PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

ADRIANO VOGEL

ADAPTIVE DEGREE OF PARALLELISM FOR THE SPAR RUNTIME

Porto Alegre
2018

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

ADAPTIVE DEGREE OF PARALLELISM FOR THE SPAR RUNTIME

ADRIANO VOGEL

Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Luiz Gustavo Fernandes, PhD
Co-Advisor: Prof. Dalvan Griebler, PhD

Porto Alegre
2018

Ficha Catalográfica

V878a Vogel, Adriano

Adaptive Degree of Parallelism for the SPar Runtime / Adriano Vogel . – 2018.
98 f.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Luiz Gustavo Fernandes.
Co-orientador: Prof. Dr. Dalvan Griebler.

1. Stream Parallelism. 2. Abstracted and Adaptive Degree of Parallelism. I. Fernandes, Luiz Gustavo. II. Griebler, Dalvan. III. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Salete Maria Sartori CRB-10/1363

Adriano Vogel

Adaptive Degree of Parallelism for the Spar Runtime

This Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on 28th , 2018.

COMMITTEE MEMBERS:

Prof. Dr. Marco Danelutto (University of Pisa)

Prof. Dr. Avelino Francisco Zorzo (PPGCC/PUCRS)

Prof. Dr. Luiz Gustavo Leão Fernandes (PPGCC/PUCRS – Advisor)

Prof. Dr. Dalvan Jair Griebler (PNPD/PUCRS - Co-Advisor)

To my family, my girlfriend, and my friends.

“Do not let arrogance go to your head and despair to your heart; do not let compliments go to your head and criticisms to your heart; do not let success go to your head and failure to your heart.”
(Roy T. Bennett)

ACKNOWLEDGMENTS

First I would like to thank my advisors for guiding this research. Second, my peers and lab colleagues for their help and technical discussions. This work was partially supported by funding from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPQ), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), and by the HiPerfCloud Project.

ADAPTIVE DEGREE OF PARALLELISM FOR THE SPAR RUNTIME

RESUMO

As aplicações de *stream* se tornaram cargas de trabalho representativas nos sistemas computacionais. Diversos domínios de aplicações representam *stream*, como vídeo, áudio, processamento de gráficos e imagens. Uma parcela significativa dessas aplicações demanda paralelismo para aumentar o desempenho. Porém, programadores frequentemente enfrentam desafios entre produtividade de código e desempenho devido às complexidades decorrentes do paralelismo. Buscando facilitar a paralelização, a DSL SPar foi criada usando atributos *(stage, input, output, and replicate)* do `C++-11` para representar paralelismo. O compilador reconhece os atributos da SPar e gera código paralelo automaticamente. Um desafio relevante que ocorre em estágios paralelos da SPar é a demanda pela definição manual do grau de paralelismo, o que consome tempo e pode induzir a erros. O grau de paralelismo é definido através do número de réplicas em estágios paralelos. Porém, a execução de diversas aplicações pode ser pouco eficiente se executadas com um número de réplicas inadequado ou usando um número estático que ignora a natureza dinâmica de algumas aplicações. Para resolver esse problema, é introduzido o conceito de um número de réplicas transparente e adaptativo para a SPar. Além disso, os mecanismos implementados e as regras de transformação são descritos para possibilitar a geração de código paralelo na SPar com um número adaptativo de réplicas. Os mecanismos adaptativos foram avaliados demonstrando a sua eficácia. Ainda, aplicações reais foram usadas para demonstrar que os mecanismos adaptativos propostos podem oferecer abstrações de alto nível sem significativas perdas de desempenho.

Palavras-Chave: Paralelismo de Stream, Paralelismo Adaptativo e Abstrato.

ADAPTIVE DEGREE OF PARALLELISM FOR THE SPAR RUNTIME

ABSTRACT

In recent years, stream processing applications have become a traditional workload in computing systems. They are traditionally found in video, audio, graphic and image processing. Many of these applications demand parallelism to increase performance. However, programmers must often face the trade-off between coding productivity and performance that introducing parallelism creates. SPar Domain-Specific Language (DSL) was created to achieve the optimal balance for programmers, with the `C++-11` attribute annotation mechanism to ensure that essential properties of stream parallelism could be represented (stage, input, output, and replicate). The compiler recognizes the SPar attributes and generates parallel code automatically. The need to manually define parallelism is one crucial challenge for increasing SPAR's abstraction level, because it is time consuming and error prone. Also, executing several applications can fail to be efficient when running a non-suitable number of replicas. This occurs when the defined number of replicas in a parallel region is not optimal or when a static number is used, which ignores the dynamic nature of stream processing applications. In order to solve this problem, we introduced the concept of the abstracted and adaptive number of replicas for SPar. Moreover, we described our implemented strategy as well as transformation rules that enable SPar to generate parallel code with the adaptive degree of parallelism support. We experimentally evaluated the implemented adaptive strategies regarding their effectiveness. Thus, we used real-world applications to demonstrate that our adaptive strategy implementations can provide higher abstraction levels without significant performance degradation.

Keywords: Stream Parallelism, Abstracted and Adaptive Degree of Parallelism.

LIST OF FIGURES

LIST OF ABBREVIATIONS

API. – Application Programming Interface

AST. – Abstract Syntax Tree

CPU. – Central Processsing Unit

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

DAG. – Directed Acyclic Graph

DSL. – Domain-Specific Language

FIFO. – First In, First Out

GMAP. – Grupo de Modelagem de Aplicações Paralelas

MIT. – Massachusetts Institute of Technology

ms. – millisecond

QoS. – Quality of Service

SPar. – Stream Parallelism

TBB. – Threading Building Blocks

UPL. – Utility Performance Library

CONTENTS

LIST OF PAPERS

- **Private IaaS Clouds: A Comparative Analysis of OpenNebula, CloudStack and OpenStack.** *24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* [VGM+16].
- **Proposta de Implementação de Grau de Paralelismo Adaptativo em uma DSL para Paralelismo de Stream.** *Escola Regional de Alto Desempenho (ERAD)* [VGF17].
- **An Intra-Cloud Networking Performance Evaluation on CloudStack Environment.** *25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* [VGSF17].
- **Grau de Paralelismo Adaptativo na DSL SPar.** *Escola Regional de Alto Desempenho (ERAD)* [VF18].
- **Performance of Data Mining, Media, and Financial Applications under Private Cloud Conditions.** *IEEE Symposium on Computers and Communications (ISCC)* [GVM+18].
- **Service Level Objectives via C++11 Attributes.** *Euro-Par: Parallel Processing Workshops, International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms* [GSV+18].
- **Autonomic and Latency-Aware Degree of Parallelism Management in SPar.** *Euro-Par: Parallel Processing Workshops, International Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing* [VGS+18].

1. INTRODUCTION

The increasing use of techniques to collect data from different sources (*e.g.,* sensors, cameras, radar) has given rise to stream applications. This new type of application has unique aspects, such as continuous data processing and varied volume of input streams. The Quality of Service (QoS) of these applications relies mainly on latency, throughput, memory, and CPU usage metrics [CQ09].

The stream processing paradigm emerged due to the need to built systems capable of processing a continuous flow of data [AGT14]. It is also a consequence of technological advances in database and distributed systems, combined with the increased usage of signal processing and big data technologies. There are a huge number of domains that must gather and analyze data in real time [AGT14]. However, this is a difficult task that presents several challenges.

Adding to the level of complexity, currently several processors are placed on the same chip. Despite the opportunity to improve performance, applications must be able to run in parallel in order to properly exploit the hardware [MRR12]. Therefore, the concept of parallel programming, which emerged based on the idea of dividing a problem into several parts, is also applied to stream processing applications as a way to improve their performance.

1.1 Motivation

Since it is possible to run stream processing applications in parallel, several programming frameworks and libraries were developed to facilitate the task of parallel programming, such as Intel Thread Building Blocks (TBB) [Rei07], FastFlow [Fas17, AMT10] and StreamIt [TKA02]. TBB [Rei07] is Intel's tool for parallel programming, which uses the C++ standard. It exploits a template interface for expressing parallelism using the Pipeline pattern, where stages of the pipeline can run in parallel with a static degree of parallelism. A comparable abstraction in TBB is performed by decomposing threads into tasks.

StreamIt [TKA02] is a programming language developed by the Massachusetts Institute of Technology (MIT) which uses a new language that was designed for coding stream processing applications. It is also based on a static degree of parallelism, although it has extensions for an adaptive degree of parallelism [SMMF15]. Another solution is FastFlow [Fas17, AMT10], which is considered a high-level and skeleton-based parallel programming library created by researchers from the University of Pisa and the University of Turin. The FastFlow programming interface is also based on C++ templates and offers a flexible runtime library. Executions with its runtime library use a static degree of parallelism.

The programming complexities involved in introducing parallelism in stream processing applications have motivated research efforts for abstractions. Yet, the coding abstraction introduced by the previously mentioned programming frameworks remain insufficient for application programmers, who are focused on developing stream processing applications [Gri16] and which may not necessarily be parallel programming experts.

In order to address the need for further abstractions in stream parallelism, SPar [GF17, GDTF15] was created as an internal DSL (Domain-Specific Language). It maintains the host language (C++) syntax and enables high-level stream parallelism through code annotations. Aiming to increase productivity and abstraction to efficiently make sequential code parallel, it was designed with `C++-11` attribute annotation mechanism. Using SPar, programmers can identify regions that are able to run in parallel and annotate the sequential code by using attributes, which are handled at the compiler level. The SPar compiler parses the attributes and generates the parallel code. The degree of parallelism in SPar is defined through the `Replicate` attribute. A static number of replicas is set for each execution by the user/programmer. Despite the fact that SPar uses high-level abstractions, it has been proven to have good performance while running applications [GFDF18, GHDF17, GHL+17, Gri16]. Listing 1.1 gives an example of a stream processing application coded using SPar attributes. In this example, the data type is a “string” and the input stream comes from a file (read in line 3). This code block is a loop with iterations and a new stream item is read and computed (line 6) for each iteration. In line 5, the attribute *Replicate* defines the degree of parallelism to 4 replicas, which is the static number of replicas used during the entire execution. An example of a code that produces an output is shown in line 8.

```
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),
     spar::Replicate(4)]]
6   { compute(stream_element); }
7   [[spar::Stage,spar::Input(stream_element)]]
8   { write_out(stream_element); }
9 }
```

Listing 1.1 – Simple stream computation using SPar to annotate parallelism. Extracted from [GDTF17].

This work addresses the problem related to the abstraction of manually setting the degree of parallelism in SPar. For the most part, defining the degree of parallelism is a complicated and time-consuming task, because the programmer has to run the same program several times to decide which is the best configuration. Also, some stream processing applications have load fluctuations which require continuous adaptations to achieve optimal

throughput [VF18]. Consequently, running and testing several configurations is not a feasible approach for stream processing applications. Unfortunately, currently neither SPar nor the other state-of-the-art frameworks (TBB, FastFlow, and StreamIt) enable ready-to-use and adaptive degree of parallelism. Hence, when the load increases, the performance can be degraded due to the use of a static number of replicas. Moreover, when the load decreases, unnecessary resources may be used, which is a waste of computing power.

An example of the complexities involved in defining the degree of parallelism is shown in Figure 1.1. Here the authors [PGB11] compared PARSEC applications' performance in a 24 core machine. This evaluation indicates that the Bodytrack application achieved the best performance when using one thread for each core, while Ferret had the best speedup when running 64 threads in the 24 core machine (3 threads per core). Thus, we can infer that the common practice of defining the degree of parallelism with one thread per core is not suitable for some applications.

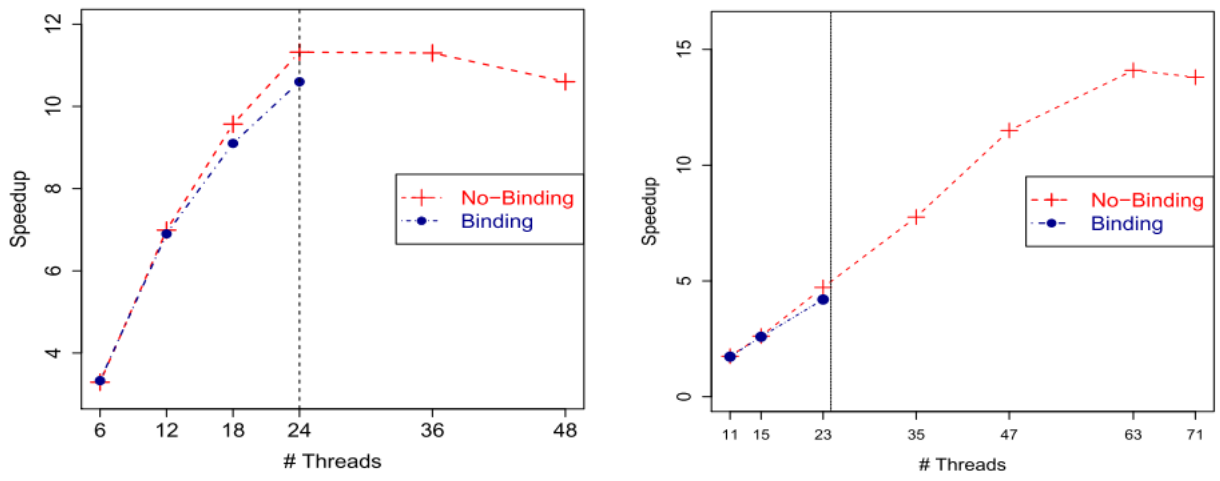


Figure 1.1 – Performance of Bodytrack (Left) and Ferret (Right). Extracted from [PGB11].

The evaluation of PARSEC applications in [PGB11] shows how relevant and at the same time how difficult it is to find an optimal degree of parallelism. However, it is important to note that the problem addressed by [PGB11] is distinct from ours. In addition to stream processing applications, they use a different runtime system and consider aspects of the operating system. The technique presented in [PGB11] is unable to adapt the degree of parallelism on-the-fly, it needs to rerun an application to adapt the degree of parallelism. Consequently, [PGB11]'s technique may not be suitable for real-world stream processing applications.

1.2 Contributions

The main goal of this work is to provide an adaptive degree of parallelism for stream processing applications. There are related studies available in the literature. Yet, they use distinct frameworks and have other goals, such as improving utilization and energy efficiency [STD16]. Whereas *e.g.* utilizes a proactive approach [MM16], [SST+15] uses self-aware adaptations for stream processing applications. Although [GSHW14] addressed the same goal, we target different scenarios and runtime libraries. The main contributions of this work are threefold:

- Providing a fully abstracted adaptive strategy to automatically adapt the degree of parallelism in SPar. By monitoring the application during its execution and taking actions to optimize the number of active replicas, programmers are no longer required to provide the degree of parallelism manually.
- Implementing strategies for a wide range of stream processing applications and considering different metrics. The adaptive strategies were designed based on a feedback loop from the control theory concept. The degree of parallelism can be adapted for different objectives (throughput, latency, and congestion). It enables QoS by maximizing throughput and also controls latency and runtime congestion.
- A performance evaluation of the strategies. We provide an implementation with real-world stream processing applications and compare the performance of adaptive to static executions.

1.3 Thesis Organization

The remainder of this work is organized as follows: the next chapter presents the background and context of this study. The related approaches are then presented in Chapter 3. Implementations are presented in Chapter 4 and Chapter 5 discusses the transformation rules for implementing the adaptive strategies in SPar. Chapter 6 consists of the experiment results. Finally, the conclusion and discussion are presented in Chapter 7.

2. BACKGROUND

We have seen the rise of a new type of application over the past few decades – the stream class. These applications arose to address the need to process data in a continuous fashion without having a predefined end to the execution (sometimes there is no end) [CQ09]. In Section 2.1, we introduce the parallel patterns that best fit the stream parallelism context. An overview of this paradigm is presented in Section 2.2. Then in Section 2.3, we describe the SPar, a DSL for stream parallelism that we intend to extend in this work. Lastly, the fundamentals for designing an adaptive degree of parallelism are shown in Section 2.4.

2.1 Structured Parallel Programming

Modern computers are now parallel and support parallelism features (*e.g.,* vector instructions, multicore processors, co-processors). However, automatic parallelization approaches for serial code are not effective [MRR12]. Consequently, a parallel execution requires explicit parallel programming in order to exploit the machine's resources. McCool et al. [MRR12] emphasizes the advances and challenges regarding parallelism:

> "*Despite the fact that computer hardware is naturally parallel, computer architects chose 40 years ago to present a serial programming abstraction to programmers. Decades of work in computer architecture have focused on maintaining the illusion of serial execution. Extensive efforts are made inside modern processors to translate serial programs into a parallel form so they can execute efficiently using the fine-grained parallel hardware inside the processor. Unfortunately, driven by the exponential increase in the number of transistors provided by Moore's Law, the need for parallelism is now so great that it is no longer possible to maintain the serial illusion while continuing to scale performance. It is now necessary for programmers to explicitly specify parallel algorithms if they want their performance to scale. Parallelism is everywhere, and it is the path to performance on modern computer architectures.*"

In theory, any application may be seen as many tasks that run concurrently to achieve a specific goal [Sen12]. Parallelism is a suitable way to scale applications' performance, but developing an efficient parallel application is considered both challenging and error-prone [MRR12, FAG+17, GDTF17]. This has motivated efforts towards increasing the level of abstraction and hiding the complexities from application programmers (*e.g.,* concurrency, synchronization, scheduling). One way to do so is to follow the methodologies for structured parallel programming.

Structured Parallel Programming can be understood as a methodology for programmers using libraries or languages to introduce parallelism. Patterns are a representation of structured parallel programming and are currently a well-accepted concept that optimizes and reuses specific parts of code. They emerge from the best practices of codifying in software engineering. Thus, Parallel Patterns use the methodology from structured parallel programming in order to enable code abstractions. The reference material presented by [MRR12] defines the term parallel pattern as: “a recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design”.

In this context, we also have skeletons [Col89] that have a similar goal of parallel patterns, but skeletons came from the High Performance Computing (HPC) community. In our specific case, we consider skeletons in the same context as parallel patterns since they are related, but simply emerged in different areas [MRR12]. Each pattern or skeleton has a specific name that describes it, making it easier to compare different patterns. Consequently, studying patterns results in vocabulary, which is one of the most relevant aspects used to design algorithms. In this case, patterns are combined to model parallel applications.

Patterns are considered universal, which means that they can be applied to all existing parallel programming systems [MRR12]. Furthermore, there are already a very large number of patterns in different areas. However, in the specific case of this work focused on stream processing applications, we are interested in pipeline and farm patterns. Pipeline and farm are within the stream parallelism category and their computations are independent tasks running in parallel [Gri16, Sen12]. In this study, the forms of parallelism described in this section are used to model parallel applications for the stream scenario (Section 2.2).

A significant part of applications can be represented by using implemented patterns called *Skeletons* [Col89, Col04]. These programming patterns are introduced by libraries or higher order functions (templates). *Skeletons* enable the modeling of parallel programs by composing typical patterns.

2.1.1 Pipeline

A pipeline is a *skeleton* that handles tasks in a producer-consumer fashion. In theory, every pipeline stage is active at once, and each one performs different operations in the input data, working in sequence like on an assembly line. A pipeline stage can be serial (one item at a time) or parallel (replicated). Even when there are serial stages, a pipeline still runs in parallel, because different stages perform operations at the same time, as shown in Figure 2.1. For instance, a pipeline’s performance is the time it takes to process a given task. Consequently, the slowest stage limits performance (*e.g.,* throughput).

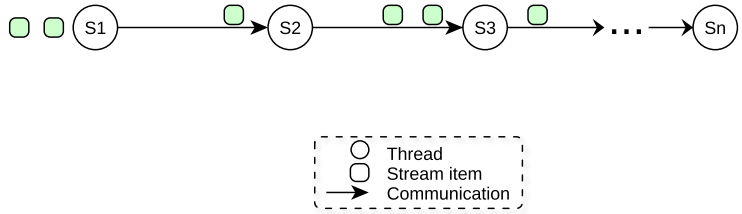


Figure 2.1 – Pipeline.

2.1.2 Farm

Task-farm is a skeleton that uses functional replication with modules [Sen12, Fas17]. Moreover, a task-farm may also be called a master-worker pattern, we simply call it a farm.

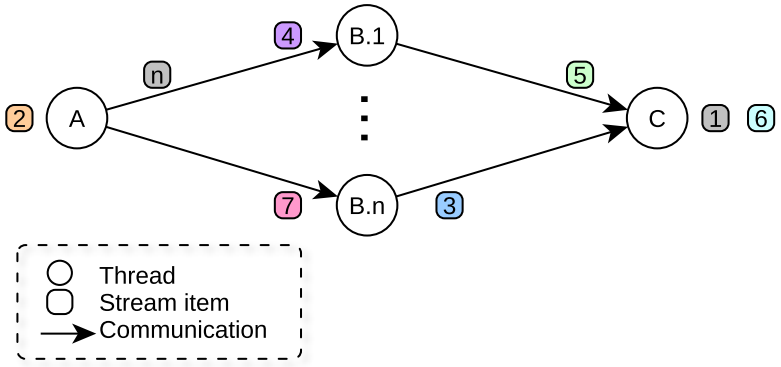


Figure 2.2 – Basic Farm.

A basic farm is shown in Figure 2.2. It has at least one activity, called an emitter, which receives the input tasks and sends them to next stage(workers), according to a scheduling policy. An emitter is then replicated with a number N of parallel agents (called workers or replicas), N being the degree of parallelism. The collector then gathers the tasks from the workers and places them in the output stream. The collector is optional and can be removed if desired.

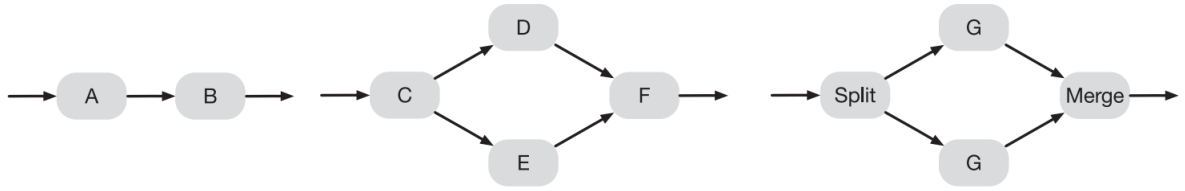
2.2 Stream Processing

A data source is called a producer when data items are consumed/analyzed by a stream processing application [AGT14]. Sources may vary and range from equipment (radar, telescopes, cameras, etc.) to files (text, image). Also, the data consumed by stream systems may have different characteristics, the three main classes are structured, semi-structured, or unstructured data.

Stream processing applications are being used more and more currently and there is an increasing demand for efficient solutions to meet the challenges they present. Therefore, research in this area is expanding rapidly. Challenges still remain when it comes to properly exploiting stream systems. Due to the large number of studies, this area lacks unified taxonomies and terminologies. Thus, [HSS+14]'s work presented a survey and cataloged the prevailing terminology and definitions with the aim of providing a common vocabulary. The authors also combined the synonymous terms and explained the differences between similar definitions. For instance, [HSS+14] defines stream processing system as a runtime system that can execute stream tasks [1].

Additionally, a survey of stream processing systems is presented by [WKWO12], which classifies the existing solutions according a set of criteria. The main challenges regarding stream processing systems are in terms of performance, scalability, and robustness [WKWO12]. These aspects are emphasized due to the demand for handling a continuous flow of data from different sources.

Hirzel et al. [HSS+14] also discusses the terminology used for parallelism in stream processing applications, the three main examples are shown in Figure 2.3. Pipeline-parallel (a) is related to the concurrent running of different stages (A,B) using the terminology of producer and consumer. We can easily see similarities with this view of pipeline parallel and the pipeline angle from the structured parallel programming (Section 2.1.1). Task-parallel (b) on the other hand, concerns the concurrent execution of different operators (D,E) and is related to structured parallel programming with stream parallel patterns, such as the farm shown in Section 2.1.2. Moreover, in data-parallel (Figure 2.3(a)) the same task G is processed by different parallel sub tasks. In this work, we are interested in stream parallelism, which is the target scenario of the SPar DSL, described in Section 2.3.



(a) Pipeline-parallel A || B. (b) Task-parallel D || E. (c) Data-parallel G || G.

Figure 2.3 – Parallelism among stream operators. Extracted from [HSS+14].

The characteristics of stream processing applications vary depending on the source and model [CQ09]. Yet, stream processing applications have several common characteristics. One of the most highlighted aspects is the continuous arrival of data items.

The input of data items in stream systems cannot be controlled and therefore can be quite unpredictable. It can vary from a very large rate (gigabytes per minute, for instance) to just a few bytes per minute. Stream systems are exploited by users through

[1] In this work, when the term stream processing system is mentioned we are referring to this definition.

applications, and thousands of such applications are available in several areas. For example, in telecommunications (online billing, network switches, signal processing), sensors and surveillance (monitoring, cameras, scanners), financial (stock trading, credit card), among others [AGT14], which are shown in Figure 2.4.

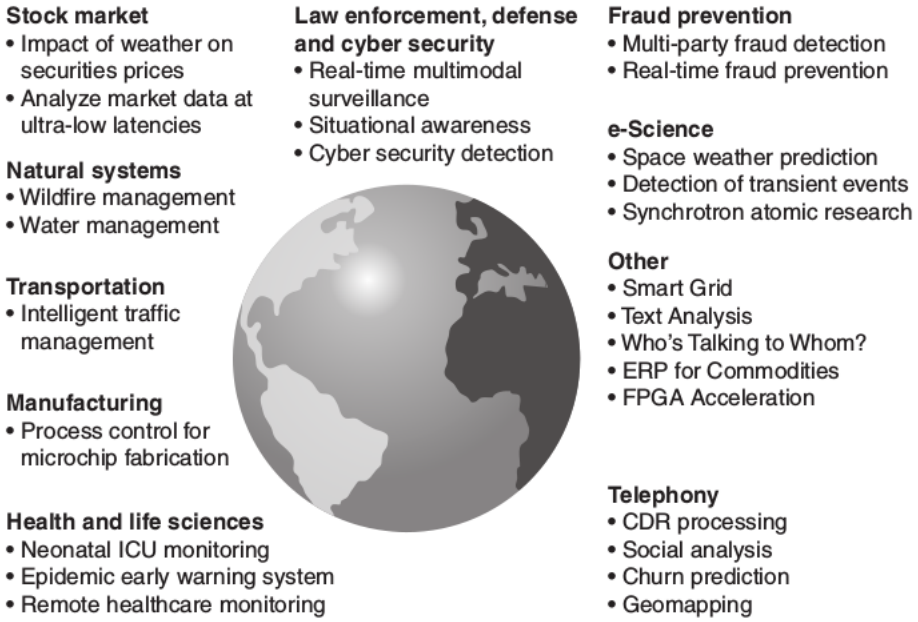


Figure 2.4 – Examples of stream processing applications. Extracted from [AGT14].

Whereas stream processing covers the class of stream processing applications, stream parallelism is broader because it models parallel applications, which can thus be applied to stream processing applications.

In order to increase the performance of stream processing applications, several types of parallelism are exploited. Stream parallelism is one variant of stream processing, where each operator performs a set of operations in stream items [Gri16]. The applications related to stream parallelism tend to require intensive use of computation resources, because applications that do not work intensively may not need to be executed in parallel or cannot efficiently use the available resources.

The large number of stream processing applications available represent a significant part of current computing systems. However, as emphasized in [Gri16], a high percentage of stream processing applications are still sequential and thus cannot run in parallel. This is the primary reason programmers face a trade-off between coding productivity and performance, thus increasing programming effort because of the need to rewrite code.

2.3 SPar

Considering the inherent challenges of parallelism in stream processing applications, SPar [GDTF17], a DSL specifically designed to facilitate stream parallelism, was been proposed. It offers high-level C++11 attributes to enable source code annotations eliminating the need to rewrite sequential code manually. SPar's compiler generates parallel code using source-to-source transformations.

All of its specific features have been designed to increase the abstraction level by using High-Level Parallelism so users do not need to manipulate low-level instructions when coding [Gri16]. Because SPar uses FastFlow [Fas17, MMPM14] as its runtime library (described in Subsection 2.3.2), programmers do not need to deal with advanced concepts, scheduling, load balancing or parallelism strategies, which are all examples of built-in functionalities provided by FastFlow.

Additionally, SPar was designed to be architecture independent, enable code portability, and coding productivity by decreasing programming effort [Gri16]. SPar also uses the C++ standard interface, C++ programmers do not need to learn a new language syntax.

