBB cannot achieve good performance due to the work stealing scheduling implementation that adds extra overhead in streaming and fine grained applications.

7.3.3 Mandelbrot Set

The Mandelbrot set is a mathematical application that aims to create a fractal image from a set of complex numbers [DM06]. It is very well known in the mathematical field, and different algorithms are available in the literature. Our application was taken from FastFlow examples repository[vi]. This is not a stream application, but can be implemented as such. Listing 7.4 presents the stream region of the application. For a given dimension, the program computes the value of each one of the pixels until a line of the image is completed (between lines 7 and 20) so that it can be printed on an interactive screen (between lines 23 and 24).

```
1  //preprocesing
2  [[spar::ToStream, spar::Input(dim,init_b,step,init_a,niter)]]
3  for(int i=0;i<dim;i++) {
4    unsigned char *M = (unsigned char *) malloc(dim);
5    double im=init_b+(step*i);
6    [[spar::Stage,spar::Input(M,i,im,niter,init_a,step,dim), spar::Output(
       M,i),spar::Replicate()]]{
7      double a,b,a2,b2,cr,k;
8      for (int j=0;j<dim;j++){
9        a=cr=init_a+step*j;
10       b=im;
11       k=0;
12       for (k=0;k<niter;k++){
13         a2=a*a;
14         b2=b*b;
15         if ((a2+b2)>4.0) break;
16         b=2*a*b+im;
17         a=a2-b2+cr;
18       }
19       M[j]= (unsigned char) 255-((k*255/niter));
20     }
21   }
22   [[spar::Stage,spar::Input(i,dim,M)]]{
23     ShowLine(M,dim,i);
24     free(M);
25   }
26 }
```

[vi] https://sourceforge.net/p/mc-fastflow/code/HEAD/tree/examples/simple_mandelbrot/

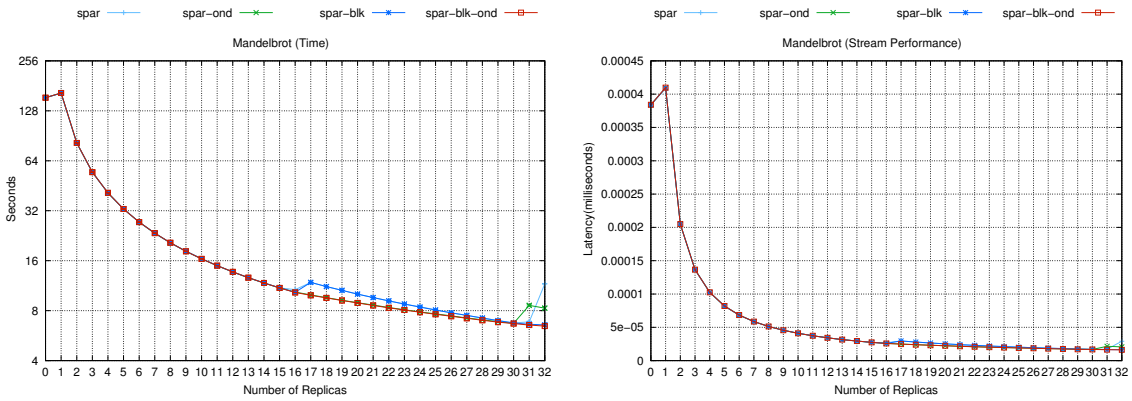
```
27 //pos-processing
```

Listing 7.4: Mandelbrot using SPar.

Since we intend to evaluate the performance later, just the most efficient annotation schema is demonstrated in Listing 7.4. When interpreting this application in a stream parallelism fashion, we have to look what the stream region is consuming from outside the region to annotate the input variables. The same attention must be paid when annotating the stage regions. Hence, we define the computation part and print as stages. Because each line of the resultant image can operate independently, we can replicate the computing stage (line 6). Note that last stage can not be replicated because printing lines in parallel will produce an incorrect image in the Mandelbrot set (it is a state-full stage).

7.3.3.1 SPar Performance

To evaluate SPar's compiler optimization flags performance, we configured the Mandelbrot set application to compute (-2.125+-1.5) to (0.875+1.5), iterate 1,024 times and produce an image of 400,000,000 pixels. The experiments were run on the Pianosau machine and standard deviations were plotted on the graphs through error bars.

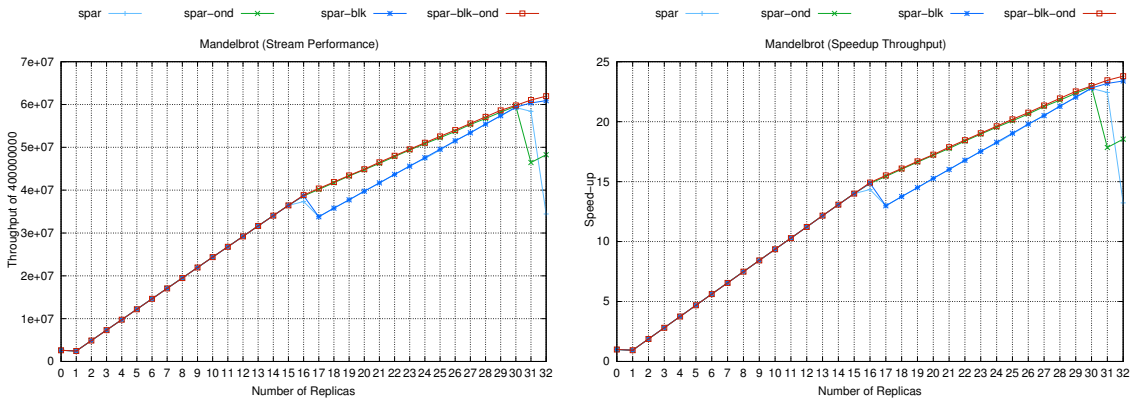


(a) Mandelbrot execution times. (b) Mandelbrot latency.

Figure 7.33: Time performance (Mandelbrot)

The completion times and latency for this application are in Figure 7.33. As can be observed, all options achieved similar results up to 16 replicas. We also had significant completion time reduction as well as low latency per pixel computed. The differences can be better understood in Figure 7.34 through the throughput rates. Note that performance degradation starts from where the replicas start to use hyper threading facilities. There is only significant degradation when `spar_ondemand` flag is not present when going up to 16 replicas. Also, scheduling influences the throughput

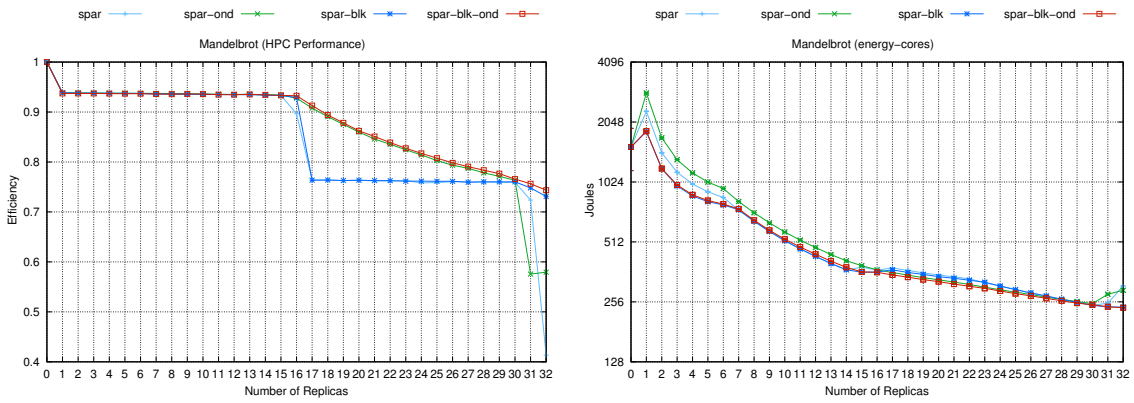
speedup. In the worst case (using 17 replicas), the program can process about 6,372,000 pixels more when using this flag.



(a) Mandelbrot throughput. (b) Mandelbrot throughput speed-up.

Figure 7.34: Stream performance (Mandelbrot)

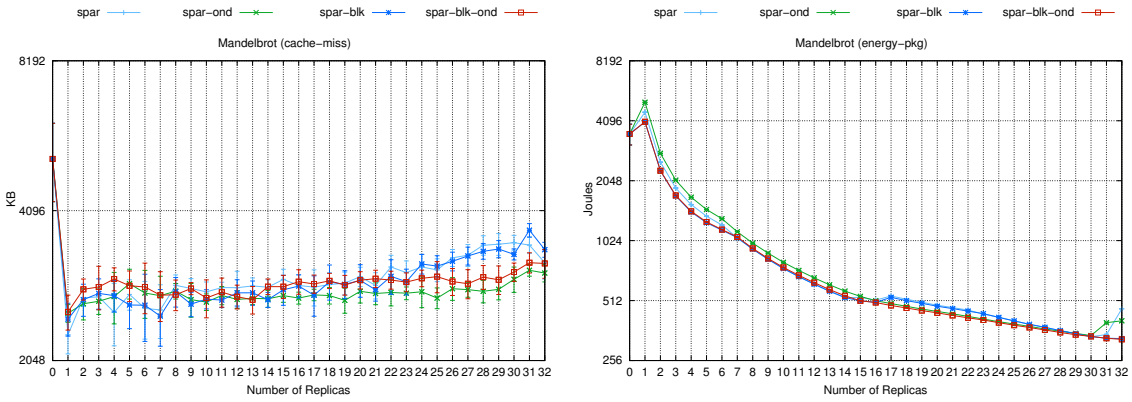
Concerning the HPC result in Figure 7.35, it becomes clear that in terms of efficiency and energy consumption, the best choice is to use `spar_ondemand` and `spar_blocking` flags together for this application along with the annotation schema.



(a) Mandelbrot HPC efficiency. (b) Mandelbrot CPU cores energy consumption.

Figure 7.35: HPC performance (Mandelbrot)

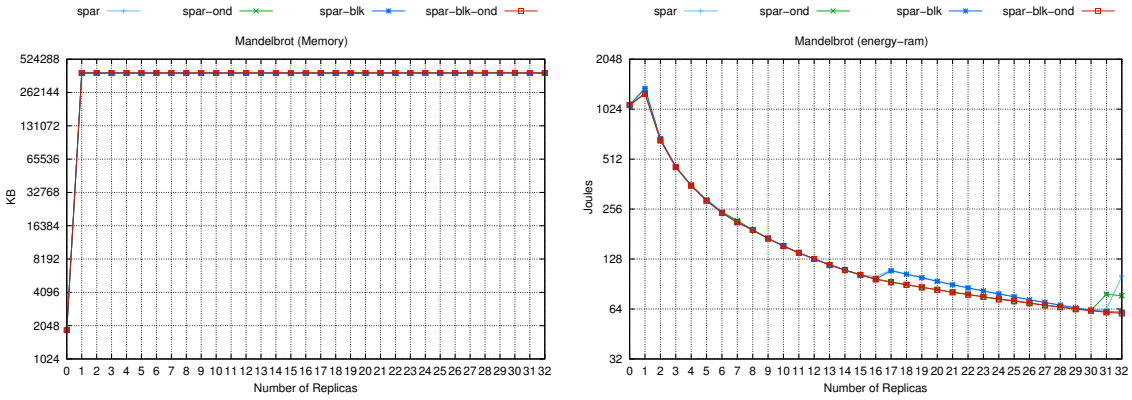
The high standard deviation of cache misses presented in Figure 7.36 does not allow us to assume that one alternative was better than another. On the other hand, results of CPU socket energy consumption reflected the results of the power consumption considering only the CPU cores. Finally, Figure 7.37 demonstrated no significant differences in memory usage, while the power consumption difference impacted the whole application's performance.



(a) Mandelbrot cache misses.

(b) Mandelbrot socket energy consumption (cache and CPUs).

Figure 7.36: CPU Socket performance (Mandelbrot)



(a) Mandelbrot memory usage.

(b) Mandelbrot memory energy consumption.

Figure 7.37: Memory performance (Mandelbrot)

7.3.3.2 Productivity Comparison

To compare the SLOC productivity, we experimented all implementation version that were used to compare performance, including OpenMP (omp), TBB (tbb), FastFlow (ff), and Pthreads (pt). The FastFlow and Pthreads versions were taken from the repository while we implemented the others. Figure 7.38 gives an idea of code productivity differences (in percentage). SPar is much more productive with respect to the state-of-the-art tools for stream parallelism (FastFlow and TBB) as well as Pthreads. Moreover, even though OpenMP should be suitable for this kind of parallelization, it requires more code than SPar because OpenMP annotations can not be made along with C++ statements, they appear on a separate line.

For analyzing the implementation, we discuss only the most productive versions.

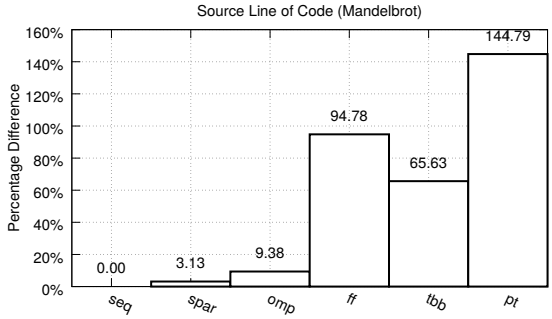


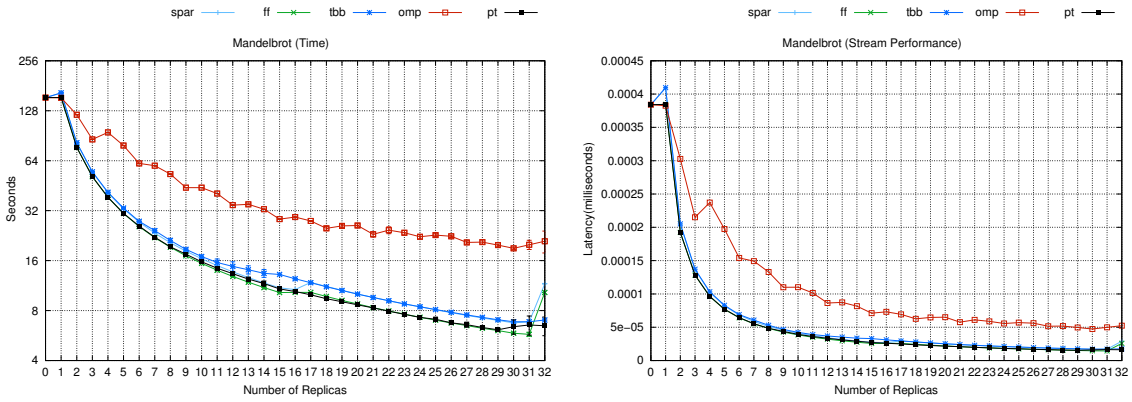
Figure 7.38: Source line of code for Mandelbrot application.

Therefore, their pseudo source code is presented in Section A.3.3. We can observe in Listing A.9 that OpenMP parallelization uses an explicit approach and terms that are particular to the programming model. In contrast, SPar annotations are high-level stream properties and the parallelism is implicit by the specification of stage replicas. On the other hand, programmers have to restructure the source code in a pipeline fashion using TBB (Listing A.11) while in Listing A.9, FastFlow restructures the program like a farm computation with the collector. In both libraries, one must implement classes and methods offered through a C++ template interface. Consequently, more code is necessary to structure and manage the data such as pointers.

7.3.3.3 Performance Comparison

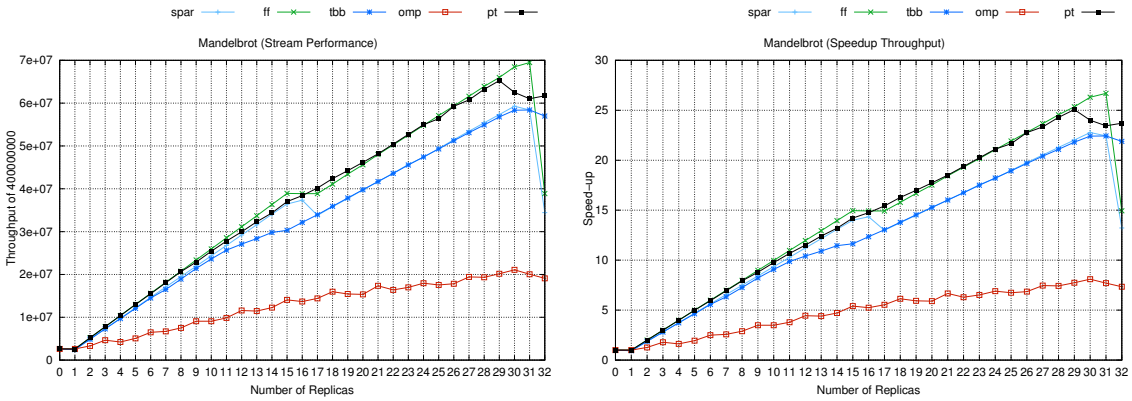
In this section, we aim to provide a performance comparison for evaluating the SPar and other state-of-the-art tools. Figure 7.39 compares execution times and latency for a set number of tested replicas. Among the SPar versions, we plot only the default without an optimization flag. We can observe that this SPar version has identical execution times with respect to hand-written FastFlow and Pthreads up to 16 replicas. To provide an identical result with more replicas, the `spar_ondemand` flag is used when compiling the program (as was observed in Figure 7.33(a)). In respect to OpenMP and TBB, SPar reduces the latency and execution time significantly. OpenMP performs the worst because the parallel region can not be entirely parallelized. Since the OpenMP runtime is not able to behave in a pipeline fashion, there is an implicit barrier after `single` and `parallel for` (see Listing A.9), where all system threads cross together.

This difference is better represented in Figure 7.40. The best SPar option is plotted in Figure A.26 to better visualize and compare with the other tools. We can see that SPar flexibility by using optimization flags allows for speedup performance while TBB does not provide such an opportunity. Although more competitive with



(a) Mandelbrot execution times. (b) Mandelbrot latency.

Figure 7.39: Time performance comparison (Mandelbrot)



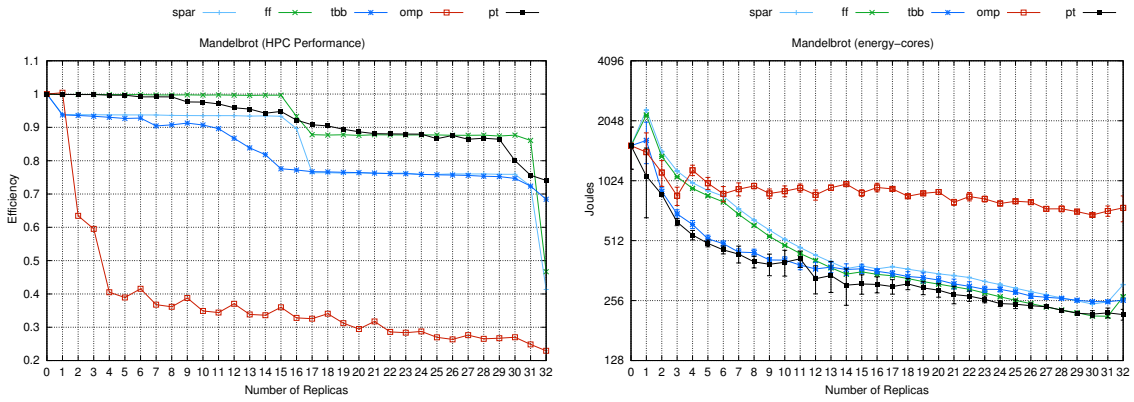
(a) Mandelbrot throughput. (b) Mandelbrot throughput speed-up.

Figure 7.40: Stream performance comparison (Mandelbrot)

FastFlow and Pthreads when using the optimization, SPar was not able to achieve identical high throughput rates. Consequently, there is an opportunity for SPar to improve code generation.

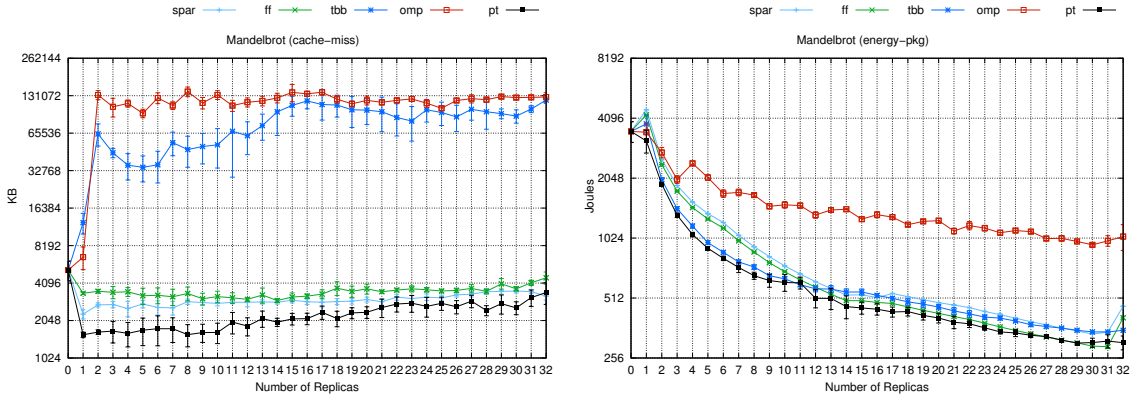
The parallel programming frameworks' efficiency and energy consumption of the CPU cores can be seen in 7.41. With the exception of OpenMP, all of the interfaces achieved a good balance of energy and efficiency. FastFlow, Pthreads and SPar also had a low number of cache misses as we can see in Figure 7.42. Yet, energy consumption is a consequence of the cores' energy consumption, which preserves a relative difference.

Figure 7.43 presents memory performance. Even using a different optimization flag, this result could not be improved in SPar. Therefore, we found another important result that permits the improvement of code generation, because manually developed FastFlow achieved better memory usage than SPar.



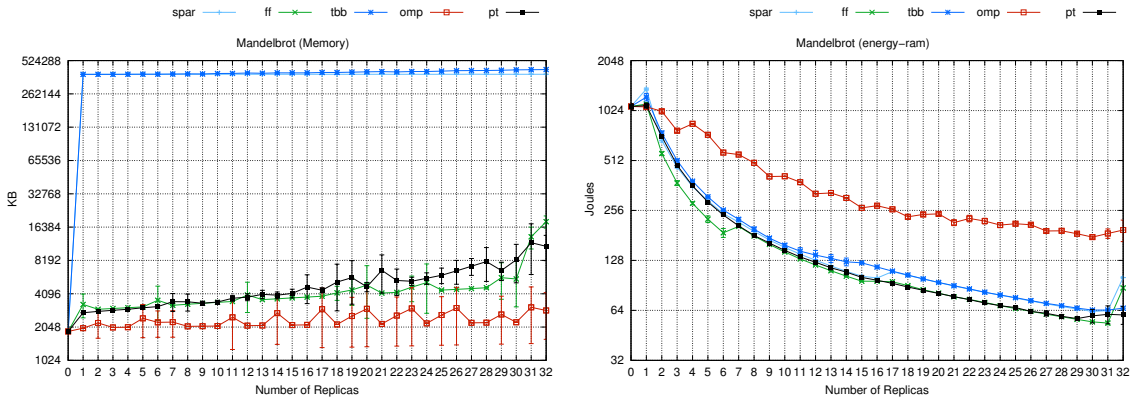
(a) Mandelbrot HPC efficiency. (b) Mandelbrot CPU cores energy consumption.

Figure 7.41: HPC performance comparison (Mandelbrot)



(a) Mandelbrot cache misses. (b) Mandelbrot socket energy consumption (cache and CPUs).

Figure 7.42: CPU Socket performance comparison (Mandelbrot)



(a) Mandelbrot memory usage. (b) Mandelbrot memory energy consumption.

Figure 7.43: Memory performance comparison (Mandelbrot)

7.3.3.4 Summary

During the discussion of the Mandelbrot set application, we demonstrated that SPar was able to again provide high-level, suitable and straightforward annotation along with the possibility to maintain the source code. In this application, optimization flags significantly improved the performance after a certain number of replicas.

When comparing productivity, SPar is significantly better than TBB and FastFlow, and slightly better on SLOCs and much more high-level than OpenMP. Moreover, OpenMP's poor performance is a consequence of its programming model that is not suitable for stream-like computations. SPar achieved similar performance in almost all the replicas tested with respect to FastFlow and TBB, which makes it competitive. Also, it is relevant to highlight that this application is not an original stream application. Therefore, some performance degradations should be expected. In general, the results were good, and the experiments were enough to find opportunities to increase SPar's performance, mainly for memory usage and when using a higher number of replicas.

7.3.4 Prime Numbers

This application is the naïve algorithm for finding prime numbers. Our implementation receives a number as an input and checks it by simply dividing it, and adding up every prime that is found. Listing 7.5 demonstrates an efficient way for introducing stream parallelism for this application by using our DSL. This algorithm is a classic example from the mathematical field used for many cryptographic applications and scientific programs.

