Received: 3 January 2020 | Revised: 2 November 2020 | Accepted: 4 December 2020

DOI: 10.1002/spe.2948

RESEARCH ARTICLE

Providing high-level self-adaptive abstractions for stream parallelism on multicores

Adriano Vogel[1] | Dalvan Griebler[1,2] | Luiz Gustavo Fernandes[1]

[1]School of Technology, Pontifical Catholic University of Rio Grande do Sul, Rio Grande do Sul, Brazil

[2]Laboratory of Advanced Research on Cloud Computing, Sociedade Educacional Três de Maio (SETREM), Rio Grande do Sul, Brazil

Correspondence
Dalvan Griebler and Adriano Vogel, School of Technology, Pontifical Catholic University of Rio Grande do Sul, Rio Grande do Sul, Brazil.
Email: dalvan.griebler@acad.pucrs.br, adriano.vogel@acad.pucrs.br

Funding information
Conselho Nacional de Desenvolvimento Científico e Tecnológico, Grant/Award Number: 437693/2018-0; Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Grant/Award Number: 001; Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul, Grant/Award Numbers: 17/2551-0000871-5, 19/2551-0001895-9

Abstract
Stream processing applications are common computing workloads that demand parallelism to increase their performance. As in the past, parallel programming remains a difficult task for application programmers. The complexity increases when application programmers must set nonintuitive parallelism parameters, that is, the degree of parallelism. The main problem is that state-of-the-art libraries use a static degree of parallelism and are not sufficiently abstracted for developing stream processing applications. In this article, we propose a self-adaptive regulation of the degree of parallelism to provide higher-level abstractions. Flexibility is provided to programmers with two new self-adaptive strategies, one is for performance experts, and the other abstracts the need to set a performance goal. We evaluated our solution using compiler transformation rules to generate parallel code with the SPar domain-specific language. The experimental results with real-world applications highlighted higher abstraction levels without significant performance degradation in comparison to static executions. The strategy for performance experts achieved slightly higher performance than the one that works without user-defined performance goals.

KEYWORDS
parallelism abstractions, parallel programming, self-adaptive systems, stream parallelism, stream processing applications

1 | INTRODUCTION

The increasing use of techniques to collect data from different sources (e.g., sensors, cameras, and radar) has given rise to stream processing applications. This new application type has unique aspects, such as continuous data processing and varied input/workload trends. There are a huge number of domains that must gather and analyze data in real (or almost real) time,[1-4] which is a complex demand.

Although parallelism is a potential solution for improving the performance of stream processing applications, it increases the level of complexity in terms of programmability. First, performance gains can be achieved because computer architectures have multiple processing units placed on a chip. Second, from the parallel programming perspective, it is more challenging because applications are expected to be implemented to properly exploit the available hardware parallelism. However, parallelism requires managing low-level routines, such as thread creation, synchronization, load distribution, and balancing. Consequently, introducing parallel routines remains too complex for application programmers, who are focused on developing stream processing applications and may not necessarily have expertise in parallel hardware and specific operating system (OS) routines.

Considering the complexities of supporting parallel executions, different parallel programming frameworks and libraries have been created to exploit stream parallelism in C++ programs, such as Intel TBB,[5] FastFlow,[7] GrPPI,[8] SPar,[9] and StreamIt6 which is a new language. There are also distributed stream processing engines (SPEs) such as Apache Storm,[10] Apache Spark,[11] and Apache Flink[12] that are not for C++ applications. These SPEs are designed for large-scale clusters using Java Virtual Machine to provide hardware abstractions. However, as shown in Reference 13, java-based SPEs achieve a severely limited performance and efficiency due to the suboptimal data serialization, memory accesses, and garbage collection. Furthermore, we have seen C++ based solutions running on a single multicore machine outperforming cluster solutions, where representative case studies are provided by PiCo[14] and WindFlow.[15] Thus, we expect that the efficient use of a multicore machine provides a level of performance suitable for a significant part of stream processing applications.

Stream processing applications have a nature of executing under changing conditions (e.g., workload, input rates, environment) and running for long or infinite time periods.[2] In this context, changes occur while the applications are executing and a configuration that sustained the quality of services becomes instantly suboptimal or unprofitable. Therefore, one way to achieve responsiveness to changes is by applying adaptation actions at run-time.[16] A relevant configuration to be continuously adapted concerns the degree of parallelism, mostly because responsiveness to changes is not possible when a static degree of parallelism is used during the entire execution. However, the aforementioned frameworks and libraries for stream processing on C++ require users/programmers to define a static and unchangeable degree of parallelism.