2.3.1 SPar Language

SPar provides five attributes to exploit key aspects of stream parallelism. The *ToStream* attribute represents the beginning of a stream region, the code block between the *ToStream* and the first *Stage* will run as the first processing stage. The defined *ToStream* often starts a stream parallel part of a given program. The number of *Stages* are defined in *ToStream*, which sets a phase for stream items to be computed by applying operations.

Another relevant feature is the *Input* attribute, which programmers can use to define the data to be processed inside a stream region. This data is then consumed in a stage using input arguments or with a variable list. In contrast, the *Output* attribute is used to set a stream result produced in a processing stage by modifying a specific list of variables.

Replicate is another very important attribute, which is used to define the degree of parallelism. It represents the number of concurrent operations performed in a parallel region (stage), which is actually the number of replicas or worker threads. Due to the fact that SPar is currently related to stateless stream operators, each replica is independent and modifies items and produces results. When all workers have finished their processing, the results may be unordered. Therefore, SPar has a compilation flag to maintain the order of the stream items that are produced *(spar-ordered)* [GDTF17].

SPar is a C++ embedded DSL. An example of SPar's annotation attributes is shown on the left-hand side of Figure 2.5. It first reads a stream, then computes the items, and

produces an output using the attributes mentioned above. The *Replicate* attribute is the number of worker threads on stateless parallel regions. In this example, the first stage's annotation *[[spar::Stage, spar::Replicate(<integer value>)]]* uses the *Replicate* attribute for setting the degree of parallelism to any integer value defined by the programmer.

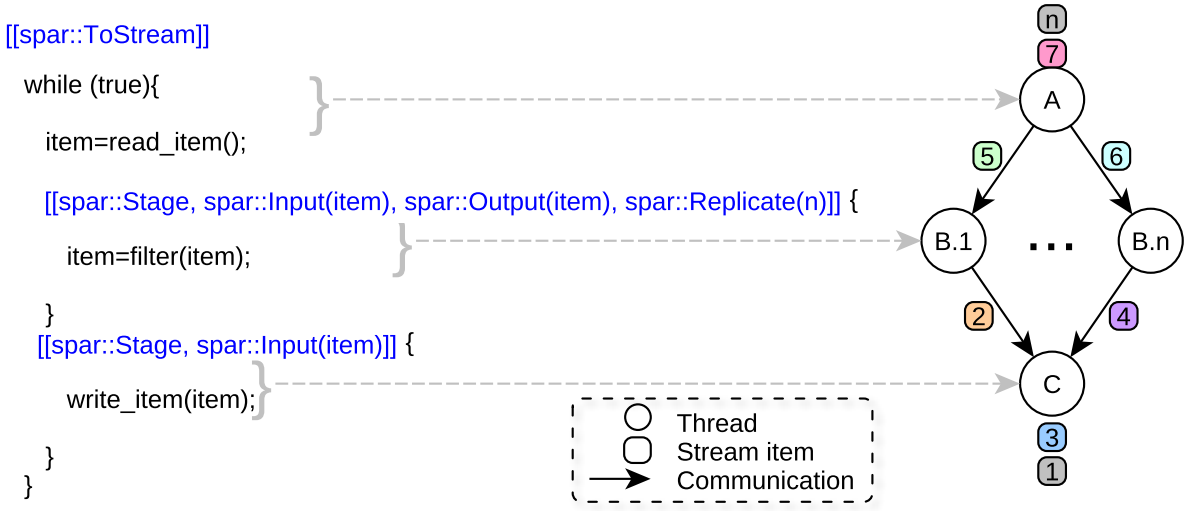


Figure 2.5 – SPar Example.

Moreover, SPar uses a compiler in order to generate parallel code. As stated by Griebler and Fernandes [GF17]:

> "The SPar compiler is designed to recognize our language and generate parallel code. It was developed by using the CINCLE (A Compiler Infrastructure for New C/C++ Language Extensions) support tools. The compiler parses the code (which is specified by a compiler flag named spar file) and builds an AST (Abstract Syntax Tree) to abstractly represent the C++ source code. Subsequently, all code transformations are made directly in the AST. Once all SPar annotations are properly transformed, another C++ code is generated, which is then compiled by invoking the GCC compiler to produce a binary output."

2.3.2 SPar Runtime

The previous section described SPar's language and how it generates parallel code, which is based on the FastFlow library. SPar also uses FastFlow's objects and runtime, because FastFlow is considered a high-level and pattern-based parallel programming library [Fas17],[MMPM14]. The runtime library has a modular architecture which provides the necessary flexibility to programmers.

The runtime library powered by FastFlow supports several parallel patterns. One of the most relevant is the pipeline, which is composed of several stages in which at least one is active during execution and each stage may update the data values when processing [Fas17]. Moreover, a farm is another relevant parallel pattern that has at least one emitter, N worker threads and sometimes a process that collects the results. The farm can be viewed as a pipeline because the heaviest stage is replicated using N replicas (threads). When a farm is implemented, the numbers of replicas, or the degree of parallelism for the execution, must to be defined. SPar generates new parallel patterns by combining FastFlow's pipeline with farm patterns.

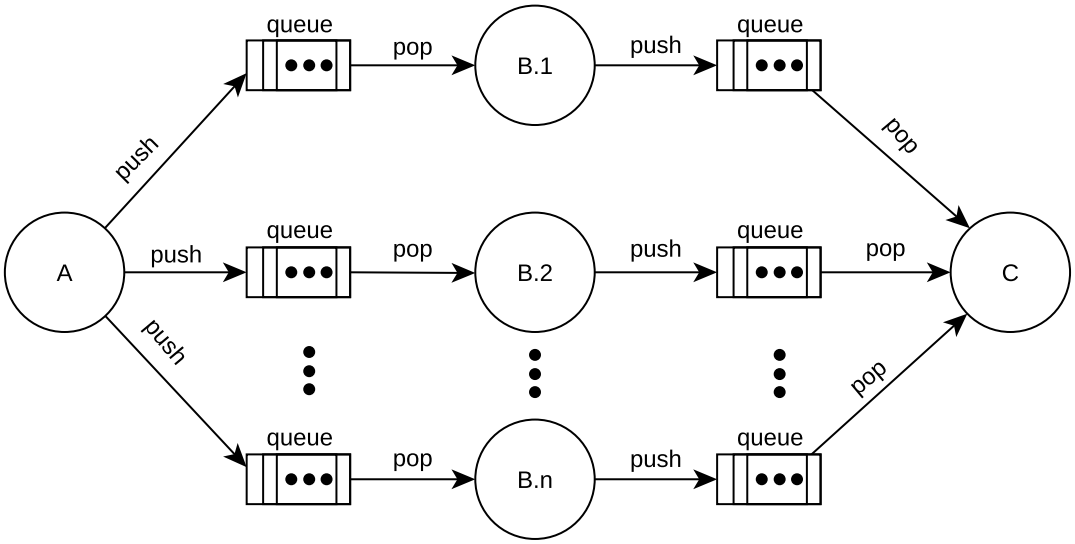


Figure 2.6 – Farm - communication queues inside SPar's runtime.

SPar's default scheduling policy is round-robin. Therefore, worker threads passively receive items to compute based on the emitter's task distribution. Moreover, auto-scheduling, which uses an on-demand policy, is also available. It optimizes the memory consumption and load balancing, and the emitter (task scheduler) continuously tries to add one task to the workers' queue. The runtime library's queues are called push and pop. A given task is added to a queue by a push operation, and a task is taken from a queue by a pop. For instance, the emitter (task scheduler) gives a task to the next stage (workers) through a push in its queue, and the worker stage gets the task with a pop in its queue, as shown in Figure 2.6. In this specific example, *n* replicas are activated (B.1, ..., B.n), each of which receive data from the previous stage and sends produced results to the subsequent stage. Communications between stages occur through shared queues. Depending on the program, they may also use a collector (last stage), which gathers the results processed by the stages. An example of a farm runtime parallelism in SPar is shown on the right-hand side of Figure 2.5.

If a worker's queue is full, the task scheduler will continuously try to add elements to its queue. Each time the scheduler is unable to add a new value to a worker's queue, there will be an event called a push lost. If the worker tries to get a new task by pop and it fails, the event will be called a pop lost.

2.4 Control Theory

Control theory is focused on automated systems. The concepts used in this work are based on [HDPT04], which uses the term feedback control for monitoring aspects such as throughput, latency or system utilization and, by triggering optimization in the next execution. Monitoring the outputs of the system and performing actions according to its behavior results in an architecture is called feedback or closed loop [HDPT04].

Control theory can be applied to stream processing applications in order to implement services that regulate their execution in order to optimize it. Yet a control framework is required to manage execution [HDPT04]. An example is a feedback control in computers is using Queueing Systems mainly for performance modelling. A simple example is shown in Figure 2.7, where the items are placed in a queue and processed by the servers (circle).

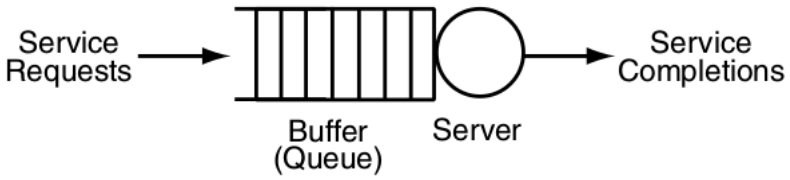


Figure 2.7 – Simple Example of Queueing System. Extracted from [HDPT04].

Figure 2.8 shows an example of a feedback control loop to manage response times in a queuing system. It is relevant to point out that such a system has a service-level objective (SLO) to maintain a given response time (constraint).

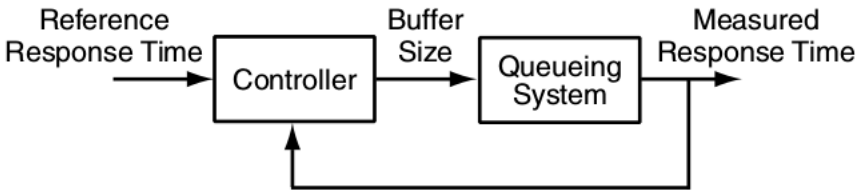


Figure 2.8 – Example of Feedback Control. Extracted from [HDPT04].

Streaming Media applications are growing and can take advantage of feedback control in order to improve their services [HDPT04]. Due to the continuous delivery of data

in stream processing applications, they can monitor the output in order to optimize future executions. For instance, sometimes the throughput must be increased according to a service objective. Figure 2.9 depicts two stream systems, each one applying different operations by request.

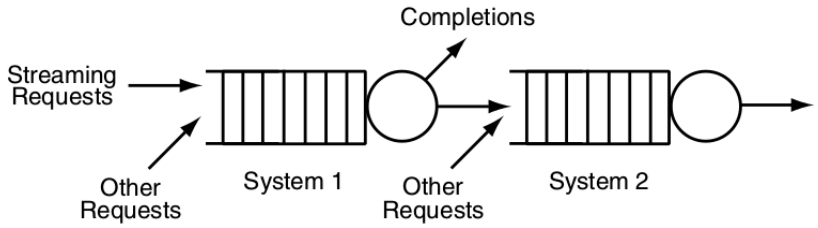


Figure 2.9 – Example of streaming. Extracted from [HDPT04].

A closed loop using a feedback control is considered a simple system model that can adapt to disturbances [HDPT04]. Thus, it fits the requirements of a stream class of applications. Related studies by [MM16] and [GSHW14] have used the properties of feedback control to optimize stream executions. In a similar vein, we are working with mechanisms to adapt the degree of parallelism using a feedback loop that optimizes based on the previous execution.

3. RELATED WORK

In this chapter the related studies that present strategies for adapting the degree of parallelism are presented and contextualized. The scope and goals were analyzed in order to select the most relevant among the variety of papers available in the area. Firstly, papers addressing streaming applications were selected. In the following paragraphs, the related studies are described and compared, such studies use adaptive (*e.g.,* elastic, automatic, self-aware, reconfigurable, dynamic, optimal) strategies for stream processing and discuss adaptiveness in respect to degrees of parallelism.

Many researchers have assessed how to best determine the optimal number of threads in parallel applications, of note are Pusukuri et al. [PGB11], Raman et al. [RKO+11] and Sridharan et al. [SGS13]. However, these approaches deal with adaptivity in non-stream processing applications. Thus, they are not feasible for streaming applications due to the specific behaviors and constraints of they present, such as the nature of load fluctuations, unpredictable input rates, and nonlinear behavior. Stream processing applications therefore require specifically targeted solutions. In the literature, there have been some works addressing the degree of parallelism problem in stream processing applications. We will describe these approaches below. Another aspect used to consider if a study is related or not is the presence of online adaptation (a.k.a. interruption-free [LDC+17]). Online adaptation is a relevant aspect of stream processing applications since it allows the system to change the degree of parallelism during the execution, without requiring the applications to rerun.

Sensi et al. [STD16, DST17, SMD17] explores stream systems, aiming to predict performance and power consumption using linear regression as a learning technique. Their goal was to reduce power consumption with “acceptable" performance losses. In order to do so, they proposed and implemented an algorithm called *NORNIR*, which enabled the application to change (number of cores, clock frequency) during run-time.

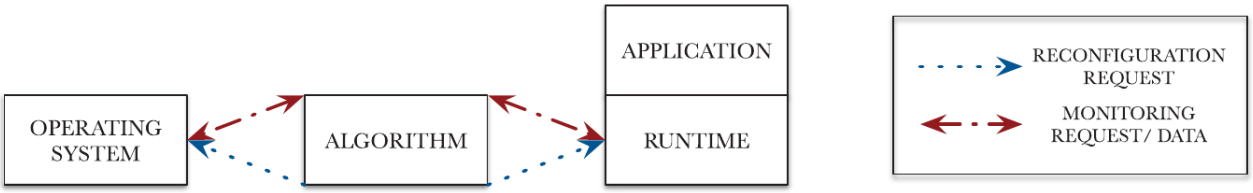


Figure 3.1 – Proposed algorithm interactions. Extracted from [STD16].

The *NORNIR* algorithm was aimed to satisfy bounds in terms of power consumption or performance, which must be defined by the user. It then triggers actions when it detects changes in the input rate or application. *NORNIR* interacts with the operating sys-

tem (OS) and with the FastFlow's runtime (Figure 3.1). Additionally, it is validated using simulations and real executions (PARSEC) [Bie11].

In addition, Matteis and Mencagli [MM16] present elastic properties for data stream processing to improve performance and energy efficiency (number of cores and frequency). They argue that using the maximum amount of resources is expensive and inefficient, and therefore promote elasticity support as a solution because it provides efficient usage according to QoS requirements and thus reduces operating costs.

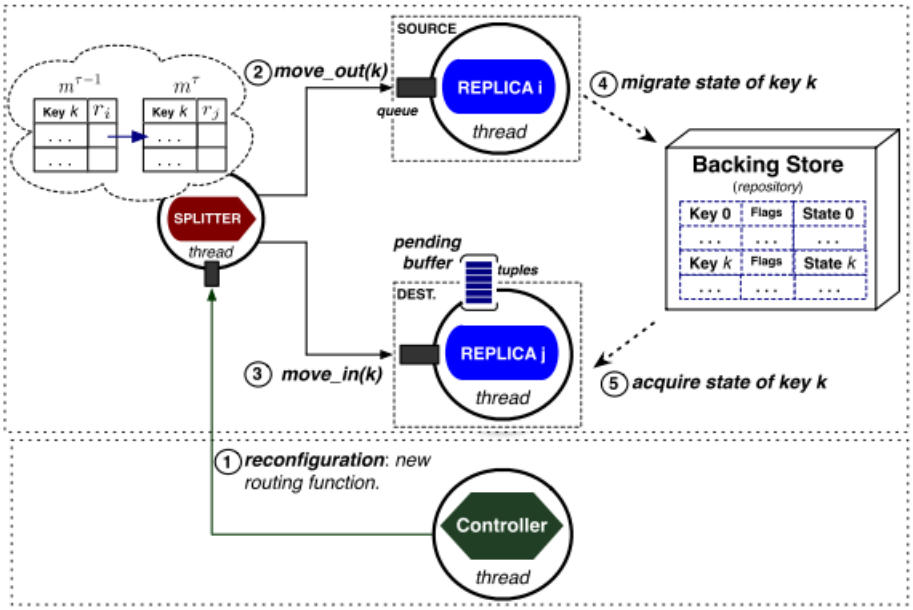


Figure 3.2 – Example of reconfiguration. Extracted from [MM16].

Their proposed model (Figure 3.2) is implemented in FastFlow's runtime. This approach concerns stateful operators, which brings additional complexities when migrating the thread states is required due to dependency among them. Figure 3.2 shows a reconfiguration example of the implementation, which uses one controller thread for monitoring the infrastructure and triggering changes. When the load increases, the controller instantiates new threads (stream replicas). Moreover, the CPUs' frequency is changed in order to reduce energy consumption using the MAMMUT library. [STD17].

An approach regarding elasticity for streaming applications, was found in Schneider et al. [SAG+09]. They extend SPADE [GAW+08], a language and compiler for developing stream processing applications, by adding elasticity support. It is implemented using a dispatcher thread, which manages the stream system (load distribution, queues). The dispatcher thread feasibility is characterized as a component that "*can increase the degree of parallelism by creating new workers or waking up old ones, and it can also decrease parallelism by putting workers to sleep*"([SAG+09]). A key component of the system is the alarm thread which "*wakes up and tells the dispatch thread that it is time to reevaluate the thread level*"([SAG+09]). Figure 3.3 shows the architecture of this proposed approach.

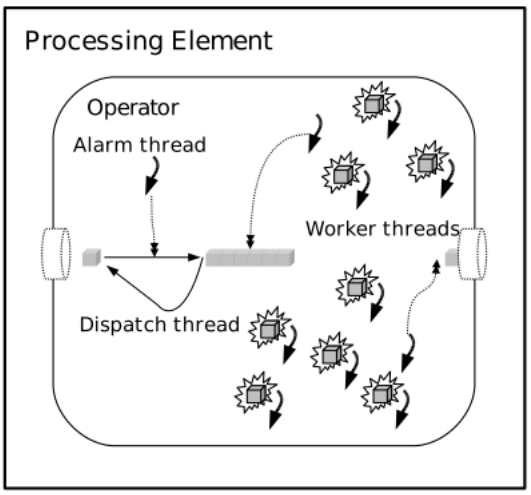


Figure 3.3 – Elastic architecture. Extracted from [SAG+09].

Bringing the term “self-aware" to stream processing applications, Su et al. [SST+15] introduce StreamAware, which is a programming model with adaptive strategies targeting dynamic environments. The aim is to allow the applications to automatically adjust during run-time. This approach defines adaptivity as one of the most important features in parallelism. The adaptive approach to stream processing applications is based on the MAPE-K loop and uses concepts from autonomic computing [KC03].

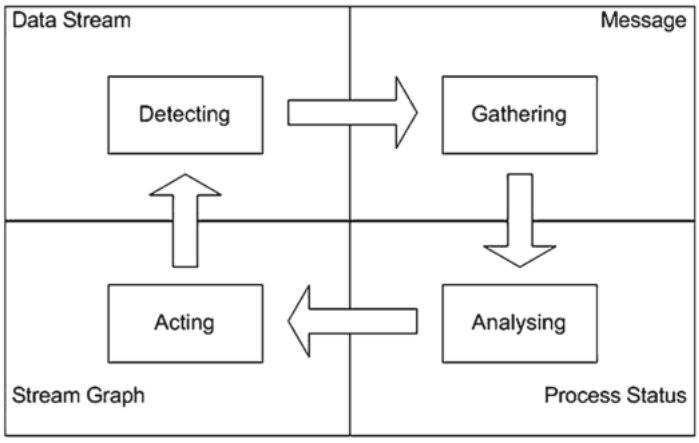


Figure 3.4 – Self-awareness properties. Extracted from [SST+15].

Su’s et al. [SST+15]’s mechanism to support adaptive features in stream processing applications has four parts: detecting, gathering, analyzing, and acting, as shown in Figure 3.4. It adjusts stream parallelism by adding or removing worker threads. The adaptive method is evaluated and validated using the Parsec benchmark suite.

On the other hand, Gedik et al. [GSHW14] use the term elastic auto-parallelization[SHGW12] to locate and parallelize parallel regions. They also address adaptation of parallelism during the execution. The paper highlights the question of the “profitability" of stream parallelism,

by asking “*How many parallel channels provide the best throughput?*" The term “parallel channels" can be understood as a specific aspect related to the degree of parallelism.

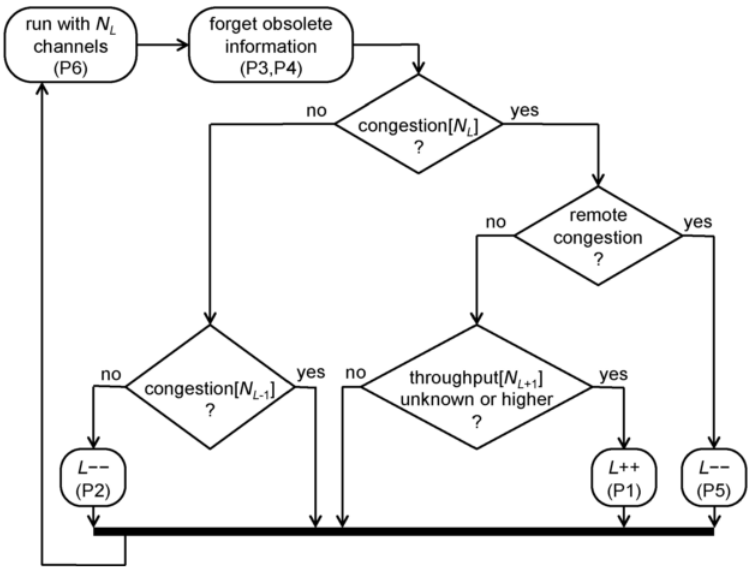


Figure 3.5 – Proposed algorithm [GSHW14].

Moreover, Gedik et al. [GSHW14] argue that the parallelism profitability problem depends on workload changes (variation) and resource availability. They propose an elastic auto-parallelization solution, which adjusts the number of channels to achieve high throughput without wasting resources. It is implemented by defining a threshold and a congestion index in order control the execution regardless of if more parallel channels are required. This approach also monitors the throughput and adapts to increase performance. Figure 3.5 shows a flowchart describing their proposed control algorithm. P1 expands (increases) the number of channels, P2 requests an action to decrease the number of channels, P3 means that a congestion was detected, and P4 indicates changes in the throughput rate. Moreover, P5 is a congestion that does not affect the throughput and in this case the number of channels are reduced. Lastly, P6 adds a set of channels.

Yet another approach to stream processing is provided by Heinze et al. [HPJF14]. This work emphasizes the complexity of determining the right point at which to increase or decrease the degree of parallelism. The authors investigate issues of elasticity in the data stream to meet requirements for auto-scaling (scaling in or out), workload independence, adaptivity, configurability, and computational feasibility. They also explore latency aspects in a distributed system, using metrics with fixed minimum, maximum, and target utilization.

Heinze et al. [HPJF14] classify the approaches for auto-scaling applications in five groups: Threshold-based, time series, reinforcement learning, queuing theory, and control theory. They argue that time series is not feasible for stream processing applications, because it considers historical data and the stream load is unpredictable. Queuing model was also excluded due to its limited adaptivity. The remaining classes were then tested with

stream processing applications. Threshold approaches are characterized by the need for the user set upper and lower bounds with respect to the resource utilization and/or performance. On the other hand, reinforcement learning is based on the system state for taking optimization actions, using a feedback control that monitors the execution and chooses another configuration the next time. Control theory is based in an independent controller that responses fast to input changes based on a feedback loop. The experiments show the proposed techniques outperforming related approaches.

Selva et al. [SMMF15] shows an approach related to adaptation in run-time for streaming languages. The StreamIt language is extended in order to allow the programmer to specify the desired throughput and the runtime system controls for the execution. Moreover, it implemented an application and system monitor to check the throughput and system bottlenecks. This approach is able to adapt the execution based on previous observation, which classifies it as reactive.

Another related work is Gulisano et al. [GJPPM+12] that presents StreamCloud to scale data stream processing applications in cloud environments. It implements parallelism to process queries and distribute the data/tasks among nodes. A major concern of this approach is to handle load balancing through optimized task distribution. Parallelism optimizations are triggered considering the number of active nodes and their loads.

Table 3.1 – Related Work Overview.

Approach	Goal	System	Mechanism	Environment	Applications	External thread
Sensi [STD16]	Performance and Power consumption	NORNIR	Linear regression and monitoring	Multicore	PARSEC	Yes
Matteis [MM16]	Performance and Power consumption	Elasticity on FastFlow	Predictive using future time horizon	Multicore	High-frequency trading	yes
Schneider [SAG+09]	Throughput and CPU utilization	SPADE	Monitoring and applying changes	Multicore	Synthetic and a radioastronomy	Yes
Su [SST+15]	Performance per Watt	StreamAware	Autonomic MAPE K loop	Distributed	PARSEC	Yes
Gedik [GSHW14]	Congestion and Throughput	Algorithm Implementation	SASO accuracy	Distributed	Finance, Twitter, PageRank and Network	Yes
Heinze [HPJF14]	Maximize utilization and control the latency	System prototype	Monitoring and applying changes	Distributed	Financial	Yes
Gulisano [GJPPM+12]	Maximize utilization and throughput	data stream processing engine	Monitoring and applying changes	Distributed	Performance and load balance	Yes
Selva [SMMF15]	Quality-of-Services	Extension of the StreamIt language	Monitoring System and applications	Multicore	Dataflow applications	*Yes*
This approach	Parallelism Abstraction (throughput and latency)	Extension of SPar DSL	Applications and runtime library	Multicore	Lane Detection, Person Recognition and Pbzip	No

Table 3.1 shows an overview of related works. The main aspects considered were the adaptation mechanism and the scenario of the applications evaluated in each study. Each approach has specific goals, implemented strategies, and a target environment. The

applications evaluated vary among the approaches, but we selected only related works that tested stream processing applications.

Our research differs from existing papers because we provide an adaptive degree of parallelism support to the SPar DSL (Section 2.3). We share with Sensi et al. [STD16], Matteis and Mencagli [MM16] the idea of using the FastFlow framework as the runtime library. However, Sensi et al.[STD16], and Matteis and Mencagli [MM16] address energy consumption aspects. The selected related works share ideas in the same application domain, but we focused particularly on stream parallelism abstractions regarding the degree of parallelism.

Moreover, the research problem presented in Gedik et al. [GSHW14] is closely related to this work, since both approaches evaluate an optimized and adaptive degree of parallelism. However, we have a different scenario and target architecture. While Gedik et al. [GSHW14] and [GJPPM+12] address distributed stream systems, our approach targets stream parallelism in shared-memory multicore systems using FastFlow as runtime library and SPar for parallelism abstraction. Consequently, we consider the algorithm implementations in related works to be limited due to their specific scenario and not sufficiently abstracted for application programmers. We aim to go further by implementing the strategy for adaptations during run-time and abstract from users the need to define the degree of parallelism. Also, the related works used an external thread to monitor execution and triggering actions. However, our approach implements adaptation inside the runtime library to reduce complexity and overhead.

In fact, the presently available solutions are complex even for experts in parallel programming and do not focus on user interfaces. Consequently, there is a demand to abstract the need to set degrees of parallelism from end-users and enable them to run their applications transparently without manual often unfeasible and error prone interventions. Additionally, using adaptation during run-time should provide an acceptable performance defined by the user and, at the same time reducing the complexities for programmers.

4. ADAPTIVE DEGREE OF PARALLELISM: DESIGN AND IMPLEMENTATION

Several stream processing applications run in parallel and also with load fluctuations. One way to respond to load fluctuation is by adapting the degree of parallelism during run-time. For instance, when the load increases, performance can be improved by increasing the degree of parallelism. On the other hand, if the load decreases, the degree of parallelism can be reduced by suspending active threads and therefore not wasting computational resources. This study is focused on the term profitability for fission dynamics (a.k.a. adaptive) [HSS+14]. Fission determines the number of replicas (degree of parallelism) for a given parallel region as well as the related parallelism profitability, which was also the research problem in related studies [GSHW14].