Even though it is not part of the stream parallelism domain, we chose this application to illustrate how it is possible to implement a non-stream-oriented application in SPar. Listing 7.5 presents only the function of the application that calculates a prime number of a given number and returns the total of primes. Therefore, the first loop will iterate for generating the numbers so that the nested loop can find all its primes. If case there is not a prime, the variable used to sum the total number of primes receives zero to not be counted.

Based on the SPar's methodology, the answer for the first question is to annotate the first loop (line 3) and reuse the iteration statement to characterize it as a stream region. Then, the region only consumes the `n` variable as input. The total is not our input, because it will be produced inside a stream operation that will be annotated as a stage. Consequently, we can reuse the iteration statement (line 5) again to

annotate the first stage, which checks primes for a given number. The other stage will be the sum of primers (line 12). As the `ToStream` region is producing the number, SPar's methodology sees it as a stream element as well as the prime variable used for adding. Therefore, these variables are consumed by the first stage and only the prime is produced in the next stage. Finally, the last stage consumes the prime and will produce the total number of primes outside the stream region when the stream ends for the function returning the data (line 14).

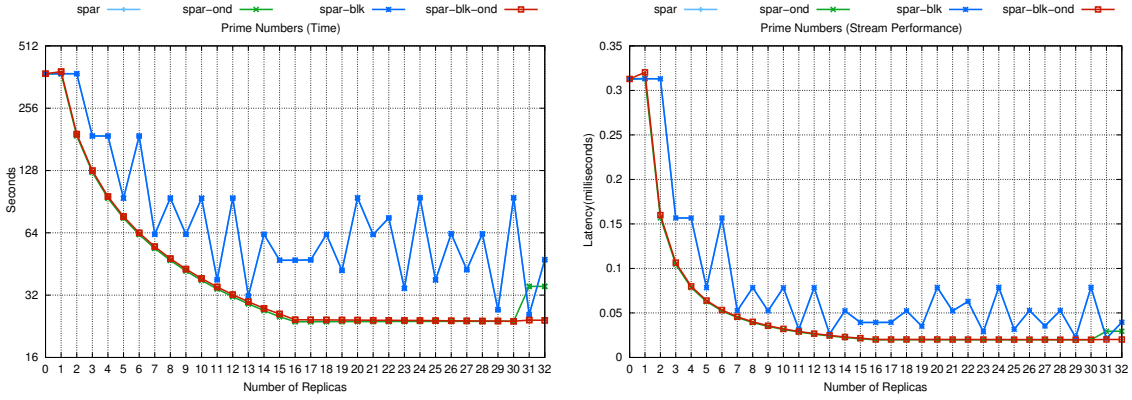
```
1  int prime_number(int n){
2    int total = 0;
3    [[spar::ToStream, spar::Input(n)]] for (int i = 2; i <= n; i++ ){
4      int prime = 1;
5      [[spar::Stage, spar::Input(i,prime), spar::Output(prime), spar::
       Replicate(workers)]] for (int j = 2; j < i; j++ ){
6        if ( i % j == 0 ){
7          prime = 0;
8          break;
9        }
10     }
11     [[spar::Stage,spar::Input(prime),spar::Output(total)]]
12     { total = total + prime; }
13   }
14   return total;
15 }
```

Listing 7.5: Prime Numbers using SPar.

We can only add replicate in the first stage, since it is possible to check whether a number is prime or not independently and sum the number of primes is a stateful operation. On the other hand, the last stage can not be replicated because at the end of the region the total sum is expected, instead of single replica sum. The next section will evaluate SPar's performance using different compiler optimization flags.

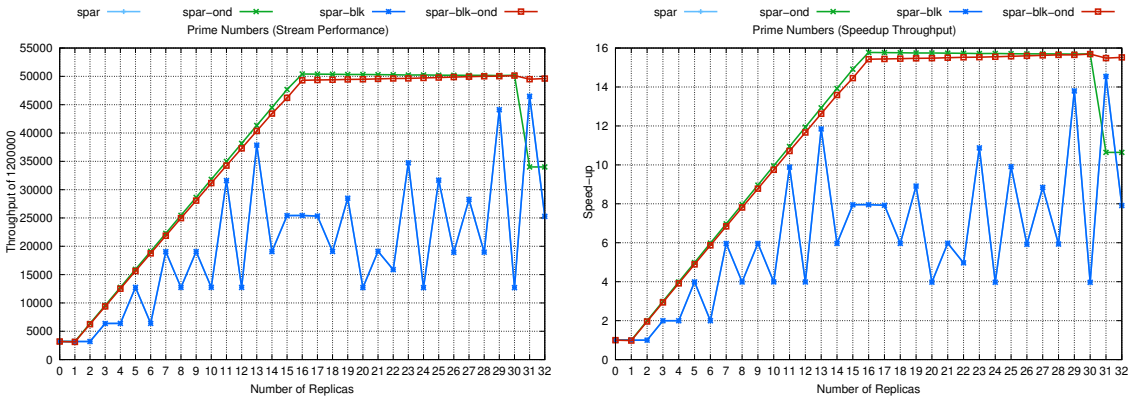
7.3.4.1 SPar Performance

Figure 7.44 presents the completion time and latency of the prime number application with all possible optimization flags. We tested the performance using the Pianosau machine setting up 1,200,000 as the size of the workload to find the primes. The results indicate that not using a `spar_ondemand` flag will significantly impact the completion time and latency of the application. This means that the scheduler plays an important control role for achieving good performance in this application. For example, Figure 7.45 presents the throughput rates, where it is possible to visualize that the difference represents two times more throughput in some cases (16 and 17 replicas) with the optimization flag.



(a) Prime Numbers execution times. (b) Prime Numbers latency.

Figure 7.44: Time performance (Prime Numbers)

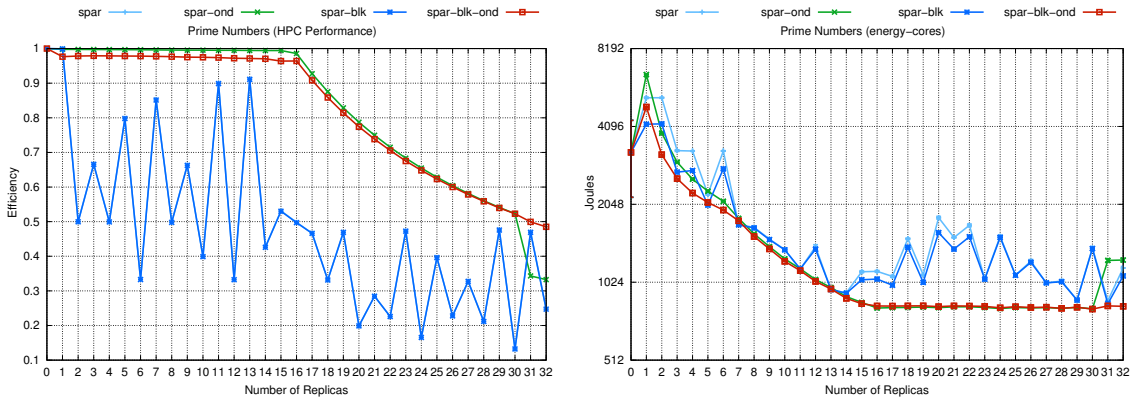


(a) Prime Numbers throughput. (b) Prime Numbers throughput speed-up.

Figure 7.45: Stream performance (Prime Numbers)

When looking at efficiency in Figure 7.46, we can see that adding the `spar_blocking` flag provides slight degradations. This occurs because the blocking mode adds extra overhead on thread synchronizations in fine-grained stream

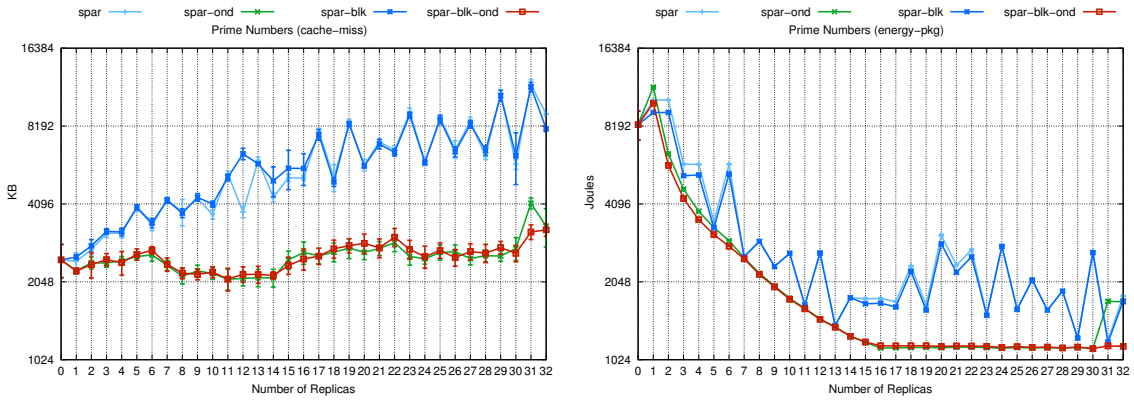
computations. The CPU cores' energy consumption was as good as the efficiency of `spar_ondemand`. Because there are intensive mathematical operations, the application cannot scale more replicas than the number of physical cores. Consequently, we can conclude that SPar was good enough to achieve expected speedup and efficiency with the help of the optimization flags.



(a) Prime Numbers HPC efficiency.

(b) Prime Numbers CPU cores energy consumption.

Figure 7.46: HPC performance (Prime Numbers)

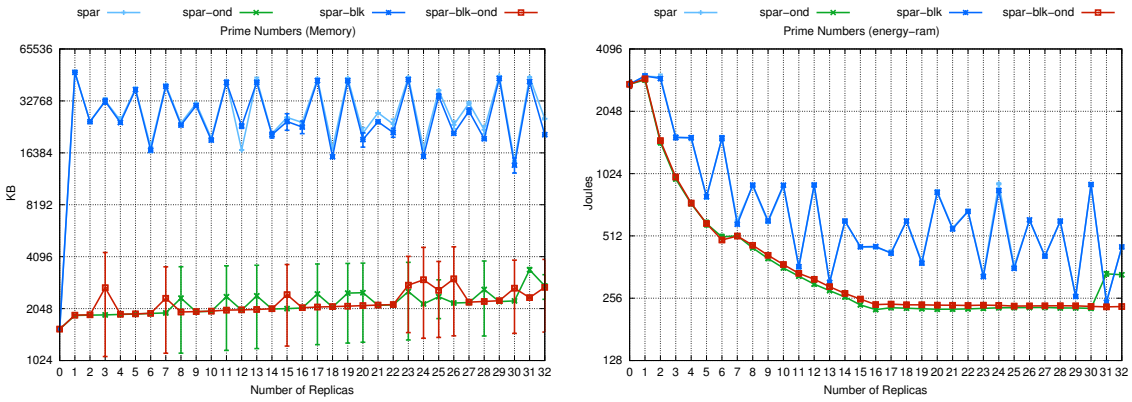


(a) Prime Numbers cache misses.

(b) Prime Numbers socket energy consumption (cache and CPUs).

Figure 7.47: CPU Socket performance (Prime Numbers)

We can also note that using optimization flags will impact cache misses (Figure 7.47) and memory usage (Figure 7.48). Once again, SPar provides performance flexibility to better fit the application's design goals.



(a) Prime Numbers memory usage. (b) Prime Numbers memory energy consumption.

Figure 7.48: Memory performance (Prime Numbers)

7.3.4.2 Productivity Comparison

To discuss and compare productivity, we considered the simple and legwork implementation of the prime number on FastFlow and TBB. The results concerning the SLOCs are presented in Figure 7.49. Even the simplest version of these tools require more code than SPar and OpenMP required the same amount of code. OpenMP achieved good coding productivity because it is a simple case and it was designed for simplifying parallelism for this kind of computation. In contrast, TBB and FastFlow provide more flexibility to the end user for implementing different parallelism versions.

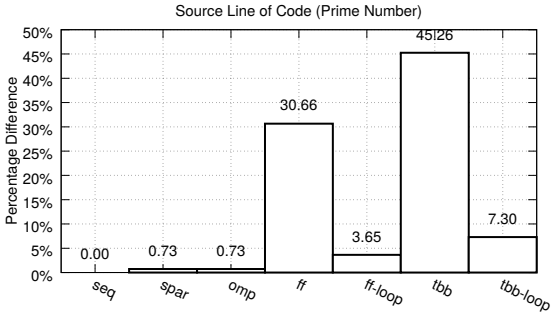


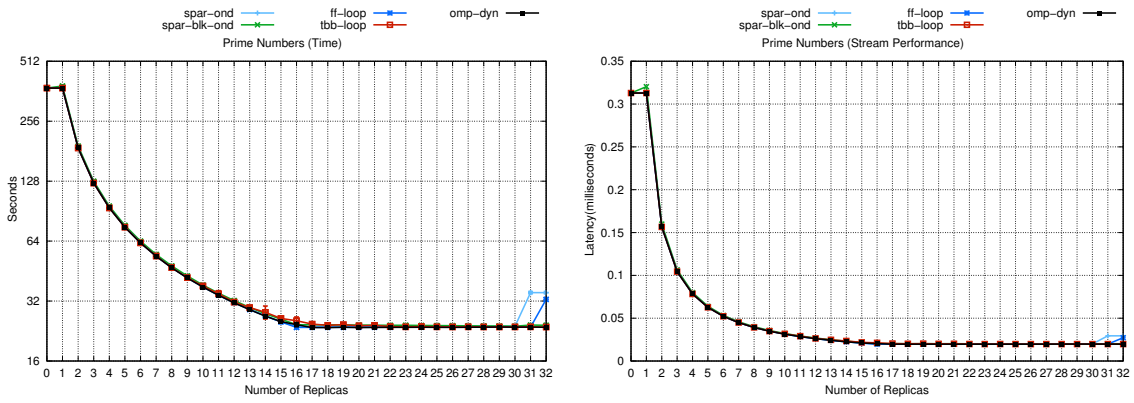
Figure 7.49: Source line of code for Prime Number application.

All pseudo code versions can be seen in Section A.3.4. In Listing A.12, we can note that OpenMP requires a single line annotation, where low-level parallel programming aspects are put in the table such as scheduling and reduction operation. Therefore, scheduling specification is not necessary to produce correct code. Yet, in later experiments it was necessary to speedup performance. In contrast, even when SPar has been used for another domain, it was able to provide efficient parallelism

without needing to add extra attributes through its annotations. On the other hand, TBB and FastFlow can provide a different alternative for better productivity in this kind of application by using the lambda function interface. However, this does not prevent the user from restructuring the "for" loop and implementing the reduction operation as can be observed in Listings A.14 and A.16.

7.3.4.3 Performance Comparison

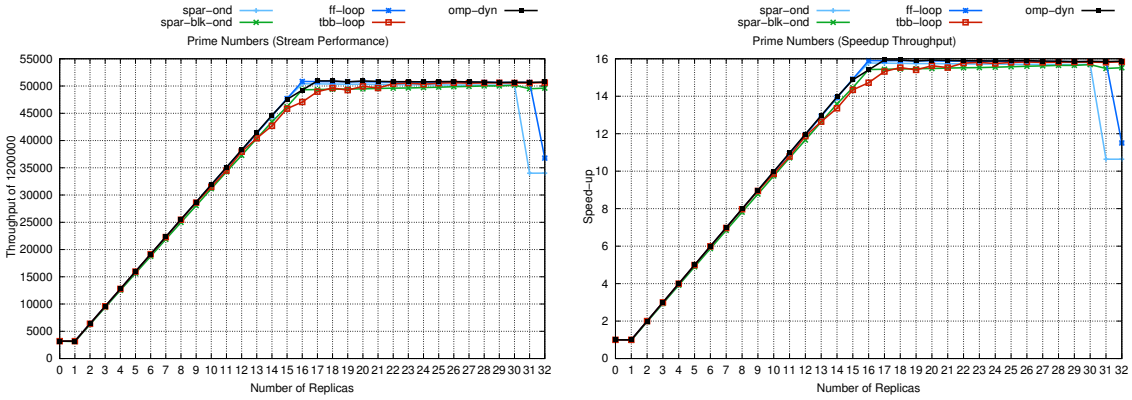
This section will compare the performance of the best solutions implemented to evaluate the transformations of SPar. As previously discussed, the best alternatives for SPar were when it used the `spar_ondemand` flag. In OpenMP, when using the default scheduler, the performance is similar as with the default compilation of SPar. Thus, we plotted its results with dynamic scheduler (omp-dyn). The results not discussed in this section are in the Appendix, specifically in Section A.1.3. Therefore, Figure 7.50 presents the execution times and latency for the number of replicas tested. All versions presented identical results, but using only the `spar_ondemand` flag along with FastFlow revealed a performance degradation in the last number of replicas. This is because of the on-demand scheduler that works intensively even when there is no stream. Thus, the blocking mode prevents such overhead, because the thread will be put to sleep when no work has to be done.



(a) Prime Numbers execution times. (b) Prime Numbers latency.

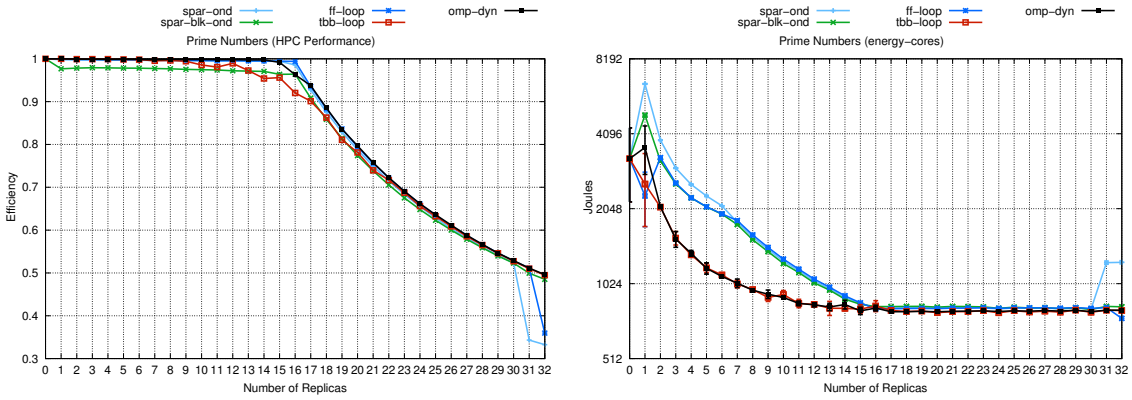
Figure 7.50: Time performance comparison (Prime Numbers)

The scheduler overhead for this application is noted in Figure 7.51 as well as the slightly worse performance difference of TBB, which is caused by its work-stealing scheduler. This is confirmed through Figure 7.52, which shows efficiency when using the CPU and energy consumption. We discovered that SPar and FastFlow versions consume more energy than OpenMP and TBB.



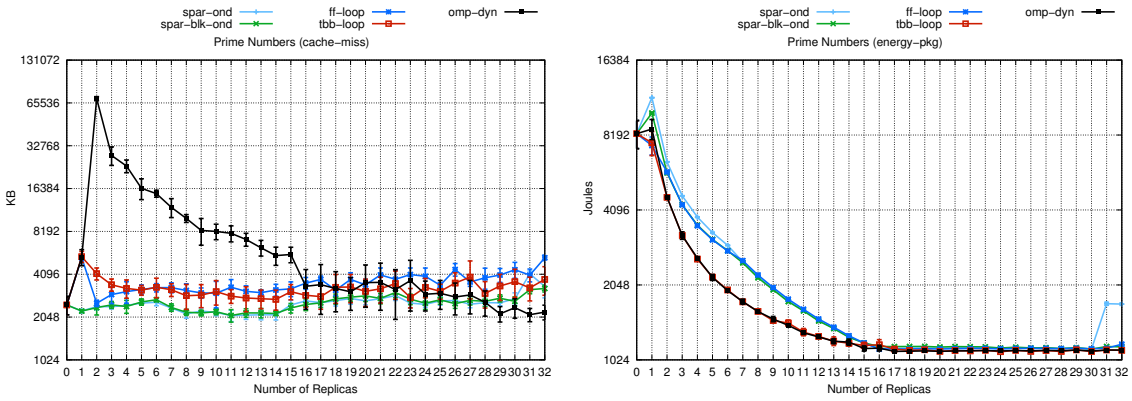
(a) Prime Numbers throughput. (b) Prime Numbers throughput speed-up.

Figure 7.51: Stream performance comparison (Prime Numbers)



(a) Prime Numbers HPC efficiency. (b) Prime Numbers CPU cores energy consumption.

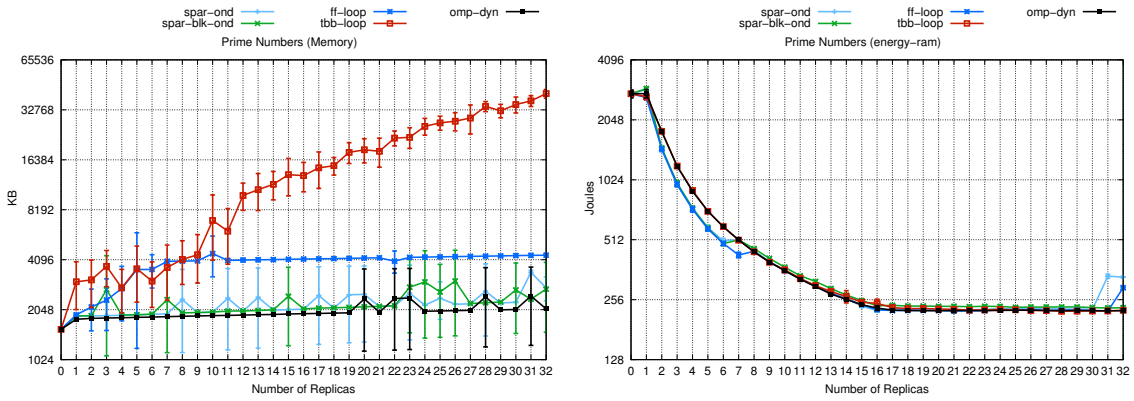
Figure 7.52: HPC performance comparison (Prime Numbers)



(a) Prime Numbers cache misses. (b) Prime Numbers socket energy consumption (cache and CPUs).

Figure 7.53: CPU Socket performance comparison (Prime Numbers)

Finally, SPar versions were able to avoid more cache misses (Figure 7.53) than all others. It also used less memory (see on Figure 7.54). Consequently, less memory energy consumption was necessary for SPar and FastFlow versions.



(a) Prime Numbers memory usage. (b) Prime Numbers memory energy consumption.

Figure 7.54: Memory performance comparison (Prime Numbers)

7.3.4.4 Summary

The prime number application provides us an interesting problem that stresses the need for performance flexibility. In the expressiveness evaluation, we could observe that SPar attributes were generic enough to annotate the parallelism of this application in a stream fashion. As in the other state-of-the-art tools, SPar requires the help of optimizations to provide high performance on data parallel computations.

Concerning the performance comparison, we concluded that SPar is also efficient for data parallel computations. Even though the prime number application is not a real stream computation, the results were good. Also, we did not expected so good and optimistic results such as less memory usage, cache efficiency and a small difference in stream performance such as throughput and latency.

7.3.5 K-Means

During recent year data analysis has become one of the hottest research topics in computer science. Among many algorithms existent in this field, one that is often used for analyzing big data sets is K-Means. It is a simple algorithm of a non-supervised machine learning to solve the clustering problem. The goal is to classify a given data set in groups. Our algorithm was taken from the code examples of Phoenix++ [TYK11]. Listing 7.6 presents the parallelized part of the algorithm to discuss and demonstrate how SPar attributes can be used for annotating the parallelism.

K-Means aims to first define `k` centroids for each cluster. These centroids are arranged at different locations bring different results. Consequently, it is necessary to place them as far as possible from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. Thus, when no data is pending, the first step is completed, which has performed the first step grouping. Then, the second phase is to recalculate the new cluster centroids from the previous step. After, a new connection must be made between the same points of the new data sets and the nearest centroid. Finally, K-Means algorithm has to iterate several times until all points are classified.

This application is not a real stream computation, but it was chosen to test SPar's expressiveness in other domains and how we could annotate parallelism correctly. Our methodology seeks to find the regions where it is possible to stream the code. K-Means has two dependent steps, making impossible to annotate them as stages (between line $3-18$ and $19-36$). When analyzing inside each step, we can see the most outside `for` loop generates a stream that is the index for the points and matrix of clusters or means. Thus, we found two stream regions. As for each iteration new values for the matrix of clusters and means are produced, they are input streams for **ToStream** annotations.

On the other hand, stream operations can be annotated in different ways using SPar. Since we do so for performance evaluation later, we exemplified the most efficient way to annotate. For instance, we could simply determine two stages in each one of the stream regions. In this case, we put all stream operations in a single stage. The input of the stage will be the same as the **ToStream** annotation plus the generated index, while the output is only the stream element modified inside the stage. Therefore, the first step stage produces the clusters as output and the second step stage produces the means. For a new iteration, the first step consumes the output of the second step and vice versa. Lastly, we can replicate each stage region because the points can be computed independently.