In addition, finding the best degree of parallelism tends to be complicated and time-consuming, because a programmer would need to run the same program several times to find the optimal configuration. Moreover, the degree of parallelism depends on a series of factors, which are related to the computer architecture and application characteristics. One example of the complexities involved in defining the degree of parallelism is shown by Pusukuri et al.,[17] where each PARSEC application was run several times to find the most performatic configuration. However, Pusukuri et al.'s[17] solution was unable to adapt to the degree of parallelism at run-time, consequently, this solution is not suitable for real-world stream processing.

A potential solution for this specific problem of finding the best degree of parallelism and adapting it at run-time can be using the theory of self-adaptive systems[18,19] applied to stream processing applications. A self-adaptive execution can abstract from programmers/users the specification of the degree of parallelism. Hence, executing in a self-adaptive mode can make stream processing applications more intelligent, which reduces human efforts and assists in error-prone activities by avoiding incorrect configurations. Moreover, self-adaptiveness is a potential way of responding to fluctuations and increasing performance/efficiency.

In this work, we propose an autonomous manager[20] that uses a feedback loop approach to change the degree of parallelism at run-time. Most of the related works are concerned with system utilization, energy efficiency improvements, and the design of self-adaptive techniques to implement predictive strategies,[21-23] which are helpful insights for our research. There are also works that implement reactive strategies.[24-27] By contrast, our goal is to provide an abstraction layer for autonomously managing the degree of parallelism. This abstraction is expected to meet the desired performance requirements, implementing new reactive self-adaptive strategies.

We demonstrated how the degree of parallelism impacts the latency of stream items in a previous work,[28] where we proposed a strategy attempting to manage latency by adapting the degree of parallelism. However, latency is only relevant for specific applications. In Reference 29, we addressed techniques for seamless parallelism configurations in video processing applications. Moreover, in Reference 30 we proposed a solution for reducing the overhead when applying self-adaptive actions. By contrast, the current work focuses on strategies for scenarios where throughput is the most important metric. In addition, here the focus is on encompassing the previous advances in new and generalizable strategies that can enable the generation of self-adaptive parallel code. Noteworthy, here we provide the following scientific contributions:

- Two new reactive self-adaptive strategies to abstract and autonomously manage the degree of parallelism in real-time stream processing applications. Unlike References 24-27, our strategies target a different computer architecture (multicore) and provide higher-level abstractions to programmers. We generate an autonomous runtime system for SPar with the designed self-adaptive strategies. The abstraction supports executions without parameter definition or can be easily customized. Related studies require programmers to have expertise concerning specific parameters such as threshold, target performance, and calibration to use the self-adaptive algorithms for distributed parallel architectures (clusters).

- Compiler transformation rules for generating an autonomous runtime system that supports the two new reactive self-adaptive strategies.
- Evaluation of our solution in favor of parallel programming abstractions using a domain-specific language for stream parallelism (named SPar). This solution allows more programmers to implement new autonomous stream processing applications with parallelism support without significant effort.
- Experiments for evaluating the performance of our solutions. We used three real-world applications, different input workload trends, and two different machine architectures.

Our article is organized as follows. Section 2 presents the context of SPar DSL. Section 3 describes aspects regarding the implementations and transformation rules in Section 4. Then, Section 5 shows experimental results. Section 6 describes the related studies addressing stream processing applications and supporting an adaptive (e.g., elastic or reconfigurable) degree of parallelism. Finally, Section 7 concludes this article.

2 | SPAR OVERVIEW

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.[9] The large number of stream processing applications available represent a significant part of current computing systems. However, a significant part of stream processing applications are still sequential (they do not run in parallel).[9] This is the primary reason that programmers face a trade-off between coding productivity and performance. Thus, introducing parallelism tends to increase the programming effort because of the need to rewrite code and the demand for handling architectural/low-level aspects.