SPar, which is the DSL extended in this work, already reduces programming effort for parallel applications by generating parallel code. The parallel code generated by SPar is skeleton-based (*eg.,* farm, pipeline), used for exploiting parallelism in concurrent regions. The replicate attribute (Section 2.3) may be filled with the number of replicas/threads or by setting an environmental variable, the replicate represents the fission [HSS+14] as well as the width of a given parallel region. Currently, the replicate attribute is a static integer value that is used during the entire program's execution, as shown in the example in Section 2.3. However, if the stream processing application presents fluctuations (*e.g.,* performance, environment, or input rates), this static execution can lead to inefficient resources usage (waste) or poor performance[1].

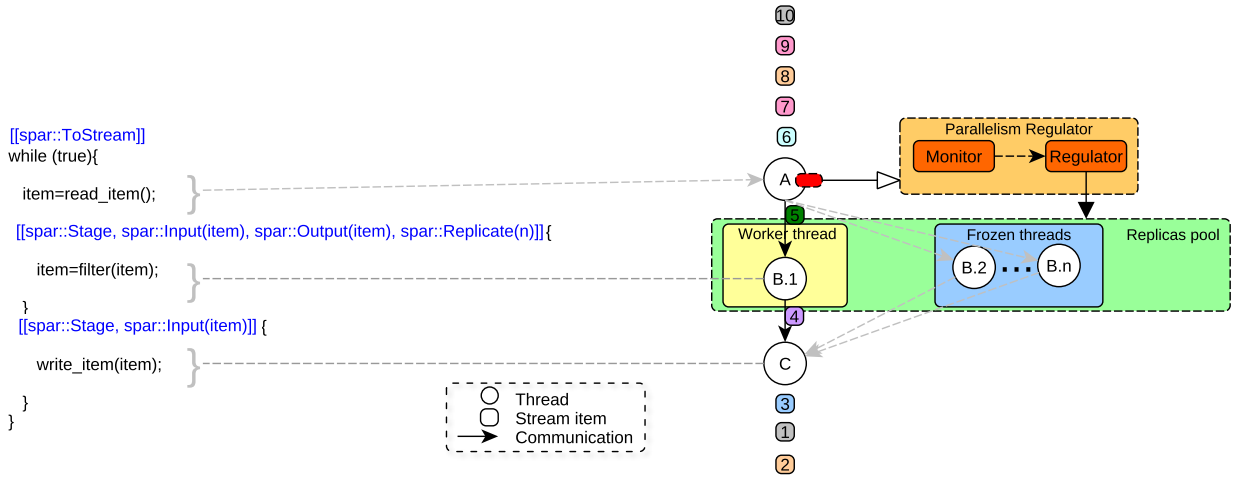


Figure 4.1 – SPar - desired adaptive mechanism.

[1]The term replicate is used to signal the degree of parallelism in SPar, and it may also be viewed as the number of threads/workers in a given parallel region. Consequently, in this study, the terms number of replicas and degree of parallelism have the same meaning and are used interchangeably.

Our desired adaptive mechanism is presented in Figure 4.1. The goal was to implement a strategy that monitors the execution and regulates the degree of parallelism during run-time. We have implemented strategies for an adaptive degree of parallelism according to the requirements and properties, shown respectively in sections 4.1 and 4.2. One strategy is presented in Section 4.4 that monitors the run-time library and its queues for detecting congestion. Other strategy were implemented to test the application's throughput (Section 4.5) and Section 4.6 analyzes the latency of stream items.

It is important to note that the scope of the adaptive strategies addressed in this study is limited to specific aspects. We are not discussing the placement problem [LQF15, HSS+14]. Future work will focus on evaluating the impact of low-level aspects such as the threads placement, CPUs affinity, and core frequency. Furthermore, the adaptive mechanisms are meant to work on replicated (a.k.a fissioned) stages. Also, on-demand was the scheduling strategy used to simulate fine task granularity and sensitive load balancing. The experiments evaluating the implemented strategies were run on a machine with 8 cores (16 Hyper-threads) and 16GB of RAM.

4.1 Requirements for Adaptive Degree of Parallelism

Heinze et al. [HPJF14] proposed a set of requirements for auto-scaling strategies for elastic data stream operators. The requirements for scaling data stream processing applications can be used to design the requirements in our specific case. These requirements are:

- **Workload Independence:** A strategy that adapts the degree of parallelism is supposed to be agnostic from the workload characteristics. We expect a mechanism to be a generic solution that does not require customization for every workload.
- **Adaptivity:** The strategy is expected to adapt continuously on-the-fly according to the workload and the fluctuations of different processing phases.
- **Transparency:** The techniques used should have *configurability* (be easy to configure for users). However, we are working with parallelism abstraction and our goal is a strategy that transparently adapts during the execution. This is required for us to reach our goal, which includes avoiding the need to manually define the degree of parallelism.
- **Computational Feasibility:** The internal algorithm used by an adaptive strategy has to be feasible enough to run and make decisions on-the-fly. Computational complexity must be low enough to quickly respond to load fluctuations without using too many resources, which would significantly affect execution.

- **Low Overhead:** The strategy should also not be too complex, consume too many resources, or cause significant overhead. The overhead is expected to be negligible. Approaches controlling the intrusiveness are considered overhead-aware. When the adaptation process only changes the needed elements (replicas), non-intrusiveness is easily met.

4.2 Desired Properties for Adaptive Degree of Parallelism

Several properties and interests can be included when designing adaptive strategies. In this work, we consider SASO (Stability, Accuracy, Settling time, Overshoot) properties [HDPT04]. Related approaches [GSHW14], [MM16] have also used SASO properties when designing their solutions.

A relevant property for an adaptive system is stability, a system is *stable* if it produces the same configurations every time under given conditions. A system is *accurate* if the “measured output converges (or becomes sufficiently close) to the reference input” [HDPT04]. *accuracy* is used in related studies to search for the degree of parallelism that optimizes performance.

Moreover, a system is expected to present *short settling times* by quickly responding to changes and reaching an optimal state. When the load fluctuates, the response should be rapid and maintain a service level objective. A strategy should also avoid *overshoot* by only using the necessary amount of resources.

4.3 Implemented Design Goals

The previously presented requirements and desired properties for adapting the degree of parallelism resulted in goals designed for our specific scenario, the SPar DSL. Accordingly, the main design goals are:

- **High-level Parallelism Abstractions:** SPar already prevents users from handling low-level aspects when parallelizing applications. Our goal is to support parallelism abstractions when defining the degree of parallelism. Consequently, meeting the properties of adaptivity and transparency.
- **Flexibility:** Flexibility support the implemented functionalities in a wide range of applications. It is not meet by implementing a solution only for a specific application. Thus, we aim to support an adaptive degree of parallelism for any application parallelized with SPar. This design goal meets the property of workload independence.

- **Performance:** The need for performance is twofold: Internally (in the mechanism) and externally (running applications). The property of computational feasibility is relevant for an adaptive execution to achieve the expected performance. Moreover, the internal aspects of its implementation are expected to have a low overhead. Consequently, we avoid using an external thread for the implemented strategies, because using external threads requires additional implementations that can increase complexity and thus resource consumption.

4.4 Adapting the Number of Replicas Based on Communication Queues

The requirements for an adaptive degree of parallelism were used to design and implement adaptivity in SPar. We studied SPar's runtime to find techniques to change the number of replicas during run-time. In the stream processing context, an effective mechanism is required to change a program's execution on-the-fly without the need to recompile or rerun. The first strategy was implemented using low-level calls to the runtime library to change the status of the replicas (active, suspended). This first strategy uses a strategy that considers metrics from the queues for adapting the degree of parallelism. Also, the strategy is reactive because it monitors the queues and then acts, which uses the concept of feedback closed-loop and controller from control theory.

4.4.1 Implementation

Figure 2.6 in Section 2.3.2 presented the the runtime library's queues in a farm skeleton. Figure 4.2 shows how we extended SPar's runtime to implement our adaptive strategy. The first stage that hosts the parallelism regulator. The regulator gets the status of each queue by monitoring and collecting the push lost values[2]. It is assumed that when a program is executing with a high number of push losses, its performance would be degraded by the congestion. A high number of push losses occurs when the replica's queues are full, meaning that they are processing fewer elements than needed, resulting in congestion. When a high number of push losses was detected, the the regulator triggered an action to increase the number of replicas. On the other hand, if the number of push losses decreased, the number of active replicas was reduced by suspending active replicas. This was in the form of iteration, every time the regulator was run, it decided whether or not to change the number of replicas.

[2]A push lost is characterized when the task scheduler fails to add a new item to a replica's queue because the queue is full.

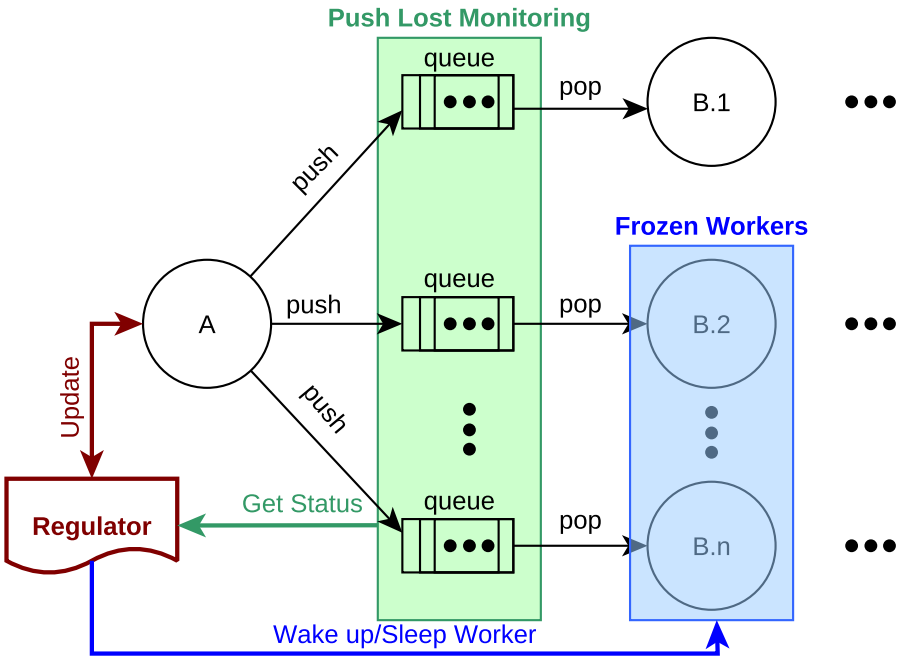


Figure 4.2 – Queues monitoring mechanism in SPar's runtime.

The regulator is implemented in the first SPar stage to avoid the need to restart the application when the number of replicas is changed. Algorithm 4.1 presents the regulator implemented in the first stage, which runs within SPar's task scheduler. The regulator has a maximum value for the number of replicas, which is defined according to the machine's CPU availability. The execution is started by the task scheduler by sending tasks to the active replicas. After a certain amount of time, the regulator collects monitoring data from the queues and decides to increase, decrease, or maintain the same number of replicas. This is performed in lines 4 and 7 of the regulator algorithm by adding (waking up) or removing (suspending) threads until it reaches the minimum or the maximum number of replicas. The task scheduler continues sending tasks and the regulator continues to monitor the push losses until the execution ends (if it does end).

Algorithm 4.1 Parallelism Regulator based on Queues

```
1: procedure REGULATOR( )
2:     while true do
3:         Sleep(timeInterval)                    ▷ Wait until the next iteration
4:         if pushLost > upperThreshold then      ▷ If true there is congestion
5:             WakeUpReplica()                    ▷ Add active replicas
6:         else if pushLost < lowerThreshold then
7:             SuspendReplica()                   ▷ Suspend active replicas
```

4.4.2 Configuration Considerations

Despite the strong motivations for an adaptive degree of parallelism, an adaptive execution tends to have additional complexities and challenges. For instance, when the execution starts all replicas need to be suspended, and only the replicas required should be awakened and then receive tasks. Consequently, the task scheduler and load balancer must be aware of the number of active replicas.

The minimum and maximum number of replicas that a given parallel stage can use must also be configured. If parallelism is required, we assume that the minimum degree is 2 (threads/replicas) and the maximum is defined according to the machine number of available processing cores. The parallelism regulator determines the number of physical cores using the UPL (Utility Performance Library) [3].

Another aspect of the configuration is the scaling factor (SF), which is how many threads/replicas are added or removed when adjusting the degree of parallelism. In the literature, the most commonly used scaling factor value is 1 (threads/replicas). Our adaptive strategy is tested with scaling factors 1 and 2. We have chosen to test with 2 also to quickly scale up or down to meet the short settling time property.

A challenge related to an adaptive degree of parallelism is *when* to change the execution. Based on the related bibliography, it is possible to consider this aspect as an unsolved problem. In related works, there is no consolidated solution to handle it, and each approach addresses it according to its specific scenario. The most common approach is time-driven, where it periodically runs in an arbitrary time interval (TI), which can be viewed as a sampling time. In [GSHW14, MM16, SHGW12, STD16] used time intervals ranging from 0.1 to 5 seconds. We consider 0.1 seconds a too low an interval and 5 seconds too high. Our adaptive strategy is tested with 0.5 and 1 second. This configuration was chosen with the idea of avoiding excessive iterations, while at the same time maintaining a correct level of sensitivity to potential workload fluctuations. Moreover, the impact of the different time intervals is tested in the adaptive strategy.

Because this first implementation adapts the number of replicas considering the runtime library's queues, it uses a simple technique based on a threshold for deciding whether or not an adaptation is needed. This configuration has to be sensitive enough to detect application fluctuations. We are using a minimum and maximum number of push losses as a threshold, ranging between 150 and 200 (thousand) occurrences. In this implementation, the user has to define the tolerated number of push losses as well as when it characterizes a congestion. Push losses may in fact vary from a machine's capacity to process (tic speed) and an application's behavior. As a consequence, setting threshold con-

[3] https://github.com/dalvangriebler/upl

figurations can be challenging even for experts in parallel programming, which is a usability limitation of this implementation.

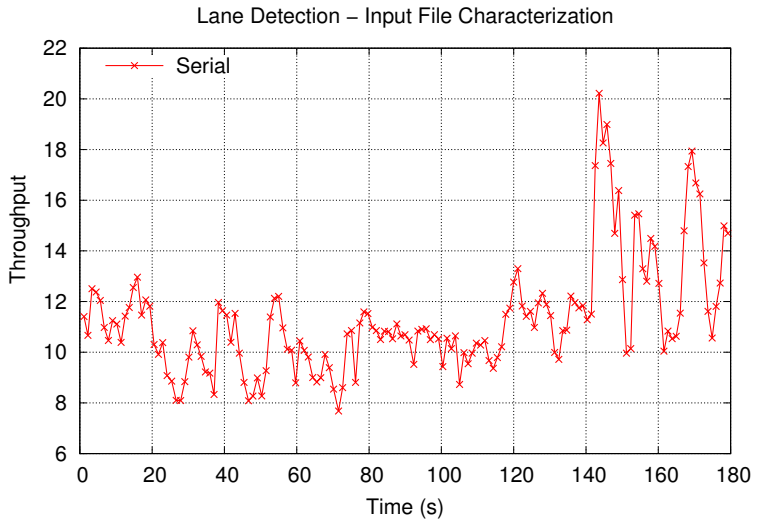


Figure 4.3 – Throughput Characterization.

The strategy was tested by adding the adaptive part to a parallel version of the Lane Detection video application (described in Section 6.2). We tested the strategy's behavior under different configurations in a video stream processing application to understand the implications of configurations and their impact on the application's performance. The aforementioned configurations used inside of our strategy were the time interval of 0.5 and 1 second and scaling factors of 1 and 2 replicas.

The throughput of a program is defined by how many tasks are processed in a given time interval. In our primary tests, we used a video file as an input to simulate a typical execution of a video streaming application. Figure 4.3 shows the throughput of a serial execution with the tested input file. This characterizes the load and reveals the variation during execution caused by the different time it took to process each task. The throughput oscillated between 7 and 20 frames per second. In the parallel version, the variation tends to be higher because multiple threads process at the same time.

An example of how this implemented strategy works, is shown in Figure 4.4. As mentioned previously, the queues' monitoring strategy adapts the degree of parallelism considering the number of push losses. The defined threshold of push losses is used by the adaptive strategy to decide if an adaptation is needed (in terms of the degree of parallelism). In this example, the execution started running with 8 replicas and kept increasing the degree of parallelism until the sixth second. The degree of parallelism was maintained the same while the push losses index was within the threshold, examples were between the eleventh and the twentieth first second. Also, the degree of parallelism was decreased when the push losses index went below the threshold, as occurred in the second 22.

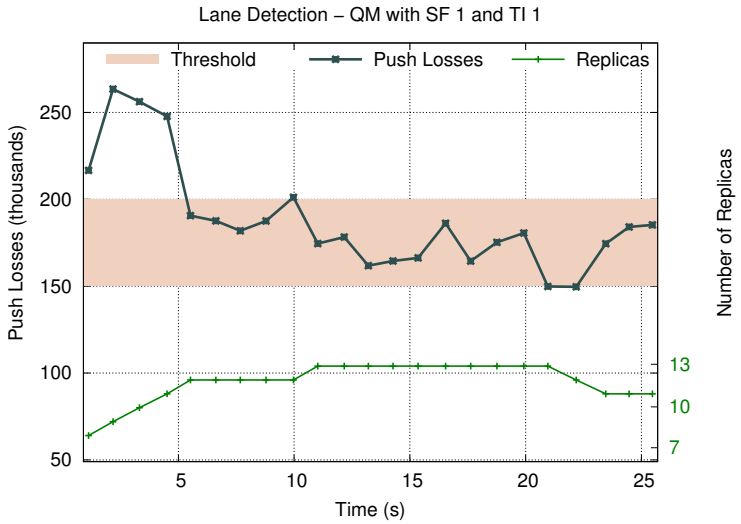


Figure 4.4 – Lane Detection - Queues Monitoring.

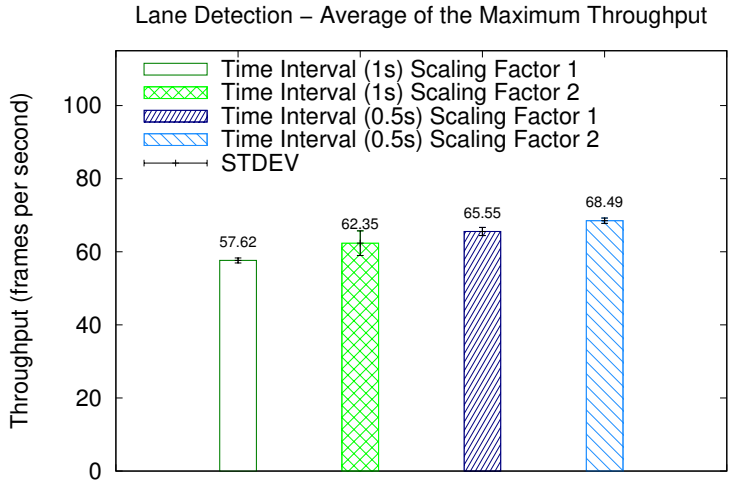


Figure 4.5 – Time Interval and Scaling Factor - Throughput.

Figure 4.5 shows the average throughput from 10 executions using different configurations. The configuration that achieved the best throughput was the one with a scaling factor of 2 (it went up and down faster) and a time interval of 0.5s, the configuration was more sensitive to application's fluctuation. The configuration with time interval 0.5s and scaling factor 1 also performed well. The configurations with a time interval of 1s also achieved high throughput rates. Commonly, in stream processing applications, fast processing does not actually mean that users will have a better experience. In these results, we are not comparing which configuration yields the best performance. In fact, we are evaluating the adaptive strategy and how the configurations impact performance. We also have shown the standard deviation of the experiments. Considering this variation, the throughput is similar in the different configurations.

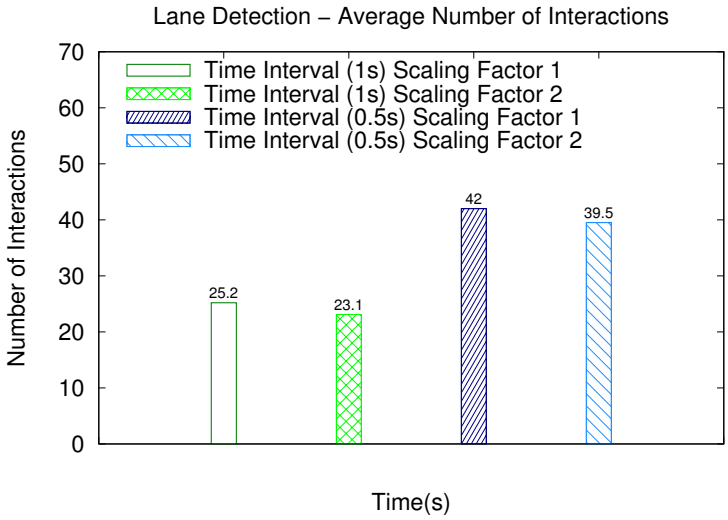


Figure 4.6 – Time Interval and Scaling Factor - Number of Iterations.

The different configurations tested presented similar throughputs (Figure 4.5). However, in addition to showing performance results, in order to understand and discuss how this adaptive strategy and its mechanism works, we also monitored several aspects (*e.g.,* resources utilization, iterations, parallelism adaptations) while the application was running. One relevant part of the adaptive strategy is an *iteration*. We consider *iteration* when the parallelism regulator runs, which occurs after the sleeping period shown in line 3 of Listing4.1.This means that the parallelism regulator will verify if an adaptation of the degree of parallelism is needed. As this routine is implemented inside the SPar's task scheduler, it can result in an additional overhead since the distribution of tasks can be delayed. Figure 4.6 shows the number of iterations for each configuration. The results reveal the impact of shorter time intervals in the number of iterations, which was expected with 0.5 as the time interval. Additionally, the results in Figure 4.5 shown that iterations do not degrade performance, the two configurations with more iterations achieved the best throughput rates.

Figure 4.7 shows how many times that the number of replicas was changed. The configuration with the best throughput has on average less than 7 adaptations. However, it is not possible to infer that fewer adaptations result in better performance, since the lowest throughput was not in the configuration with more adaptations. For instance, the negligible impact of adaptation is highlighted when comparing the two configurations that both used 0.5s as a time interval, because they presented similar performance. However, the configuration with a scaling factor 2, had only half of the adaptations compared to scaling factor 1. In fact, the configuration with time interval 0.5s and scaling factor 2 achieved the best throughput, because it increased the degree of parallelism more quickly.

Another relevant aspect monitored during the execution was CPU usage. Figure 4.8 shows the percent of the average load collected every second in each configuration scenario. These results demonstrate the impact of the number of replicas in CPU usage.

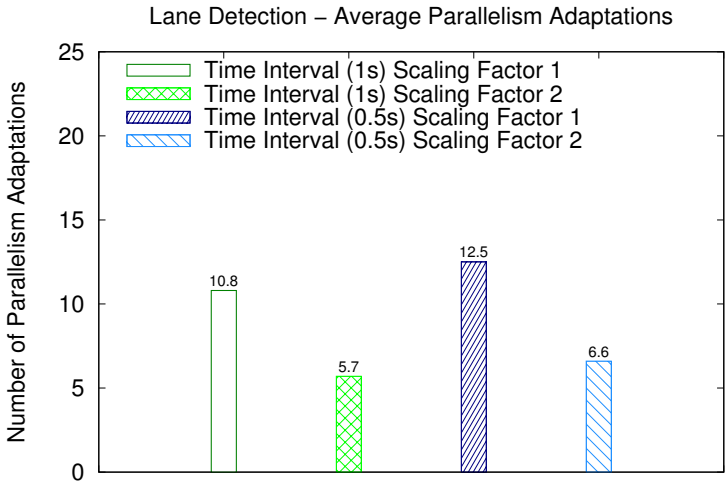


Figure 4.7 – Time Interval and Scaling Factor - Number of Parallelism Adaptations.

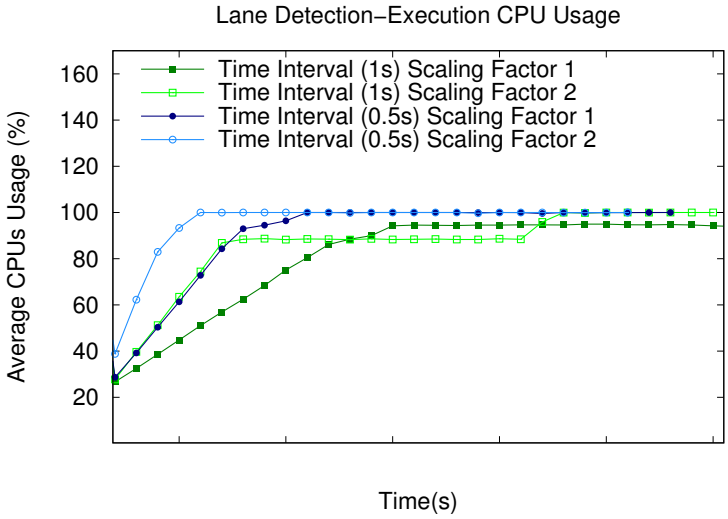


Figure 4.8 – Time Interval and Scaling Factor - CPUs Utilization.

This was done to test efficiency and whether adding more replicas these threads would actually process tasks. Thus, more replicas will certainly increase CPU usage, which with a balanced number of replicas can improve the performance and/or with too many replicas may cause resource contention and degrade the performance.as The execution with the best throughput was that which most quickly increased CPU usage.

In addition to the CPU load, the total memory consumption was also monitored and is shown in Figure 4.9. Despite the previously mentioned contrasts among the configurations, memory usage is very similar. Because each replica is a thread and needs some additional memory, and therefore higher variation can occur.

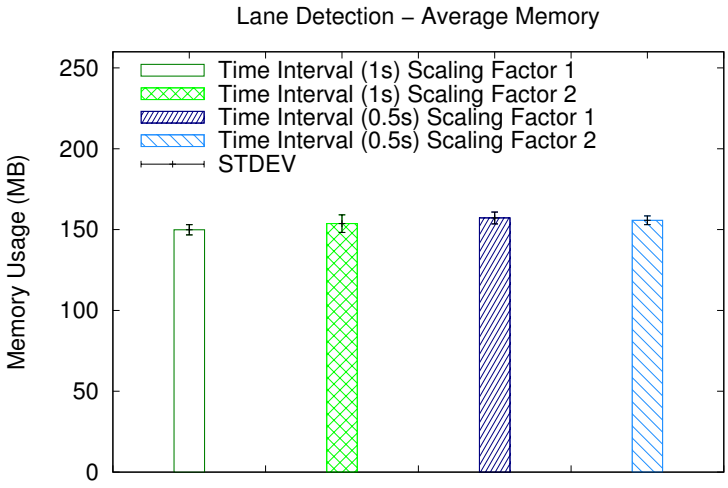


Figure 4.9 – Time interval and scaling Factor - Memory Usage.

4.5 Adapting the Number of Replicas Based on the Throughput

The previous section presented the first strategy for adapting the degree of parallelism in SPar. The limitations that were highlighted called for better ways to adjust the number of replicas. The primary requirement (from Section 4.1) that caused this demand was transparency. In the strategy based on the runtime library's queue, sensitive parameters still needed to be defined by the user, such as the threshold of push losses. However, this is not feasible for application programmers since thresholds vary among applications and require a deep understanding of workload characteristics [LBMAL12].

Defining a performance goal is presumably easier for application programmers than defining a low-level parameter of the runtime system. Therefore, we studied ways to handle the configuration challenges and abstract them from programmers to meet the requirement of a transparent degree of parallelism. Hence, we implemented a strategy that adapts the degree of parallelism based on the application's throughput.

4.5.1 Implementation

This strategy optimizes the execution based on a closed-loop system from control theory [HDPT04]. It monitors the execution considering a performance metric to attempt to optimize the execution in the next iteration. Figure 4.10 shows the design of the implemented strategy with throughput as the performance goal. Even with many iterations, the adaptations occur on-the-fly without the need to rerun the application.