Note that the vector of points is not a stream, because it is a static data source

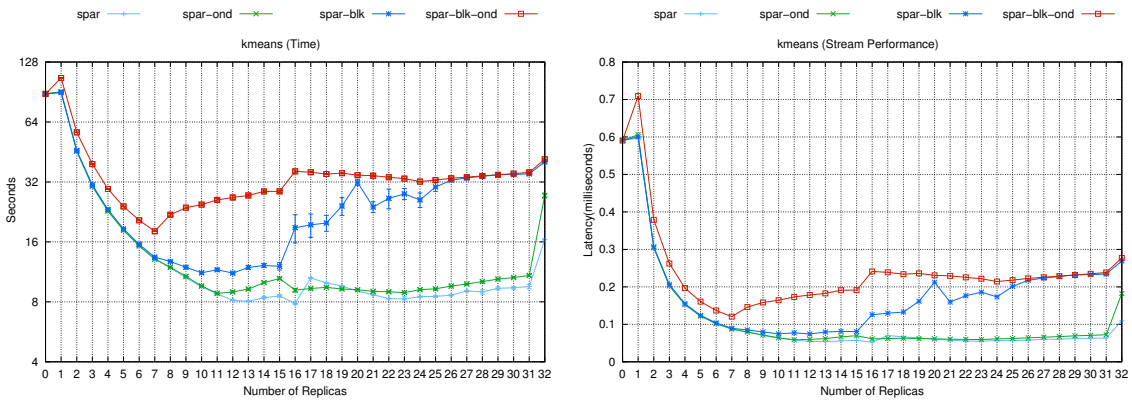
that was previously loaded from a data set file. Also, the amount of K-Means iteration is determined by the first step stage (line 16). The next section will present the performance results for this application.

```
1  while (modified){
2    modified = false;
3    [[spar::ToStream,spar::Input(means)]] for (int i = 0; i < num_points;
       i++){
4      [[spar::Stage, spar::Input(i,means), spar::Output(clusters), spar::
       Replicate()]]{
5      unsigned int min_dist = get_sq_dist(points[i], means[0]);
6      int min_idx = 0;
7      for (int j = 1; j < num_means; j++){
8        unsigned int cur_dist = get_sq_dist(points[i], means[j]);
9        if (cur_dist < min_dist){
10         min_dist = cur_dist;
11         min_idx = j;
12       }
13     }
14     if (clusters[i] != min_idx){
15       clusters[i] = min_idx;
16       modified = true;
17     }}
18   }
19   [[spar::ToStream,spar::Input(clusters)]] for (int i = 0; i < num_means
       ; i++){
20     [[spar::Stage, spar::Input(i,clusters), spar::Output(means), spar::
       Replicate()]]{
21     int* sum = (int *)malloc(dim * sizeof(int));
22     memset(sum, 0, dim * sizeof(int));
23     int grp_size = 0;
24     for (int j = 0; j < num_points; j++){
25       if (clusters[j] == i){
26         add_to_sum(sum, points[j]);
27         grp_size++;
28       }
29     }
30     for (int j = 0; j < dim; j++){
31       if (grp_size != 0){
32         means[i][j] = sum[j] / grp_size;
33       }
34     }
35     free(sum);}
36   }
37 }
```

Listing 7.6: K-Means application using SPar.

7.3.5.1 SPar Performance

To evaluate the performance of SPar and its optimization flags, we created the workload using 150,000 points to classify in 1,500 groups. The experiment ran in Pianosa machine as in the previous experiments. We also used the same annotation schema with different compilation flags. Figure 7.55 presents the execution times and latency for the replicas tested. Note that the number of replicas is the same for each stream region. The graphs clearly point out that using `spar_blocking` flag retrogrades the latency and completion time with respect to other versions. The reason for such degradation is attributed to the stream behavior that is fine grained and intensive. Consequently, when more replicas are assigned, more blocking synchronization is necessary. This becomes even worse when used with `spar_ondemand`, because more communication is performed to scheduling the stream elements.

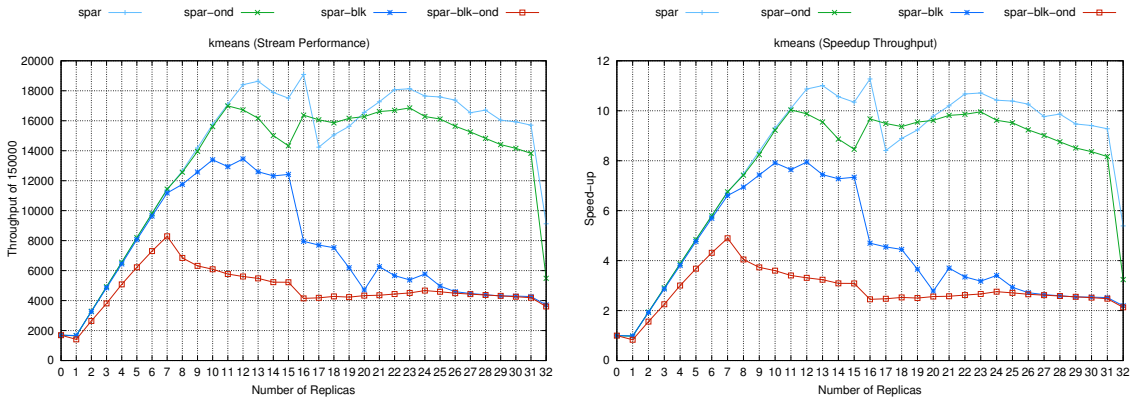


(a) K-Means execution times. (b) K-Means latency.

Figure 7.55: Time performance (K-Means)

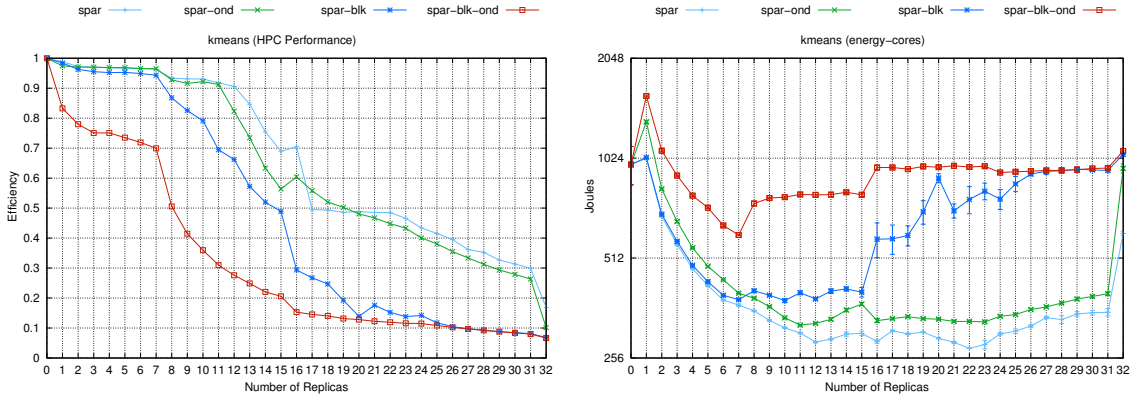
The impact of the differences can be observed in Figure 7.56 by the throughput and speedup. In general, when comparing the highest rates of the worst and best versions, the difference is about 10,000 points for the given time of the replica. This is very significant for the application's performance. Moreover, we can also see the efficiency and energy consumption in Figure 7.56, where no version was not good enough from 12 replicas up to 32.

With respect to the cache misses, all optimization flags presented similar performance. This can be seen in Figure 7.58. However, energy consumption was dramatically different, where the least efficient SPar version also consumed much more energy. This is an indication of overheads by the parallel code generated. Finally, the memory usage (Figure 7.59) was also significantly different for the versions without the `spar_ondemand` flag, which was similar to the previous applications because of the impact of FastFlow queues.



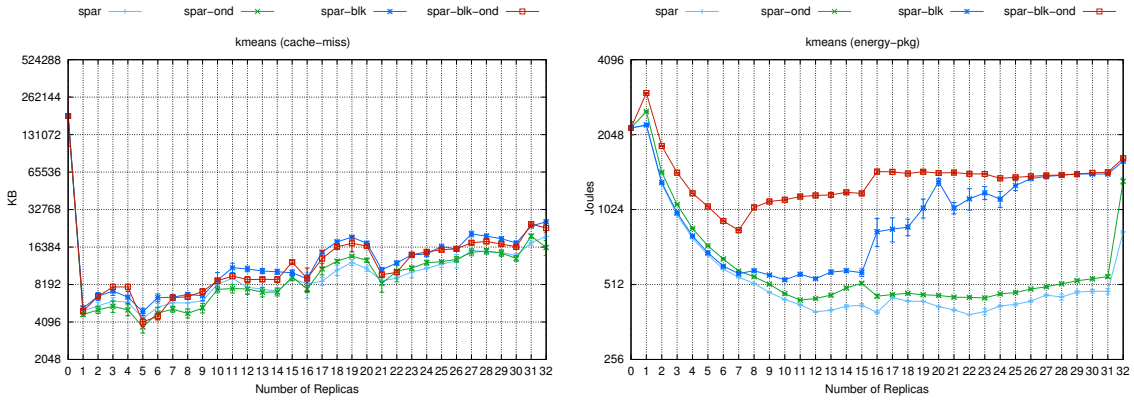
(a) K-Means throughput. (b) K-Means throughput speed-up.

Figure 7.56: Stream performance (K-Means)



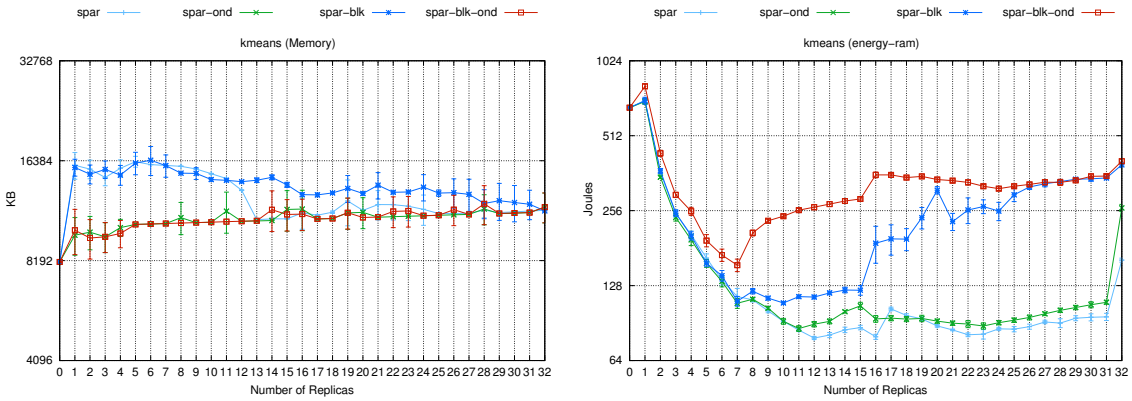
(a) K-Means HPC efficiency. (b) K-Means CPU cores energy consumption.

Figure 7.57: HPC performance (K-Means)



(a) K-Means cache misses. (b) K-Means socket energy consumption (cache and CPUs).

Figure 7.58: CPU Socket performance (K-Means)



(a) K-Means memory usage. (b) K-Means memory energy consumption.

Figure 7.59: Memory performance (K-Means)

7.3.5.2 Productivity Comparison

Since we tested alternative implementations using low-level patterns with FastFlow and TBB, it is not necessary to repeat such implementations to compare productivity because we noted before that this requires much more effort to implement. Also, in terms of data parallelism performance, the parallel for interface was demonstrated to be better suited for this purpose. Figure 7.60 presents the percentage difference of SLOCs when using each one of the programming frameworks. TBB and FastFlow require slightly more code because they need to add their library. While SPar and OpenMP, only annotations are necessary. The parallelized parts of the code are in Section A.3.5.

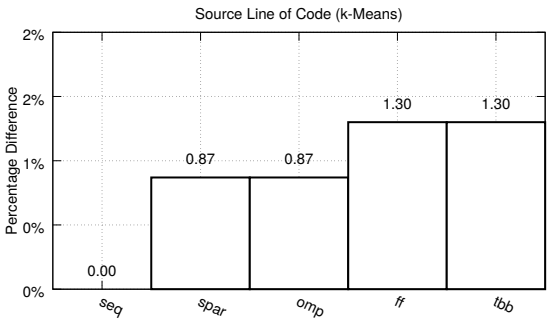


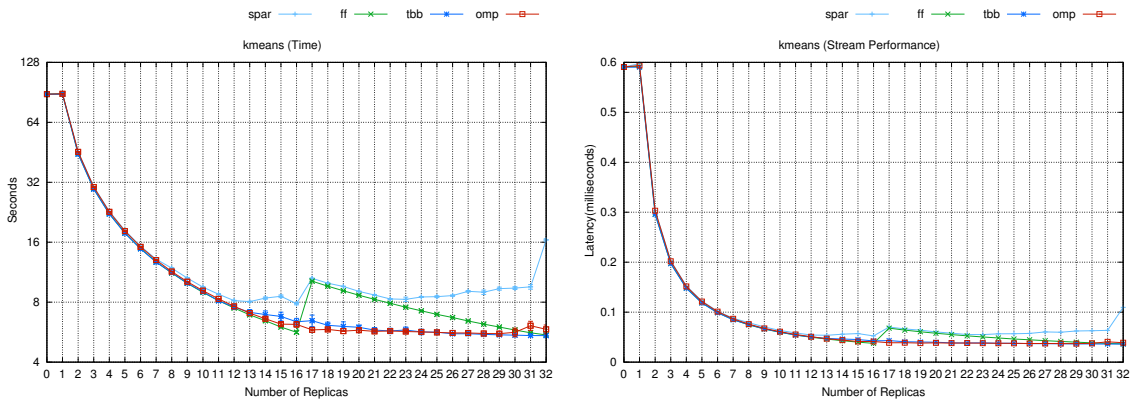
Figure 7.60: Source line of code for K-Means application.

When comparing SPar (Listing 7.6) to OpenMP (Listing A.17), SPar achieved the same productivity because it is placed along with the iteration statement and the stage must be introduced. Consequently, this is one case where OpenMP provides better productivity, because it was designed to be simple when there is easy data

parallelism. In contrast, the high-level SPar attributes are generic enough to annotate such parallelism. Specifically in this application, we can conclude that OpenMP and SPar provide equivalent coding productivity, while TBB (Listing A.19) and FastFlow (A.18) still requires minimal code rewriting in data parallelism.

7.3.5.3 Performance Comparison

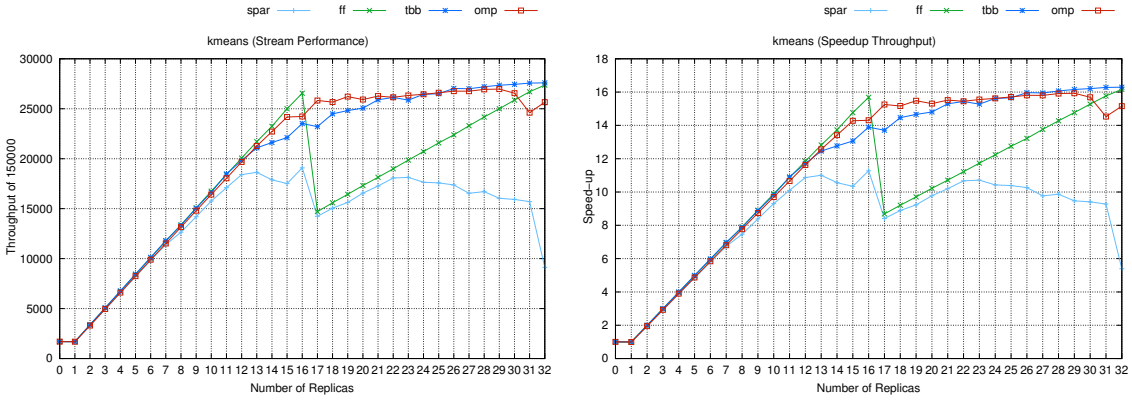
To compare performance, we used the same implementation version of the productivity comparison and the default SPar version, which presented the best performance among the options. Figure 7.61 plots the results of execution time and latency. We can see that SPar achieved the worst performance with respect to the others. Fastflow also had significant performance degradation starting with 17 replicas up to 30. This initial degradation achieved the same peak of SPar, which coincides with hyper threading facilities usage of the machine. We also expected that OpenMP and TBB would suffer some degradation starting at 17 replicas. Figure 7.62 shows that only FastFlow was able to achieve a linear throughput speedup until physical cores are full with replicas. The advantage is that less replicas are needed by FastFlow parallelism when compared to OpenMP and TBB, which only achieved the highest speedup by using 26 replicas.



(a) K-Means execution times. (b) K-Means latency.

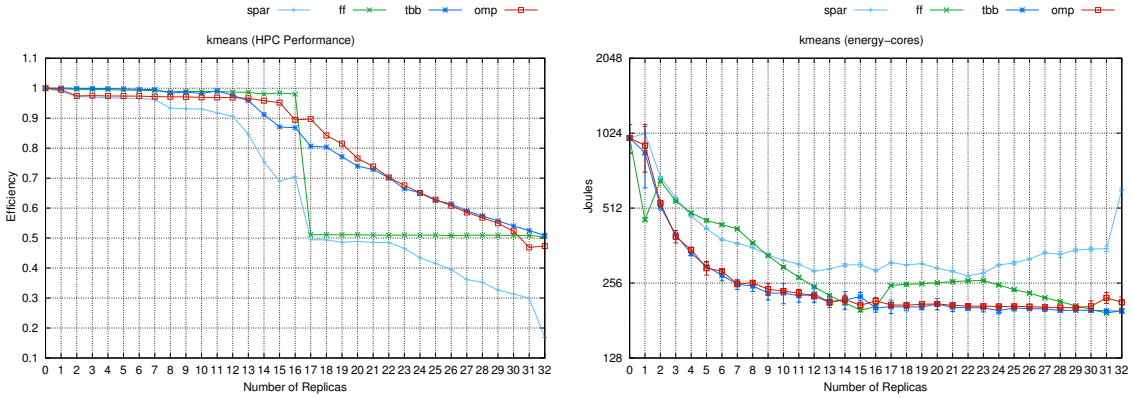
Figure 7.61: Time performance comparison (K-Means)

We can prove the efficiency of the programming frameworks in Figure 7.63 as well as evaluate the energy consumption of the CPU cores. As in the other applications, SPar and FastFlow usually consume more energy than the others. However, less cache misses are generated as can be observed in Figure 7.64. Concerning memory usage, there were contrasts among the interfaces, but OpenMP outperforms all other versions.



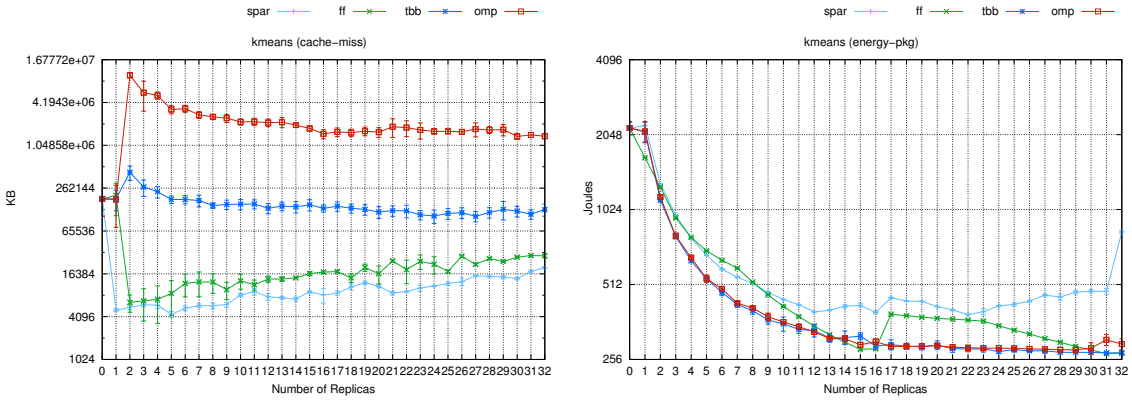
(a) K-Means throughput. (b) K-Means throughput speed-up.

Figure 7.62: Stream performance comparison (K-Means)



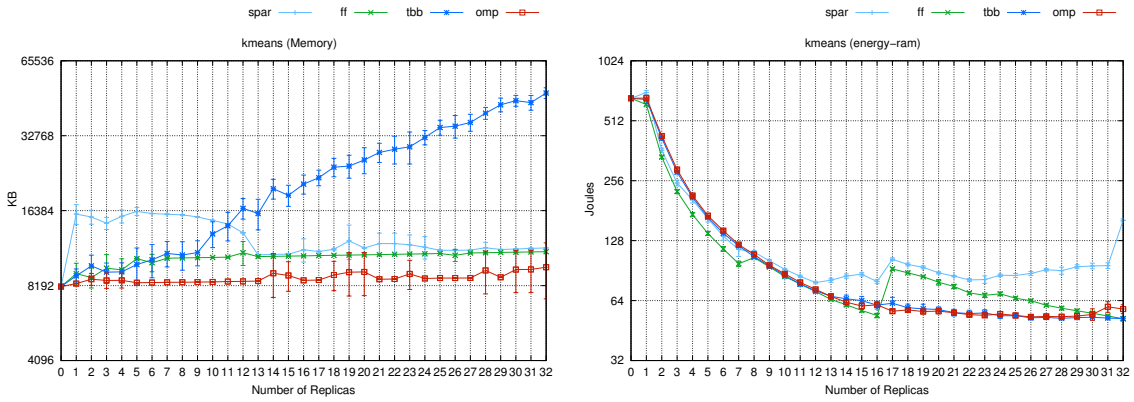
(a) K-Means HPC efficiency. (b) K-Means CPU cores energy consumption.

Figure 7.63: HPC performance comparison (K-Means)



(a) K-Means cache misses. (b) K-Means socket energy consumption (cache and CPUs).

Figure 7.64: CPU Socket performance comparison (K-Means)



(a) K-Means memory usage. (b) K-Means memory energy consumption.

Figure 7.65: Memory performance comparison (K-Means)

7.3.5.4 Summary

K-means application provided us a different challenge for SPar. Its interface was generic enough to annotate the parallelism in a stream fashion. The usage of SPar's optimization flags resulted in performance degradation. Also, coding productivity when considering SLOCs is not significant different because the interfaces provided suitable mechanisms for implementing this kind of parallelism. However, TBB and FastFlow still require more code rewriting when using lambda function.

SPar did not achieve similar performance as other specialized programming frameworks for data parallelism. Even FasFlow's interface for data parallelism suffers some degradations. Although not favorable to SPar, results give us interesting insights for investigating in the future alternatives to achieve better performance for data parallelism. However, our top research goal was to enable performance and productivity in stream parallelism applications.

7.4 Cluster Environment

This section shows the results of the experiments targeting the cluster environment to evaluate SPar's performance and code portability. In fact, we used two of the same applications previously discussed and we applied different transformation rules to each one of them. In case, the transformation rules used by the compiler in the multi-core environment are the same when manually generated for the respectively application in the cluster environment.

7.4.1 Sobel Filter

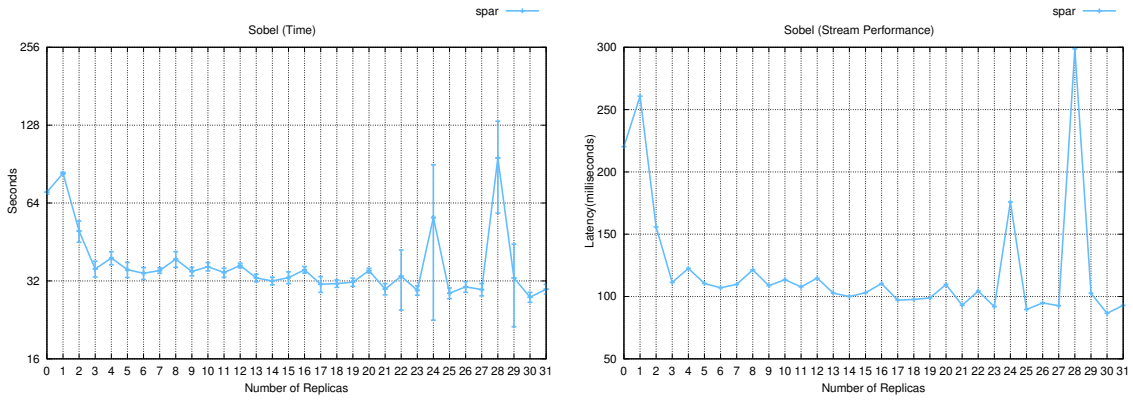
This application was previously discussed and annotated with SPar in Section 7.3.1. We provided two previous versions, but in this section we only generate the code manually for the second version, which is denoted as $[[T_0]]\{\square_0, [[S_0, R_n]]\{\square_1\}\}$. There are no any new attributes inserted and transformations follow Rule 6.1 described in Section 6.6. Our goal is to demonstrate that transformation rules are generalized and enough to achieve code portability.

In the cluster environment, our experiments were set up with balanced and unbalanced workloads. For the balanced workload 320 images were used with 3000x2250 resolution. On the other hand, the unbalanced workload was composed of 1,280 images, and four different resolutions were selected (800x600, 1024x768, 1600x1200 and 3000x2250). We ran the application ten times to take an average duration of the Dodge cluster (see the configuration in Section 7.2.2). The standard deviations are plotted through error bars in the graphs.

Figure 7.66 presents the graphs of execution time (a) and latency (b) for the replicas tested. The application reduced the completion time and latency in half in almost all replicas tested. If compared to the multi-core environment, the same performance is not expected because there is a network and distributed file system. The problem is that our application does not reduce the completion time when increasing the number of replicas. The throughput and speedup helped us to better understand the impacts in Figure 7.67.

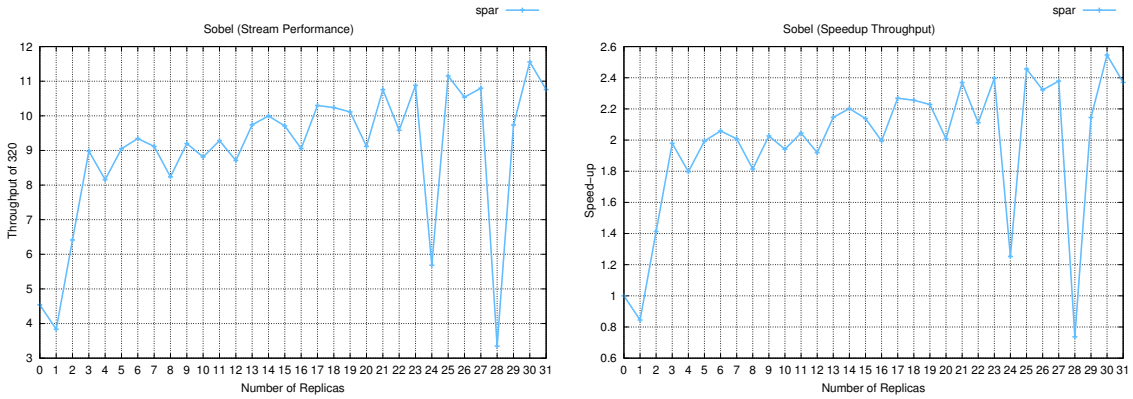
As previously discussed, the Sobel filter performs more in disk than CPU and the latency and overhead increases because of the distributed file system. In case, it will be hard to achieve speedups greater than two using this workload, since it is not concerning the environment. In the Appendix Section A.2 the results use unbalanced workloads. They presented similar results even though there were the highest number

of images. As the image size varies with smaller images, the latency was lower such as in Figure A.27(b) demonstrates.



(a) Sobel Filter execution times. (b) Sobel Filter latency.

Figure 7.66: Time performance using balanced workload (Sobel Filter)



(a) Sobel Filter throughput. (b) Sobel Filter throughput speed-up.

Figure 7.67: Stream performance using balanced workload (Sobel Filter)

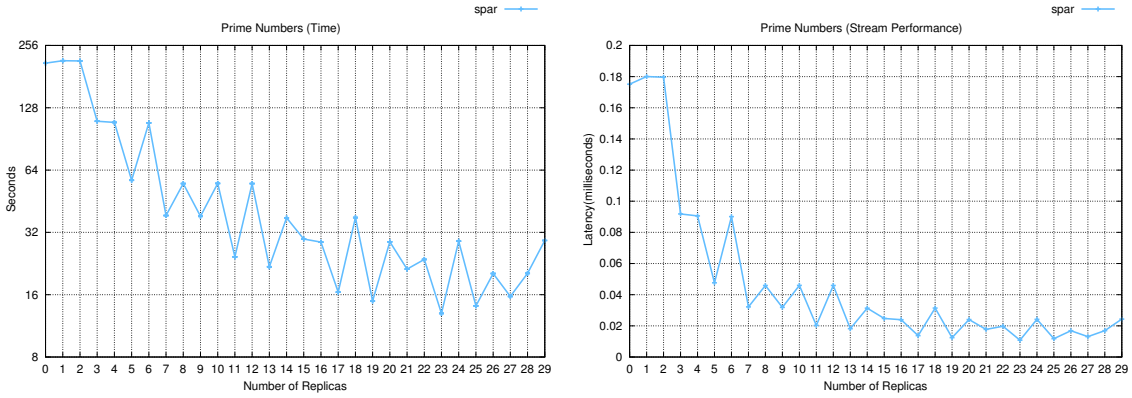
Another important aspect concerning the results is that we are using an older machine in the cluster environment and less sophisticated hardware and network. Unfortunately, the disk bottleneck impacts much more in the cluster environment. Thus, we can conclude that even providing code portability, performance may be not the same as the other environment. It depends not only on the code generated, but also on the hardware aspects and application constraints.

7.4.2 Prime Number

In the prime number application, the annotation schema is exactly the same as discussed before in Section 7.3.4 and can be denoted as $[[T_0]]\{\square_0, [[S_0, R_n]]\{\square_1\}, [[S_1]]\{\square_2\}\}$.

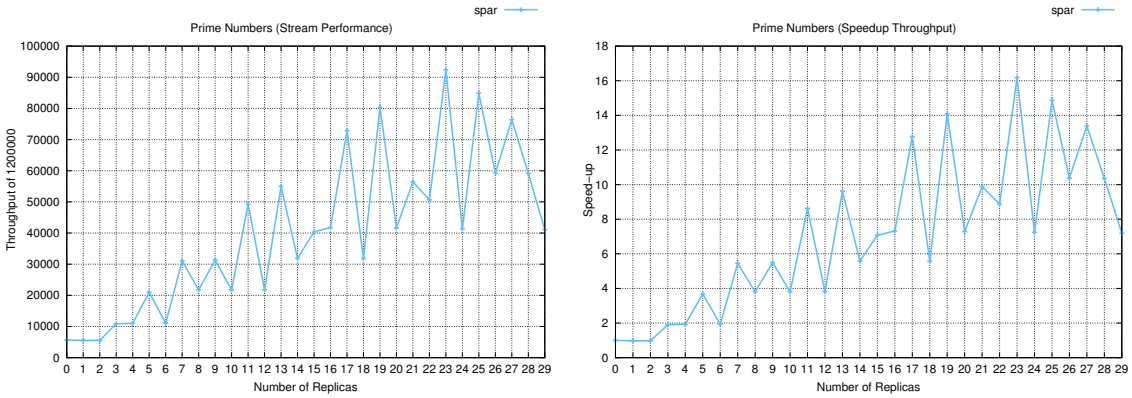
Consequently, we applied Rule 6.3 to manually generate the code, so that we can demonstrate that code portability is possible. The experiment was conducted in the Dodge cluster and the problem size was the same as in multi-core, which is to find the primers from 1 to 1,200,000 numbers.

Figure 7.68 presents the results concerning completion time metric. The contrasts are similar as presented in Figure 7.44 when the code generated is without optimization flags, which is performing in a round robin fashion. Thus, this result was also expected in a cluster environment, which can better observed in Figure 7.69 through the throughput rates and speedup. As the machines are different (in the cluster and multi-core environments), the results in the original source code were also different for latency and completion time (with 0 replica).



(a) Prime Numbers execution times. (b) Prime Numbers latency.

Figure 7.68: Time performance (Prime Numbers)



(a) Prime Numbers throughput. (b) Prime Numbers throughput speed-up.

Figure 7.69: Stream performance (Prime Numbers)

We already highlighted the importance of the scheduler in this application in Section 7.3.4. Unfortunately, for the cluster environment, our runtime did not providing this optimization. We expect in the future to also support an on-demand

scheduler to enable better performance for applications where performance depends on the scheduler. Unlike the Sobel filter application, we demonstrated through this experiment that we were also able to provide performance portability along with code portability, like a CPU bound application.

7.5 Summary

In this chapter, we first performed experiments in the multi-core environment to evaluate and compare Spar's performance and code productivity to TBB, FastFlow, OpenMP, and eventually Pthreads. Our performance results were similar as those tuning manually and better than TBB and OpenMP for streaming applications. Also, our experiments demonstrated that we reduced the programming effort significantly and increased the coding productivity compared to TBB and FastFlow. As expected, SPar was able to annotate data parallel computations and maintain productivity, but provided less performance than OpenMP.

In the cluster environment, we evaluated the code portability by using two identical annotated applications previously tested for the multi-core environment. Thus, code portability was granted through the transformation rules proposed for both environments. However, performance portability also depends on application features (*e.g.*, the disk bottleneck), and runtime support for scheduling optimizations.

Part IV

Discussions

8

Conclusions

This chapter will present the conclusions of the thesis.

Contents

8.1 Overview

This dissertation has contributed to the fields of domain-specific language design and support tools for high-level stream parallelism. The support tools proposed include the Compiler Infrastructure for New C/C++ Languages Extensions (CINCLE). CINCLE uses several standard tools to provide a simpler and more efficient environment/infrastructure for generating standard C++ embedded DSLs, especially for introducing the C++ attribute mechanism. Moreover, CINCLE was implemented to support the user with aggressive source-to-source code transformations (AST to AST) as well as to provide an AST that is fully compliant with the standard grammar. Also, it provides a set of APIs and a tree visualization library to increase productivity and accelerate the learning curve when designing DSLs.

CINCLE provides essential features and capabilities for high-level abstraction that have been used to build an embedded C++ DSL for stream parallelism (named as SPar). SPar helps users to annotate parallelism with only five attributes that are implemented using the standard C++ attribute mechanism. The language terms are related only the streaming domain, avoiding low-level parallel programming aspects such as models, scheduling policies, load balancing, and others. SPar essentially aims to enhance code productivity without significantly degraded performance. We also created a methodology where developers ask themselves five questions that will instruct them what to do when annotating the code. Thus, it guides them reduce programming effort and achieve efficient stream parallelism.

The design of the SPar language was also essential to introduce code portability. We therefore created generalized transformation rules to translate SPar attributes to parallel patterns. These rules are independent of the target architecture and programming framework, and can therefore be easily implemented into a compiler algorithm or manually generated. First, the SPar compiler was implemented to perform source-to-source code transformations automatically targeting multi-cores using the FastFlow framework and accomplish the interpretation and semantic analysis of the attributes. Then, we demonstrated how the translations target code portability by manually generating the code for clusters using the MPI library.

Lastly, we performed a set of experiments to evaluate productivity, performance, and code portability. We picked five different applications that were annotated with SPar as well as implemented with TBB, FastFlow, OpenMP, and later Pthreads. Thus, productivity and performance were evaluated in a multi-core environment. Code portability was tested by implementing two of the five applications in a cluster environment without changing the source code tested in the multi-core experiments, while using the same transformation rules used to generate the multi-core versions.

8.2 Assessments

In the dissertation we achieved all of the established goals. The first goal was *to provide support tools that enable parallel programmers to create standard C++ embedded DSLs*. We met this goal by creating CINCLE. It was sufficient to support us in the implementation of the SPar compiler. We were able to perform complex source-to-source transformations and semantic analysis using CINCLE. Also, in Section 5.7, we highlighted that CINCLE simplified DSL design and, through real use cases (Section 4.9), that it performed quite well.

Our second goal was *to provide high-level stream parallelism and coding productivity without significant performance degradation in multi-core systems*. We achieved this goal by creating SPar. The experiments (Section 7.3) and presentation in Chapter 5 revealed that SPar provides a simpler vocabulary with only five attributes sufficient to annotate different kinds of applications and parallelism. High-level stream parallelism is achieved because C++ programmers do not have to be aware of any low-level parallel programming aspects. The results of the experiments proved that SPar is more productive without demonstrating any significant performance degradation in stream applications. Moreover, it is able to provide the same productivity in data parallel applications with competitive performance (*e.g.* K-Means and Prime Numbers applications) when compared to other frameworks (TBB, FastFlow and OpenMP). In addition, SPar performed similar to FastFlow (implemented manually), illustrating that the automatic code generation designed was efficient and does not add significant overhead.

The third goal was *to introduce code portability in multi-core and cluster systems*. We met this goal by introducing generalized transformation rules from SPar attributes targeting parallel patterns. We demonstrated that it is possible to provide code portability in SPar because we were able to transform code integration the transformation rules in the compiler to automatically generate parallel code using FastFlow (multi-cores) and manually generate code using the MPI library (clusters). Although code portability was achieved, the results demonstrated that performance portability depends on application characteristics and scheduling optimizations of the runtime support.

In addition to our main goals, we have met the programming framework aims as well as support application level DSL to entirely abstract parallelism exploitation. We achieved a vertical validation of the framework in a collaborative research project that created a DSL for geospatial visualizations (named as GMaVis) [Led16]. GMaVis provides a descriptive language that generates a robust C++ code along with SPar

annotations to take advantage of parallelism in multi-core architectures. It is important to highlight this study because we validated our perspective and the results demonstrated increased code productivity and simplicity for the designer of the DSL application. These results verify the thesis in a completely different scenario.

8.3 Limitations

The limitations relate to real use cases and experiments regarding productivity, performance, and code portability. We cannot generalize performance and productivity in an application that is not stream-oriented. SPar is not able to deal with state-full stages and stages with feedbacks channels. This is also a research challenge in the parallel programming field that has not been solved in a completely abstract and general way. In SPar, we recommend that state-full stages should not be replicated.

CINCLE's infrastructure was initially made to deal with C++ attributes. Even though we proposed it as a more generic support tool, there is no guarantee at the moment that it also works for other kinds of language extensions that are not C++ annotations. There are limitations concerning the AST because Bison cannot solve all ambiguities. Therefore, we need to test CINCLE with bigger amount of code and an algorithm must be created to address complex ambiguities after the creation of the AST. SPar's code generation was not a problem, but it would become a problem if we were to perform DataFlow and other sophisticated analysis at compilation time.

Our generalized transformation rules are not generic enough when other domains are taken into account. Although generalized, the rules we used only deal with SPar attributes and may produce only farm, pipeline, and compositions of pipeline with farm stages. They are so-called generalized because they were created to enable one to produce more complex transformations for other SPar annotation sentences allowed by its semantics. Consequently, code portability inherits these limitations either integrating them into the compiler algorithm to perform changes automatically in the AST or for manually generating code.

8.4 Considerations

The dissertation also contributes to the wider scenario of computer science dealing with parallel patterns, FastFlow runtime, and the C++ community. Concerning parallel patterns, we stress the fact that they were proposed independently of a target architecture, and we have proven that they can also be integrated with a different

scenario such as DSLs. For instance, SPar provides a higher abstraction level that can be translated into parallel code using a parallel patterns approach. On the other hand, using FastFlow we further contributed to stresses its performance and flexibility to exploit parallelism. We bring significant insights to different performance metrics when compared to other state-of-the-art frameworks. Because SPar can easily combine different annotation sentences, we can further expand the test scenarios to FastFlow runtime so that it can be improved in the future and support better performance and flexibility.

Using the standard C++ attribute mechanism, we contributed by providing a use case and pointing out the challenges that were not clear before starting the DSL design. This work has proven that this mechanism is a suitable alternative to providing high-level abstraction and powerful source-to-source transformations. Unfortunately, we found out that there is weak support in the documentation and there is no clear understanding what can be provided. We are happy with the design of CINCLE, which can now also be used for other DSLs aiming at parallelism abstractions. Moreover, SPar is a real use case that can be proposed to directly integrate into the grammar, because it is a "*de-facto*" embedded in C++ language.

9

Future Work

This chapter will present and discuss potential future work related to the contributions of this dissertation.

Contents

9.1 Programming Framework

This dissertation has achieved and presented contributions in our programming framework. They may be extended in different ways in the coming years to improve our findings in high-level parallel programming, compiler-based tools, and the stream parallelism domain.

In addition to the opportunities to expand and improve our specific contributions, which will be described in detail in the next sections, we will also give some ideas of what we expect to achieve in the customization space and support space in the next years (see Figure 3.1). In the customization space, our collaborative work in [Led16] only targeted a small set of applications. There are a variety of applications to be explored and we are seeking DSLs in the application layer that will be embedded with C++ to reuse the CINCLE infrastructure, because GMaVis is an external DSL (a completely new language). For the support space, there is the possibility to create/reuse different runtimes targeting virtual machines, GPU, DSP, FPGA, and others.

9.1.1 CINCLE

CINCLE is not ready to release to the scientific community because it is still being improved. Considering the time spent in related projects to consolidate their tools, we have a great deal of work ahead in terms of software development and testing to make CINCLE a competitive tool in the C++ DSL design space. In addition to the technical aspects and the clear documentation that need to be completed, in the future we aim to provide a better API by using C++ templates and object oriented approaches. Moreover, we intend to improve CINCLE to support more complex source code analysis regarding data-flow, pattern matching, automatic code parallelization, other internal optimizations, and compiler techniques for code generation.

9.1.2 SPar

The performance experiments have demonstrated opportunities for SPar to generate more efficient code by using FastFlow with respect to memory and energy. One possible future project would be to evaluate the FastFlow runtime to optimize code generation, combining different sized queues to increase memory performance. Another

possible optimization would be to implement dynamic changing of the blocking and non-blocking mode to improve energy efficiency.

Concerning the DSL interface and capabilities, it would be interesting to support users by offering stages for feedback communication. Another option would be to add sliding window option in the Stage attribute. This could give more opportunities for fine tuning streaming applications. Moreover, new optimization flags could be created to target different schedulers to increase performance for a wider set of applications and architectures.

At the present moment we only support the replication of stateless stages. The benchmark characterization of StreamIt is presented in Figure 9.1. This is the result of streaming applications for the filter types (stages). As we can observe, these applications are almost always composed of stateless stages, only six percent were state-full. In fact, half of this six percent are avoidable by using internal compiler techniques. Therefore, as future work, we can investigate techniques to avoid state-full stages while the rest are unavoidable in benchmark applications.

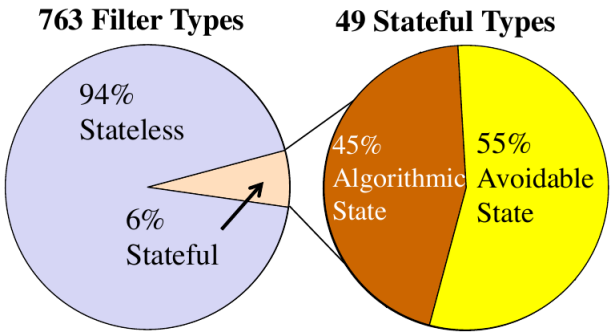


Figure 9.1: Statistics of StreamIt benchmarks [TA10]. Extracted from [Won12].

When transformation rules for clusters are also integrated in the SPar compiler, a future work could be to exploit hybrid parallelism. Here we can combined code generation to exploit clusters with multi-core machines. Because there is one MPI process inside each machine, it could be equipped with FastFlow code to exploit lightweight multi-thread parallelism. This implementation could optimize the performance and have better scale for some cluster applications.

9.1.3 Transformation Rules

Our transformation rules target only stream parallel patterns. As future work, we plan to include data parallel patterns like map and reduce, when possible. An example would be the following SPar sentence $[[T_0]]\{[[S_0, R_n]]\{\square_1\}\}$. In this case, there is a

`ToStream` annotation in front of the "for" loop and inside of the loop the immediately sentence is a stage block declaration. Due to the fact that it has the attribute replicate, we can assume that it can run independently. Also, there is no $\square$ between the annotation T_0 and S_0. Consequently, we could apply the map parallel pattern or transform it into a parallel "for". This is just one hypothetical case to achieve better performance transformation rules when data parallel computations are annotated using SPar. There will probably be other similar cases. Therefore, there are many possibilities for future investigation if it works well for maintaining consistency with different parallel architecture targets.

9.2 Experiments

There is a lack of streaming application benchmarks in C++ and most of the state-of-the-art benchmarks are low-level and use old C code. This causes many incompatibilities when compiling with the new standard compiler. A future work would be to create a standard C++ benchmark that tests all stream parallelism properties. Also, it would be interesting to do experiments with robust applications. In this case, a big challenge for future work would be to implement some of the PARSEC and StreamIt benchmarks.

In the cluster environment, we have tested only two transformation rules. A future work is to conduct more experiments in this environment with all of the applications experimented in the dissertation. Other experiments to benchmark complex transformations rules would be interesting to observe their efficiency. Also, to test if equivalent transformation rules are also equivalent in performance.

Concerning code productivity, it would be interesting to expand the experiments taking into account other metrics such as those presented in [SS96], where experiments were conducted with students. Also, software engineering methods could be applied such as the COCOMO II model [BAB+00] to predict cost and programming effort.

Part V

Complements

10

Bibliography

[AAC+11] Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Pierre Villalon. PIPS Is not (only) Polyhedral Software. In *First International Workshop on Polyhedral Compilation Techniques*, IMPACT' 11, page 6, Chamonix, France, April 2011.

[ACD+15] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Pool Evolution: A Domain Specific Parallel Pattern. *International Journal of Parallel Programming (IJPP)*, pages 1–21, March 2015.

[AD07] Marco Aldinucci and Marco Danelutto. Skeleton-based Parallel Programming: Functional and Parallel Semantics in a Single Shot. *Computer Languages, Systems and Structures*, 33(3):179–192, October 2007.

[ADA+12] Marco Aldinucci, Marco Danelutto, Lorenzo Anardu, Massimo Torquati, and Peter Kilpatrick. Parallel Patterns + Macro Data Flow for Multi-Core Programming. In *20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 27–36, Garching, Germany, Febuary 2012. IEEE.

[ADK+11] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating Code on Multi-Cores with FastFlow. In *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 170–181, Bordeaux, France, September 2011. Springer Berlin Heidelberg.

[ADK+12] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An Efficient Unbounded Lock-Free Queue for Multi-core Systems. In *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 662–673, Rhodes Island, Greece, August 2012. Springer Berlin Heidelberg.

[ADKT12] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting Heterogeneous Architectures Via Macro Data Flow. *Parallel Processing Letters*, 22(2):12, May 2012.

[ADKT14] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. FastFlow: High-Level and Efficient Streaming on Multi-core. In *Programming Multi-core and Many-core Computing Systems*, volume 1 of *Parallel and Distributed Computing*, page 14, Pisa, Italy, March 2014. Wiley.

[ADM+09] Marco Aldinucci, Marco Danelutto, Massimiliano Meneghin, Peter Kilpatrick, and Massimo Torquati. Efficient Streaming Applications on Multi-Core with FastFlow: the Biosequence Alignment Test-Bed. In *Parallel Computing: From Multicores and GPU's to Petascale*, volume 19 of *Advances in Parallel Computing*, pages 273–280, Lyon, France, September 2009. IOS Press.

[ADP+14] Marco Aldinucci, Maurizio Drocco, Guilherme Peretti Pezzi, Claudia Misale, Fabio Tordini, and Massimo Torquati. Exercising High-Level Parallel Programming on Streams: A Systems Biology use Case. In *34th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 51–56, Madrid, Spanish, July 2014. IEEE.

[AGJ+14] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 647–658, New Orleans, Louisana, November 2014. IEEE Press.

[AGLF15a] Daniel Adornes, Dalvan Griebler, Cleverson Ledur, and Luiz G. Fernandes. A Unified MapReduce Domain-Specific Language for Distributed and Shared Memory Architectures. In *The 27th International Conference on Software Engineering & Knowledge Engineering*, page 6, Pittsburgh, USA, July 2015. Knowledge Systems Institute Graduate School.

[AGLF15b] Daniel Adornes, Dalvan Griebler, Cleverson Ledur, and Luiz G. Fernandes. Coding Productivity in MapReduce Applications for Distributed and Shared Memory Architectures. *International Journal of Software Engineering and Knowledge Engineering*, 25(10):1739–1741, December 2015.

[AGT14] Henrique C. M. Andrade, Bugra Gedik, and Deepak S. Turaga. *Fundamentals of Stream Processing.* Cambridge University Press, New York, USA, 2014.

[AI91] Corinne Ancourt and François Irigoin. Scanning Polyhedra with DO Loops. In *Third ACM SIGPLAN Symposium on Principles and Prac-*

tice of Parallel Programming, PPOPP '91, pages 39–50, Williamsburg, Virginia, USA, April 1991. ACM.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Person Addison Wesley, Boston, USA, 2007.

[Ami12] Mehdi Amini. *Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators*. PhD thesis, École Nationale Supérieure des Mines de Paris, Paris, French, December 2012.

[AMT10] Marco Aldinucci, Massimiliano Meneghin, and Massimo Torquati. Efficient Smith-Waterman on Multi-Core with FastFlow. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 195–199, Pisa, Italy, February 2010. IEEE.

[And13] Quinton Anderson. *Storm Real-time Processing Cookbook*. PACKT, Birmingham, UK, 2013.