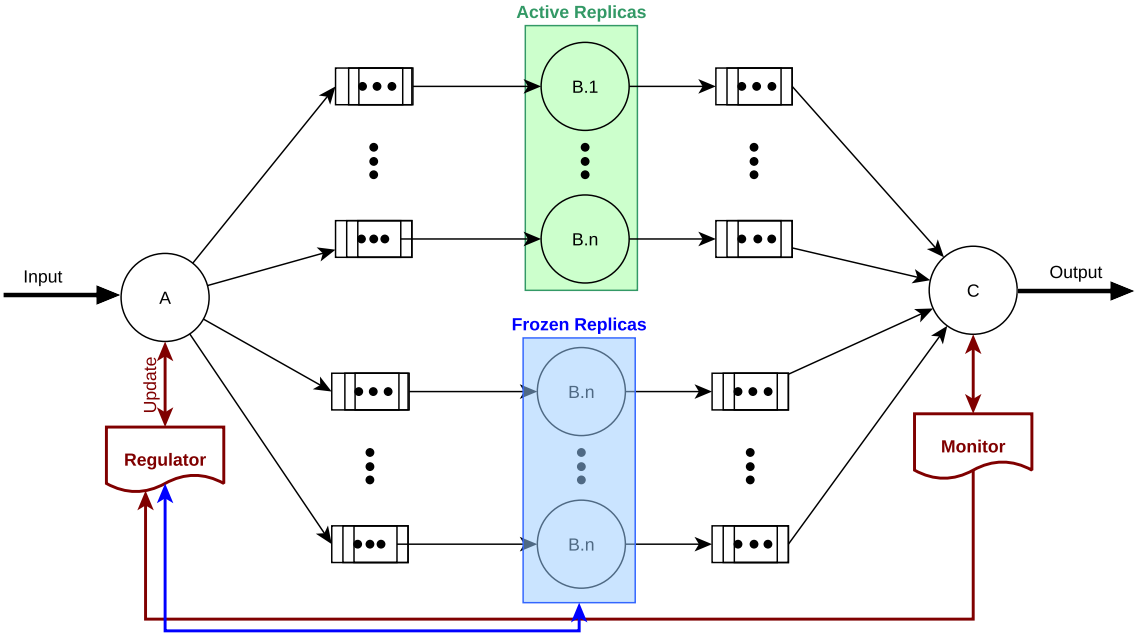


Figure 4.10 – Regulator and Monitor in SPar's runtime.

We implemented this strategy through an algorithm that changes the number of replicas and continuously monitors the program's execution, as represented in Algorithm 4.2. As shown in lines 4 through 7, the decision whether to change the degree of parallelism is based on the target (expected) and measured (actual) throughput, which is the main difference compared to the adaptation approach based on queues.

Algorithm 4.2 Parallelism Regulator based on Throughput

```
1: procedure REGULATOR( )
2:     while true do
3:         Sleep(timeInterval)                          ▷ Wait until the next iteration
4:         if Throughput < Target then                  ▷ If throughput is below the target
5:             WakeUpReplica()                          ▷ Add active replicas
6:         else if Throughput > Target + Threshold then
7:             SuspendReplica()                         ▷ Suspend active replicas
```

The throughput value (tasks/second) is calculated in the last stage. Listing 4.3 shows a representation of the monitor that calculates the number of tasks processed in the current iteration. The number of tasks processed in the current iteration is calculated by subtracting the previous total number of tasks from the current total number of tasks. In each iteration, the throughput is the result of dividing the number of processed tasks by the execution time.

The regulator runs in the first stage along with the SPar's task scheduler. The pool of replicas is managed the same as the previous strategy. The difference lies in the last stage, which gathers the tasks and also measures the throughput. The throughput rates are stored and accessed by the regulator in the first stage. This flow provides information to the

Algorithm 4.3 Throughput Monitor

```
1: procedure MONITOR( )
2:     while true do
3:         Sleep(timeInterval)                          ▷ Wait until the next iteration
4:         numOfProcTasks = currentTask − lastProcTask
5:         Throughput = numOfProcTasks/timeInterval
```

regulator to decide if an adaptation is required. This strategy only requires the definition of a target throughput.

The number of replicas is changed by freezing active threads if the throughput is higher than the target. Furthermore, because throughput tends to oscillate, we aim to avoid unnecessary adaptations. There is no consensus in previous works as to when to reduce the degree of parallelism. Each approach is specific and implements the strategy considering its scenario. In our strategy, we added a percentage value, acting as a threshold, which is a value that can be tolerated when the actual throughput is higher, but close to the target. 20% was the most suitable value for video applications during our empirical tests. The threshold is used in line 6 of the Algorithm 4.2. However, it is important to note that this threshold may not be as feasible with a significantly different workload. Also, the input file used to simulate a real-world scenario presents oscillations in the time taken to process the frames. The actual throughput is reduced because some frames require more time to be processed and cause load fluctuations.

An essential assessment of the feasibility of this strategy was performed using the same application and input used in Section 4.4. We simulated a potential condition called dynamic throughput that occurs when the target performance is changed during the execution.This requires the number of replicas in a parallel stage to be adapted to meet the new performance goal. In this case, the number of replicas is changed considering the number of tasks processed. In this video application, a task is a video frame. Also, it is relevant to emphasize that this strategy allows one to test different conditions, which is why this strategy presented more results than the previous one. Furthermore, the main goal of these experiments was to evaluate the effectiveness of the adaptive strategy. Thus, performance will be evaluated and discussed in the following chapters.

Figure 4.11 shows a simulated scenario where the target throughput is dynamic during the execution, starting with an arbitrary value of 40 tasks as the target throughput. In this simulation, when the execution time reaches 8 seconds, the target throughput is changed to 18. At the 16 seconds, the performance goal is increased to 50. When the execution reaches 13 seconds, the target throughput is changed to 70. Also, in this experiment the strategy that adapts the parallelism is configured with a scaling factor 1 and a time interval of 0.5 seconds. When this time interval occurs, an iteration is triggered to verify if the degree of parallelism must be adapted. Also, the execution starts with replicate 2, which is the degree of parallelism. This value was used to have a minimum degree of parallelism

and adapt continuously to pursue the target throughput by adding more replicas when the throughput was below the target.

The input file used to simulate a real-world scenario presented oscillations in the time taken to process the frames. Some frames required more time to be processed, resulting in load fluctuations because it reduced the actual throughput. For instance, throughput fluctuations can be seen in Figure 4.3 and in Figure 4.11 between seconds 4 and 6 where with the same number of replicas the throughput goes below the target throughput. Hence, at 7 seconds, more replicas were activated in the poll aiming to increase the throughput.

Another relevant aspect is observed after 13 seconds, when the target throughput was higher (70). The combination of the load fluctuations from the input and number of replicas resulted in a high variation of the actual throughput. This strategy reacts to fluctuations in the execution and predicting such load fluctuations is a challenge because streams items are processed in real time under heavy and uncertain workload trends [HPJF14].

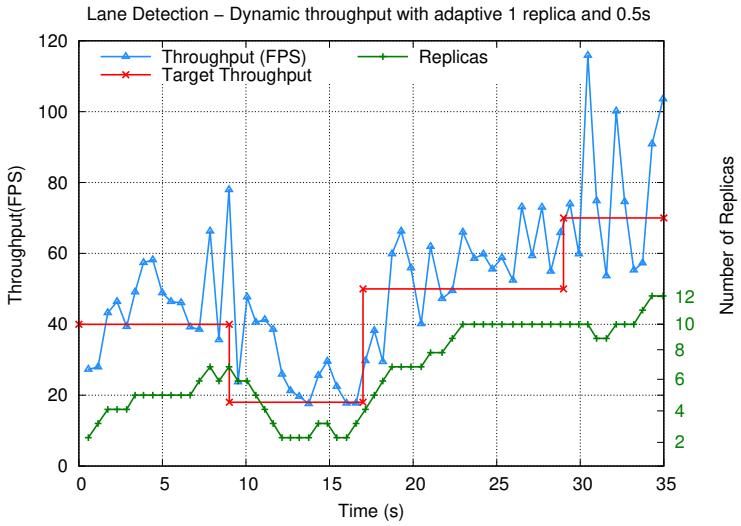


Figure 4.11 – Dynamic throughput with scaling factor 1 and time interval 0.5s .

While Figure 4.11 had a scaling factor of 1, in Figure 4.12 the scaling factor was 2, which means that two replicas can be added or removed on each adaptation. Consequently, there are more throughput variations. Scaling factor 2 reacts faster to changes, which could results in a short settling time at the price of less stability. Results show the need for a balance between the requirements and desired properties.

In Figure 4.13, we present the same dynamic throughput simulation, the adaptive degree of parallelism, and input video file. However, the time interval between each iteration is 1 second in this experiment. A higher time interval means fewer iterations. The measured performance tends to be stable since it is an average of a longer time period, but the settling time tends to be increased. Comparing the different configurations, we conclude that the configurations performed as predicted. Using a time interval of 1s, the actual throughput is closer to the target, but it takes a more time to reach the target throughput. For example,

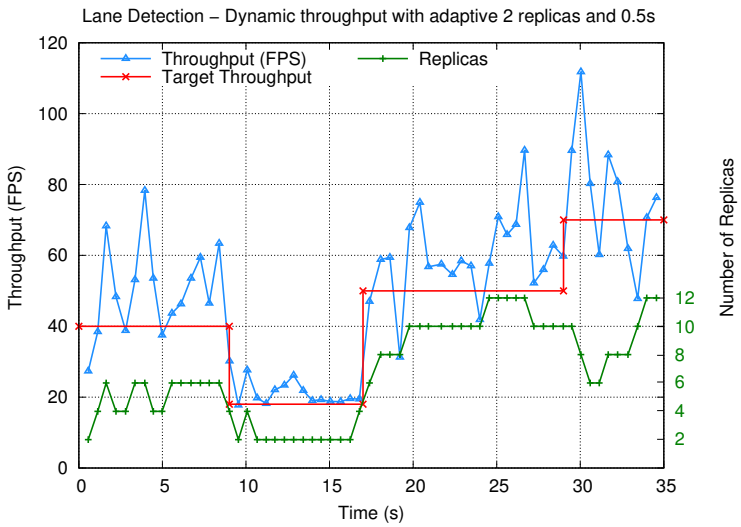


Figure 4.12 – Dynamic throughput with scaling factor 2 and time interval 0.5s .

at 16 seconds of the execution shown in Figure 4.11, the target throughput was updated to 50. This configuration with the time interval of 0.5s took 2.7 seconds to achieve the target throughput. On the other hand, the configuration with the time interval of 1s took 5.8 seconds to reach the target throughput. In short, these distinct settling times show how significantly configurations affect applications' performance.

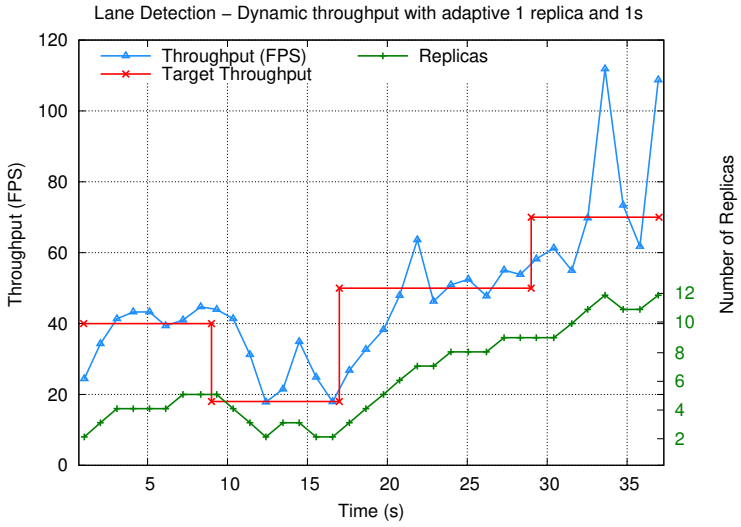


Figure 4.13 – Dynamic throughput with scaling factor 1 and time interval 1s .

Comparing the Figures 4.14 and 4.13, we can see the same trends from scaling factor 1. The time interval 1 presented fewer fluctuations since the average performance is from a longer period. Also, the settling time was higher with 1s as a time interval. The configuration with time interval 0.5s was more sensitive, and seems to be more suitable for video streaming applications by responding fast to changes.

In Figure 4.15, we present an experiment with a static target throughput of 60 frames per second and a scaling factor of 1, using the same input video from previous

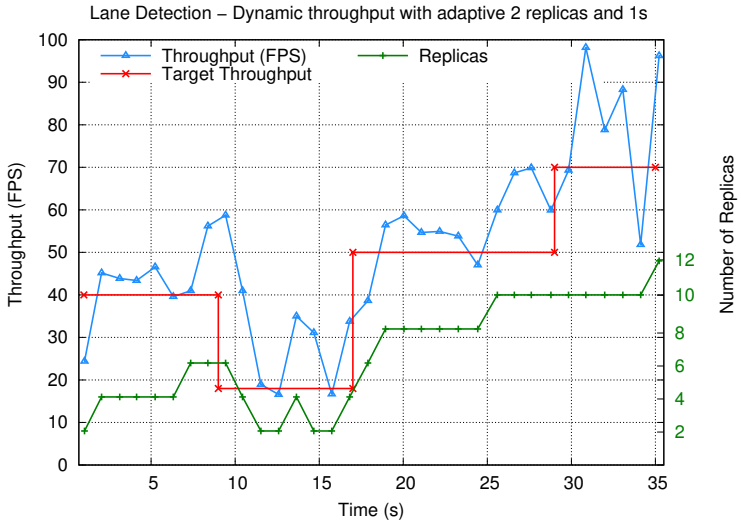


Figure 4.14 – Dynamic throughput with scaling factor 2 and time interval 1s .

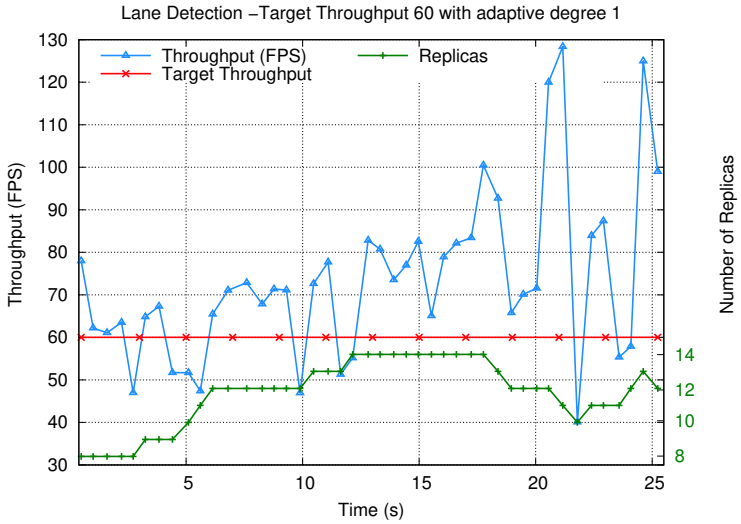


Figure 4.15 – Target throughput 60 with scaling factor 1 and time interval 0.5s.

experiments. In the tested machine,the number of replicas used varies from 8 to 14. As the number of replicas impacts the actual throughput, we decided to start the execution using half of the total available cores[4] (with Hyper-threads). We started the execution with more replicas because the target throughput was higher. In previous experiments, it is possible to notice a need to increase the number of replicas right after the execution starts.

Figure 4.16 shows an experiment again using 60 as a target throughput with a scaling factor of 2 replicas. A comparison between Figures 4.15 and 4.16 showed no significant contrasts in the throughput caused by the differences in the scaling factor. The load from the input file caused most of the throughput fluctuations. Both configurations reacted to changes and pursued the target throughput. It is noteworthy that in some specific instances, even using the maximum number of replicas, it was not possible to achieve the target throughput.

[4]Using the UPL library

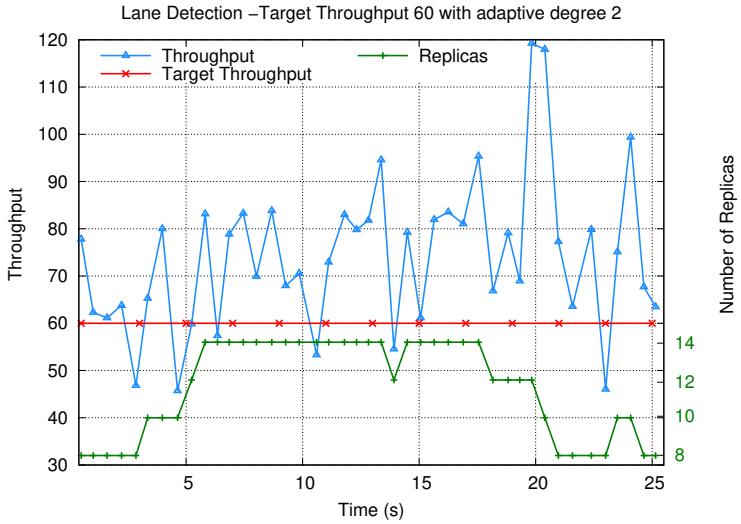


Figure 4.16 – Target throughput 60 with scaling factor 2 and time interval 0.5s.

However, it was not caused by the adaptive mechanism, but is a consequence of the machine's limited processing capability.

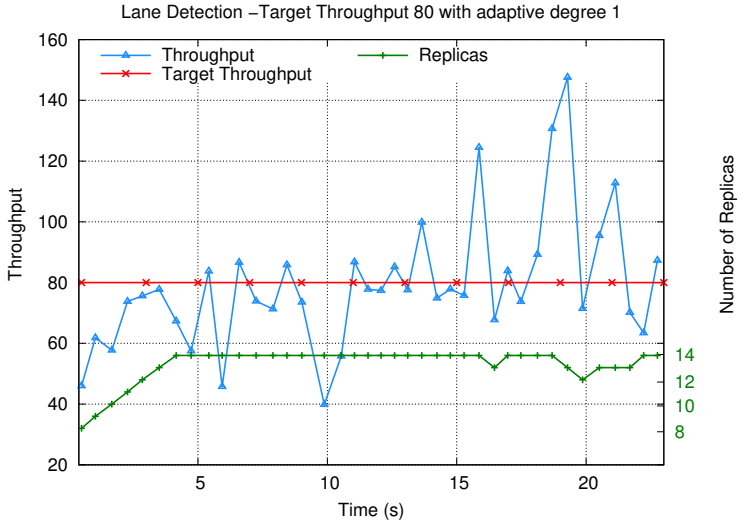


Figure 4.17 – Target throughput 80 with scaling factor 1 and time interval 0.5s.

Also, an experiment with a higher target throughput, 80 tasks, is shown in Figures 4.17 and 4.18. The machine could not always meet the target throughput with this high-performance demand, even when using the maximum replicas. This is due to the hardware and scalability limits of the tested application. Both configurations started with 8 replicas and the number was increased to improve the throughput. The maximum number of replicas (14) was the most commonly used configuration. We observed that the scaling factor of 2 achieved a shorter settling time (2 seconds) and scaling factor of 1 only used all replicas at the fourth second.

In this Section, we have seen experiments with different configurations and target throughput. It was shown that the throughput is affected by the configurations. In addition,

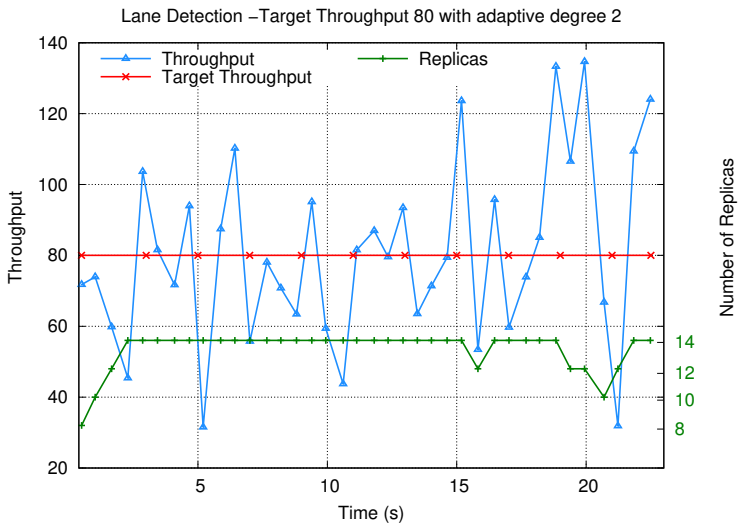


Figure 4.18 – Target throughput 80 with scaling factor 2 and time interval 0.5s.

we aim to evaluate how the final throughput (average of an entire execution) can be affected by configurations, which will be presented in the next section.

4.5.2 Comparison of Configurations

We saw in previous experiments that using different target throughput and configurations. Applications and their inputs vary a great deal. Therefore, it is a challenge to have a configuration suitable for all scenarios. The possible configurations of the adaptive strategy are the time interval, between each monitoring, and the scaling factor, concerning the number of replicas added or removed from the pool. However, it is not intuitive for application programmers to define these parameters. As a consequence, we need to test and characterize the impact on performance caused by configurations in order to abstract the complexities involved.

Figure 4.19 presents the throughput performance of the four configurations related to the adaptive strategy. In this case, a high target throughput was set to achieve the maximum performance. It is worth nothing that the throughput presented is an average of the entire execution. When an execution ended, the throughput was calculated considering the number of tasks processed and the total time used. The result of each configuration is therefore an average of 10 runs as well as error bars showing the standard deviation, which is the factor of variation in the throughput results.

The highest average throughput was achieved by the configuration with a time interval of 0.5 and scaling factor of 2, which was the only execution with a visible but minimal standard deviation. The short settling time and rapid scaling of this configuration, which was observed in previous results, explains why it had the best performance. In this experiment,

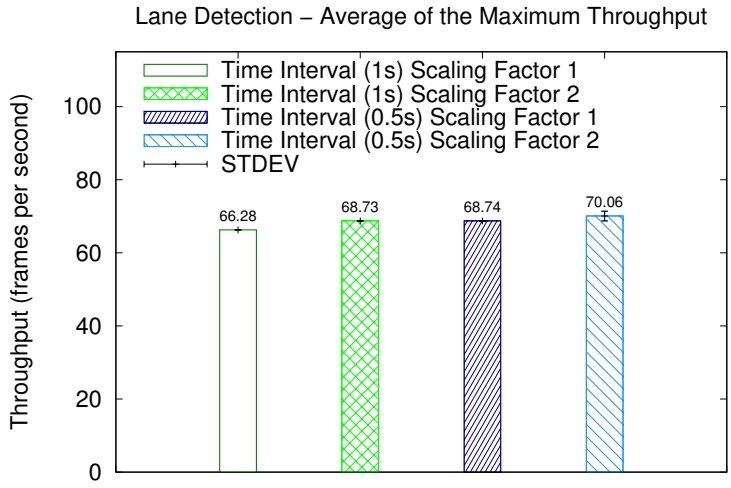


Figure 4.19 – Maximum throughput of each configuration.

we did not consider efficiency or resource consumption. The goal was to simulate a scenario demanding maximum performance. In Chapter 6, we compare the performance and resource consumption of an adaptive degree of parallelism to static executions.

4.6 Adapting the Number of Replicas Based on the Latency

In the previous section we discussed the strategy for adapting the number of replicas according to a target throughput. Latency is another relevant aspect for stream processing applications [MM16, Gri16]. In this study, we consider latency as the time taken to process a stream item. It is quite relevant because stream items can be sensitive to latency, thus quick response time may be a performance requirement for several stream processing applications. For instance, a lane detection application may need to process video frames and detect the lanes under a specific latency constraint. Because this application could be used by an autonomous car in order to detect the lanes and use the results of the processed feed to keep the car on the road. Proper responses to the detected lanes can not be determined if the latency is too high.

When latency is a constraint, the number of replicas can be regulated for managing the time to process the stream items. In the previous section, we discussed aspects for adapting the parallelism without considering the impact of the number of replicas in the latency. There can be a correlation between throughput and latency. Achieving a high throughput using many replicas tends to increase the latency. On the other hand, using too few replicas decreases the throughput and latency. Consequently, there is a trade-off between throughput and latency, which tends to require a balance between the two performance goals.

The challenge is that a high throughput is commonly pursued, and at the same time low latency may also be necessary. Throughput and latency are shown on the left side of Figure 4.20, which adds the monitored latency in the configuration shown in Figure 4.18. Here when the throughput increased, the latency decreased and vice- versa. These aspects highlight that the workload from the input video was the primary source of load fluctuations. The throughput decreased because fewer frames were processed in a given instant. The frames demanded more resources and delayed the output, which also resulted in higher latency.

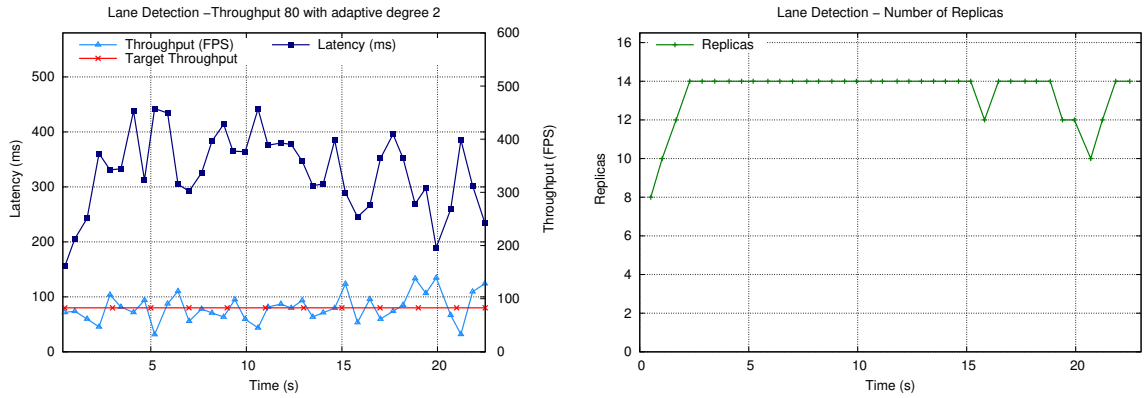


Figure 4.20 – Latency and throughput of stream items (Left) and number of replicas (Right).

The throughput tends to increase when more replicas are added. In the right side of Figure 4.20 shows the number of replicas used during the execution. By comparing the latency and number of replicas it is possible to identify a relation between them. For instance, from the beginning of execution until the third second, the number of replicas is increased to improve the throughput. However, this also significantly increased latency. This same event also occured in the time interval between seconds 20 and 23. The relation between the number of replicas and latency can also be seen after the 18 seconds when the number of replicas decreased from 14 to 10 and the latency dropped from 400 to 200 milliseconds.

The effect of the number of replicas was also tested in this study with a static number of replicas, a fixed degree of parallelism that a given program runs during its entire execution. Figure 4.21 shows the latency of executions during the first 35 seconds of the lane detection application, using the same input of the previous experiments. Moreover, because the number of replicas changes the performance, some executions ended before others according to the number of replicas. Ignoring the total execution time, the best latencies can be seen in executions ranging from 2 to 6 replicas, because 6 is a degree that had one thread per physical core (8) considering the 2 additional/required threads (one for the runtime library scheduler and other to gather tasks). This comparison is relevant to demonstrate the optimal number of replicas that can be used when latency is a constraint.

Complementing the representation of the execution shown in Figure 4.21, Figure 4.22 shows the average latency concerning each execution with different numbers of repli-

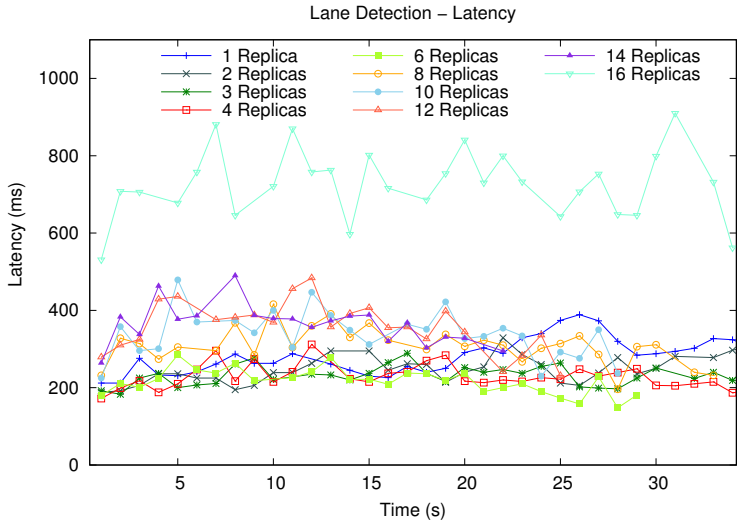


Figure 4.21 – Impact in latency caused by the number of replicas.