[APD+15] Marco Aldinucci, Guilherme Peretti Pezzi, Maurizio Drocco, Concetto Spampinato, and Massimo Torquati. Parallel Visual Data Restoration on Multi-GPGPUs using Stencil-Reduce Pattern. *International Journal of High Performance Computing Application*, 29(4):461–472, 2015.

[ATD+13] Marco Aldinucci, Fabio Tordini, Maurizio Drocco, Massimo Torquati, and Mario Coppo. Parallel Stochastic Simulators in System Biology: the Evolution of the Species. In *21th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 410–419, Belfast, UK, February 2013. IEEE.

[BAB+00] Barry W. Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer, and Bert Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall, Upper Saddle River, United States, 2000.

[BC05] Anne Benoit and Murray Cole. Two Fundamental Concepts in Skeletal Parallel Programming. In *International Conference on Computational Science (ICCS)*, volume 3515 of *LNCS*, pages 764–771, USA, May 2005. Springer.

[BDLT13] Daniele Buono, Marco Danelutto, Silvia Lametti, and Massimo Torquati. Parallel Patterns for General Purpose Many-Core. In *21th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 131–139, Belfast, UK, February 2013. IEEE.

[BE05] Ayon Basumallik and Rudolf Eigenmann. Towards Automatic Translation of OpenMP to MPI. In *19th Annual International Conference on Supercomputing*, ICS '05, pages 189–198, Cambridge, Massachusetts, June 2005. ACM.

[BGP14] Suresh Boob, Horacio Gonzalez–Vélez, and Alina Madalina Popescu. Automated Instantiation of Heterogeneous Fast Flow CPU/GPU Parallel Pattern Applications in Clouds. In *22th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 162–169, Torino, Italy, February 2014. IEEE.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Symposium on Principles and Practice of Parallel Programming*, volume 30 of *PPOPP '95*, pages 207–216, USA, August 1995. ACM.

[BMD+11] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesús Labarta. Productive Cluster Programming with OmpSs. In *17th International Conference on Parallel Processing*, Euro-Par'11, pages 555–566, Bordeaux, France, August 2011. Springer-Verlag.

[BME07] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. Programming Distributed Memory Sytems Using OpenMP. In *IEEE International on Parallel and Distributed Processing Symposium*, IPDPS' 2007, pages 1–8, Long Beach, CA, March 2007. IEEE.

[BML+12] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Aurangzeb, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P. Midkiff. The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation. *International Journal of Parallel Programming*, 41(6):753–767, August 2012.

[BPD+12] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. Productive Programming of GPU Clusters with OmpSs. In *26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 557–568, Shanghai, China, May 2012. IEEE Computer Society.

[BSL+11] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, October 2011. IEEE Computer Society.

[CCA+10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 21, San Jose, CA, USA, April 2010. USENIX Association.

[Cha16] Charm++. Parallel Programming Framework. http://charmplusplus.org/, Febuary 2016.

[Cil16] Intel Cilk. Intel® Cilk Plus. https://www.cilkplus.org/, Febuary 2016.

[CJvdP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. MIT Press, London, UK, 2007.

[Cla16] Clang. The Clang's Documentation. http://clang.llvm.org/docs/, Febuary 2016.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, USA, 1989.

[Col04] Murray Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, March 2004.

[CSB+11] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A Domain-specific Approach to Heterogeneous Parallelism. In *16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 35–46, San Antonio, TX, USA, March 2011. ACM.

[DBM+09] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Centus: A Source-to-Source Compiler Infrastructure for Multicores. *IEEE Computer*, 42(12):36–42, 2009.

[DDST14] Marco Danelutto, Luca Deri, Daniele De Sensi, and Massimo Torquati. Deep Packet Inspection on Commodity Hardware using FastFlow. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 92–99, Munich, Germany, September 2014. IOS Press.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications ACM*, 51(1):107–113, January 2008.

[DGS+16] Marco Danelutto, Jose Daniel Garcia, Luis Miguel Sanchez, Rafael Sotomayor, and Massimo Torquati. Introducing Parallelism by using

REPARA C++11 Attributes. In *24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, page 5. IEEE, February 2016.

[DJP+11] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, page 12, Seattle, Washington, November 2011. ACM.

[DK14] Marco Danelutto and Haileyesus Alemu Kifle. Stream parallel computations on GPUs. In *International Workshop on High-level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU)*, pages 26–32, Viena, Austria, January 2014. ACM.

[DM06] V Dolotin and A Morozov. *The Universal Mandelbrot Set*. World of Science Press, Singapore, US, 2006.

[DT14] Marco Danelutto and Massimo Torquati. Loop Parallelism: A New Skeleton Perspective on Data Parallel Patterns. In *22th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 52–59, Torino, Italy, February 2014. IEEE.

[DT15] Marco Danelutto and Massimo Torquati. Structured Parallel Programming with "core" FastFlow. In *Central European Functional Programming School: 5th Summer School*, volume 8606 of *Lecture Notes in Computer Science*, pages 29–75, Cluj-Napoca, Romania, July 2015. Springer International Publishing.

[DTK15] Marco Danelutto, Massimo Torquati, and Peter Kilpatrick. A Green Perspective on Structured Parallel Programming. In *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 430–437, Turku, Finland, March 2015. IEEE.

[Fas16] FastFlow. FastFlow website. http://mc-fastflow.sourceforge.net/, Febuary 2016.

[FHLLB09] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and Other Cilk++ Hyperobjects. In *Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, Calgary, AB, Canada, August 2009. ACM.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, Boston, USA, 2010.

[Fur14] Ash Furrow. *Functional Reactive Programming on iOS*. Leanpub, 2014.

[GAF14] Dalvan Griebler, Daniel Adornes, and Luiz G. Fernandes. Performance and Usability Evaluation of a Pattern-Oriented Parallel Programming Interface for Multi-Core Architectures. In *The 26th International Conference on Software Engineering & Knowledge Engineering*, pages 25–30, Vancouver, Canada, July 2014. Knowledge Systems Institute Graduate School.

[GCC16a] GCC. The GNU Compiler Collection. https://gcc.gnu.org/, Febuary 2016.

[GCC16b] GNU GCC. Plugins (online documentation). https://gcc.gnu.org/onlinedocs/gccint/Plugins.html, March 2016.

[GDTF15] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz G. Fernandes. An Embedded C++ Domain-Specific Language for Stream Parallelism. In *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing*, ParCo'15, pages 317–326, Edinburgh, Scotland, UK, September 2015. IOS Press.

[GF13] Dalvan Griebler and Luiz G. Fernandes. Towards a Domain-Specific Language for Patterns-Oriented Parallel Programming. In *Programming Languages - 17th Brazilian Symposium - SBLP*, volume 8129 of *Lecture Notes in Computer Science*, pages 105–119, Brasilia, Brazil, October 2013. Springer Berlin Heidelberg.

[GHJV02] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, USA, 2002.

[GHLL+98] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference*. MIT Press, London, England, 1998.

[Gho11] Debasish Ghosh. *DSLs in Action*. Manning publications Co., Stamford, CT, USA, 2011.

[Gor10] Michael I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 2010.

[Gre14] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall, Boston, USA, 2014.

[Gri12] Dalvan J. Griebler. Proposta de uma Linguagem Específica de Domínio de Programação Paralela Orientada a Padrões Paralelos: Um Estudo de Caso Baseado no Padrão Mestre/Escravo para Arquiteturas Multi-Core. Master's thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, March 2012.

[GSFS15] Jose Daniel García, Rafael Sotomayor, Javier Fernández, and Luis Miguel Sánchez. Static Partitioning and Mapping of Kernel-Based Applications Over Modern Heterogeneous Architectures. *Simulation Modelling Practice and Theory*, 58(1):79–94, November 2015.

[Gue11] Serge Guelton. *Building Source-to-Source Compilers for Heterogeneous Targets*. PhD thesis, École Nationale Supérieure des Mines de Paris, Paris, French, October 2011.

[Gus11] John L. Gustafson. Little's Law. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1038–1041. Springer US, 2011.

[GVL10] Horacio González-Vélez and Mario Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Software Practice & Experience*, 40(12):1135–1160, November 2010.

[Had16] Hadoop. Apache Hadoop. http://hadoop.apache.org/, Febuary 2016.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan-Kaufmann, Boston, USA, 2011.

[HSWO14] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying Scalable Graph Processing with a Domain-Specific Language. In *Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 11, Orlando, FL, USA, March 2014. ACM.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In *5th International Conference on Supercomputing*, ICS '91, pages 244–251, Cologne, West Germany, June 1991. ACM.

[ISO11a] ISO/IEC-14882:2011. Information Technology - Programming Languages - C++. Technical report, International Standard, Geneva, Switzerland, August 2011.

[ISO11b] ISO/IEC-9899:2011. Information technology - Programming languages - C. Technical report, International Standard, Geneva, Switzerland, December 2011.

[ISO14] ISO/IEC-14882:2014. Information Technology - Programming Languages - C++. Technical report, International Standard, Geneva, Switzerland, December 2014.

[JLF+05] Troy A. Johnson, Sang-Ik Lee, Long Fei, Ayon Basumallik, Gautam Upadhyaya, Rudolf Eigenmann, and Samuel P. Midkiff. Experiences in Using Cetus for Source-to-source Transformations. In *17th International Conference on Languages and Compilers for High Performance Computing*, LCPC'04, pages 1–14, West Lafayette, IN, September 2005. Springer-Verlag.

[Kar05] Bjorn Karlsson. *Beyond the C++ Standard Library: Introduction to Boost*. Addison Wesley Professional, Boston, USA, 2005.

[KB16] Adrian Kaebler and Gary Bradski. *Learning OpenCV Computer Vision in C++ with the OpenCV library*. O'Reily Press, Sebastopol, CA, 2016.

[KKWZ15] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis*. O'Reilly Media, Sebastopol, CA, USA, 2015.

[KmWH10] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier, Burlington, MA, USA, 2010.

[LA14] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries*. PACKT Publishing, Birmingham, UK, 2014.

[Led16] Cleverson Ledur. GMaVis: A Domain-Specific Language for Large-Scale Geospatial Data Visualization Supporting Multi-core Parallelism. Master's thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, March 2016.

[Lei09] Charles E. Leiserson. The Cilk++ Concurrency Platform. In *46th Annual Design Automation Conference*, DAC '09, pages 522–527, San Francisco, California, July 2009. ACM.

[Lev09] John R. Levine. *Flex & Bison*. O'Reilly Media Inc., Sebastopol, CA, 2009.

[LGMF15] Cleverson Ledur, Dalvan Griebler, Isabel Manssuor, and Luiz G. Fernandes. Towards a Domain-Specific Language for Geospatial Data Visualization Maps with Big Data Sets. In *ACS/IEEE International Conference on Computer Systems and Applications*, AICCSA'15, page 8, Marrakech, Marrocos, November 2015. IEEE.

[LLS+13] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. On-the-fly Pipeline Parallelism. In *ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 140–151, Portland, Oregon, USA, June 2013. ACM.

[Ló14] Sergio Aldea López. *Compile-Time Support for Thread-Level Speculation*. PhD thesis, Universidad de Valladolid, Valladolid, Spain, July 2014.

[Mil15] Joshua John Milthorpe. *X10 for High-Performance Scientific Computing*. PhD thesis, Australian National University, Australia, March 2015.

[Mis14] Claudia Misale. Accelerating Bowtie2 With a Lock-Less Concurrency Approach and Memory Affinity. In *22th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 578–585, Torino, Italy, February 2014. IEEE.

[MMPSC08] Daniel Millot, Alain Muller, Christian Parrot, and Frédérique Silber-Chaussumier. STEP: A Distributed OpenMP for Coarse-Grain Parallelism Tool. In *OpenMP in a New Era of Parallelism: 4th International Workshop*, IWOMP'08, pages 83–99, West Lafayette, IN, USA, May 2008. Springer Berlin Heidelberg.

[MRR12] Michael McCool, Arch D. Robison, and James Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, MA, USA, 2012.

[MS12] Donald Miner and Adam Shook. *MapReduce Design Patterns*. O'Reilly, Sebastopol, CA, USA, 2012.

[MSM05] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, Boston, USA, 2005.

[MW08] Jens Maurer and Michael Wong. Towards Support for Attributes in C++ (Revision 6). Technical report, The C++ Standards Committee, September 2008.

[NBF96] Bradford Nicbols, Dick Buttlar, and Jacqueline P. Farrell. *Pthreads Programming*. O'Reilly, Sebastopol, USA, 1996.

[Omp16] OmpSs. The OmpSs Programming Model. https://pm.bsc.es/ompss, Febuary 2016.

[Ope16] OpenMP. Open Multi-Processing API specification for parallel programming. http://openmp.org/, Febuary 2016.

[OSV10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala 2nd*. Artima Press, Walnut Creek, California, 2010.

[Par13] Terrence Par. *The Definitive ANTRL 4 Reference*. The Pragmatic Programmers, 2013.

[PBAL13] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Self-Adaptive OmpSs Tasks in Heterogeneous Environments. In *IEEE 27th International Symposium on Parallel Distributed Processing*, IPDPS '13, pages 138–149, Boston, MA, May 2013. IEEE Computer Society.

[PC11] Antoniu Pop and Albert Cohen. A Stream-Computing Extension to OpenMP. In *6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 5–14, Heraklion, Greece, January 2011. ACM.

[PC13] Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs. *ACM Transactions on Architecture and Code Optimization*, 9(4):53:1–53:25, January 2013.

[Pop12] Antoniu Pop. *Leveraging Streaming for Deterministic Parallelization: an Integrated Language, Compiler and Runtime Approach*. PhD thesis, École Nationale Supérieure des Mines de Paris, Paris, French, June 2012.

[PPL16] PPL. The Pervasive Parallelism Laboratory. https://ppl.stanford.edu/projects, Febuary 2016.

[Pro14] REPARA Project. D3.3: Static Partitioning Tool. Technical report, University of Pisa, Pisa, Italy, December 2014.

[Pro15] REPARA Project. D4.3: Source Code Transformations for Coarse Grained Parallelism. Technical report, University of Pisa, Pisa, Italy, February 2015.

[QSYS04] Dan Quinlan, Markus Schordan, Qing Yi, and Andreas Saebjornsen. Classification and Utilization of Abstractions for Optimization. In *Leveraging Applications of Formal Methods: First International Symposium*, ISoLA 2004, pages 57–73, Paphos, Cyprus, Greece, October 2004. Springer Berlin Heidelberg.

[Qui03] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, USA, 2003.

[RCJ11] Eric C. Reed, Nicholas Chen, and Ralph E. Johnson. Expressing Pipeline Parallelism Using TBB Constructs: A Case Study on What Works and What Doesn'T. In *Compilation of the Co-located Workshops of SPLASH*, SPLASH '11 Workshops, pages 133–138, Portland, Oregon, USA, October 2011. ACM.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly, Sebastopol, CA, USA, 2007.

[REP16] REPARA. Reengineering and Enabling Performance and poweR of Applications. http://repara-project.eu/?page_id=244, Febuary 2016.

[RJ15] James Reinders and Jim Jeffers. *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*. ElSci, United States, 2015.

[RO10] Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, Eindhoven, The Netherlands, October 2010. ACM.

[ROS16] ROSE. ROSE@LLNL: Making Compiler Technology Accessible. http://rosecompiler.org/, Febuary 2016.

[RR10] Thomas Rauber and Gudula Rünger. *Parallel Programming for Multicore and Cluster Systems*. Springer, New York, USA, 2010.

[SBL+14] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems*, 13(4):25, July 2014.

[Sch14] Boris Schaling. *The Boost C++ Libraries*. XML Press, 2014.

[SGA+13] Robert Soulé, Michael I. Gordon, Saman Amarasinghe, Robert Grimm, and Martin Hirzel. Dynamic Expressivity with Static Optimization for Streaming Languages. In *7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 159–170, Arlington, Texas, USA, July 2013. ACM.

[SLB+11] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 609–616, Bellevue, Washington, USA, June 2011. ACM.

[SLJ+14] Mehrzad Samadi, Janghaeng Lee, Anoushe D. Jamshidi, Scott Mahlke, and Amir Hormati. Scaling Performance via Self-Tuning Approximation for Graphics Engines. *ACM Transaction Computer Systems*, 32(3):7:1–7:29, September 2014.

[SQ03] Markus Schordan and Dan Quinlan. A Source-To-Source Architecture for User-Defined Optimizations. In *Modular Programming Languages: Joint Modular Languages Conference*, JMLC 2003, pages 214–223, Klagenfurt, Austria, August 2003. Springer Berlin Heidelberg.

[SS96] Duane Szafron and Jonathan Schaeffer. An Experiment to Measure the Usability of Parallel Programming Systems. *Concurrency: Practice and Experience*, 8(2):147–166, March 1996.

[Str14] Bjarne Stroustrup. *Programming: Principles and Practice Using C++*. Addison-Wesley Professional, San Francisco, USA, 2014.

[Str16] StreamIt. Website. http://groups.csail.mit.edu/cag/streamit, Febuary 2016.

[Suj14] Arvind Krishna Sujeeth. *Productivity and Performance with Embedded Domain-Specific Language*. PhD thesis, Stanford University, Stanford, USA, May 2014.

[SUPT14] Alessandro Secco, Irfan Uddin, Guilherme Peretti Pezzi, and Massimo Torquati. Message Passing on InfiniBand RDMA for Parallel Run-Time Supports. In *22th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 130–137, Torino, Italy, February 2014. IEEE.

[TA10] William Thies and Saman Amarasinghe. An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design. In *International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 365–376, Austria, September 2010. ACM.

[TBB16] Intel TBB. Intel® Threading Building Blocks. http://threadingbuildingblocks.org, Febuary 2016.

[TDM+14] Fabio Tordini, Maurizio Drocco, Ivan Merelli, Luciano Milanesi, Pietro Liò, and Marco Aldinucci. NuChart-II: A Graph-Based Approach for the Analysis and Interpretation of Hi-C Data. In *11th International meeting on Computational Intelligence methods for Bioinformatics and Biostatistics (CIBB)*, volume 1 of *Lecture Notes in Bioinformatics*, pages 1–13, Cambridge, UK, June 2014. Springer.

[Thi09] William Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 2009.

[TKA02a] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *11th International Conference on Compiler Construction*, CC '02, pages 179–196, Grenoble, France, April 2002. Springer-Verlag.

[TKA02b] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *11th International Conference on Compiler Construction*, CC '02, pages 179–196, Grenoble, France, April 2002. Springer-Verlag.

[TYK11] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular MapReduce for Shared-memory Systems. In *Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 9–16, San Jose, California, USA, June 2011. ACM.

[VBD+13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering Designing, Implementing and Using Domain-Specific Languages*. Ebook, Germany, 2013.

[vH06] William von Hagen. *The Definitive Guide to GCC*. Apress, Berkeley, CA, 2006.

[Won12] Eric Wong. Optimizations in Stream Programming for Multimedia Applications . Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 2012.

[Wri10] Steve Wright. *Digital Compositing for Film and Video*. Focal Press and Elsevier, Oxford, UK, 2010.

[X1016] X10. Performance and Productivity at Scale. http://x10-lang.org/, Febuary 2016.

[ZLRA08] David Zhang, Qiuyuan J. Li, Rodric Rabbah, and Saman Amarasinghe. A Lightweight Streaming Layer for Multicore Execution. *SIGARCH Computer Architecture News*, 36(2):18–27, May 2008.

A

Appendix

This chapter presents all appendixes to complement the discussions of the thesis.

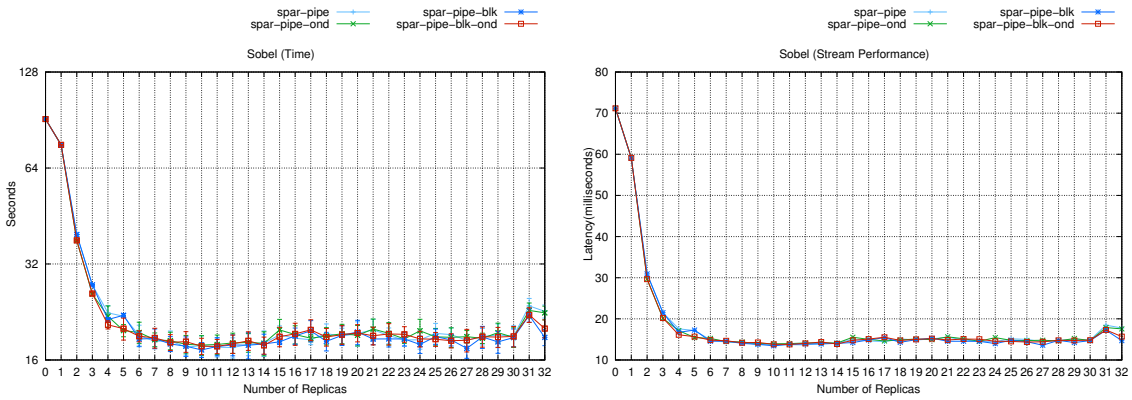
Contents

A.1 Complementary Results on Multi-Core

During this section, complementary results of the applications when running in a multi-core environment are plotted.

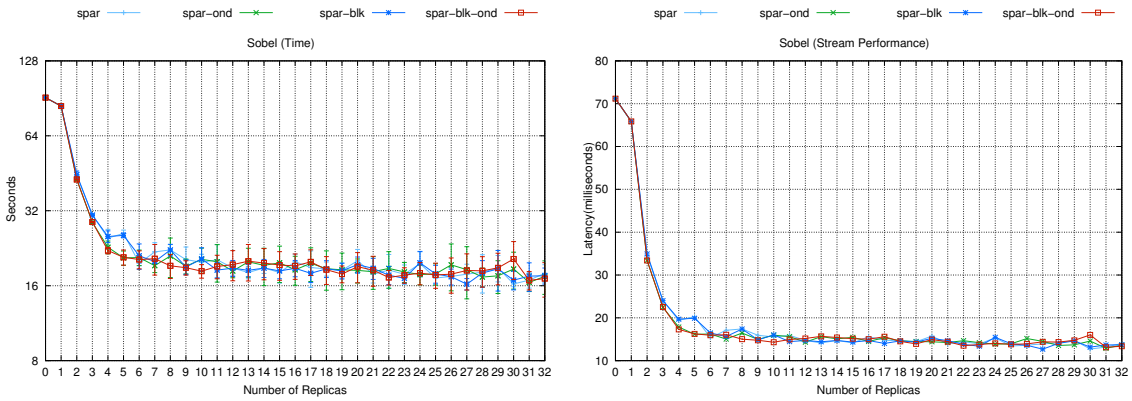
A.1.1 Filter Sobel SPar Performance

This section is just complementing the results concerning the Filter Sobel application, instead using unbounded workload for comparing the SPar optimization flags and application versions.



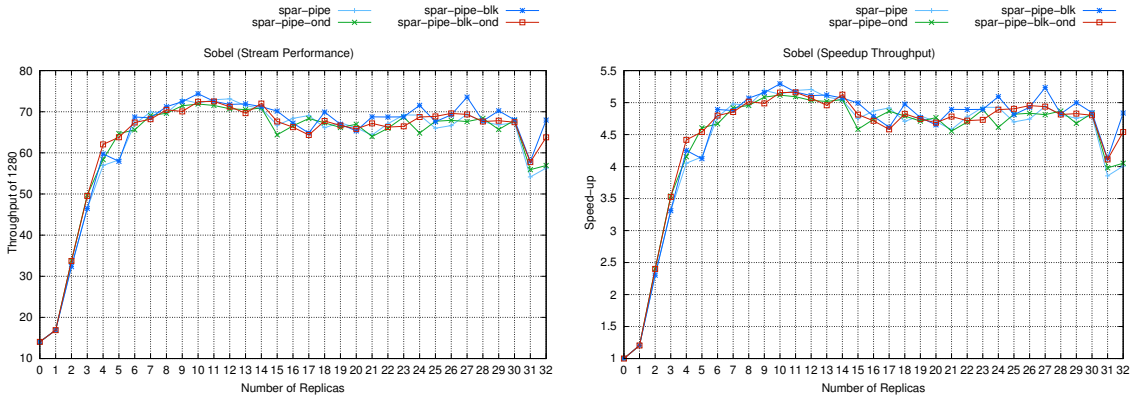
(a) Filter Sobel execution times. (b) Filter Sobel latency.

Figure A.1: Time performance using unbalanced workload (pipe-like)



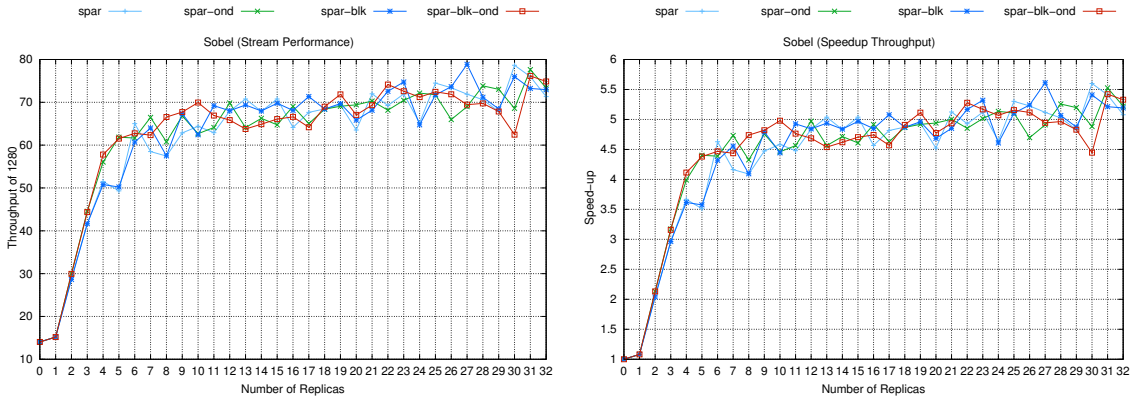
(a) Filter Sobel execution times. (b) Filter Sobel latency.

Figure A.2: Time performance using unbalanced workload (farm-like)



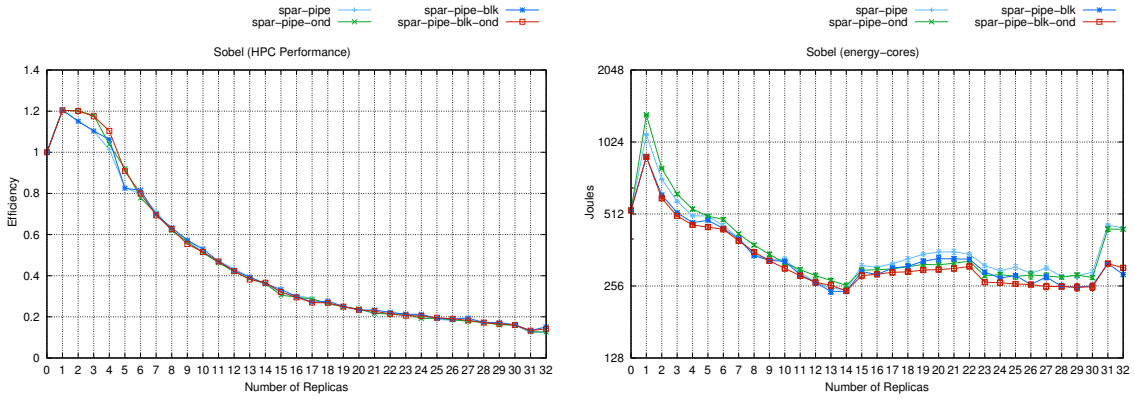
(a) Filter Sobel throughput. (b) Filter Sobel throughput speed-up.

Figure A.3: Stream performance using unbalanced workload (pipe-like)



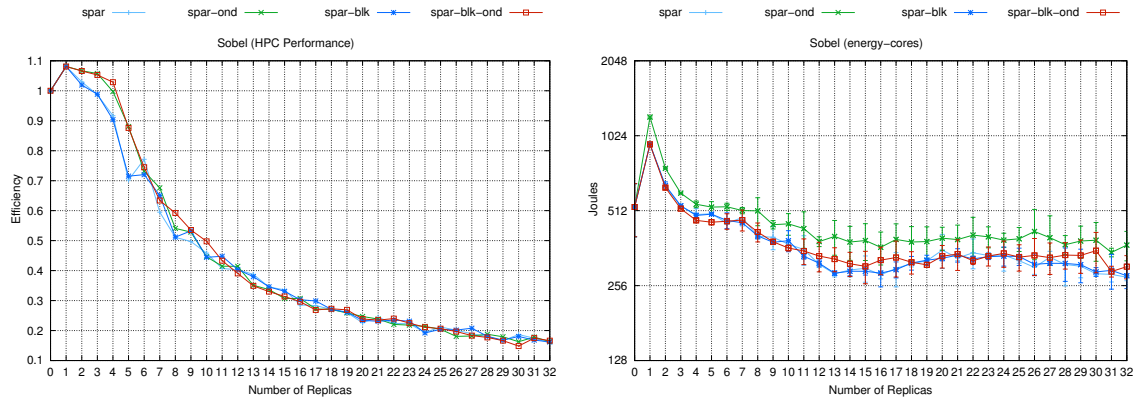
(a) Filter Sobel throughput. (b) Filter Sobel throughput speed-up.

Figure A.4: Stream performance using unbalanced workload (farm-like)



(a) Filter Sobel HPC efficiency. (b) Filter Sobel CPU cores energy consumption.

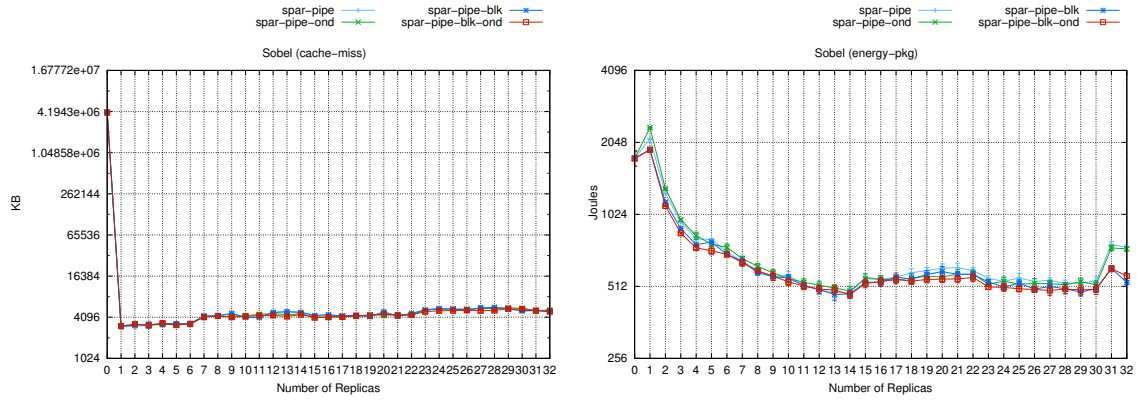
Figure A.5: HPC performance using unbalanced workload (pipe-like)



(a) Filter Sobel HPC efficiency.

(b) Filter Sobel CPU cores energy consumption.

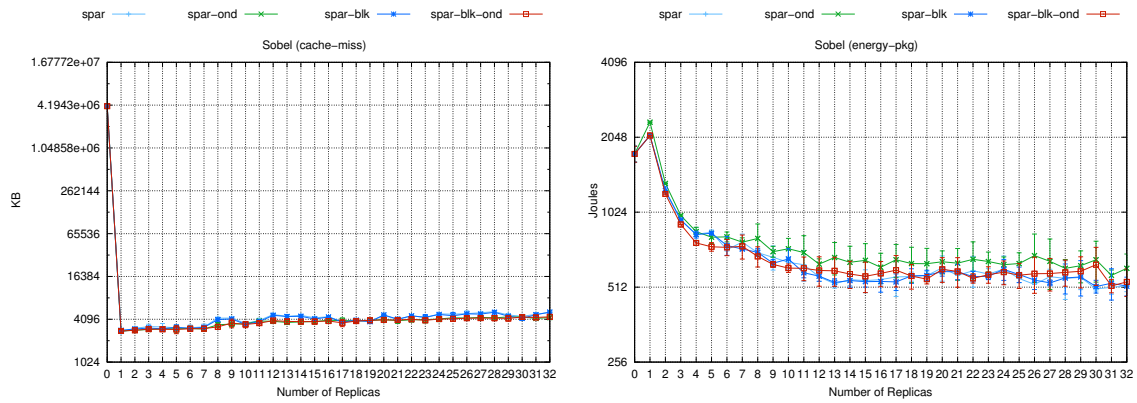
Figure A.6: HPC performance using unbalanced workload (farm-like)



(a) Filter Sobel cache misses.

(b) Filter Sobel socket energy consumption (cache and CPUs).

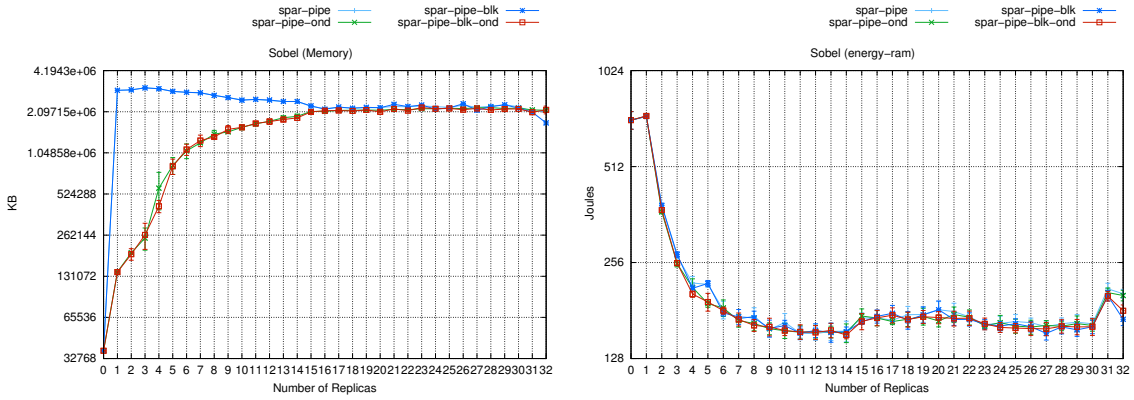
Figure A.7: CPU Socket performance using unbalanced workload (pipe-like)



(a) Filter Sobel cache misses.

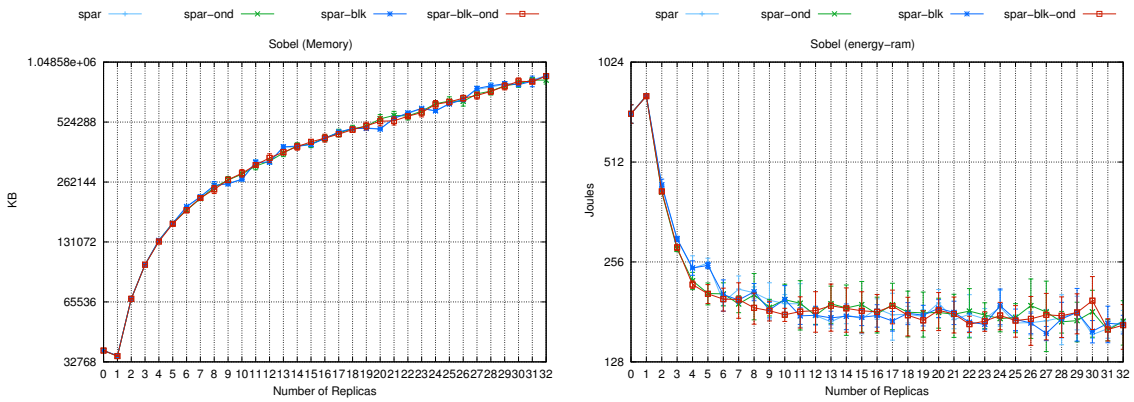
(b) Filter Sobel socket energy consumption (cache and CPUs).

Figure A.8: CPU Socket performance using unbalanced workload (farm-like)



(a) Filter Sobel memory usage. (b) Filter Sobel memory energy consumption.

Figure A.9: Memory performance using unbalanced workload (pipe-like)

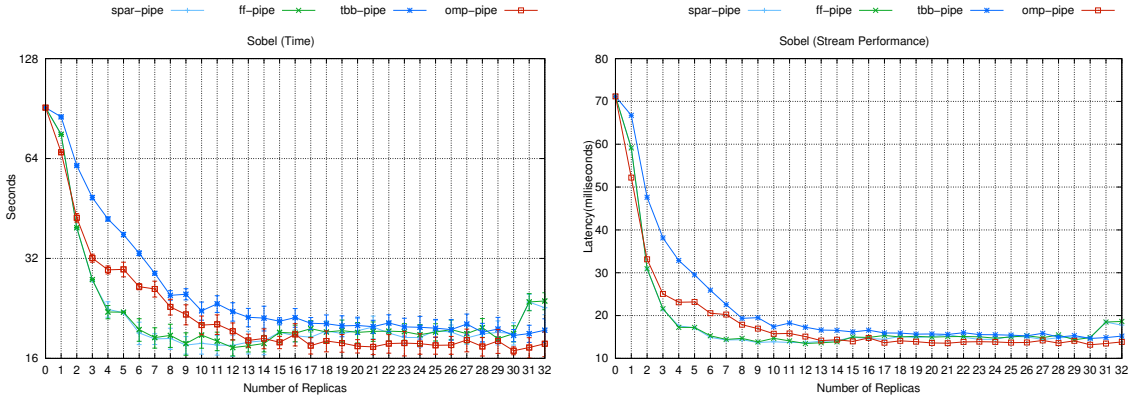


(a) Filter Sobel memory usage. (b) Filter Sobel memory energy consumption.

Figure A.10: Memory performance using unbalanced workload (farm-like)

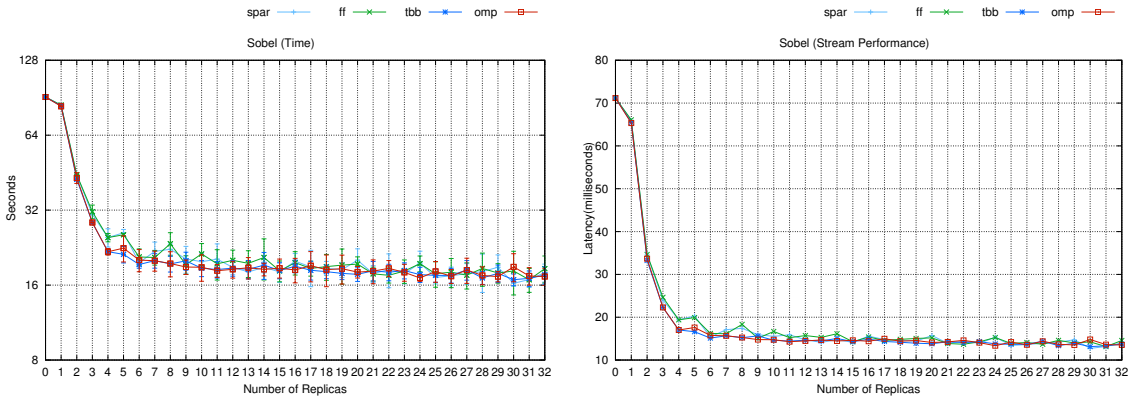
A.1.2 Filter Sobel Performance Comparison

This section is presenting more results concerning the Filter Sobel application using unbounded workload for the performance comparison.



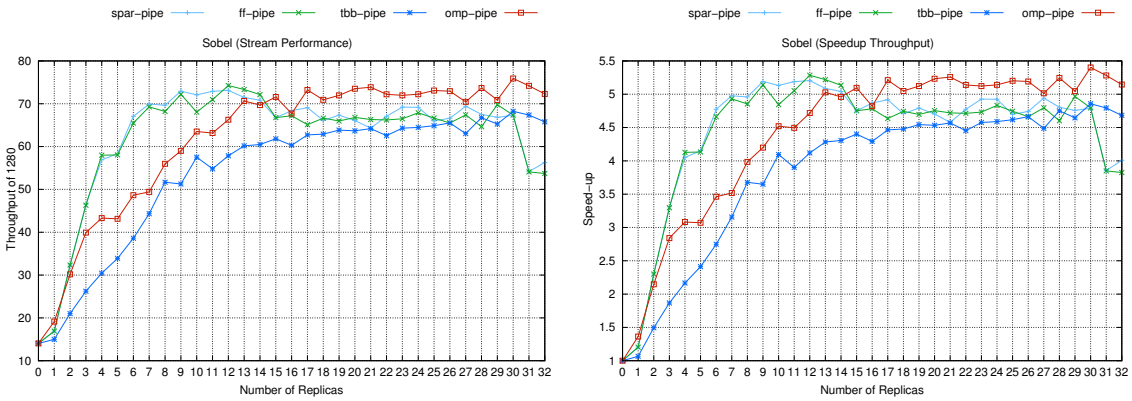
(a) Filter Sobel execution times. (b) Filter Sobel latency.

Figure A.11: Time performance comparison using unbalanced workload (Listing 7.1)



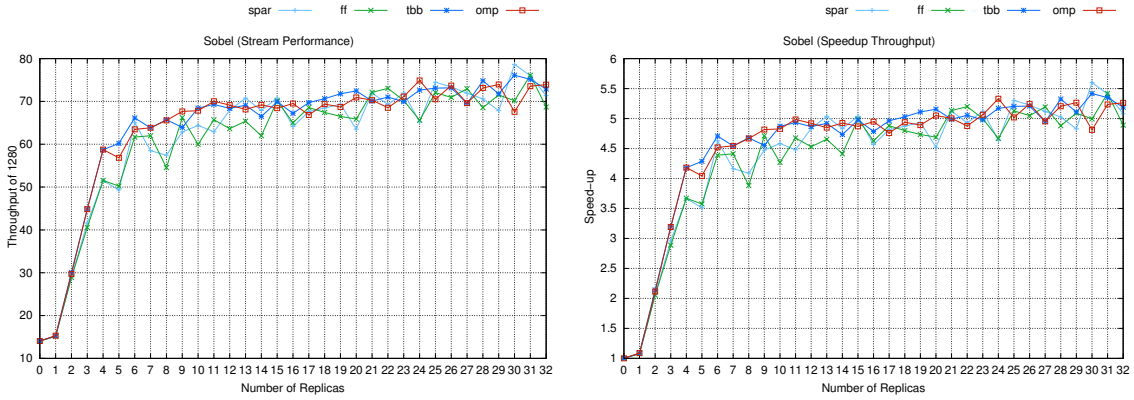
(a) Filter Sobel execution times. (b) Filter Sobel latency.

Figure A.12: Time performance comparison using unbalanced workload (Listing 7.2)



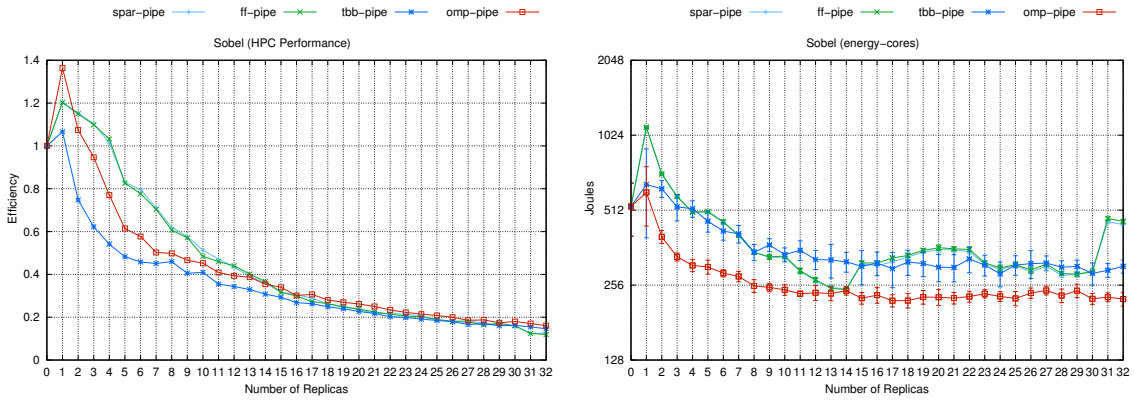
(a) Filter Sobel throughput. (b) Filter Sobel throughput speed-up.

Figure A.13: Stream performance comparison using unbalanced workload (Listing 7.1)



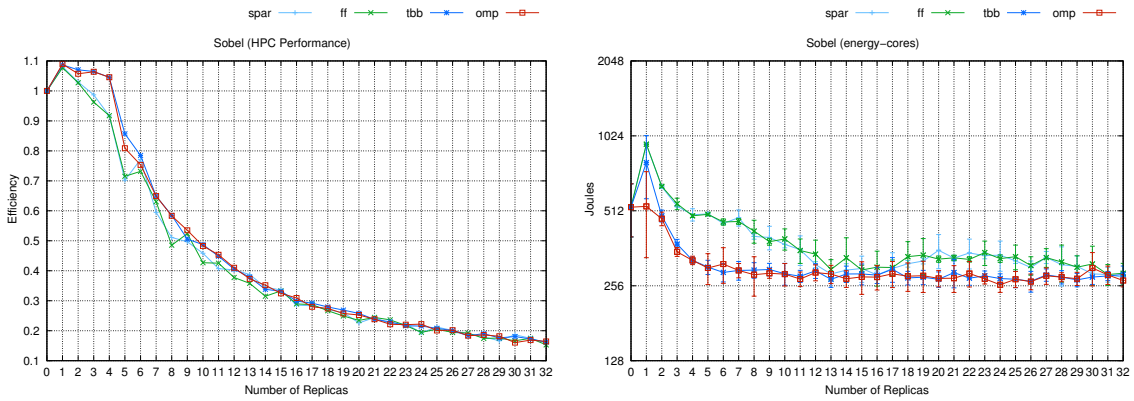
(a) Filter Sobel throughput. (b) Filter Sobel throughput speed-up.

Figure A.14: Stream performance comparison using unbalanced workload (Listing 7.2)



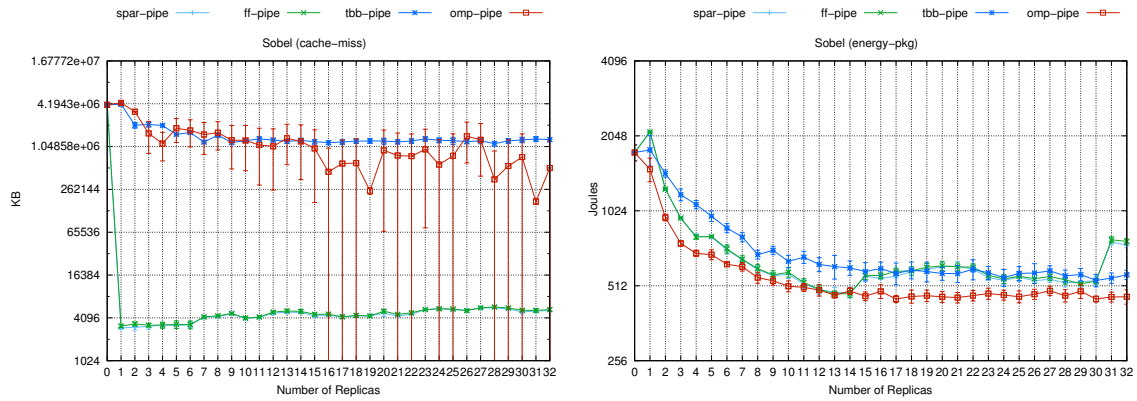
(a) Filter Sobel HPC efficiency. (b) Filter Sobel CPU cores energy consumption.

Figure A.15: HPC performance comparison using unbalanced workload (Listing 7.1)



(a) Filter Sobel HPC efficiency. (b) Filter Sobel CPU cores energy consumption.

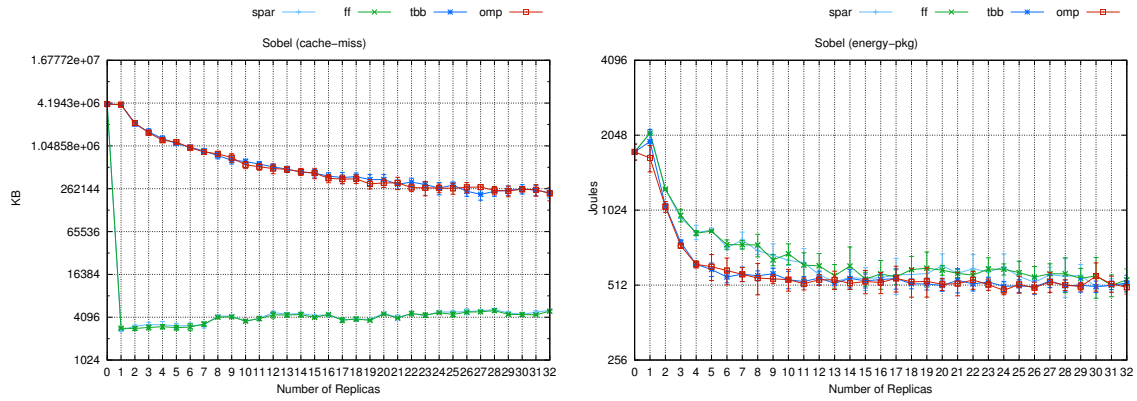
Figure A.16: HPC performance comparison using unbalanced workload (Listing 7.2)



(a) Filter Sobel cache misses.

(b) Filter Sobel socket energy consumption (cache and CPUs).