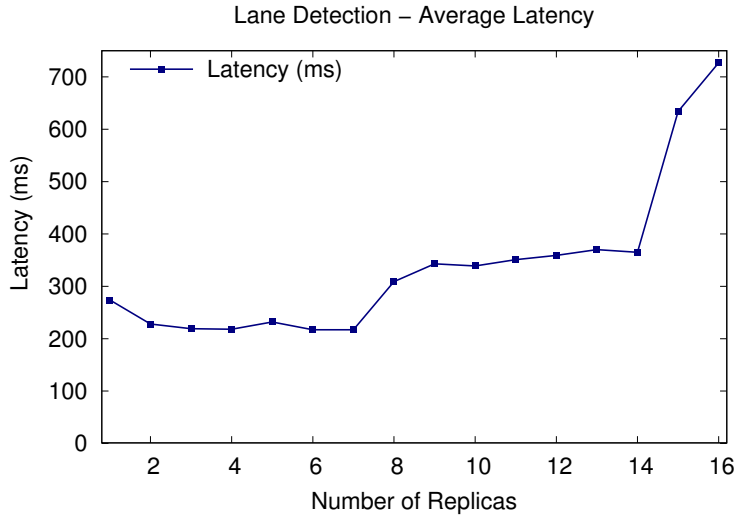


Figure 4.22 – Average latency of executions with replicas.

cas. In this machine, the latency started to increase when more than 7 replicas were used. A significant increase in latency was observed with 15 and 16 replicas, because of the limited number of Hyper-threads resulted in additional concurrency between the parallel region and the other sequential stages (scheduler and gather). The additional overhead, using more than 14 replicas, changed the number used as the maximum degree of parallelism in the adaptive strategies to the total number of cores (with Hyper-threads) reduced by 2. For instance, in a machine with 24 cores, the maximum number of workers in a given parallel region would be 22.

4.6.1 Implementation

In this section, we have discussed how the number of replicas affects the latency of stream items. Measuring and understanding the latency of stream processing applications is a complicated and time-consuming task. We aimed to abstract the parallelism challenges for latency sensitive applications. We proposed a new strategy for transparently managing the latency by monitoring the latency of stream items and managing execution by adjusting the number of replicas. Figure 4.10 shows the strategy we used for adapting the number of replicas, considering the monitored throughput. The same logic was also used for adapting the number of replicas in our latency strategy. However, instead of monitoring the throughput, the latency of stream items was monitored and accessible to the regulator in the first stage.

The mechanism for changing the number of replicas is similar to the previous strategy. When latency is a constraint, the use of the Hyper-threads will be avoided due to the additional overhead caused. Moreover, the number of replicas can vary from 2 to the total number of physical cores less 1. In Figure 4.22 (where 8 is the number of physical cores), the best latencies were between 2 and 7 replicas. For example, in order to optimize the latency, if a machine has 12 physical and 24 Hyper-threads, the maximum number of replicas will be 11.

Algorithm 4.4 shows the implementation for controlling the latency by adjusting the number of replicas. Unlike a throughput strategy that increases the number of replicas for improving performance, this strategy reduces the number of replicas when the latency is higher than the target. However, using only a few replicas causes a lower throughput. Therefore, it requires a balance to be struck between throughput and latency constraints. The main part of this algorithm is shown between line 4 and 7 in Algorithm 4.4, where the latency is periodically verified and the number of replicas is modified when necessary.

Algorithm 4.4 Parallelism Regulator

```
1: procedure REGULATOR( )
2:     while true do
3:         Sleep(timeInterval)                    ▷ Wait until the next iteration
4:         if Latency > Constraint then           ▷ Latency is too high
5:             SuspendReplica()
6:         else if Latency < Constraint then
7:             WakeUpReplica()
```

Moreover, it is relevant to note that latency is monitored for each task. This approach still acts like a feedback loop that has a *monitor()* in the last stage, calculating the average latency of the tasks processed in the current iteration. The calculated latency is

saved so that the *regulator()* in the first stage can decide if the number of replicas must be adapted.

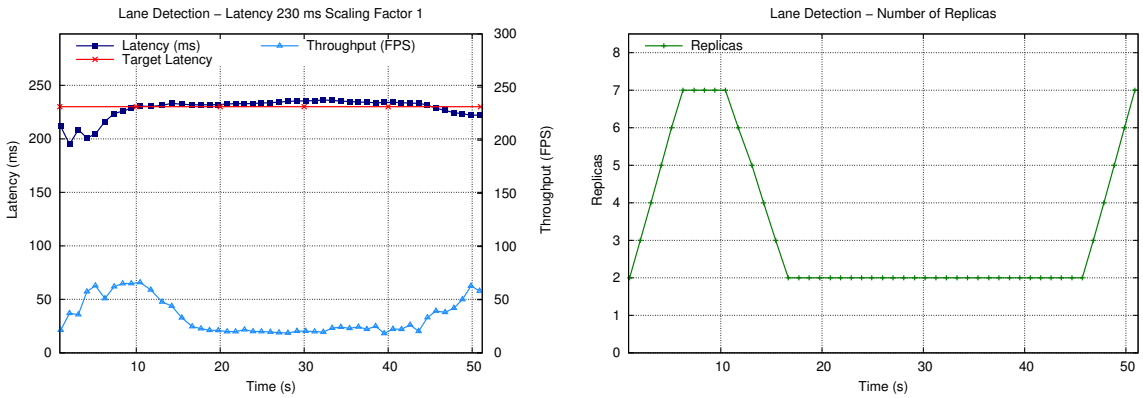


Figure 4.23 – Latency Constraint of 230 ms (Left) and Replicas used (Right).

Executing a stream processing application with the maximum throughput without controlling the latency is not suitable for several of these applications. At the same time, using a minimum number of replicas to reduce latency tends to result in poor throughput performance. Therefore, the latency strategy for adapting the number of replicas tries to improve the performance by increasing the number of replicas when the latency is below the target. We tested our strategy for latency with the same application and input used by the previous strategies. The left side of Figure 4.23 shows the throughput and latency, and the right side presents the number of replicas used during the execution.

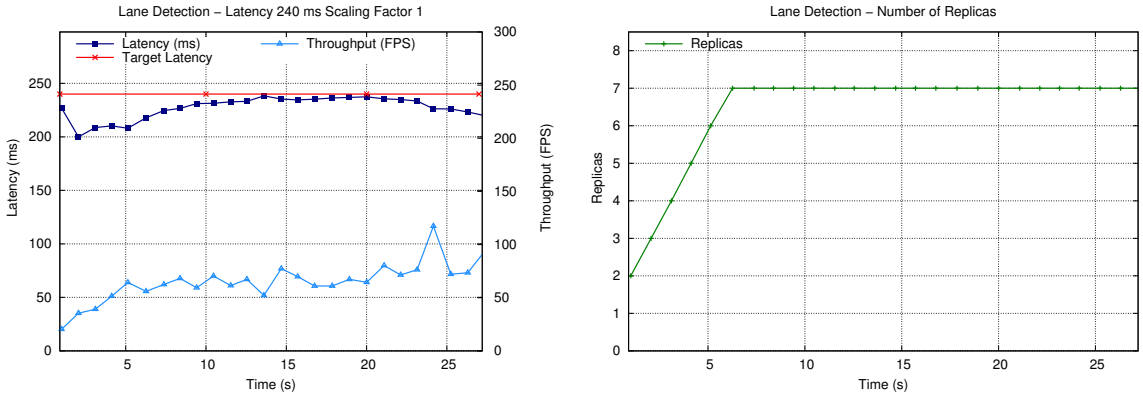


Figure 4.24 – Latency Constraint of 240 ms (Left) and Replicas used (Right).

In Figure 4.23, the rigorous latency goal of 230 milliseconds was not always achievable due to the machine processing limits, which was also observed with static parallelism in Figure 4.21. The relevant aspect of this strategy is that it reduced the number of replicas when the latency increased. For instance, between 15 and 45 seconds , the target latency was not achieved, but our strategy did reduce the number of replicas to the minimum, trying to reduce latency. After 45 seconds, when the latency decreased, the strategy increased the number of replicas, thereby, improving the throughput.

Figure 4.24 shows an experiment that tolerates higher latencies (240 milliseconds). Consequently, it was possible to increase the number of replicas and throughput. In this case, the target latency was met using 7 replicas, which resulted in a significantly higher throughput rate. Comparing Figures 4.23 and 4.24 it is possible to see significant contrasts in terms of latency, throughput, and number of replicas. However, the target latency difference was only 10 milliseconds, and the strategy had to make different decisions regarding the degree of parallelism. These results demonstrate the effectiveness of the strategy to meet the target latency as well as how sensitive the execution is to 10 milliseconds in the definition of the target latency. Also, these results show that the latency strategy is quite accurate and emphasize how susceptible the execution is to an incorrect parameter.

4.7 Adapting the Number of Replicas Without User-Defined Parameters

Previously, we presented an approach for adapting the number of replicas regarding a target throughput (Sections 4.5). We demonstrated that is possible to adapt the parallelism and optimize the performance during run-time. The strategies presented used a target performance to compare with the actual performance and therefore adapt the number of replicas. As a consequence, the programmer has to know and define the target performance. Yet, this can be a usability challenge since application programmers often have no performance expertise. Thus, the previous approaches met the requirement for adapting the degree of parallelism, but lead to a demand for a potentially complex definition of a target performance.

It is a challenge for an adaptive strategy to adapt the degree of parallelism without a target performance. It is expected to be a trade-off between the complexities for defining parameters and a complete transparent execution. Defining parameters according to specific demand increases the flexibility at the price of additional complexities. On the other hand, running transparently increases the abstraction level by reducing complexities, but tends to result in less flexibility as well as poorer performance.

Algorithm 4.5 Parallelism Regulator without user-defined parameters

```
1: procedure REGULATOR( )
2:     while true do
3:         Sleep(timeInterval)                          ▷ Wait until the next iteration
4:         if Throughput < OneOfPrevious then    ▷ If throughput is lower than a previous
5:             WakeUpReplica()                                  ▷ Add active replicas
6:         else if Throughput > AverageOfPrevious then
7:             SuspendReplica()                               ▷ Suspend active replicas
```

In Algorithm 4.5 is presented as a new strategy for a regulator algorithm. It adapts the number of replicas without demanding a target performance. Instead, it considers other

information during the execution. It can use the maximum number of replicas, as an abstraction. However, this can result in an overshoot. This implemented strategy decides whether to adapt the number of replicas by comparing the performance in the previous three executions. For instance, this strategy continuously monitors the throughput and, if the throughput is lower than one of the previous executions, it increases the number of replicas. On the other hand, if the current throughput is higher than the average throughput of the previous executions, it reduces the number of replicas. Also, this strategy uses a throughput monitor in the last stage that calculates the actual throughput, shown in Algorithm 4.3. Consequently, this implementation is a new strategy derived from the throughput strategy, shown in Section 4.5.

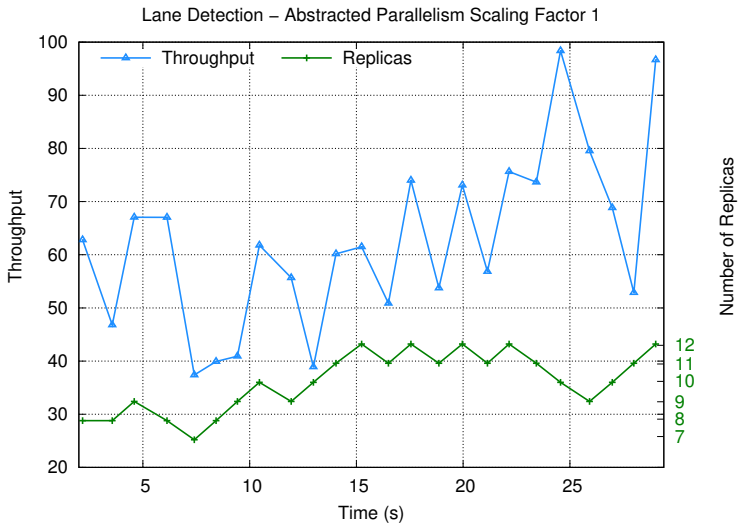


Figure 4.25 – Adaptive Strategy without user-defined parameters - Scaling Factor 1.

Figure 4.25 presents the results of the throughput and the number of replicas used during execution using the strategy without user-defined parameters. Our experiment had a scaling factor of 1 replica and monitoring time interval of 1 second. As observed in the previous experiments, the throughout oscillates during the execution, which is mainly caused by workload trend. It is notable that the number of replicas is not always stable since it varies during the execution. We decided to rely on the previous 3 executions to be sensitive enough to the workload fluctuations. In addition to that, when the execution started, there was no prior information to analyze. In this specific experiment, the number of replicas varied too much because it adapted based on the actual application's throughput. However, under more stable throughput conditions it is likely that there would be fewer changes in the number of replicas, but this depends on the application and its input characteristics.

In Figure 4.25 it is again possible to identify throughput fluctuations. Moreover, this strategy for adapting the degree of parallelism successfully adjusted the execution. The number of replicas varied from 7 to 12 and the throughput oscillated from 40 to almost a 100 frames per second. The final performance of this strategy will be evaluated in Chapter 6.

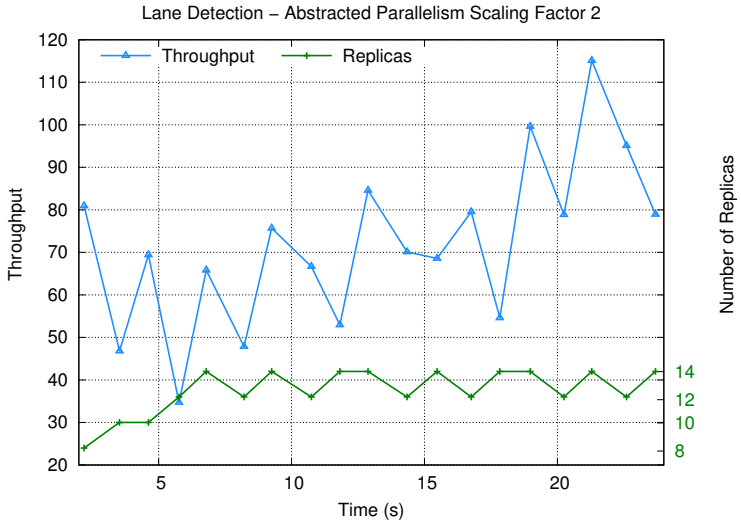


Figure 4.26 – Adaptive strategy without user-defined parameters - Scaling Factor 2.

In Figure 4.26, a similar experiment to Figure 4.25 is presented. Yet here we did not have a scaling factor of 2. In this case, the execution faster scaled up or down. However, under the throughput oscillations, it is not possible to identify settling patterns in the number of replicas. This number was changed several times between 12 and 14 replicas. This strategy tries to use a number of replicas according to the workload characteristics without user-defined parameters. However, it may result in performance losses, which is the trade-off for fully abstracting the target performance.

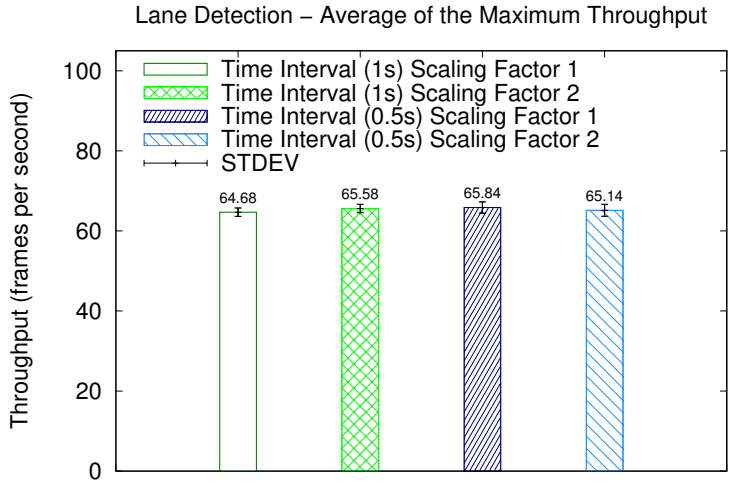


Figure 4.27 – Maximum throughput of each configuration.

We also evaluated the four configurations used for testing the adaptive degree of parallelism. Figure 4.27 shows the average throughput of 10 runs for each configurations. We noticed small contrasts as well as a low standard deviation. The average throughput was similar in the distinct configurations. The performance and resources consumption of this strategy will be evaluated in the Chapter 6.

4.8 Remarks

In this chapter, we discussed the motivations and requirements for an adaptive degree of parallelism and the implemented strategies were characterized using the Lane Detection application. Also, we tested the strategies for adapting the number of replicas with different configurations, considering two time intervals and scaling factors of 1 and 2.

The first strategy considering the runtime library's queues, highlighted that it is possible to adjust the program during run-time and adapt the number of replicas on-the-fly. This strategy demonstrated the alternative of using different configurations. Despite the demand for configuring low-level parameters, this strategy has met the requirements and desired properties. However, we demonstrated that is not intuitive nor is it suitable for application programmers to configure low-level parameters.

Therefore, we implemented an improved strategy that pursued a given performance by adapting the number of replicas based on throughput. In Section 4.5, we presented a solution that required a target throughput as a configuration and optimized the execution. The experiments revealed that the target performance might vary during execution as well as the demand for resources. The time interval and scaling factor impacted settling time. Moreover, balancing between the desired properties to achieve low settling times, stability, and avoiding overshooting is also required. In Section 4.6, we demonstrated the impact on latency caused by the number of replicas. Here we also proposed a strategy based on the latency of stream items.

Furthermore, we implemented a strategy that adapts the number of replicas without the need to set a target performance. Because many application programmers do not have performance expertise, this strategy tries to maximize the throughput without a user-defined parameter (Section 4.7). The advantage of more abstraction comes at the price of less flexibility.

Considering the requirements for an adaptive degree of parallelism (Section 4.1), the strategies also met the goals for adaptive and transparent executions. The overhead and computational feasibility as well as the workload independence will be demonstrated in Chapter 6. The overhead and feasibility are tested by monitoring the consumption of resources, and workload independence is shown by using different applications and their inputs.

The desired SASO properties were also met by the strategies. The execution of the strategies is stabilized by explicitly comparing the target and actual performance to adapt the degree of parallelism. The strategies are also stabilized by using time intervals between iterations and the short settling times are pursued by using short time intervals and rapid scaling configurations. Also, overshoot is avoided by decreasing the degree of parallelism when suitable.

We designed transformation rules for SPar based on the implemented strategies. These rules are for generating parallel code with support for an adaptive number of replicas. We intend to offer the option for users to define how to adapt the number of replicas. The more suitable approaches are based on target throughput and latency as well as the strategy without the user-defined parameter. Our idea is that a compilation flag will allow the user to choose between adaptation based on the latency or throughput.

The following chapters will present the transformation rules for supporting an adaptive degree of parallelism in SPar in addition to an assessment of the performance of the adaptive strategies compared to the static case under different real-world applications.

5. ADAPTIVE NUMBER OF REPLICAS IN SPAR

In this chapter, we introduce how the adaptive degree of parallelism in the previous chapters can be integrated into SPar. It already generates parallel code with a static number of replicas. Our goal is to design transformation rules for generating custom code that support our proposed adaptive strategies.

5.1 Source-to-Source Transformation Rules

Transformation rules are used within SPar to perform parallel code generation. These rules are related to the SPar runtime shown in Section 2.3.2 and according to the chosen parallel patterns. As described in Section 2.3, SPar transforms an annotation sequence into parallel code by composing parallel patterns. This can be represented using functional semantics.

5.1.1 SPar Existing Rules

According to [GF17, Gri16, GDTF17], a farm parallel pattern can be expressed using a functional semantics as follows. ***Farm()*** accepts one to three arguments (*farm*(*B*), *farm*(*A*, *B*), *farm*(*A*, *B*, *C*)). The arguments represent the farm task scheduler (known as "emitter", *A*) that sends input tasks to the workers, the farm worker (*B*) that computes the tasks, and the farm collector (*C*) that gathers results from the workers. The emitter and the collector are optional. When the emitter and collector are not present, default scheduling and gathering policies are implemented in the farm. Each one of the three elements only accepts a code block/sequence of commands as an argument.

```
[[spar::ToStream]]{
      f1();
   [[spar::Stage, spar::Replicate(N)]]{   ⇒   farm(A(f1),B(f2));
      f2();
   }
}
```

SPar Rule 1.

SPar already has transformation rules to represent the code generation. Rule 1 represents a code generation from a SPar annotation to a farm parallel pattern. The *ToStream* code region is the first stage with the `f1` (a sequence of commands), which is assigned to

the task scheduler (emitter) (*A*(...)). Furthermore, `f2` (a sequence of commands) is the second stage, which is replicated (*B*(...)). The `Replicate` attribute in the `Stage` annotation (f2) defines the parallelism with a given degree *N*.

Rule 2 has one more `Stage` annotation, which is another sequence of commands represented by `f3`. It is still a transformation from a SPar's sentence to the farm parallel pattern. However, `f3` is assigned to *C* that gathers the tasks, `f2` to *B*, and `f1` to *A*.

```
[[spar::ToStream]]{
    f1();
  [[spar::Stage,
  spar::Replicate(N)]]{
    f2();                          ⇒  farm(A(f1), B(f2), C(f3));
  }
  [[spar::Stage]]{
    f3();
  }
}
```

SPar Rule 2.

5.1.2 Adaptive Rules

In addition to the existing rules, we designed new rules that cover the adaptive degree of parallelism for the SPar DSL. Adaptive Rule 1 is similar to Rule 1, but *A* also runs the adaptive strategy. The adaptive part is included by using the regulator method. *RegulatorQ*() is a new method of class *A* from the queues strategy (shown in Listing 4.1). This regulator returns the replica IDs to send to the next task, and also monitors the runtime queues to decide if the parallelism degree needs to be adapted.

The goal of Adaptive Rule 1 is that the `f1()` code block is the same as in Rule 1. In this case, without *C* is difficult to collect the actual performance during the execution because each worker replica outputs its own result independently, which is the reason this rule is related to the queues (two-stage) strategy.

```
farm(A(f1),B(f2))
        ⇓
farm(A(f1).RegulatorQ(), B(f2));
```

Adaptive Rule 1.

Adaptive Rule 2 was designed with one more stage than Adaptive Rule 1, to gather the tasks from the replicas. *A* and *C* have the code implementing the adaptive number of replicas. This rule is designed for *Farm*(*A*, *B*, *C*). There is a method inside *A* that has

the code from the Listing 4.2, represented by *RegulatorT*(). The regulator and the monitor of Adaptive Rule 2 are called *RegulatorT*() and *MonitorT*() because they are based on the throughput strategy. *RegulatorT*() returns to *A* the ID of the replica to send the next task, and the regulator periodically accesses the data collected by *MonitorT*() and decides whether the number of replicas should be adapted. In *C*, *MonitorT*() implements the code shown in Listing 4.3.

```
farm(A(f1), B(f2), C(f3));
                ⇓
farm(A(f1).RegulatorT(), B(f2), C(f3).MonitorT());
```

Adaptive Rule 2.

Adaptive Rule 3 has the same logic of Adaptive Rule 2. The difference is that Adaptive Rule 3 uses another regulator and monitor which implement the adaptive strategy based on latency. The *RegulatorL*() returns the ID of the replica to send to the next task, while the *MonitorL*() periodically measures and stores the latency accessible to the regulator. The interactions between the monitor and regulator achieve the feedback loop properties for adapting the degree of parallelism.

```
farm(A(f1), B(f2), C(f3));
                ⇓
farm(A(f1).RegulatorL(), B(f2), C(f3).MonitorL());
```

Adaptive Rule 3.

We designed these rules to enable the strategies for adaptive degree of parallelism in SPar. The experiments with the strategy shown in the previous Chapter were manually coded, after generating the default SPar parallel code. However, one goal of this study was to generate a custom parallel code using SPar that runs with an adaptive number of replicas. It is possible to use these rules to implement the adaptive part into SPar's code generation inside of its compiler infrastructure. In the next Chapter, we will evaluate the performance of the adaptive executions by comparing them to the static case.

5.2 Flags for Adaptive Number of Replicas

Compilation flags are a way of enabling the user to exploit a supported capability. The previously presented strategy and transformations rules should be available for SPar users through flags in order for them to customize their parameters. Adaptive Rule 2 and 3 can generate the code required, and the user can customize according to the following flags:

- **spar_adaptive-throughput [*target*] [*scaling_factor*]** : This flag accepts the target throughput in tasks/second as well as the scaling factor as parameters, which aims to provide customization to user. The number of replicas will be regulated to meet the target throughput. Furthermore, if the parameters have not been defined, this approach will exploit the strategy without user-defined parameters. The number of replicas will be regulated transparently, aiming to maximize performance without overshooting.
- **spar_adaptive-latency [*constraint*] [*scaling_factor*]**: This configuration adapts the number of replicas by latency, it accepts the latency constraint and scaling factor as parameters. If the parameters are not defined, the strategy transparently monitors latency and controls the number of replicas. This approach tries to use the maximum number of replicas to maximize performance while meeting the latency constraint.

Moreover, when a farm only has two stages, and the adaptive flags are defined, the compiler and runtime can be configured to use the strategy based on queues. In this case, the degree of parallelism is adaptive, although it will not be possible to meet a target performance.

6. RESULTS

In this chapter, we introduce a set of experiments aimed to evaluate the strategies and their mechanisms for an adaptive degree of parallelism. We tested the strategies under different supported configurations and also compared these results to executions with a static degree of parallelism. Section 6.1 describes the experimental methodology and we then evaluate the performance and consumption of resources using real-world applications.

6.1 Experimental Methodology

This Section presents the methodology used for conducting the experiments. We present the applications used to test the adaptive strategies as well as the machine used in the test environment. Furthermore, we describe the performance metrics for evaluating the strategies.

6.1.1 Test Applications

The strategies were evaluated by implementing them in existing parallel applications. In fact, real world applicability was the key criterion for selecting applications. In addition, applications were selected with different application characteristics and QoS requirements. For instance, some applications chosen perform simple processing and outputting results under a regular workload trend, while other applications can perform more complex processing under an irregular (variant) workload trend. We aim to validate our strategy, which was created with the goal of making any application parallelizable using SPar and exploiting the strategies for an adaptive number of replicas. The tested applications are:

- **Lane Detection:** is a video application for detecting lane lines, targeting self-driving cars.
- **Person Recognition:** is a video application for detecting and identifying people. The recognizer part uses a database of images to verify if a detected person is known.
- **Bzip2:** is a data compression application. We used a parallel version of Pbzip2 from POSIX-Threads adapting it to SPar.

6.1.2 Test Environment

We used the same machine to test the performance of different tested applications. The tests were run on a multicore machine with 2 Sockets Intel(R) Xeon(R) CPU 2.40GHz (8 cores-16 threads), which has a memory of 16GB - DDR3 1066 GHz. Moreover, the operating system used was Ubuntu Server 16.04. Also, this environment was dedicated for this experiments, which means that no other workload was run at the same time.

6.1.3 Performance Evaluation

The characteristics of the adaptive strategies considering the desired properties were presented in Chapter 4. Here we evaluate the adaptive strategies by comparing their maximum performance to hand-coded static parallel applications. Therefore, we ran a set of experiments to evaluate the effectiveness of the adaptive strategies. The following performance aspects were evaluated:

- **Throughput:** is calculated from the number of processed items and time. Here we present the final throughput, which derived by taking the total number of processed elements in a given execution divided by the time.
- **CPU Consumption:** is related to the utilization of computing power. This value is presented as a percentage from 0 to 100. It is calculated considering the load from each of the CPUs.
- **Memory Usage:** is collected at the end of a given execution, which an amount of memory space used for the specific execution.
- **Runtime library:** several aspects can be monitored from the runtime library. In this work, we collected statistics from the pop losses. In Section 4.4 we used the push losses, which are a different metric. A pop lost (shown in Section 2.3.2) is collected from the replica queues and a count of how many times this occurred during a given execution. A pop loss is characterized when a replica (worker thread) tries to get a new stream item to process from its queue and it fails, meaning the task scheduler is unable to send tasks fast enough to all replicas. We monitored this aspect to evaluate the adaptive strategy, because in this scenario the task scheduler is also in charge of regulating the number of replicas. This additional demand can overload the task scheduler.

Each execution was repeated 10 times and the results presented are an arithmetic means of these. Moreover, the results are presented in terms of throughput as well as

their respective standard deviation. We also address parallelism aspects at a higher and abstracted level. For instance, when our mechanism adds a new replica, the operating system will do its best to place the thread in available cores.