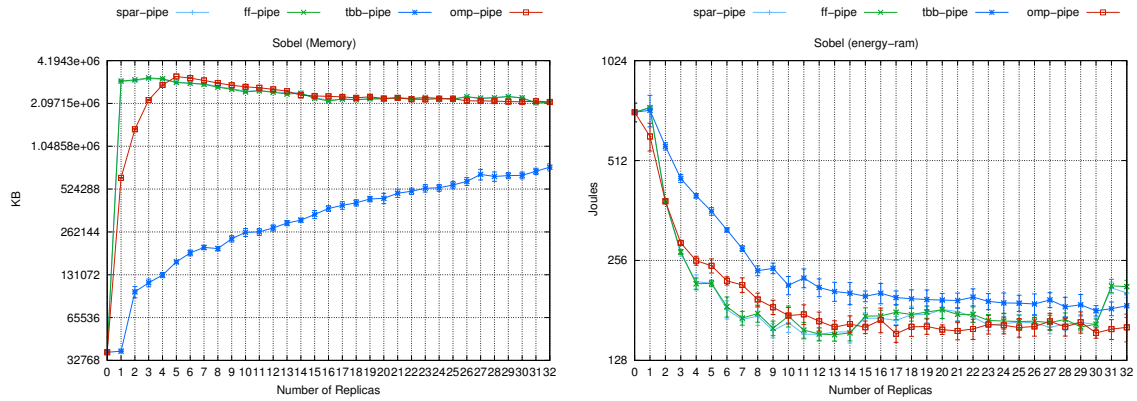
Figure A.17: CPU Socket performance comparison using unbalanced workload (Listing 7.1)



(a) Filter Sobel cache misses.

(b) Filter Sobel socket energy consumption (cache and CPUs).

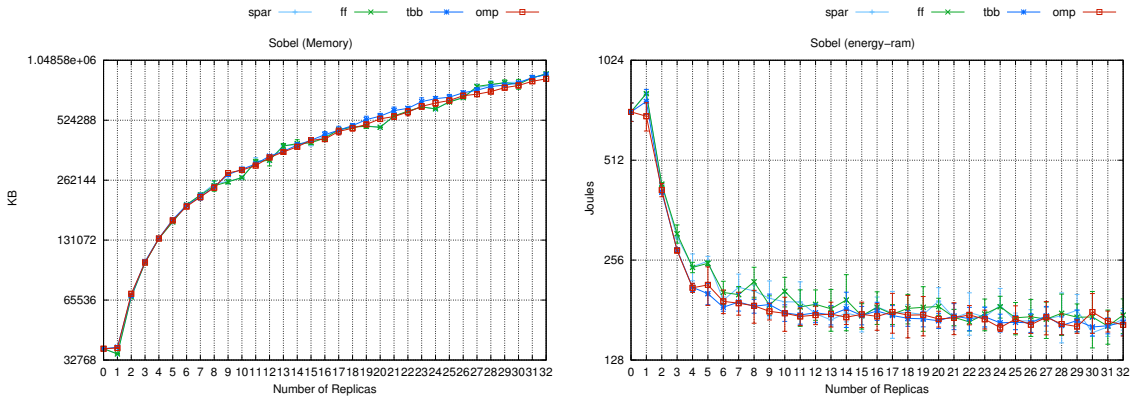
Figure A.18: CPU Socket performance comparison using unbalanced workload (Listing 7.2)



(a) Filter Sobel memory usage.

(b) Filter Sobel memory energy consumption.

Figure A.19: Memory performance comparison using unbalanced workload (Listing 7.1)



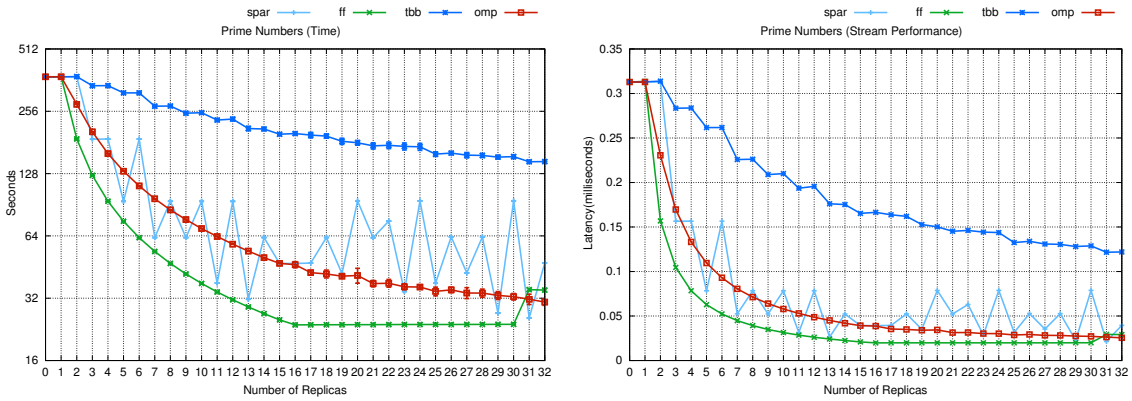
(a) Filter Sobel memory usage.

(b) Filter Sobel memory energy consumption.

Figure A.20: Memory performance comparison using unbalanced workload (Listing 7.2)

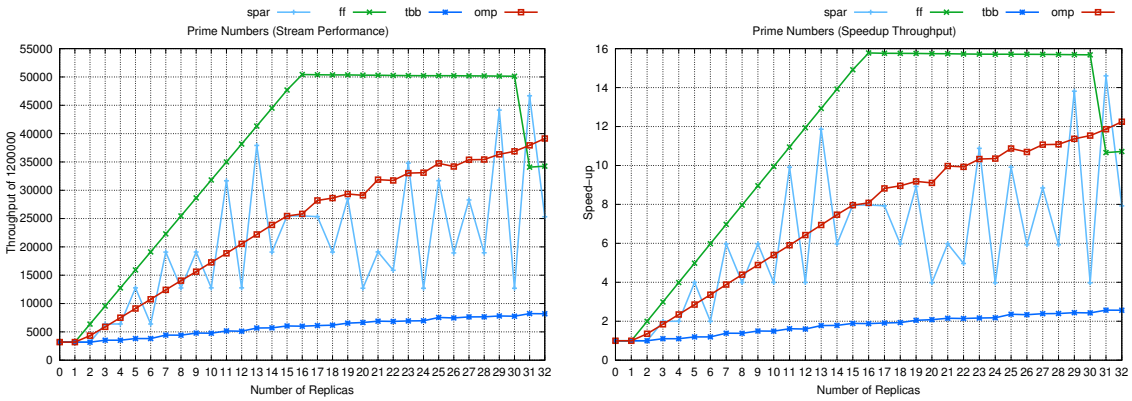
A.1.3 Prime Numbers Performance Comparison

In this section, complementary results are presented for the Prime Numbers application running in multi-core.



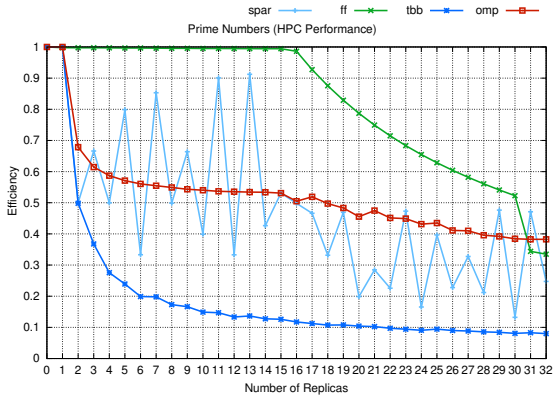
(a) Prime Numbers execution times. (b) Prime Numbers latency.

Figure A.21: Time performance comparison (Prime Numbers Default)

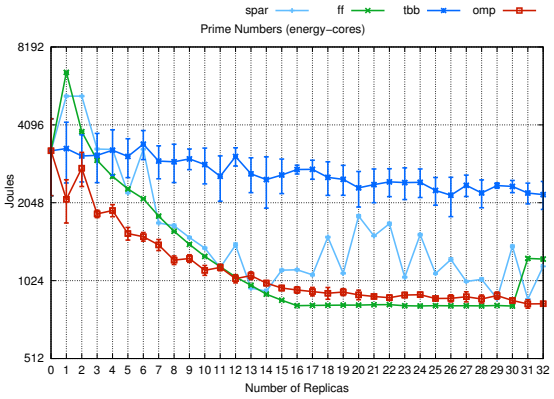


(a) Prime Numbers throughput. (b) Prime Numbers throughput speed-up.

Figure A.22: Stream performance comparison (Prime Numbers Default)

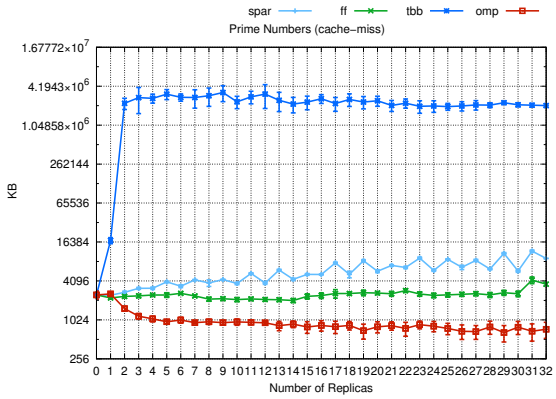


(a) Prime Numbers HPC efficiency.

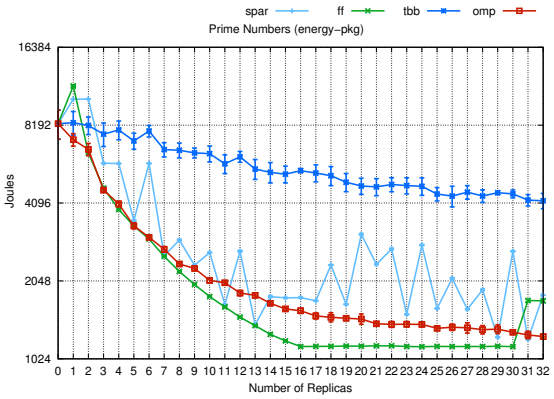


(b) Prime Numbers CPU cores energy consumption.

Figure A.23: HPC performance comparison (Prime Numbers Default)

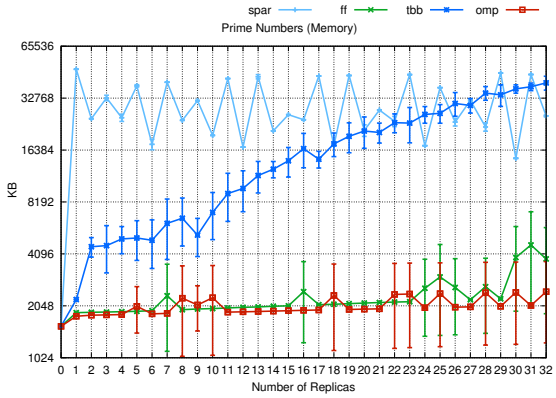


(a) Prime Numbers cache misses.

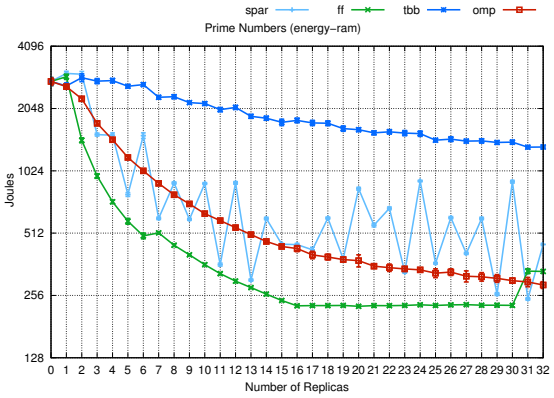


(b) Prime Numbers socket energy consumption (cache and CPUs).

Figure A.24: CPU Socket performance comparison (Prime Numbers Default)



(a) Prime Numbers memory usage.

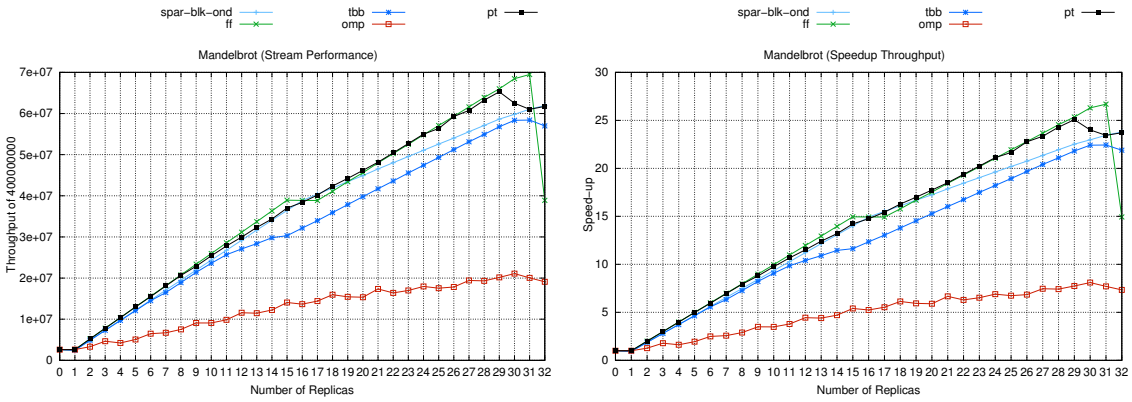


(b) Prime Numbers memory energy consumption.

Figure A.25: Memory performance comparison (Prime Numbers Default)

A.1.4 Mandelbrot Set Performance Comparison

Some complementary results about the Mandelbrot set application are demonstrated in this section that run in a multi-core machine.

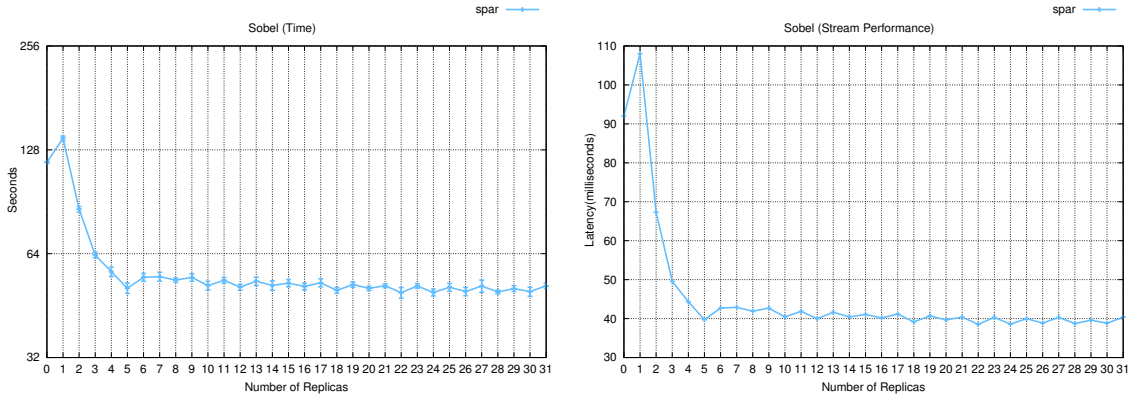


(a) Mandelbrot throughput. (b) Mandelbrot throughput speed-up.

Figure A.26: Stream performance comparison (Mandelbrot)

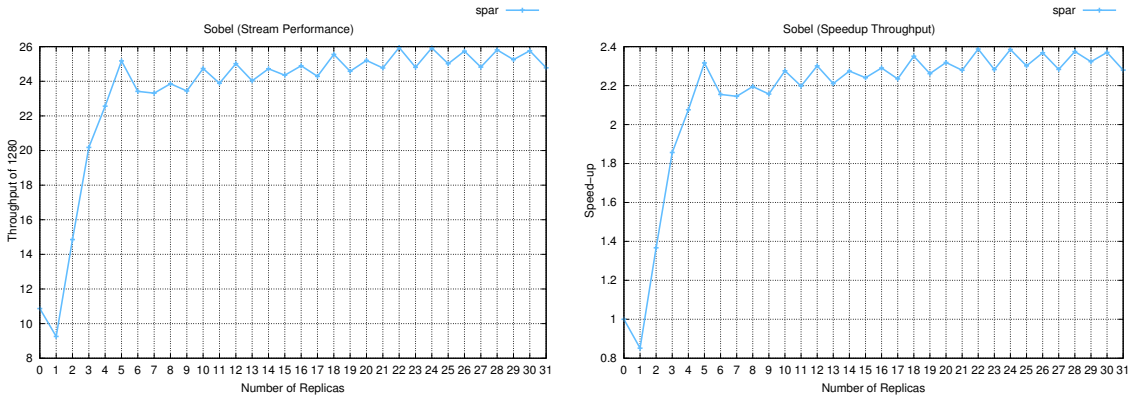
A.2 Complementary Results on Cluster

This section will present all complementary results for evaluating the performance on cluster environment.



(a) Filter Sobel execution times. (b) Filter Sobel latency.

Figure A.27: Time performance using unbalanced workload (Filter Sobel)



(a) Filter Sobel throughput. (b) Filter Sobel throughput speed-up.

Figure A.28: Stream performance using unbalanced workload (Filter Sobel)

A.3 Sources for Coding Productivity

This section will present all applications pseudo codes used to evaluate and compare the coding productivity.

A.3.1 Filter Sobel

```
//global declaration
int main(int argc, char *argv[]){
#pragma omp parallel num_threads(workers)
{
#pragma omp single
{
  //open directory ...
  DIR *dptr = opendir(...);
  struct dirent *dfptr;
  while((dfptr = readdir(dptr)) != NULL){
    //preprocessing
    if (file_extension == "bmp"){
      tot_img++;
      #pragma omp task
      {
        //Reads the image ...
        image = read(name,height,width);
        //Applies the Sobel
        new_image=sobel(image,height,width);
        //Writes the image ...
        write(newname,new_image,height,width);
      }
    }else{
      tot_not++;
    }
  }
  //end processing
}
}
  return 0;
}
```

Listing A.1: Filter Sobel using OpenMP.

```
//global declaration
int main(int argc, char *argv[]){
#pragma omp parallel num_threads(workers)
{
#pragma omp single
{
  //open directory ...
  DIR *dptr = opendir(...);
  struct dirent *dfptr;
  while((dfptr = readdir(dptr)) != NULL){
    //preprocessing
    if (file_extension == "bmp"){
      tot_img++;
      //Reads the image ...
      image = read(name,height,width);
      #pragma omp task depend(in:image[tot_img]) depend(out:new_image[tot_img])
      {
        //Applies the Sobel
        new_image=sobel(image,height,width);
      }
      #pragma omp task depend(in:new_image[tot_img])
      {
        //Writes the image ...
        write(newname,new_image,height,width);
      }
    }else{
      tot_not++;
    }
  }
}
}
  //end processing
  return 0;
}
```

Listing A.2: Filter Sobel using OpenMP (pipeline-like).

```
#include <ff/farm.hpp>
using namespace ff;
struct FF_Stream {
  FF_Stream(char *name, char *newname):
   name(name),newname(newname) {};
```

```
6
7       char *name;
8       char *newname;
9   };
10  struct Emitter_ff: ff_node_t<FF_Stream> {
11    const std::string filepath;
12    unsigned int tot_img;
13    unsigned int tot_not;
14    Emitter_ff(const std::string filepath, unsigned int tot_img, unsigned int tot_not):
15        filepath(filepath),tot_img(tot_img),tot_not(tot_not){}
16
17      FF_Stream *svc(FF_Stream *){
18      //open directory ...
19      DIR *dptr = opendir(...);
20      struct dirent *dfptr;
21      while((dfptr = readdir(dptr)) != NULL){
22        //preprocessing
23        if (file_extension == "bmp"){
24          tot_img++;
25          FF_Stream *stream = new FF_Stream(name,newname, ...);
26          ff_send_out(stream);
27        }else{
28          tot_not++;
29        }
30      }
31        return EOS;
32      }
33   };
34  void StageReplicate_Spar(FF_Stream *in){
35    //Reads the image ...
36    image = read(in->name,height,width);
37    //Applies the Sobel
38    new_image=sobel(image,height,width);
39    //Writes the image ...
40    write(in->newname,new_image,height,width);
41  }
42  FF_Stream *StageReplicate_ff (FF_Stream *in, ff_node*const){
43    StageReplicate_Spar(in);
44    delete in;
45    return (FF_Stream*)GO_ON; //end of the stream
46  }
47  //global declaration
48  int main(int argc, char *argv[]){
49    Emitter_ff Emitter(interating,filepath,tot_img,tot_not);
50    ff_Farm<FF_Stream> StageReplicate(StageReplicate_ff,workers);
51    StageReplicate.add_emitter(Emitter);
52    StageReplicate.set_scheduling_ondemand();
53    StageReplicate.remove_collector();
54    if (StageReplicate.run_and_wait_end()<0) { // executes the farm
55      error("Running farm\n");
56      return -1;
57    }
58    tot_img=Emitter.tot_img;
59    tot_not=Emitter.tot_not;
60  }
```

Listing A.3: Filter Sobel using FastFlow (farm-like).

```
1   #include <ff/pipeline.hpp>
2   using namespace ff;
3   struct FF_Stream {
4     FF_Stream(char *name, char *newname,...):
5      name(name),newname(newname), ... {};
6       //list of variable
7       char *name;
8       char *newname;
9       unsigned char* image;
10      unsigned char* new_image;
11    ...
12  };
13  struct Emitter_ff: ff_node_t<FF_Stream> {
14    const std::string filepath;
15    unsigned int tot_img;
16    unsigned int tot_not;
17    Emitter_ff(const std::string filepath, unsigned int tot_img, unsigned int tot_not):
18        filepath(filepath),tot_img(tot_img),tot_not(tot_not){}
19
20      FF_Stream *svc(FF_Stream *){
21        //open directory ...
22      DIR *dptr = opendir(...);
23      struct dirent *dfptr;
24      while((dfptr = readdir(dptr)) != NULL){
25        //preprocessing
26        if (file_extension == "bmp"){
27          tot_img++;
28          //Reads the image ...
```

```
29          image = read(name,height,width);
30          FF_Stream *stream = new FF_Stream(name,newname, ...);
31          ff_send_out(stream);
32        }else{
33          tot_not++;
34        }
35      }
36      return EOS;
37      }
38 };
39 void StageReplicate_Spar(FF_Stream *in){
40   //Applies the Sobel
41   in->new_image=sobel(in->image,in->height,in->width);
42 }
43 void Stage_Spar(FF_Stream *in){
44   //Writes the image ...
45   write(in->newname,in->new_image,in->height,in->width);
46 }
47 FF_Stream *StageReplicate_ff (FF_Stream *in, ff_node*const){
48   StageReplicate_Spar(in);
49   return in;
50 }
51 FF_Stream *Stage_ff (FF_Stream *in, ff_node*const){
52   Stage_Spar(in);
53   delete in;
54   return (FF_Stream*)GO_ON; //end of the stream
55 }
56 //global declaration
57 int main(int argc, char *argv[]){
58   Emitter_ff Emitter(filepath,tot_img,tot_not);
59   ff_Farm<FF_Stream> StageReplicate(StageReplicate_ff,workers);
60   StageReplicate.add_emitter(Emitter);
61   StageReplicate.set_scheduling_ondemand();
62   StageReplicate.remove_collector();
63   struct MultiInput_Stage: ff_minode_t<FF_Stream> {
64     FF_Stream *svc(FF_Stream *stream) {
65       return Stage_ff(stream, this);
66     }
67   };
68   MultiInput_Stage S;
69   ff_Pipe<> pipe(StageReplicate,S);
70   if (pipe.run_and_wait_end()<0) { // executes the pipeline
71     error("Running pipeline\n");
72     return -1;
73   }
74   tot_img=Emitter.tot_img;
75   tot_not=Emitter.tot_not;
76 }
```

Listing A.4: Filter Sobel using FastFlow (pipe-like).

```
1  #include <tbb/pipeline.h>
2  #include <tbb/task_scheduler_init.h>
3  struct TBB_Stream {
4    TBB_Stream(char *name, char *newname):
5     name(name),newname(newname) {};
6
7      char *name;
8      char *newname;
9  };
10 class Emitter_tbb: public tbb::filter {
11 public:
12     Emitter_tbb(const std::string filepath, unsigned int tot_img, unsigned int tot_not);
13     void *operator()(void *);
14   const std::string filepath;
15   unsigned int tot_img;
16   unsigned int tot_not;
17 };
18 Emitter_tbb::Emitter_tbb(const std::string filepath, unsigned int tot_img, unsigned int tot_not) :
19     tbb::filter(serial_in_order),filepath(filepath),tot_img(tot_img),tot_not(tot_not)
20 {}
21 void * Emitter_tbb::operator()(void *) {
22   struct dirent *dfptr;
23   //open directory ...
24   DIR *dptr = opendir(...);
25   struct dirent *dfptr;
26   while((dfptr = readdir(dptr)) != NULL){
27     //preprocessing
28     if (file_extension == "bmp"){
29       tot_img++;
30       TBB_Stream *stream = new TBB_Stream(name,newname);
31       return stream;
32     }else{
33       tot_not++;
34     }
35   }
```

```
36    return NULL;
37  }
38  class StageReplicate_tbb: public tbb::filter {
39  public:
40      StageReplicate_tbb();
41      void * operator()( void *input );
42  };
43  StageReplicate_tbb::StageReplicate_tbb(): tbb::filter(parallel){}
44  void *StageReplicate_tbb::operator()(void *input){
45    TBB_Stream *in = static_cast<TBB_Stream*>(input);
46    //Reads the image ...
47    image = read(in->name,height,width);
48    //Applies the Sobel
49    new_image=sobel(image,height,width);
50    //Writes the image ...
51    write(in->newname,new_image,height,width);
52    delete in;
53    return NULL;
54  }
55  //global declaration
56  int main(int argc, char *argv[]){
57    tbb::task_scheduler_init init(workers);
58    tbb::pipeline pipeline;
59    Emitter_tbb Emitter(filepath,tot_img,tot_not);
60    pipeline.add_filter(Emitter);
61    StageReplicate_tbb StageReplicate;
62    pipeline.add_filter(StageReplicate);
63    pipeline.run(workers);
64    tot_img=Emitter.tot_img;
65    tot_not=Emitter.tot_not;
66  }
```