6.2 Lane Detection

Lane Detection is an application for self-driving vehicles that recognizes and writes linear patterns [GHDF17] to a video file. The application iterations are shown in Figure 6.1. The *Capture()* function generates a sequence of images, every image is an actual frame and implements computer vision functions [GHDF17]. The function *Segment()* processes the images bottom area, which is where the lane tends to appear. Moreover, the function *Canny()* applies a filter to detect the edges and then the *HoughT()* filter detects the strait lines and the number of elements in *HoughP()*. The fifth step is the *Bitwise()* function that marks the lines and the *Writer()* function writes the output image to a file.

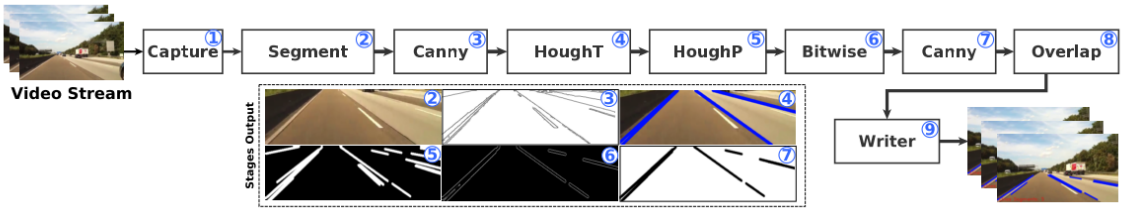


Figure 6.1 – Lane Detection - Workflow. Extracted from [GHDF17].

A real-world example of a Lane Detection output is shown in Figure 6.2. In Chapter 4 we used Lane Detection to present our adaptive strategies. In this approach, we implemented the strategies for an adaptive number of replicas (presented in Chapter 4) to a parallel version of the Lane Detection application. Moreover, we used the ordering implemented in SPar [GHDF18] to maintain the order of the video frames.

Figure 4.3 in Chapter 4 presented the throughput of input 1 that is used in this comparison as well. Additionally, we used another input file called input 2 with more variations, characterized as a serial execution in Figure 6.3.

6.2.1 Performance of Adaptive Strategies

We ran a modified version of Lane Detection that received the code modifications from the adaptive strategies in its first and last stage. Figures 4.19 and 4.27 in Chapter 4 show the throughput performance of configurations of the adaptive strategies using input 1.

Figure 6.2 – Lane Detection - Processed Image.

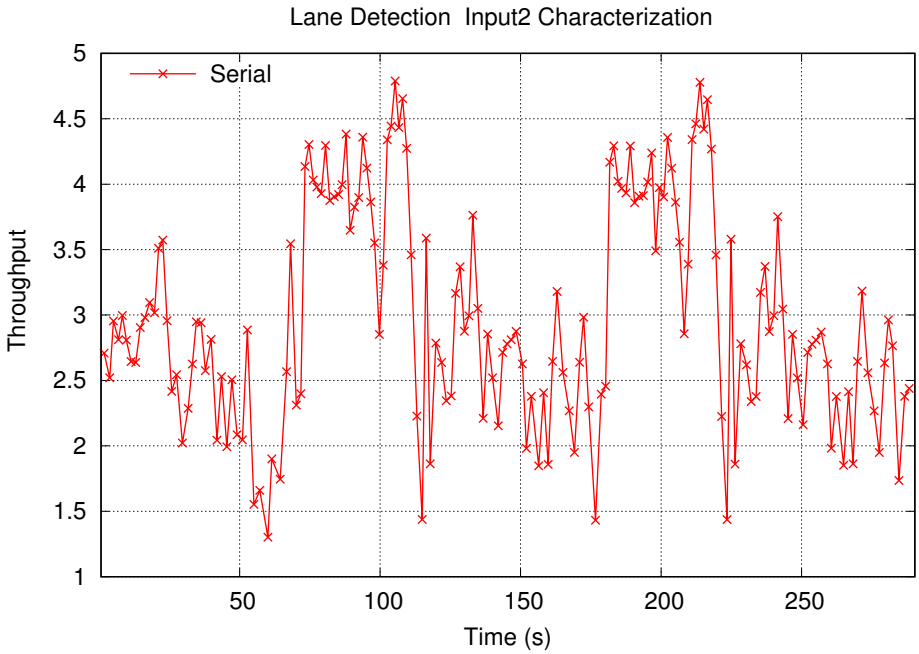


Figure 6.3 – Throughput Characterization input 2.

The goal was to achieve maximum performance. As a consequence, a high target throughput was set for the strategy based on throughput. Moreover, it is relevant to note that the throughput is an average of the entire time used, considering the number of tasks processed and the total time taken.

In addition to the results with input 1, we wanted to assess performance trends with a different input. Therefore we used input 2, which has more load fluctuations demanding a varied amount of resources. If we had used this input for validating the strategy in Chapter 4 the fluctuation would have caused more parallelism degree adaptations. However, in this evaluation our goal was to achieve maximum performance of the adaptive strategies. Consequently, the strategies were expected to use the maximum number of replicas.

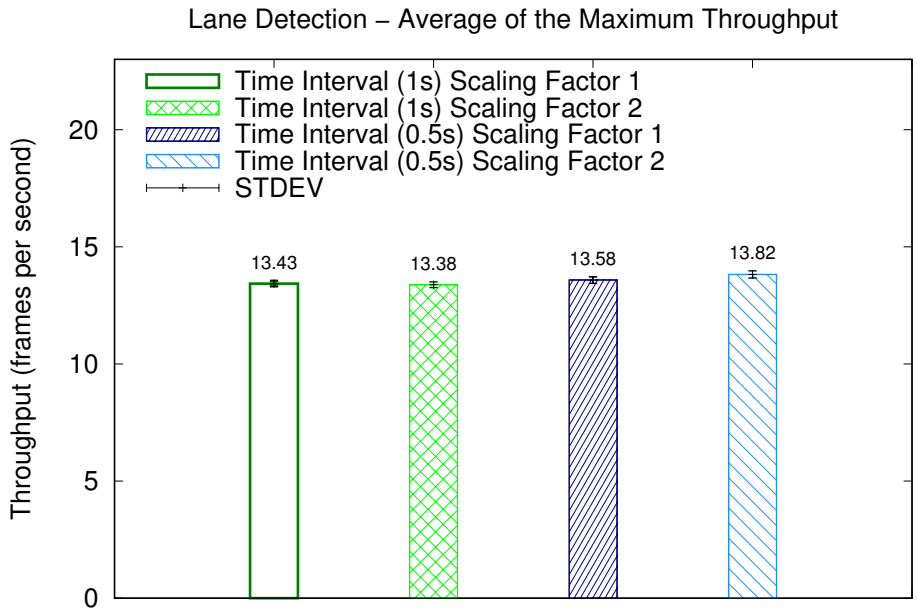


Figure 6.4 – input 2 - Performance of Target Throughput Configurations.

Figure 6.4 shows the throughput of input 2, using the adaptive target performance strategy. In this case, the goal was to maximize throughput. Again, the best throughput was achieved by the configuration with time interval 0.5s and Scaling Factor of 2, because this configuration scaled faster and, as shown in Chapter 4, the additional number of iterations did not affect the performance. It is important to note that the standard deviation was minimal and the performance of the different configurations is similar. The minor contrast between the configurations was only caused by the different settling times.

Moreover, the configuration of the strategy for an adaptive degree of parallelism without user-defined parameters was tested with input 2. Figure 6.5 presents the performance of each configuration, and there is a higher standard deviation than the strategy with a target throughput. This deviation certainly occurred because of the variations in the throughput that caused the strategy to trigger more reconfigurations. However, the minor performance contrast concerning the configurations was caused by the different settling times. Also, the throughput achieved by the different configurations was very similar, which can be placed inside the standard deviation range.

6.2.2 Performance Comparison

This Section compares the adaptive strategies to regular parallel executions that use a static (a.k.a. fixed) number of replicas, ranging in this machine from 2 to 16 replicas. It is important to note that in this evaluation we only considered performance, aiming

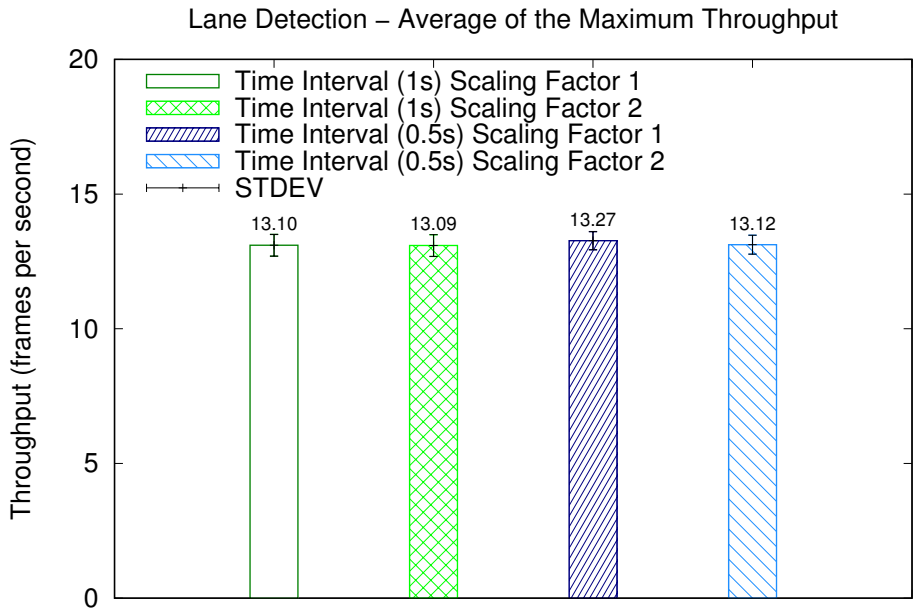


Figure 6.5 – input 2 - Performance of the strategy without user-defined parameters.

to evaluate the impact of the abstractions for an adaptive and transparent degree of parallelism. The adaptive strategies tend to have a more elaborated execution with monitoring and adaptations, which can reduce the overall performance.

Figure 6.6 presents the results using input 1 of the lane detection application. As expected, in the static executions the throughput increased as the number of replicas was increased until it reached the maximum performance of the application. We plotted results from the two adaptive strategies shown in the previous section. In the machine used, the adaptive executions started using 8 replicas, since it is the half of the total number of available cores collected in the adaptive algorithm. The throughput results are an average of the final throughput and the error bars are plotted representing the standard deviation.

The throughput of the static executions is shown regarding each number of replicas. However, the throughput from adaptive executions is the same for all replicas, since any number of replicas could be used during the execution. Figure 6.6 shows the throughput of the different executions using input 1. When comparing the two tested adaptive strategies, it is notable that the strategy with a target performance achieved a better throughput than the strategy without user-defined parameters. This is because the target performance approaches scales faster and with high target performance it uses the maximum number of replicas.

In the executions with static parallelism, the highest throughput was achieved with 14 replicas. The performance of the adaptive strategy with target performance was almost as good as the best static parallelism configuration (14 replicas). This demonstrates that even

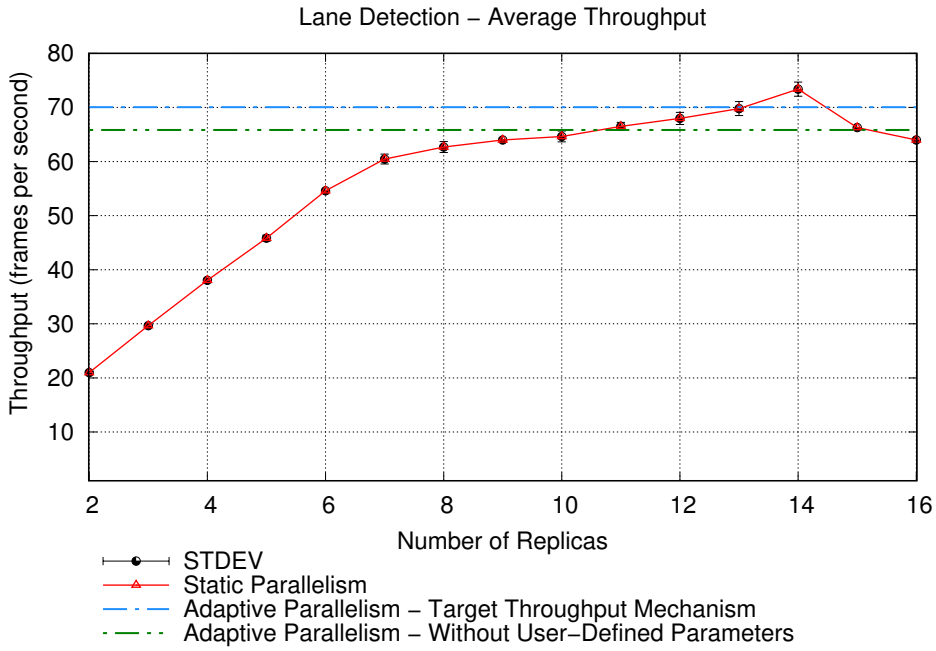


Figure 6.6 – Lane Detection - input 1 Throughput.

with the additional parts implemented the adaptive strategy can achieve great performance, similar to the best static.

Figure 6.7 shows the result of another experiment with the lane detection application using a different input. This input uses a video with a higher resolution, which requires more computation and decreases the frame processing rate. Comparing the three scenarios, it is notable that the execution with a static number of replicas achieved the best throughput with 13 replicas and it was higher than the adaptive strategies when using between 10 and 14 replicas. The two adaptive strategies perform similarly, the approach with a target throughput was slightly better than that which lacked user-defined parameters.

Using the different inputs in the lane detection application achieved the same performance outcome. For input 1, the best static performance was 4.52% better than the adaptive strategy with target throughput, while input 2 was 5.95%.

Figure 6.8 shows the utilization of CPUs from the execution of Figure 6.6. The executions using a static number of replicas tend to use a similar percentage of CPU resources since their execution does not change. On the other hand, the executions using an adaptive number of replicas present variations because the number of replicas changes and consequently, the resource usage. According to the regular characteristics of witnessed in the static execution, we decided for the sake of visual clarity, to only plot the results with 10 or more static replicas.

The CPU utilization results are calculated as an average from the load of each core. For instance, to achieve a 100% average utilization, all cores must have 100% utilization. Moreover, it is relevant to emphasize that the result from CPU utilization came from

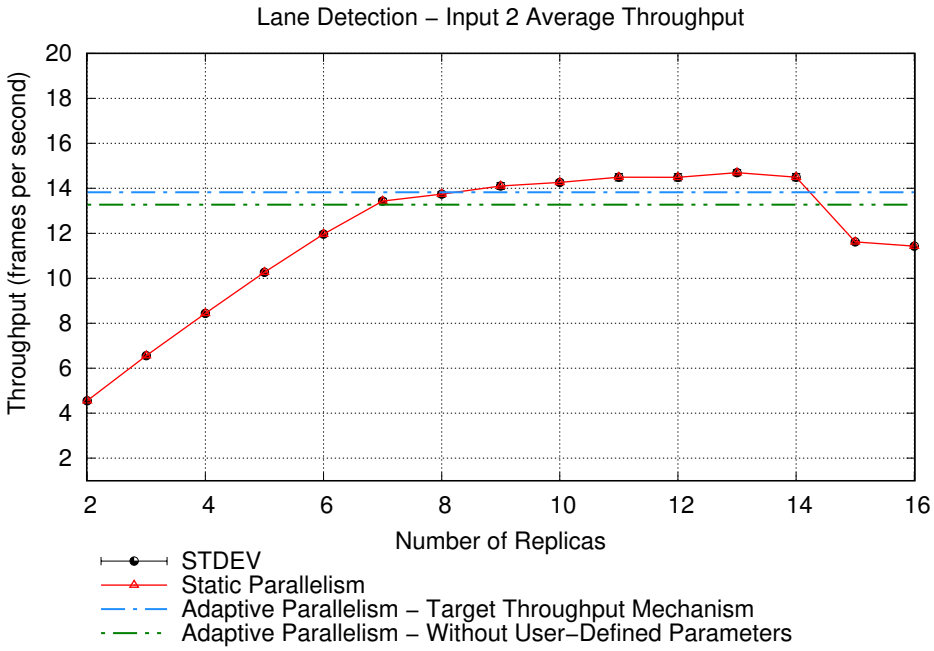


Figure 6.7 – Lane Detection - input 2 Throughput.

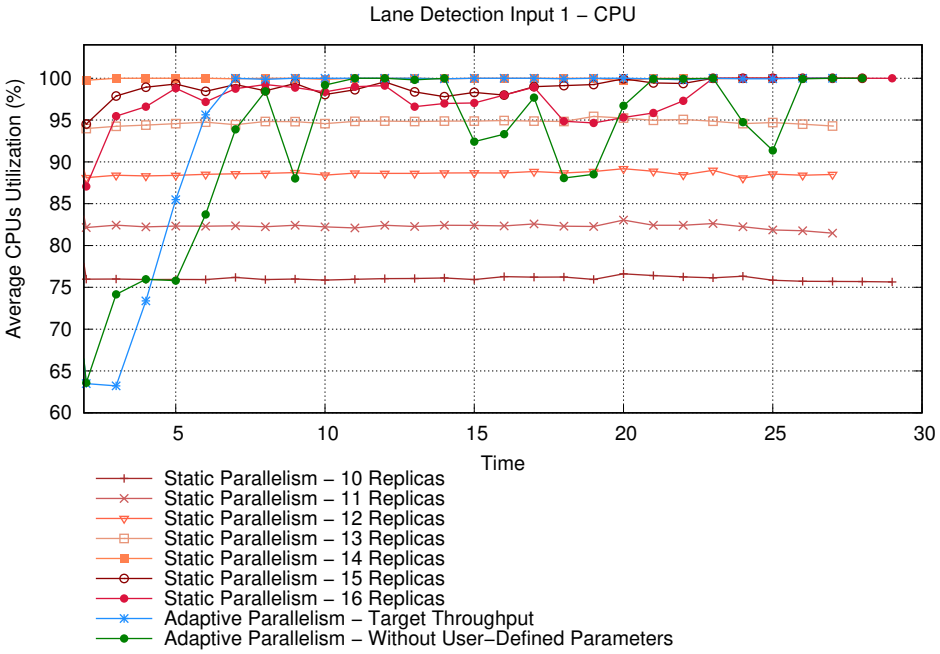


Figure 6.8 – Lane Detection input 1 - Average CPU utilization.

monitoring an execution from each scenario, and the data was collected every 1 second and plotted in a timely fashion. Previously, in the throughput of the adaptive strategies, we selected the best performing configuration to compare with the static executions. In this part, it was not feasible to present results monitoring all configurations and the performance of the different adaptive configurations was almost the same. As a consequence, we selected from both adaptive strategy the configuration with scaling factor of 2 and time interval 1 as a representative configuration that was compared to the static executions.

In Figure 6.8, which is the CPUs' average utilization from the input 1, it is possible to note that the adaptive strategies consumed less resources than the static execution, and yet achieved a similar performance. Furthermore, the same trend can be viewed in Figure 6.9, which is the CPUs' average from input 2. In this experiment, the contrasts are even higher. For instance, the small performance difference seen previously is because the static execution consumed more resources and consequently processed faster. However, mainly the strategy without user-defined parameters optimized the usage of resources by decreasing the number of replicas when suitable, as revealed in Section 4.7 of Chapter 4.

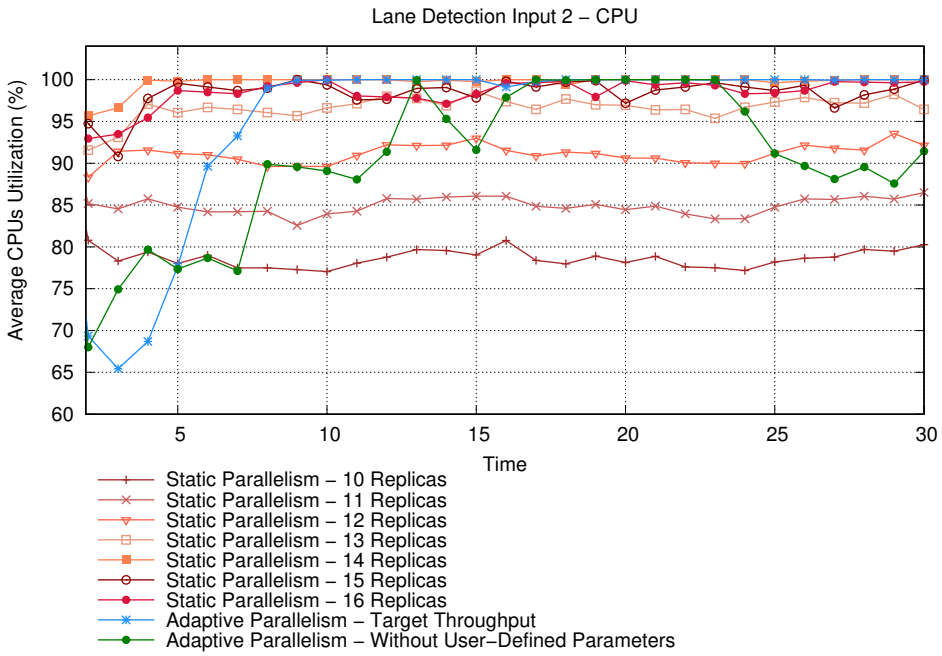


Figure 6.9 – Lane Detection input 2 - Average CPU utilization.

Furthermore, several aspects from the execution are relevant to evaluate the strategies' executions. We have already presented results of the CPUs' utilization, these results are complemented with the memory usage because it is relevant to evaluate the amount of resources that a given program demands in order to run. We collected the total memory usage using the UPL library, and the results are shown as an average of the executions.

Figure 6.10 shows the memory usage of the execution from lane detection using input 1. It shows how the number of replicas impacts memory usage. For instance, 2 replicas needed an average of 100 MB, while 12 replicas used 150 MB. The adaptive strategies used almost the same amount of memory and less than the static execution with more 13 replicas [1]. As adaptive strategies have additional processing parts, they could use additional memory. Yet, the results from memory usage of the adaptive strategies demonstrated no additional resource demands, which is essential to meet the requirement for running under a low overhead.

[1] these executions achieved a higher throughput

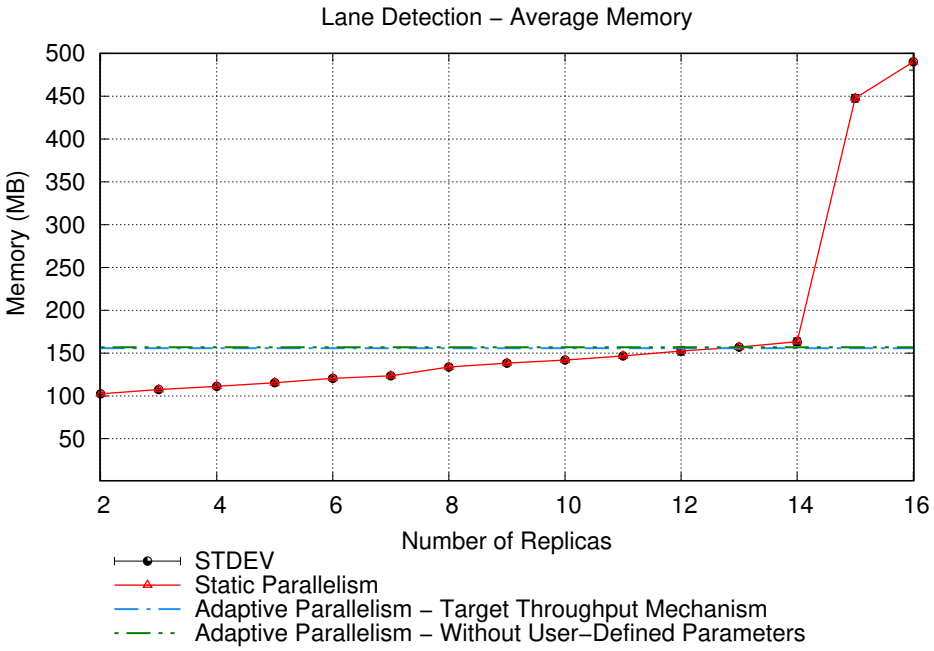


Figure 6.10 – Lane Detection input 1 - Average Memory Usage.

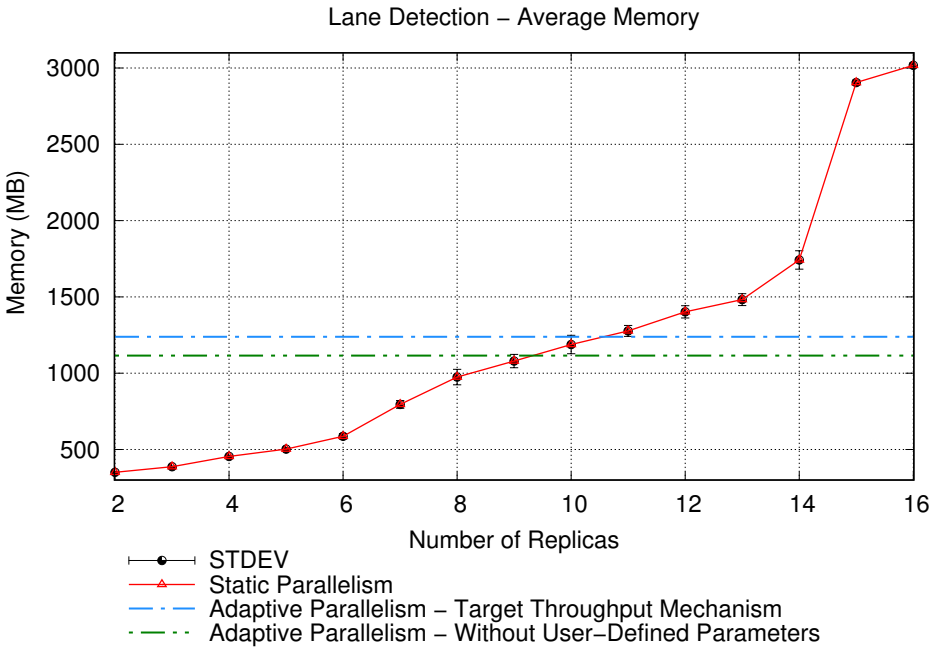


Figure 6.11 – Lane Detection input 2 - Average Memory Usage.

The memory usage of input 2 is shown in Figure 6.11. In this results, the adaptive strategies consumed even less memory compared to the static executions, again following the trend from the number of replicas used. However, in this input, the throughput contrast between adaptive and static executions was slightly higher. Moreover, input 2 consumed much more memory than input 1. This aspect is not related to the parallelism strategies but to the input load.

6.3 Person Recognition

This application is used to recognize people in video streams. The workflow of this applications is shown in Figure 6.12. It starts with the *Capture()* operation that receives the frames as well as detects the faces using the *Detector()* part. The detected faces are marked with a red circle, which the *Recognizer()* function compares with the training set of faces. When the face comparison matches, they are marked with a green circle. The output is produced by the *Writer()* that writes the frames and adds the marked faces.

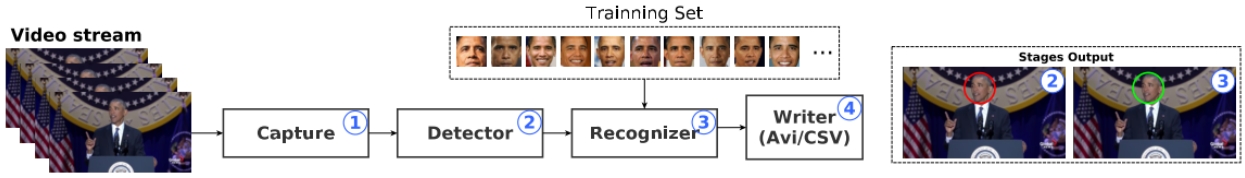


Figure 6.12 – Person Recognition - Workflow. Extracted from [GHDF17].

6.3.1 Performance of Adaptive strategies

The lane detection application was extensively evaluated in Chapter 4 and compared to executions with different configurations. However, we assumed that the proposed strategy should also be tested under different application's performance trends. People recognition has different behavior than lane detection. On the one hand, lane detection only detects and marks road lanes. On the other hand, person recognition has a more complex execution that processes the video detecting faces and comparing them to a database of images. The performance of person recognition was evaluated with an MPEG-4 video of 1.36MB (640x360 pixels) using a training set of 10 images with faces to be recognized in the video.

The code blocks that implemented the adaptive degree of parallelism for each strategy were easily integrated to the parallel version of person recognition. Also, following the structure from lane detection, we first compared the impact of the configurations on the adaptive strategies and then distinguished between the adaptive and static executions.