Listing A.5: Filter Sobel using TBB (farm-like).

```
1   unsigned int tot_img=0, workers=1, ninter=1, tot_not=0;
2   #include <tbb/pipeline.h>
3   #include <tbb/task_scheduler_init.h>
4   struct TBB_Stream {
5     TBB_Stream(char *name, char *newname, ...):
6      name(name),newname(newname), ... {};
7      //list of variable
8       char *name;
9       char *newname;
10      unsigned char* image;
11      unsigned char* filtered_image;
12    ...
13  };
14  class Emitter_tbb: public tbb::filter {
15  public:
16      Emitter_tbb(const std::string filepath, unsigned int tot_img, unsigned int tot_not);
17      void *operator()(void *);
18    const std::string filepath;
19    unsigned int tot_img;
20    unsigned int tot_not;
21  };
22  Emitter_tbb::Emitter_tbb(const std::string filepath, unsigned int tot_img, unsigned int tot_not) :
23      tbb::filter(serial_in_order),filepath(filepath),tot_img(tot_img),tot_not(tot_not)
24  {}
25  void * Emitter_tbb::operator()(void *) {
26    //open directory ...
27    DIR *dptr = opendir(...);
28    struct dirent *dfptr;
29    while((dfptr = readdir(dptr)) != NULL){
30      //preprocessing
31      if (file_extension == "bmp"){
32        tot_img++;
33        //Reads the image ...
34        image = read(name,height,width);
35        TBB_Stream *stream = new TBB_Stream(name,newname, ...);
36        return stream;
37      }else{
38        tot_not++;
39      }
40    }
41    return NULL;
42  }
43  class StageReplicate_tbb: public tbb::filter {
44  public:
45      StageReplicate_tbb();
46      void * operator()( void *input );
47  };
48  StageReplicate_tbb::StageReplicate_tbb(): tbb::filter(parallel){}
49  void *StageReplicate_tbb::operator()(void *input){
50    TBB_Stream *in = static_cast<TBB_Stream*>(input);
51    //Applies the Sobel
52    in->new_image=sobel(in->image,in->height,in->width);
```

```
53    return in;
54  }
55  class Stage_tbb: public tbb::filter {
56  public:
57      Stage_tbb();
58      void * operator()( void *input );
59  };
60  Stage_tbb::Stage_tbb(): tbb::filter(serial_in_order){}
61  void *Stage_tbb::operator()(void *input){
62    TBB_Stream *in = static_cast<TBB_Stream*>(input);
63    //Writes the image ...
64    write(in->newname,in->new_image,in->height,in->width);
65    delete in;
66    return NULL;
67  }
68  //global declaration
69  int main(int argc, char *argv[]){
70    tbb::task_scheduler_init init((workers+2));
71    tbb::pipeline pipeline;
72    Emitter_tbb Emitter(interating,filepath,tot_img,tot_not);
73    pipeline.add_filter(Emitter);
74    StageReplicate_tbb StageReplicate;
75    pipeline.add_filter(StageReplicate);
76    Stage_tbb S;
77    pipeline.add_filter(S);
78    pipeline.run(workers);
79    tot_img=Emitter.tot_img;
80    tot_not=Emitter.tot_not;
81  }
```

Listing A.6: Filter Sobel using TBB (pipe-like).

A.3.2 Video OpenCV

```
1   #include <ff/farm.hpp>
2   #include <ff/pipeline.hpp>
3   using namespace ff;
4   struct Stream{
5       Stream(Mat src, Mat res, int channel, Size S):src(src),res(res),channel(channel),S(S){}
6       Mat src;
7       Mat res;
8       int channel;
9       Size S;
10  };
11  struct Source : ff_node_t<Stream> {
12      int channel;
13      Size S;
14      Mat src;
15      Mat res;
16      Source(int channel,Size S):channel(channel),S(S) {}
17      Stream* svc(Stream *) {
18          for(;;){
19              total_frames++;
20              inputVideo >> src;
21              if (src.empty()) break;
22              Stream *s = new Stream(src,res,channel,S);
23              ff_send_out(s);
24          }
25          return EOS;
26      }
27  };
28  Stream *StageReplicate(Stream *t, ff_node*const){
29      vector<Mat> spl;
30      split(t->src, spl);                 // process - extract only the correct channel
31      for (int i =0; i < 3; ++i){
32          if (i != t->channel){
33              spl[i] = Mat::zeros(t->S, spl[0].type());
34          }
35      }
36      merge(spl, t->res);
37      cv::GaussianBlur(t->res, t->res, cv::Size(0, 0), 3);
38      cv::addWeighted(t->res, 1.5, t->res, -0.5, 0, t->res);
39      Sobel(t->res,t->res,-1,1,0,3);
40      return t;
41  };
42  struct Drain: ff_node_t<Stream> {
43      Stream *svc (Stream * t) {
```

```
        outputVideo << t->res;
        delete t;
        return GO_ON;
    }
};
ff_OFarm<Stream> F(StageReplicate,workers);
Source E(channel,S);
F.setEmitterF(E);
Drain  drain;
F.setCollectorF(drain);
if (F.run_and_wait_end()<0) {
    error("running pipe");
    return -1;
}
```

Listing A.7: Video OpenCV using FastFlow.

```
#include <tbb/pipeline.h>
#include <tbb/task_scheduler_init.h>
using namespace tbb;
struct Stream{
    Stream(Mat src, Mat res, int channel, Size S):src(src),res(res),channel(channel),S(S){}
    Mat src;
    Mat res;
    int channel;
    Size S;
};
class Source: public tbb::filter {
public:
    Source(int channel,Size S);
    void *operator()(void *);
    int channel;
    Size S;
    Mat src;
    Mat res;
};
Source::Source(int channel,Size S): tbb::filter(serial_in_order),channel(channel),S(S){}
void * Source::operator()(void *) {
    for(;;){
        total_frames++;
        inputVideo >> src;
        if (src.empty()) break;
        Stream *s = new Stream(src,res,channel,S);
        return s;
    }
    return NULL;
};
class StageReplicate: public tbb::filter {
public:
    vector<Mat> spl;
    StageReplicate();
    void * operator()( void *input );
};
StageReplicate::StageReplicate(): tbb::filter(parallel){}
void *StageReplicate::operator()(void *input){
    Stream *t = static_cast<Stream*>(input);
    cv::split(t->src, spl);
    for (int i =0; i < 3; ++i){
        if (i != t->channel){
            spl[i] = Mat::zeros(t->S, spl[0].type());
        }
    }
    merge(spl, t->res);
    cv::GaussianBlur(t->res, t->res, cv::Size(0, 0), 3);
    cv::addWeighted(t->res, 1.5, t->res, -0.5, 0, t->res);
    Sobel(t->res,t->res,-1,1,0,3);
    return t;
}
class Drain: public tbb::filter {
public:
    Drain();
    void * operator()( void *input );
};
Drain::Drain():tbb::filter(serial_in_order){}
void *Drain::operator()(void *input){
    Stream *t = static_cast<Stream*>(input);
    outputVideo << t->res;
    delete t;
    return NULL;
};
tbb::task_scheduler_init init(workers);
tbb::pipeline pipeline;
Source E(channel,S);
pipeline.add_filter(E);
StageReplicate SR;
pipeline.add_filter(SR);
```

```
Drain D;
pipeline.add_filter(D);
pipeline.run(workers);
```

Listing A.8: Video OpenCV using TBB.

A.3.3 Mandelbrot

```
unsigned char *M = (unsigned char *) malloc(dim);
#pragma omp parallel num_threads(workers)
{
  for(int i=0;i<dim;i++) {
    double a,b,a2,b2,cr,k;
    double im=init_b+(step*i);
    #pragma omp for
    for (int j=0;j<dim;j++){
        a=cr=init_a+step*j;
        b=im;
        k=0;
        for (k=0;k<niter;k++){
            a2=a*a;
            b2=b*b;
            if ((a2+b2)>4.0) break;
            b=2*a*b+im;
            a=a2-b2+cr;
        }
        M[j]= (unsigned char) 255-((k*255/niter));
    }
    #pragma omp single
    {
      ShowLine(M,dim,i);
    }
  }
}
```

Listing A.9: Mandelbrot using OpenMP.

```
#include <ff/farm.hpp>
#include <ff/spin-lock.hpp>
#include <ff/mapping_utils.hpp>
using namespace ff;
static lock_t lock;
typedef struct outitem {
  unsigned char * M;
  int line;
} ostream_t;
const double range=3.0;
const double init_a=-2.125,init_b=-1.5;
double step = range/((double) DIM);
int dim = DIM;
int niter = ITERATION;
class Worker: public ff_node {
public:
    void * svc(void * task) {
    int * t = (int *)task;
    ostream_t * oi = (ostream_t*)malloc(sizeof(ostream_t));
    oi->M = (unsigned char *) malloc(dim*sizeof(char));
    int i =   oi->line = *t;
    int j,k;
    double im,a,b,a2,b2,cr;
    im=init_b+(step*i);
    for (j=0;j<dim;j++){
    a=cr=init_a+step*j;
    b=im;
    k=0;
    for (k=0;k<niter;k++){
      a2=a*a;
      b2=b*b;
      if ((a2+b2)>4.0) break;
      b=2*a*b+im;
      a=a2-b2+cr;
      }
    oi->M[j] =  (unsigned char) 255-((k*255/niter));
    }
    return oi;
```

```
    }
};
class Worker2: public ff_node {
public:
    void * svc(void * task) {
    int * t = (int *)task;
    ostream_t * oi = (ostream_t*)malloc(sizeof(ostream_t));
    oi->M = (unsigned char *) malloc(dim*sizeof(char));
    int i =   oi->line = *t;
    int j,k;
    double im,a,b,a2,b2,cr;
    im=init_b+(step*i);
    for (j=0;j<dim;j++)   {
    a=cr=init_a+step*j;
    b=im;
    k=0;
    for (k=0;k<niter;k++)
      {
      a2=a*a;
      b2=b*b;
      if ((a2+b2)>4.0) break;
      b=2*a*b+im;
      a=a2-b2+cr;
      }
    oi->M[j] =  (unsigned char) 255-((k*255/niter));
    }
    spin_lock(lock);
    ShowLine(oi->M,dim,oi->line);
    spin_unlock(lock);
      free(oi->M);
    free(oi);
    return GO_ON;
    }
};
class Collector: public ff_node {
public:
  void * svc(void * task) {
  ostream_t * t = (ostream_t *)task;
  ShowLine(t->M,dim,t->line);
  free(t->M);
  free(t);
  return GO_ON;
  }
private:
  int init;
};
class Emitter: public ff_node {
public:
  Emitter(int max_task):ntask(max_task) {};
  void * svc(void *) {
  int * task = new int(dim-ntask);
  --ntask;
  if (ntask<0) return NULL;
  return task;
  }
private:
  int ntask;
};
ff_farm<> farm(false, dim);
std::vector<ff_node *>w;
for (int k=0;k<workers;k++){
    w.push_back((ncores>=4)? ((ff_node*)new Worker) : ((ff_node*)new Worker2));
}
farm.add_workers(w);
Emitter E(dim);
farm.add_emitter(&E);
Collector C;
if (ncores>=4){
    farm.add_collector(&C);
}
if (farm.run_and_wait_end()<0) {
  error("running farm\n");
  return -1;
}
```

Listing A.10: Mandelbrot using FastFlow.

```
#include <tbb/pipeline.h>
#include <tbb/task_scheduler_init.h>
struct TBB_Stream {
  TBB_Stream(unsigned char *M, int i, double im, int niter, double init_a, double step, int dim):
  M(M),i(i),im(im),niter(niter),init_a(init_a),step(step),dim(dim){}
  unsigned char *M;
  int i;
  double im;
  int niter;
```

```
10    double init_a;
11    double step;
12    int dim;
13  };
14  class Emitter_tbb: public tbb::filter {
15  public:
16      Emitter_tbb(int dim, double init_b, double step, double init_a, int niter, int i);
17      void *operator()(void *);
18      int dim;
19      double init_b, step, init_a;
20      int niter;
21      int i;
22  };
23  Emitter_tbb::Emitter_tbb(int dim, double init_b, double step, double init_a, int niter, int i) :
24      tbb::filter(serial_in_order),dim(dim),init_b(init_b),step(step),init_a(init_a),niter(niter),i(
        i)
25  {}
26  void * Emitter_tbb::operator()(void *) {
27    while((i++)<dim) {
28      unsigned char *M = (unsigned char *) malloc(dim);
29      double im=init_b+(step*i);
30      TBB_Stream *stream = new TBB_Stream(M,i,im,niter,init_a,step,dim);
31      return stream;
32    }
33    return NULL;
34  }
35  class StageReplicate_tbb: public tbb::filter {
36  public:
37      StageReplicate_tbb();
38      void * operator()( void *input );
39  };
40  StageReplicate_tbb::StageReplicate_tbb(): tbb::filter(parallel){}
41  void *StageReplicate_tbb::operator()(void *input){
42    TBB_Stream *in = static_cast<TBB_Stream*>(input);
43    double a,b,a2,b2,cr,k;
44    for (int j=0;j<in->dim;j++){
45        a=cr=in->init_a+in->step*j;
46        b=in->im;
47        k=0;
48        for (k=0;k<in->niter;k++){
49            a2=a*a;
50            b2=b*b;
51            if ((a2+b2)>4.0) break;
52            b=2*a*b+in->im;
53            a=a2-b2+cr;
54        }
55        in->M[j]= (unsigned char) 255-((k*255/in->niter));
56    }
57    return in;
58  }
59  class Stage_tbb: public tbb::filter {
60  public:
61      Stage_tbb();
62      void * operator()( void *input );
63  };
64  Stage_tbb::Stage_tbb(): tbb::filter(serial_in_order){}
65  void *Stage_tbb::operator()(void *input){
66    TBB_Stream *in = static_cast<TBB_Stream*>(input);
67    ShowLine(in->M,in->dim,in->i);
68    free(in->M);
69    delete in;
70    return NULL;
71  }
72  tbb::task_scheduler_init init((workers+2));
73  tbb::pipeline pipeline;
74  int i=0;
75  Emitter_tbb Emitter(dim,init_b,step,init_a,niter,i);
76  pipeline.add_filter(Emitter);
77  StageReplicate_tbb StageReplicate;
78  pipeline.add_filter(StageReplicate);
79  Stage_tbb S;
80  pipeline.add_filter(S);
81  pipeline.run(workers);
```

Listing A.11: Mandelbrot using TBB.

A.3.4 Prime Numbers

```
int prime_number ( int n ){
  int total = 0;
  #pragma omp parallel for shared (n) reduction (+:total) num_threads(workers) schedule(dynamic)
  for (int i = 2; i <= n; i++ ){
    int prime = 1;
    for (int j = 2; j < i; j++ ){
      if ( i % j == 0 ){
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  return total;
}
```

Listing A.12: Prime Numbers using OpenMP.

```
#include <ff/pipeline.hpp>
using namespace ff;
struct Spar_Stream {
  Spar_Stream(int i, int prime):
   i(i),prime(prime) {};
    int i;
    int prime;

};
void StageReplicate_Spar(Spar_Stream *in){
  for (int j = 2; j < in->i; j++ ){
    if ( in->i % j == 0 ){
      in->prime = 0;
      break;
    }
  }
}
struct ToStream_ff: ff_node_t<Spar_Stream> {
  int n;
  ToStream_ff(int n):
      n(n) {}

    Spar_Stream *svc(Spar_Stream *in){
      for (int i = 2; i <= n; i++ ){
      int prime = 1;
      Spar_Stream *stream = new Spar_Stream(i,prime);
      ff_send_out(stream);
    }
    return EOS;
    }
};
Spar_Stream *StageReplicate_ff (Spar_Stream *in, ff_node*const){
  StageReplicate_Spar(in);
  return in;
}
struct Collector: ff_node_t<Spar_Stream> {
  Collector(int total):total(total) {}
  int total;
  Spar_Stream *svc(Spar_Stream *in) {
    total = total + in->prime;
  delete in;
  return GO_ON;
  }
};
int prime_number ( int n ){
  int total = 0;
  ToStream_ff ToStream(n);
  Collector C(total);
  ff_Farm<Spar_Stream> StageReplicate(StageReplicate_ff,workers);
  StageReplicate.add_collector(C);
  StageReplicate.add_emitter(ToStream);
  StageReplicate.set_scheduling_ondemand();
  if (StageReplicate.run_and_wait_end()<0) { // executes the farm
    error("Running farm\n");
    return -1;
  }
  total=C.total;
  return total;
}
```

Listing A.13: Prime Numbers using FastFlow.

```
#include <ff/parallel_for.hpp>
using namespace ff;
int prime_number ( int n ){
```

```
  int total = 0;
  auto reduceF = [](int& total, int elem) { total += elem; };
  auto bodyF   = [](int i, int &total) {
    int prime = 1;
    for (int j = 2; j < i; j++ ){
      if ( i % j == 0 ){
        prime = 0;
        break;
      }
    }
    total+=prime;
  };
  ParallelForReduce<int> pfr(workers);
  pfr.parallel_reduce(total, 0, 0, n, 1, 1, bodyF, reduceF, workers);
  return total;
}
```

Listing A.14: Prime Numbers using FastFlow (parallel for).

```
#include <tbb/pipeline.h>
#include <tbb/task_scheduler_init.h>
struct Stream_tbb {
  Stream_tbb(int i, int prime):
   i(i),prime(prime) {};
    int i;
    int prime;
};
void StageReplicate(Stream_tbb *in){
  for (int j = 2; j < in->i; j++ ){
    if ( in->i % j == 0 ){
      in->prime = 0;
      break;
    }
  }
}
class Emitter_tbb: public tbb::filter {
public:
    Emitter_tbb(int n, int i);
    void *operator()(void *);
    int n;
    int i;
};
Emitter_tbb::Emitter_tbb(int n,int i) :
    tbb::filter(serial_in_order),n(n),i(i)
{}
void * Emitter_tbb::operator()(void *) {
  while((i++) <= n){
    int prime = 1;
    Stream_tbb *stream = new Stream_tbb(i,prime);
    return stream;
  }
  return NULL;
}
class StageReplicate_tbb: public tbb::filter {
public:
    StageReplicate_tbb();
    void * operator()( void *input );
};
StageReplicate_tbb::StageReplicate_tbb(): tbb::filter(parallel){}
void *StageReplicate_tbb::operator()(void *input){
  Stream_tbb *in = static_cast<Stream_tbb*>(input);
  if (in != NULL){
    StageReplicate(in);
    return in;
  }
  return NULL;
}
class Stage_tbb: public tbb::filter {
public:
    Stage_tbb(int total);
    void * operator()( void *input );
    int total=0;
};
Stage_tbb::Stage_tbb(int total): tbb::filter(serial_in_order),total(total){}
void *Stage_tbb::operator()(void *input){
  Stream_tbb *in = static_cast<Stream_tbb*>(input);
  total = total + in->prime;
  delete in;
  return NULL;
}
int prime_number ( int n ){
  int total = 0;
  int i=2;
  tbb::task_scheduler_init init(workers);
  tbb::pipeline pipeline;
  Emitter_tbb Emitter(n,i);
```

```
68    pipeline.add_filter(Emitter);
69    StageReplicate_tbb SR;
70    pipeline.add_filter(SR);
71    Stage_tbb S(total);
72    pipeline.add_filter(S);
73    pipeline.run(workers);
74    total = S.total;
75    return total;
76  }
```

Listing A.15: Prime Numbers using TBB.

```
1  #include "tbb/parallel_reduce.h"
2  #include "tbb/blocked_range.h"
3  #include <tbb/task_scheduler_init.h>
4  using namespace tbb;
5  int prime_number ( int n ){
6    affinity_partitioner ap;
7    task_scheduler_init init(workers);
8    int total = 0;
9    auto bodyF   = [](const blocked_range<int> &r, int in) -> int {
10     for (int i=r.begin();i!=r.end();++i) {
11       int prime = 1;
12       for (int j = 2; j < i; j++ ){
13         if ( i % j == 0 ){
14           prime = 0;
15           break;
16         }
17       }
18       in+=prime;
19     }
```

Listing A.16: Prime Numbers using TBB (parallel for).

A.3.5 K-Means

```
1  while (modified){
2      modified = false;
3      #pragma omp parallel for schedule(dynamic)
4      for (int i = 0; i < num_points; i++){
5          unsigned int min_dist = get_sq_dist(points[i], means[0]);
6          int min_idx = 0;
7          for (int j = 1; j < num_means; j++){
8              unsigned int cur_dist = get_sq_dist(points[i], means[j]);
9              if (cur_dist < min_dist){
10                 min_dist = cur_dist;
11                 min_idx = j;
12             }
13         }
14         if (clusters[i] != min_idx){
15             clusters[i] = min_idx;
16             modified = true;
17         }
18     }
19     #pragma omp parallel for schedule(dynamic)
20     for (int i = 0; i < num_means; i++){
21         int* sum = (int *)malloc(dim * sizeof(int));
22         memset(sum, 0, dim * sizeof(int));
23         int grp_size = 0;
24         for (int j = 0; j < num_points; j++){
25             if (clusters[j] == i){
26                 add_to_sum(sum, points[j]);
27                 grp_size++;
28             }
29         }
30         for (int j = 0; j < dim; j++){
31             if (grp_size != 0){
32                 means[i][j] = sum[j] / grp_size;
33             }
34         }
35         free(sum);
36     }
37 }
```

Listing A.17: K-Means using OpenMP.

```
#include <ff/parallel_for.hpp>
using namespace ff;
ParallelFor pf(atoi(argv[1]), false);
while (modified){
  modified = false;
  pf.parallel_for(0,num_points,[points,means,&clusters,&modified](int i){
    unsigned int min_dist = get_sq_dist(points[i], means[0]);
    int min_idx = 0;
    for (int j = 1; j < num_means; j++)     {
      unsigned int cur_dist = get_sq_dist(points[i], means[j]);
      if (cur_dist < min_dist){
        min_dist = cur_dist;
        min_idx = j;
      }
    }
    if (clusters[i] != min_idx){
      clusters[i] = min_idx;
      modified = true;
    }
  });
  pf.parallel_for(0,num_means,[points,&means,clusters,modified](int i){
    int* sum = (int *)malloc(dim * sizeof(int));
    memset(sum, 0, dim * sizeof(int));
    int grp_size = 0;
    for (int j = 0; j < num_points; j++){
      if (clusters[j] == i){
        add_to_sum(sum, points[j]);
        grp_size++;
      }
    }
    for (int j = 0; j < dim; j++){
      if (grp_size != 0){
        means[i][j] = sum[j] / grp_size;
      }
    }
    free(sum);
  });
}
```

Listing A.18: K-Means using FastFlow.

```
#include "tbb/tbb.h"
using namespace tbb;
tbb::task_scheduler_init init(atoi(argv[1]));
while (modified){
  modified = false;
  parallel_for(0,num_points,[points,means,&clusters,&modified](int i){
    unsigned int min_dist = get_sq_dist(points[i], means[0]);
    int min_idx = 0;
    for (int j = 1; j < num_means; j++){
      unsigned int cur_dist = get_sq_dist(points[i], means[j]);
      if (cur_dist < min_dist){
        min_dist = cur_dist;
        min_idx = j;
      }
    }
    if (clusters[i] != min_idx) {
      clusters[i] = min_idx;
      modified = true;
    }
  });
  parallel_for(0,num_means,[points,&means,clusters,modified](int i){
    int* sum = (int *)malloc(dim * sizeof(int));
    memset(sum, 0, dim * sizeof(int));
    int grp_size = 0;
    for (int j = 0; j < num_points; j++){
      if (clusters[j] == i){
        add_to_sum(sum, points[j]);
        grp_size++;
      }
    }
    for (int j = 0; j < dim; j++){
      if (grp_size != 0){
        means[i][j] = sum[j] / grp_size;
      }
    }
    free(sum);
  });
}
```

Listing A.19: k-Means using TBB.