The strategy with a target performance, which was high to maximize performance in this case, and its configuration is shown in Figure 6.13. The throughput from different configurations was very similar, placed between 5.3 and 5.5 frames per second. In Chapter 4 we saw how the different settling times impact on throughput. However, here we presented the final throughput from executions that were normalized by the average calculation.

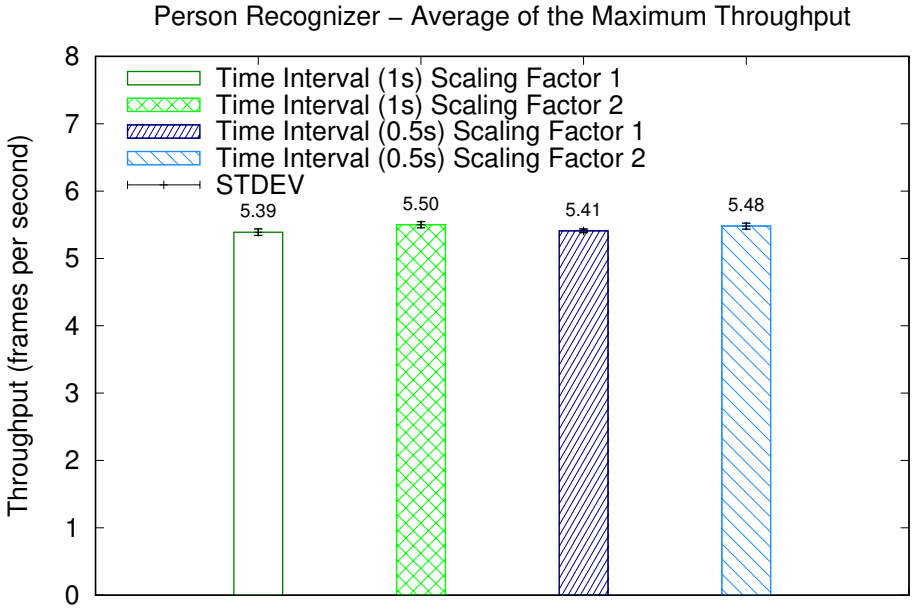


Figure 6.13 – Person Recognition - Performance of Target Throughput Configurations.

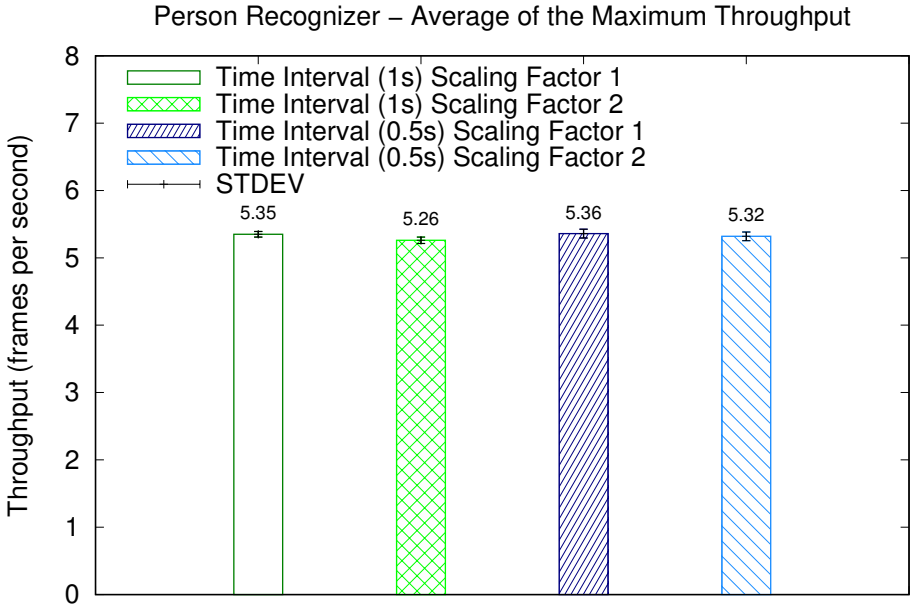


Figure 6.14 – Person Recognition - Performance of the strategy without user-defined parameters.

Moreover, the configurations of the adaptive strategy without user-defined parameters were tested as shown in Figure 6.14. Once again, the contrast between configurations was minimal, from 5.2 to 5.3 frames per second. Considering the standard deviation in the error bars, it was not possible to demonstrate differences. However, even the standard deviation was minimal. In short, the adaptive strategy under the different configuration proved stable and had similar performance.

6.3.2 Performance Comparison

In this Section, we compare the performance of adaptive strategies to executions with static parallelism. Figure 6.15 shows the throughput average. As this application differs from lane detection, the results were also a bit different. The best throughput of static executions was achieved with 8 replicas, which is the number of physical cores. Regarding the performance of the adaptive strategies, the one with a target throughput again outperformed the strategy without user-defined parameters.

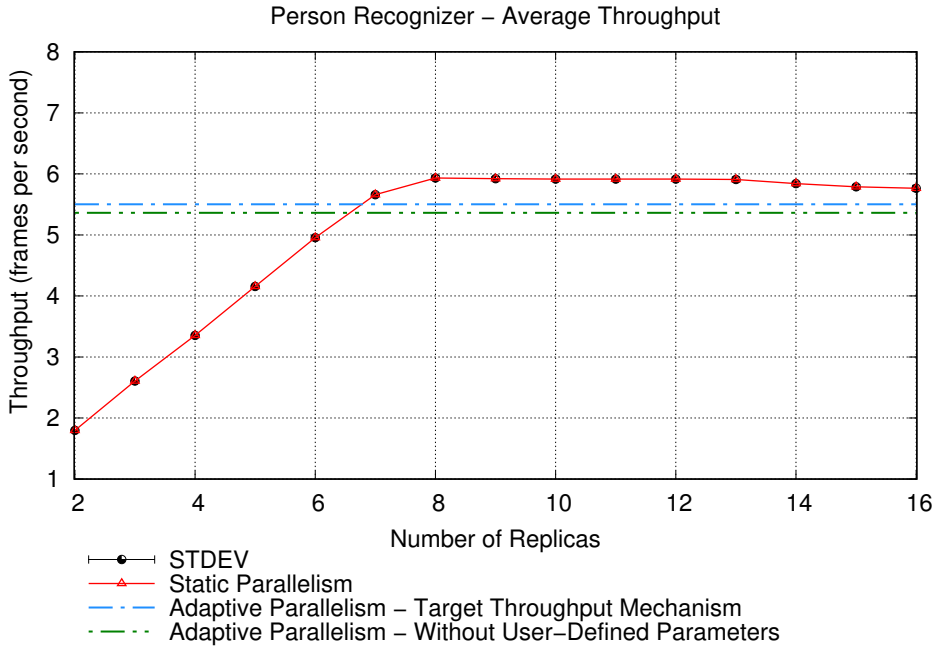


Figure 6.15 – Person Recognition - Average Throughput.

It is relevant to note that this performance result is certainly very dependent on the tested machine. For instance, in the performance of the static execution of this application, we can see a different result than in [GHDF17]. The adaptive executions with target throughput achieved a throughput similar to the static using 7 replicas. Moreover, this adaptive strategy had a throughput 7.28% lower than the best static execution.

Moreover, we monitored the consumption of resources while the application was running. Figure 6.16 shows the CPUs' utilization for each execution. In short, the utilization trends are very similar the results of lane detection. However, in this application more resources does not entail better performance. The static execution with 8 replicas that yielded the best throughput utilized less CPU than the other static executions with higher degrees of parallelism.

Memory usage was also collected from each execution, as shown in Figure 6.17. The usage of the adaptive strategy was similar and in the same range of static execution with 12 to 14 replicas. The best static performance with 8 replicas also consumed less

Figure 6.16 – Person Recognition - Average CPU utilization.

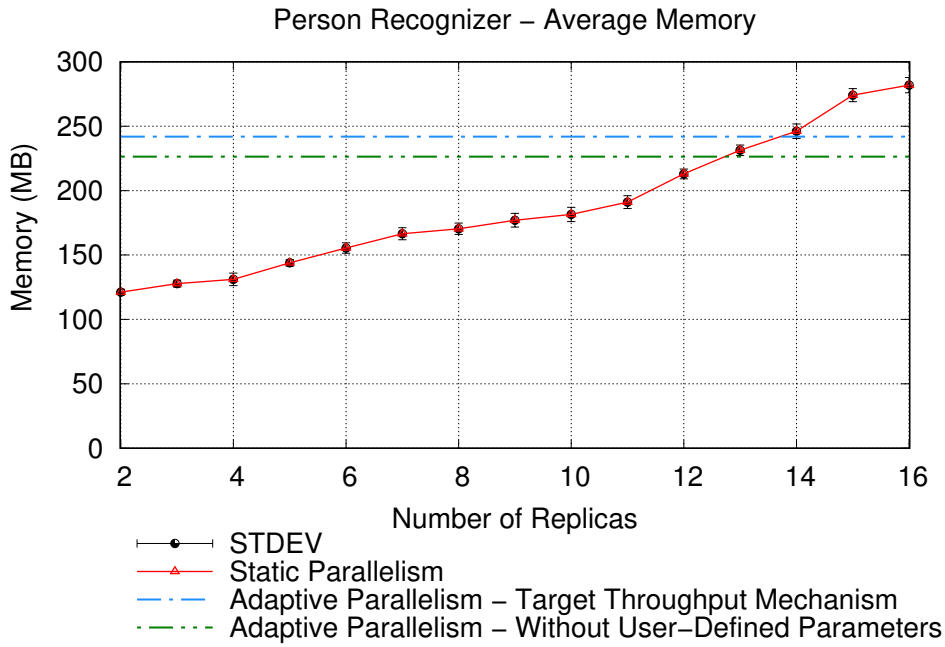


Figure 6.17 – Person Recognition - Average Memory Usage.

memory space. The results from this application demonstrated that it would be relevant for the adaptive strategies to consider the performance variations from different machines and applications to pursue potential performance optimizations. In fact, the adaptive strategies abstract complexities at the price of performance losses. The challenge is that it is almost impossible for a single approach to have optimal performance in all possible scenarios (*e.g.,* applications, machines, architectures, inputs).

6.4 Bzip2

Bzip2 is a data compression application that uses Burrows-Wheeler algorithm for sorting and Huffman coding. This application is built on top of libbzip2, which is a library for data compression [Sew07, GHL+17]. The compression has a workflow based on entities, as shown in Figure 6.18. The first step is completed by reading the input, followed by a compression stage, and finally writing to the output channel. The arrows represent the communication between the stages.

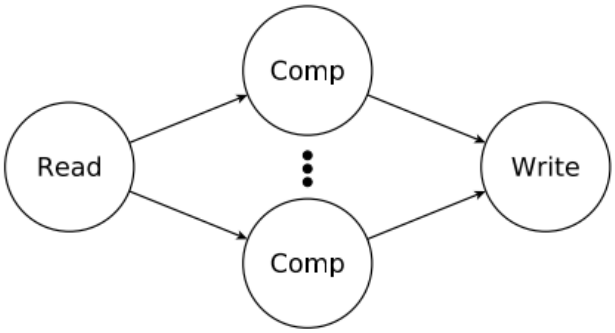


Figure 6.18 – Bzip2 Compression - Workflow. Extracted from [GHL+17].

The compression step can be viewed as a pipeline, the read stage, the parallel compression stage and, the sequential writing stage. The parallel version of Bzip2 (Pbzip2) using SPar is shown in [GHL+17], and this parallel version with the static *replicate* attribute is used in this work. We implemented the strategies for an adaptive degree of parallelism in this parallel version. We evaluated this new adaptive implementation regarding performance and resource consumption with respect to the original parallel version.

6.4.1 Performance of Adaptive Strategies

A parallel version of Bzip2 was also used for evaluating the adaptive strategies. This application is distinguished from the one previously tested because the goal is to compress the files as quickly as possible. As a consequence, Bzip2 is evaluated regarding its throughput. Moreover, we used an ISO file of 704.2 MB as input to simulate a workload.

We first evaluated the throughput of the adaptive strategies. Then the performance of the adaptive strategies were compared to executions with a static number of replicas. Figure 6.19 shows the performance of the four configurations of the adaptive strategy with a target throughput. In this case, the target performance is how many tasks are expected to be processed in a given time interval (1 second). Thus, to achieve a high throughput, the configurations were defined to process as many tasks as possible.

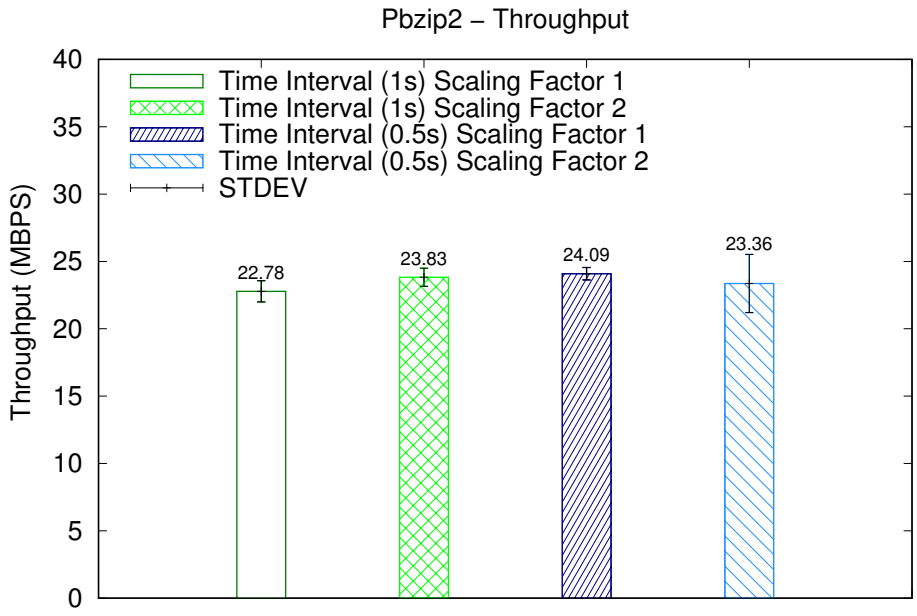


Figure 6.19 – Compression - Performance of Target Throughput Configurations.

The performance of configurations with a target throughput were similar. The best throughput, in MegaBytes per second (MBPS), came from the configuration with a time interval of 0.5s and scaling factor of 1. However, when considering the impact of the standard deviation, it is difficult to identify contrasts between the configurations. In this application, configurations with time interval 0.5s could react quickly, while scaling factor of 2 can be too sensitive to load fluctuations.

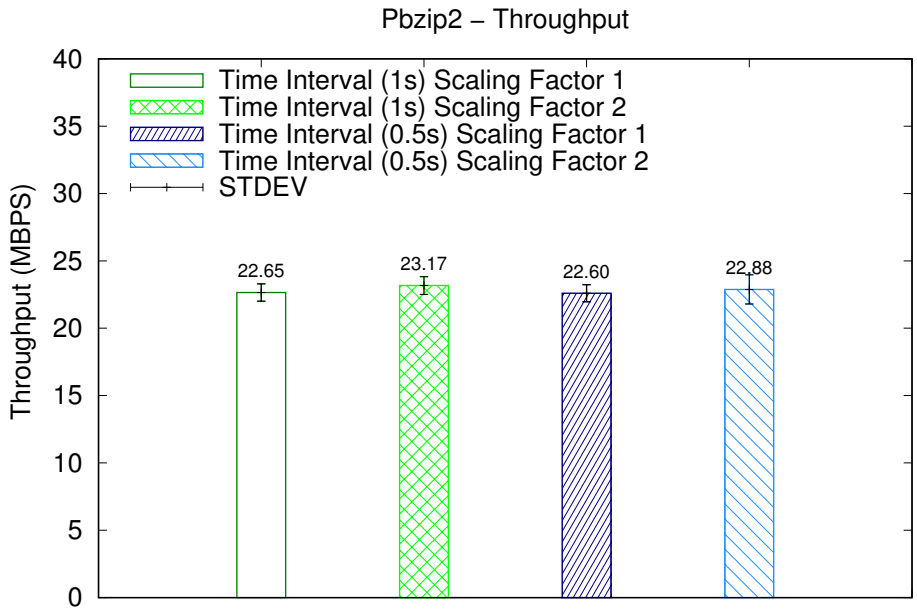


Figure 6.20 – Compression - Performance of the strategy without user-defined parameters.

Figure 6.20 shows the performance of configurations for an adaptive degree of parallelism without user-defined parameters. Again, when comparing the different configurations there are only minor performance differences as well as a low standard deviation. The maximum contrast between the best and worst performing configurations was 2.47%. The configuration that achieved the best performance used a time interval of 1s and scaling factor of 2. Contrasting with the previous applications, Bzip2 tends to perform better with a time interval of 1s. This is due to the fact that Bzip2 is more regular, not requiring constant iterations with short time intervals.

6.4.2 Performance Comparison

Performance is a relevant aspect of the executions using an adaptive number of replicas. A suitable and reasonable way is to compare with static and hand coded parallel executions. In this section, we compare Pbzip2's performance with adaptive strategies to the performance of executions with a static number of replicas.

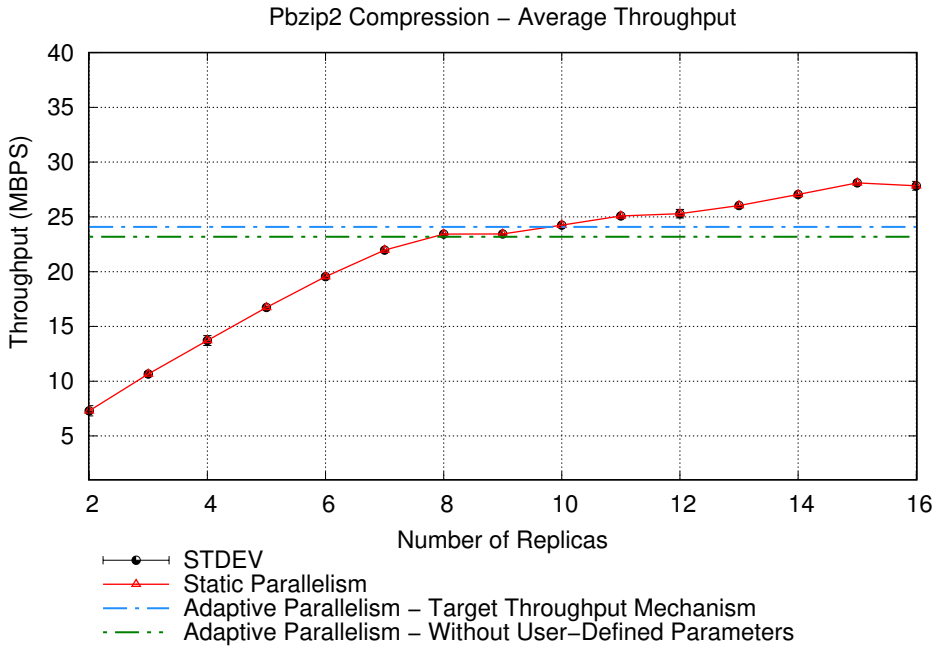


Figure 6.21 – Compression - Average Throughput.

Figure 6.21 shows the throughput of the adaptive strategies as well as the static execution under several numbers of replicas. The strategy with a target throughput was slightly better than without user-defined parameters. The static executions show the impact of the number of replicas on performance. In fact, throughput increased from 7 MBPS with 2 replicas to 28 MBPS with 15 replicas. It is noteworthy that in this application, the throughput was achieved with 15 static replicas, this number of replicas in the same machine resulted

in significant performance losses in the previous tested applications. Consequently, even with the executions with a static degree of parallelism it is possible to identify contrasts in applications' performance.

Moreover, the standard deviation was plotted with the executions. However, it is not possible to identify it since the deviation was minimal. The adaptive strategy performed similar from 8 to 11 replicas. The static parallelism execution with 15 replicas was significantly better than the throughput of the adaptive strategies. In this application, the highest difference in performance was 14.27% between the best static and the adaptive executions. The major source of higher performance differences was the small input, the execution time of the adaptive strategies was around 30 seconds. Short execution time is more affected by the settling times of the adaptive strategy, which varies from 2 to 6 seconds. During settling times, the adaptive strategies run with a sub-optimal degree of parallelism, which decreases the average performance.

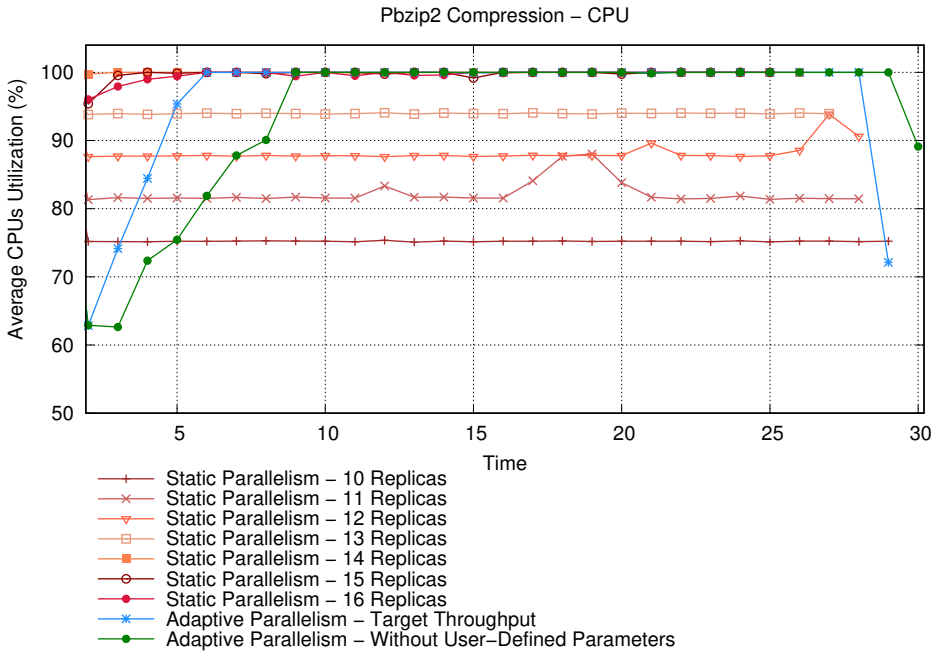


Figure 6.22 – Compression - Average CPU utilization.

CPUs' utilization during the execution was also monitored while the application was running. Figure 6.22 shows the average CPU utilization collected every second of each execution. The execution with the adaptive number of replicas, on average, presented more variation. For instance, the adaptive executions started using 8 replicas resulting in a lower CPU utilization. The execution with a target throughput took around 6 seconds to achieve 100% CPU utilization, which means this was the settling time to reach the optimal number of replicas and maximum performance. Moreover, the execution without user-defined parameters took even longer, almost 9 seconds to achieve the highest degree of parallelism. Thus, again, the executions with adaptive strategies increased the execution time. However, they consumed less CPU than the best performing static executions.

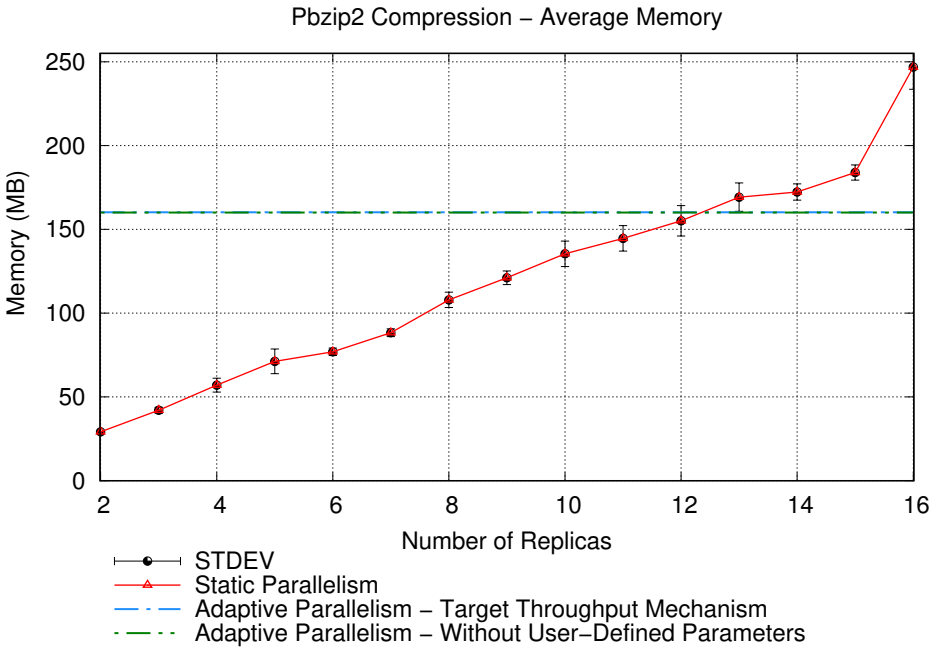


Figure 6.23 – Compression - Average Memory Usage.

The memory usage was also collected. Again, it is possible to see that the adaptive strategies consume less memory. In fact, resource usage is based on the number of replicas used. The best performing static executions consumed significantly more memory space.

Table 6.1 shows the number of pop loss events, a pop lost was introduced in Sections 2.3.2 and 6.1.3. We decided to collect the number of pop losses to understand the impact on run-time caused by the adaptive strategies. In short, a pop loss occurs when the task scheduler is unable to send tasks to the replicas fast enough, which can be caused by different factors. However, because the adaptive strategies have additional parts, several iterations run inside the task scheduler, which could result in performance overhead.

The results shown in Table 6.1 represent a million occurrences. In lane detection with input 1, the adaptive strategies did not have the highest pop losses. However, with input 2 both adaptive strategies presented a significantly higher number of pop losses, and the same result occurred in person recognition and pbzip2.

The percentage of performance contrasts between static and adaptive strategies varies in many ways, depending mainly on the number of replicas and application characteristics. It is not possible to neglect the negative impact of pop losses on performance. However, its impact is not very clear, since the performance is affected by many aspects with pop loss being only one of them. For instance, monitoring the CPU utilization it was possible to conclude that the adaptive strategy consumed less resources and consequently achieved a lower performance. Moreover, there are cases where a high number of the pop losses did not degrade performance. An example of such an event was in pbzip2, where the

static execution with more pop losses was with 16 replicas. However, pbzip2 with 16 replicas yielded one of the best performances (Figure 6.21).

Table 6.1 – Average Number of Pop Losses (in Millions).

Execution	Lane Detection - input 1	Lane Detection - input 2	Person Recognition	Pbzip2 Compression
Static Parallelism 2	13.08	11.4	9.6	0.019
Static Parallelism 3	12.8	11.07	11.5	0.032
Static Parallelism 4	12.1	9.6	10.2	0.039
Static Parallelism 5	12.2	9.7	8.1	0.044
Static Parallelism 6	9.4	8.6	7.6	0.051
Static Parallelism7	11.08	10.3	9.1	0.074
Static Parallelism 8	10.6	10.7	9.3	0.075
Static Parallelism 9	10.02	9.9	8.9	0.17
Static Parallelism 10	10.4	9.8	8.1	0.10
Static Parallelism 11	8.9	8.6	7.6	0.09
Static Parallelism 12	9.9	8.7	8.2	0.28
Static Parallelism 13	10.1	8.3	5.7	0.24
Static Parallelism 14	7.2	7.4	5.3	0.26
Static Parallelism 15	15.3	15.8	14.5	0.76
Static Parallelism 16	14.7	15.1	12.2	1.3
Adaptive Strategy - Target Throughput	10.4	22.1	17.9	3.3
Adaptive Strategy - without user-defined parameters	11.6	20.4	28.4	5.4

6.5 Performance Overview

In Chapter 4 the adaptive strategies were characterized demonstrating their effectiveness to adapt the degree of parallelism on-the-fly. Moreover, in previous sections of this chapter several performance results were presented, these results evaluated the performance of adaptive strategies compared to executions with a static degree of parallelism. Indeed, it is challenging to properly evaluate the adaptive strategy because so many experiments were run. As a consequence, in this section, we present a performance overview of the adaptive strategies.

Figure 6.24 shows the average throughput regarding the different configurations of the adaptive strategies. In this Figure, QM refers to the queues' monitoring strategy, TT to the strategy based on a target throughput, and WUDP is related to the abstracted strategy without user-defined parameters. In general, the configuration a with a scaling factor of 2 (goes up and down faster) and a time interval of 0.5s achieved slightly better performance because it is more sensitive to application fluctuations. The configuration with time interval 0.5s and scaling factor 1 also performed well. The configurations with a time interval of 1s also achieved high throughput rates. In these results, we are not comparing which configuration yielded the best performance, instead we are evaluating how the adaptive strategy behaved under different configurations.

A summary of the performance results from the tested application is shown in Figure 6.25. The results are summarized for three strategies: the two tested in this chapter

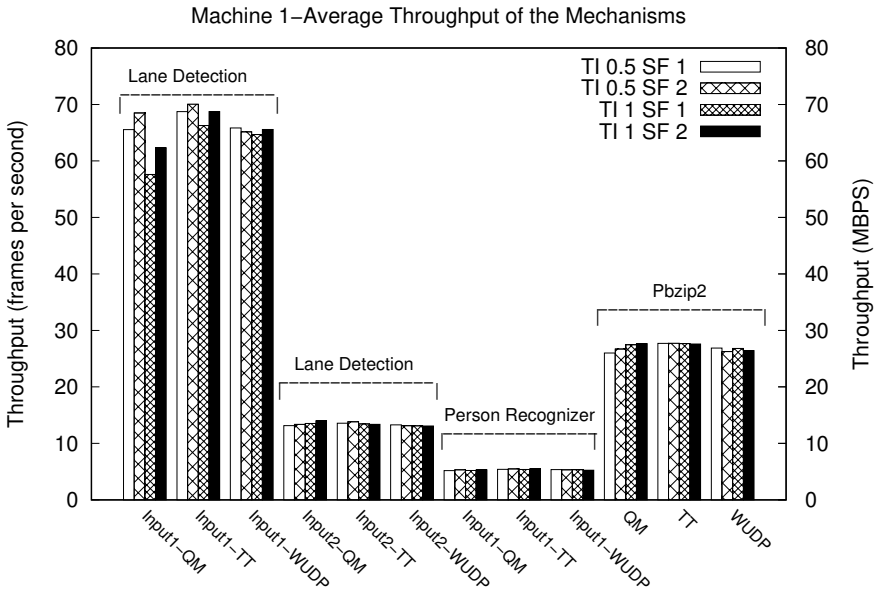


Figure 6.24 – Throughput Average - Configuration of the strategies.

related to the target and to the strategy without user-defined parameters, and the strategy based on queues (shown in Section 4.4). In order to summarize the results of the static executions, we only present the most relevant degree of parallelism. The adaptive strategies with a target throughput (TT) had better performance. However, in each scenario the static degree of parallelism with a variant number of replicas performed slightly better. For instance, in Lane Detection and Person Recognizer applications, the best performance was achieved by 14 static replicas. On the other hand, in Pbzip2 the best performing configuration was with 16 static replicas.

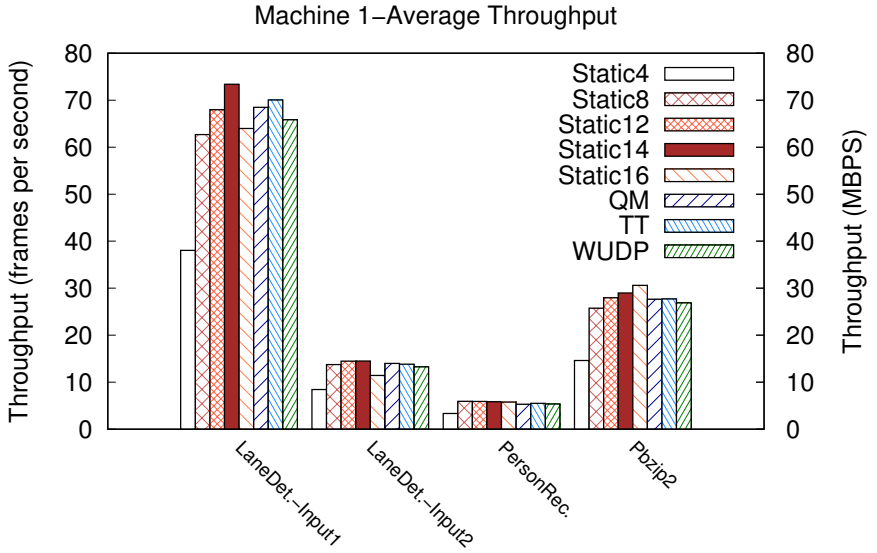


Figure 6.25 – Summary of Average Throughput.

The difference in performance between the configurations tends to be small. We expect different configurations to be better for each scenario, which is the reason we offer this customization. If transparent execution is desired, scaling factor 2 performs better (than 1), while the quick changes in configuration using time interval 0.5s also tends to improve performance.

The results from the previous experiments can also be compared to the best static execution. Consequently, the results were plotted in a graph for each application, the percentage relates to each adaptive strategy and is calculated compared to the best performing static execution. For instance, a high percentage would mean that the adaptive strategy significantly reduced the final performance of a given application. Figure 6.26 shows the percentage of performance losses of adaptive strategies in the Lane Detection application. The left-hand side shows the results from input 1, which demonstrate that the highest losses (10.2%) occurred with the strategy without user-defined parameters. The best performing strategy in input 1 was based on the target throughput, while with input 2 the best performance was achieved by the queues strategy. It is important to emphasize that for each application the queues strategy required a manual definition of the threshold configuration, which had to be sensitive enough to adapt to the degree of parallelism.

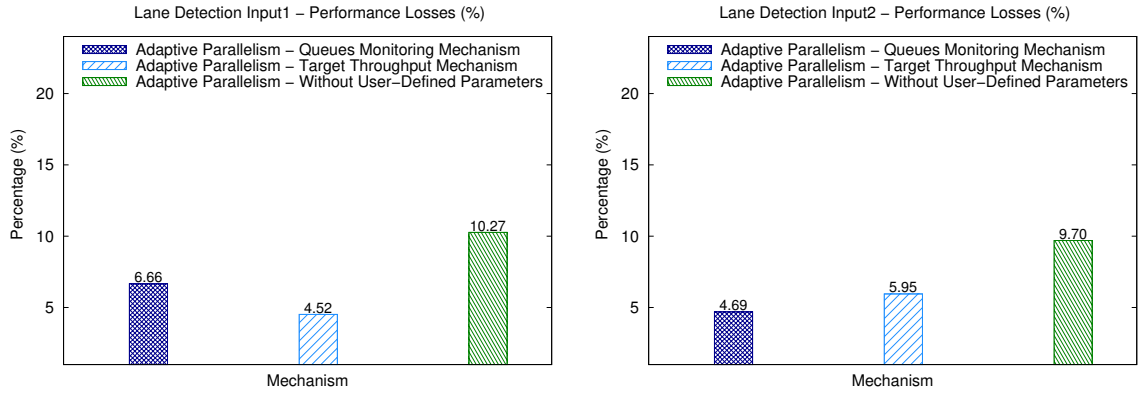


Figure 6.26 – Performance Losses (%) Compared to the Best Static Static Execution in Lane Detection - input 1 (Left) and input 2 (Right).

The left-hand side of Figure 6.27 shows the results from the Person Recognizer application, which is the percentage of performance that each strategy lost compared to the best static execution. Again, the best performance was achieved by the strategy with a target throughput. On the other hand, the queues strategy presented the highest loss (10.2%), which shows that this strategy and its threshold was unable to achieve the best performance considering only the queues from the runtime library.

Moreover, the highest performance losses of the adaptive strategies occurred in the Pbzip compression, shown on the right-hand side of Figure 6.27. This application was more affected by the adaptive strategies because its behavior is more regular, constantly adapting the degree of parallelism degraded the performance. This type of application potentially

does not require iterations and changes to the configurations periodically or for short time intervals. Consequently, this application could take advantage of a steady period that would enable the application to run for longer time intervals without adaptive iterations.

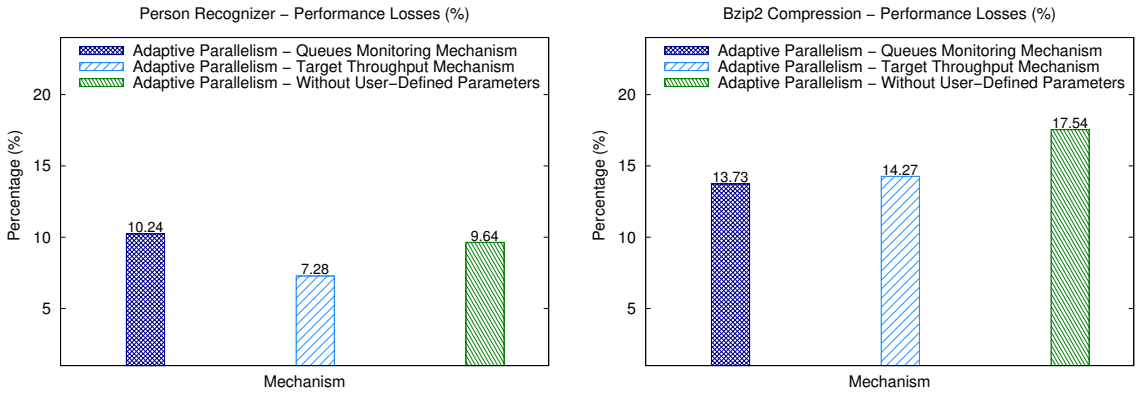


Figure 6.27 – Performance Losses (%) Compared to the Best Static Static Execution - Person Recognizer (Left) and Pbzip2 Compression (Right).

The impact on performance caused by the adaptive strategies is different in each application. In some cases, the performance losses in adaptive strategies were high. However, we have to consider several aspects. The performance of executions with a static degree of parallelism also varied between applications and machines. The best performance of each application was achieved with a different number of replicas. A significant or even a huge amount of time would be needed for a programmer to manually find the number of replicas that achieves the best performance for a specific application and machine. Moreover, the best performing configuration in stream processing applications often varies due to input and environmental changes, which further complicates using a static degree of parallelism. Thus, even with losses in performance, the adaptive strategy can be suitable for stream processing applications, which transparently and continuously adapt the applications' execution.

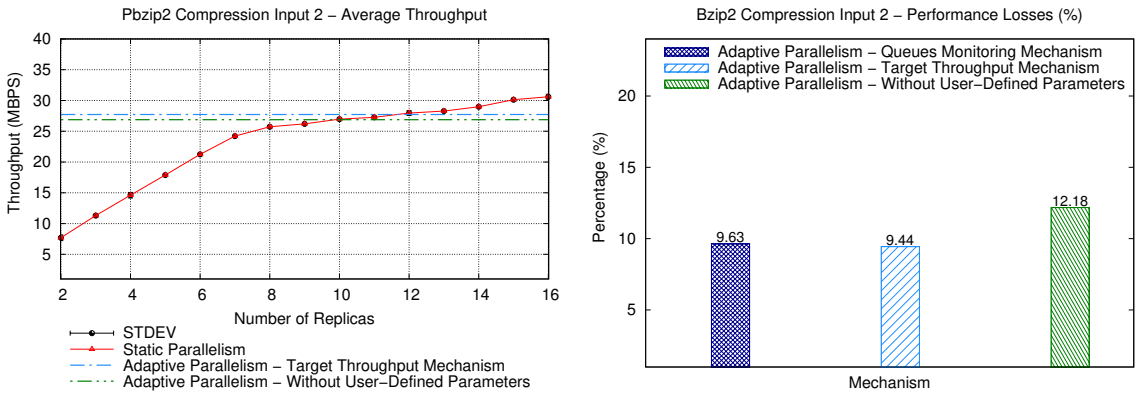


Figure 6.28 – Pbzip2 Compression input 2 - Throughput (Left) and Percentage (Right) .

The significant performance losses seen on the right-hand side of Figure 6.27 instigated efforts towards a deeper understanding of its causes. A potential reason of such

contrasts is the input used, which is small and only took a few seconds to be processed. Consequently, such short executions could increase the execution time in adaptive strategies because of the settling times when the program starts its execution. This affirms that if the adaptive execution uses a sub-optimal degree of parallelism it can significantly reduce the overall performance in short executions. We decided to evaluate the compression with a larger input. The selected input is a file with 6.3GB composed of a dump of all the abstracts from the English Wikipedia, previously used by [STD16]. Figure 6.28 shows the additional Pbzip results, on the left-hand side the throughput and on the right-hand side the percentage of performance losses compared to the best static execution. In this experiment, it was possible to note that the adaptive strategies achieved better performance. The throughput was closed to 13 static replicas, while in the worst case the strategy without user-defined parameters had 12.18% performance degradation with respect to the best static execution.

Moreover, we also included the results from the strategy based on queues, which also had a fair performance. The best performing adaptive strategy used a target throughput. It is important to note that in this specific application and machine, the best performance was achieved with 16 static replicas, while the adaptive strategy used 14 as the maximum number of replicas. The static execution with 14 replicas was only 4.38% better than the adaptive strategy using a target throughput. However, the fact that in this specific case the performance was not optimized using adaptations cannot be neglected. Consequently, future efforts in performance optimizations must find the best performing configurations for applications with regular characteristics.

6.6 Remarks

Several experiments were conducted in this study. Our key outcomes are the following:

- The final performance of adaptive strategies is not significantly affected by time intervals (0.5 - 1 second) or scaling factors (1 - 2 replicas).
- The performance trends of an application tend to be similar, even with different inputs.
- The number of replicas that yielded the best performance with static parallelism executions varies depending on different applications and machine architectures.
- The adaptive strategies have shown a reasonable overhead regarding performance and without consuming more resources (CPU and memory). In fact, the tolerated overhead depends on the programmers' performance objectives and their skills in parallel programming. How much performance can be sacrificed in order to have adaptive and high-level parallelism abstractions, depends on the specific demands and scenarios.

7. CONCLUSION

In this study we introduced the concept of adaptive degrees of parallelism in SPar, which is a new parallelism abstraction. It is crucial to employ an adaptive degree of parallelism for stream processing applications because the workload varies. Yet, this is complex for application programmers.

We aimed to provide adaptive capabilities in SPar. Our main contribution was to eliminate the need for programmers to provide the number of replicas manually. We implemented strategies to monitor the execution and adapt the degree of parallelism when needed. Thus eliminating the need for manual, complex, and time-consuming definition of the degree of parallelism in SPar.

Moreover, we presented different strategies and their mechanisms for adapting the number of replicas on-the-fly. Our strategies considered different information for deciding if optimization is required, using performance metrics such as throughput, latency, and queue congestion. Thus, we presented an option that does not require users to enter any parameters, neither the number of replicas nor a performance goal. Therefore, the programmer has the flexibility to choose whether they want to define target performance.

The adaptive strategies that were implemented can be generated along with the SPar parallel code, by using the designed transformation rules. Consequently, real-world stream processing applications can benefit from the adaptive degree of parallelism support. The performance of adaptive strategies that were implemented in our applications was evaluated and compared to the executions with a static degree of parallelism. Experiment results demonstrated the effectiveness of our solution when adjusting the number of replicas during the execution time. When running with a static number of replicas, it tends only to outperform the adaptive strategy if using the maximum amount of resources. Also, it is important to note that the static number of replicas requires a manual definition of the degree of parallelism.

With the achieved results, we concluded that the adaptive strategies increased the level of abstraction without compromising the performance or consuming more computational resources. The strategy based on a target throughput achieved the best performance. The option without user-defined parameters was easier to use, but had slightly inferior performance. The programmer needs performance expertise to use the strategies based on target performance (throughput, latency), which is why an alternative without a target performance was provided.

The requirements and implementations presented are specific to the problem of abstracting parallelism in SPar. Moreover, the methods used for controlling the number of replicas may only be effective in the SPar runtime. The results from executions using an adaptive degree of parallelism are only related to the tested applications, but there are

similar trends for other stream processing applications. Other aspects from different levels such as thread placement, affinity, and core frequency are not considered in this study.

We intend to extend this study in many ways. Firstly, we aim to test on different machines and implement in other real-world applications. Moreover, we hope to continue improving the adaptive strategies, most of the essential parts involve balancing between stability and performance, and also analyze other ways to introduce a grace period between interaction under a stable workload. Moreover, efforts could optimize the performance of the adaptive strategies. When performance is the primary goal, it would be relevant to implement a statistical methodology for testing the results. Furthermore, the techniques used in related works could also be implemented and tested in our scenario.

In the future, the adaptive strategies could be implemented using online learning to adapt the degree of parallelism. For instance, the strategy using the runtime library's queues could use a learning algorithm instead of the threshold. Thus, the need for high-level abstraction as well as performance could be met. Furthermore, the strategy can be extended to run in distributed cluster environments as well as in cloud and fog environments, increasing the flexibility of the strategies by taking advantage of elastic properties from the infrastructure level.

REFERENCES

[AGT14] Andrade, H.; Gedik, B.; Turaga, D. “Fundamentals of Stream Processing: Application Design, Systems, and Analytics”. Cambridge University Press, 2014, 558p.

[AMT10] Aldinucci, M.; Meneghin, M.; Torquati, M. “Efficient Smith-Waterman on Multi-core with FastFlow”. In: Proceedings of the Euromicro Conference on Parallel, Distributed and Network-based Processing, 2010, pp. 195–199.

[Bie11] Bienia, C. “Benchmarking Modern Multiprocessors”, Ph.D. Thesis, Department of Computer Science, Princeton University, Princeton, 2011, 153p.

[Col89] Cole, M. I. “Algorithmic Skeletons: Structured Management of Parallel Computation”. Pitman London, 1989, 137p.

[Col04] Cole, M. “Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming”, *Parallel Computing*, vol. 30–3, Mar 2004, pp. 389–406.

[CQ09] Chakravarthy, S.; Qingchun, J. “Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing”. Springer US, 2009, 324p.

[DST17] Danelutto, M.; Sensi, D. D.; Torquati, M. “A Power-Aware, Self-Adaptive Macro Data Flow Framework”, *Parallel Processing Letters*, vol. 27–01, Mar 2017, pp. 1–20.

[FAG+17] Floratou, A.; Agrawal, A.; Graham, B.; Rao, S.; Ramasamy, K. “Dhalion: Self-Regulating Stream Processing in Heron”, *Very Large Data Base Endowment*, vol. 10–12, Aug 2017, pp. 1825–1836.

[Fas17] FastFlow . “FastFlow (FF) Website”. last access in Dec, 2017, Source: http://mc-fastflow.sourceforge.net/, 2017.

[GAW+08] Gedik, B.; Andrade, H.; Wu, K.-L.; Yu, P. S.; Doo, M. “SPADE: The System S Declarative Stream Processing Engine”. In: Proceedings of the ACM International Conference on Management of Data, 2008, pp. 1123–1134.

[GDTF15] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “An Embedded C++ Domain-Specific Language for Stream Parallelism”. In: Proceedings of the International Conference on Parallel Computing - Parallel Computing: On the Road to Exascale, 2015, pp. 317–326.

[GDTF17] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “SPar: A DSL for High-Level and Productive Stream Parallelism”, *Parallel Processing Letters*, vol. 27–01, Mar 2017, pp. 1739–1741.

[GF17] Griebler, D.; Fernandes, L. G. “Towards Distributed Parallel Programming Support for the SPar DSL”. In: Proceedings of the International Conference on Parallel Computing - Parallel Computing: On the Road to Exascale, 2017, pp. 110–120.

[GFDF18] Griebler, D.; Filho, R. B. H.; Danelutto, M.; Fernandes, L. G. “High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2”, *International Journal of Parallel Programming*, vol. 10766, Feb 2018, pp. 1–19.

[GHDF17] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “Higher-Level Parallelism Abstractions for Video Applications with SPar”. In: Proceedings of the International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms, 2017, pp. 698–707.

[GHDF18] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “Stream Parallelism with Ordered Data Constraints on Multi-Core Systems”, *Journal of Supercomputing*, vol. 75, Jul 2018, pp. 1–20.

[GHL+17] Griebler, D.; Hoffmann, R. B.; Loff, J.; Danelutto, M.; Fernandes, L. G. “High-Level and Efficient Stream Parallelism on Multi-core Systems with SPar for Data Compression Applications”. In: Proceeding of the Simpósio em Sistemas Computacionais de Alto Desempenho, 2017, pp. 16–27.

[GJPPM+12] Gulisano, V.; Jimenez-Peris, R.; Patino-Martinez, M.; Soriente, C.; Valduriez, P. “StreamCloud: An Elastic and Scalable Data Streaming System”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 23–12, Jan 2012, pp. 2351–2365.

[Gri16] Griebler, D. “Domain-Specific Language & Support Tool for High-Level Stream Parallelism”, Ph.D. Thesis, Faculdade de Informática, PUCRS, Porto Alegre, 2016, 243p.

[GSHW14] Gedik, B.; Schneider, S.; Hirzel, M.; Wu, K.-L. “Elastic Scaling for Data Stream Processing”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 25–6, Jun 2014, pp. 1447–1463.

[GSV+18] Griebler, D.; Sensi, D. D.; Vogel, A.; Danelutto, M.; Fernandes, L. G. “Service Level Objectives via C++11 Attributes”. In: Proceedings of the Euro-Par: Parallel Processing Workshops, 2018, pp. 1–12.

[GVM+18] Griebler, D.; Vogel, A.; Maron, C. A. F.; Maliszewski, A. M.; Schepke, C.; Fernandes, L. G. “Performance of Data Mining, Media, and Financial Applications under Private Cloud Conditions”. In: Proceedings of the IEEE Symposium on Computers and Communications, 2018, pp. 1–7.

[HDPT04] Hellerstein, J. L.; Diao, Y.; Parekh, S.; Tilbury, D. M. “Feedback Control of Computing Systems”. John Wiley & Sons, 2004, 456p.

[HPJF14] Heinze, T.; Pappalardo, V.; Jerzak, Z.; Fetzer, C. “Auto-scaling Techniques for Elastic Data Stream Processing”. In: Proceedings of the International Conference on Data Engineering Workshops, 2014, pp. 296–302.

[HSS+14] Hirzel, M.; Soulé, R.; Schneider, S.; Gedik, B.; Grimm, R. “A Catalog of Stream Processing Optimizations”, *ACM Computing Surveys*, vol. 46–4, Apr 2014, pp. 46:1–46:34.

[KC03] Kephart, J. O.; Chess, D. M. “The Vision of Autonomic Computing”, *Computer*, vol. 36–1, Jan 2003, pp. 41–50.

[LBMAL12] Lorido-Botrán, T.; Miguel-Alonso, J.; Lozano, J. A. “Auto-scaling Techniques for Elastic Applications in Cloud Environments”, Master’s Thesis, Department of Computer Architecture and Technology, University of Basque Country, Basque Country, 2012, 44p.

[LDC+17] Liu, X.; Dastjerdi, A. V.; Calheiros, R. N.; Qu, C.; Buya, R. “A Stepwise Auto-Profiling Method for Performance Optimization of Streaming Applications”, *ACM Transactions on Autonomous and Adaptive Systems*, vol. 12–4, Nov 2017, pp. 24:1–24:33.

[LQF15] Lepers, B.; Quéma, V.; Fedorova, A. “Thread and Memory Placement on NUMA Systems: Asymmetry Matters”. In: Proceedings of the USENIX Annual Technical Conference, 2015, pp. 277–289.

[MM16] Matteis, T. D.; Mencagli, G. “Keep Calm and React with Foresight: Strategies for Low-latency and Energy-efficient Elastic Data Stream Processing”. In: Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, 2016, pp. 13:1–13:12.

[MMPM14] Marco, A.; Marco, D.; Peter, K.; Massimo, T. “Fastflow: High-Level and Efficient Streaming on Multicore”. Wiley-Blackwell, 2014, chap. 13, pp. 261–280.

[MRR12] McCool, M.; Reinders, J.; Robison, A. “Structured Parallel Programming: Patterns for Efficient Computation”. Elsevier Science, 2012, 406p.

[PGB11] Pusukuri, K. K.; Gupta, R.; Bhuyan, L. N. “Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring”. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2011, pp. 116–125.

[Rei07] Reinders, J. “Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism”. O’Reilly Media, 2007, 336p.

[RKO+11] Raman, A.; Kim, H.; Oh, T.; Lee, J. W.; August, D. I. “Parallelism Orchestration using DoPE: the Degree of Parallelism Executive”. In: Proceedings of the The ACM Special Interest Group on Programming Languages Notices, 2011, pp. 26–37.

[SAG+09] Schneider, S.; Andrade, H.; Gedik, B.; Biem, A.; Wu, K. L. “Elastic Scaling of Data Parallel Operators in Stream Processing”. In: Proceedings of the IEEE International Symposium on Parallel Distributed Processing, 2009, pp. 1–12.

[Sen12] Sensi, D. D. “DPI Over Commodity Hardware: Implementation of a Scalable Framework Using FastFlow”, Master’s Thesis, Computer Science Department, University of Pisa, Pisa, 2012, 302p.

[Sew07] Seward, J. “Bzip2 and Libbzip2, version 1.0. 5: A Program and Library for Data Compression”. last access in Dec, 2017, Source: http://www.bzip.org, 2007.

[SGS13] Sridharan, S.; Gupta, G.; Sohi, G. S. “Holistic Run-time Parallelism Management for Time and Energy Efficiency”. In: Proceedings of the ACM International Conference on Supercomputing, 2013, pp. 337–348.

[SHGW12] Schneider, S.; Hirzel, M.; Gedik, B.; Wu, K.-L. “Auto-Parallelizing Stateful Distributed Streaming Applications”. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2012, pp. 53–64.

[SMD17] Sensi, D. D.; Matteis, T. D.; Danelutto, M. “Nornir: A Customisable Framework for Autonomic and Power-Aware Applications”. In: Proceedings of the Euro-Par Parallel Processing Workshops, 2017, pp. 42–54.

[SMMF15] Selva, M.; Morel, L.; Marquet, K.; Frenot, S. “A Monitoring System for Runtime Adaptations of Streaming Applications”. In: Proceedings of the Euromicro Conference on Parallel, Distributed and Network-based Processing, 2015, pp. 27–34.

[SST+15] Su, Y.; Shi, F.; Talpur, S.; Wang, Y.; Hu, S.; Wei, J. “Achieving Self-Aware Parallelism in Stream Programs”, *Cluster Computing*, vol. 18–2, Jun 2015, pp. 949–962.

[STD16] Sensi, D. D.; Torquati, M.; Danelutto, M. “A Reconfiguration Algorithm for Power-Aware Parallel Applications”, *ACM Transactions on Architecture and Code Optimization*, vol. 13–4, Dec 2016, pp. 43:1–43:25.

[STD17] Sensi, D. D.; Torquati, M.; Danelutto, M. “Mammut: High-level Management of System Knobs and Sensors”, *SoftwareX*, vol. 6, 2017, pp. 150–154.

[TKA02] Thies, W.; Karczmarek, M.; Amarasinghe, S. “StreamIt: A Language for Streaming Applications”. In: Proceedings of the International Conference on Compiler Construction, 2002, pp. 179–196.

[VF18] Vogel, A.; Fernandes, L. G. “Grau de Paralelismo Adaptativo na DSL SPar”. In: Proceedings of the Escola Regional de Alto Desempenho, 2018, pp. 237–238.

[VGF17] Vogel, A.; Griebler, D.; Fernandes, L. G. “Proposta de Implementação de Grau de Paralelismo Adaptativo em uma DSL para Paralelismo de Stream”. In: Proceedings of the Escola Regional de Alto Desempenho, 2017, pp. 1–2.

[VGM+16] Vogel, A.; Griebler, D.; Maron, C. A. F.; Schepke, C.; Fernandes, L. G. “Private IaaS Clouds: A Comparative Analysis of OpenNebula, CloudStack and OpenStack”. In: Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2016, pp. 672–679.

[VGS+18] Vogel, A.; Griebler, D.; Sensi, D. D.; Danelutto, M.; Fernandes, L. G. “Autonomic and Latency-Aware Degree of Parallelism Management in SPar”. In: Proceedings of the Euro-Par: Parallel Processing Workshops, 2018, pp. 1–12.

[VGSF17] Vogel, A.; Griebler, D.; Schepke, C.; Fernandes, L. G. “An Intra-Cloud Networking Performance Evaluation on CloudStack Environment”. In: Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2017, pp. 1–5.

[WKWO12] Wu, S.; Kumar, V.; Wu, K.-L.; Ooi, B. C. “Parallelizing Stateful Operators in a Distributed Stream Processing System: How, Should You and How Much?” In: Proceedings of the ACM International Conference on Distributed Event-Based Systems, 2012, pp. 278–89.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br