Considering the inherent challenges of high-level parallelism abstraction in stream processing applications, SPar[1]was specifically designed to simplify the stream parallelism exploitation in C++ programs for multicore systems.[9] It offers a standard C++11 annotation language to avoid sequential source code rewriting. SPar also has a compiler that generates parallel code using source-to-source transformation technique. SPar uses FastFlow[31] as its runtime library (described in Subsection 2.2), where all low-level parallel programming advanced concepts and implementations (scheduling, load balancing or parallelism strategies) are resolved by SPar's compiler (Section 2.3).

2.1 | SPar language

Attributes can be used in C++ to express parallelism in code annotations.[32] In this vein, SPar's language is composed of five attributes to express the key properties of the stream parallelism. Listing 1 is showing the use of the attributes in the source code annotation. The `ToStream`attribute represents the beginning of a stream region, which is the code block between the `ToStream` and the first `Stage` (lines 1 and 2). The `ToStream` attribute therefore labels where a stream parallel region starts in a given program. The `Stages` are defined inside the `ToStream` region to label the computing phases where stream items will be processed, like an assembly line. Usually, stream processing applications will never end like in Listing 1, but the users may want to finish the application at some point during the execution. They can do so by introducing a stop condition in the `while` loop as well as introducing an `if` condition before the first stage that breaks the current loop, which can be any kind of loop from the host language.

```
1  [[spar::ToStream]] while(1){
2   i = read_item();
3   [[spar::Stage,spar::Input(i),spar::Output(i),spar::Replicate(n)]]
4   {
5    i = filtering(i);
6   }
7   [[spar::Stage,spar::Input(i)]]{
8   write_item(i);
9   }
10 }
```

Listing 1: Demonstrative example of the SPar language

[1]SPar's home page: https://gmap.pucrs.br/spar

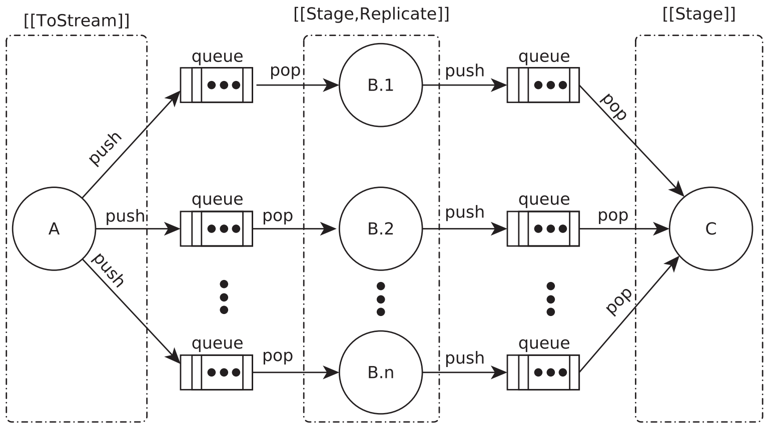


FIGURE 1 SPar runtime: Activity graph and communication queues

Once the `Stage` and `ToStream` are annotated, the `Input` or `Output` attributes are inserted to define the input and output data dependencies. The attribute arguments can be one or more variables from different data types, which label the stream items that will be consumed or produced by a given region. Finally, the `Replicate` attribute may be inserted in the attribute list of `Stage` to define the degree of parallelism of that region. The argument of this attribute is the degree of parallelism (the number of stage replicas), which can be a constant integer number or variable. SPar is not able to automatically manage stateful operations. Thus, only `Stage`s with stateless operations can actually be replicated without any user intervention.

2.2 | SPar runtime

The SPar compiler performs source-to-source transformations by generating calls to the FastFlow library. The compiler interprets the annotations in the source code, performs semantic analysis, and applies transformation rules based on stream parallel patterns.[9] The outcome is that from the high-level SPar annotations, parallel patterns such Farm, Pipeline or a combination of Pipeline with Farm stages are generated using the FastFlow algorithmic skeleton interface. The parallel code generated is what we call the SPar runtime.

Figure 1 depicts the activity graph and communication between stages of the SPar runtime system generated from the example of Listing 1. This provides an overall idea how it works. Note that the `Replicate` attribute applies the replication role over the `Stage`. Each replicated stage has its own input and output lock-free queue, where this term refers to an operation that eventually completes after a given number of steps.[33,34] The first stage is actually the code left inside a `ToStream` region, which generates stream items for the subsequent stages. It is also responsible for scheduling items, which by default is round-robin. However, users may need an on-demand scheduler that is made possible through a compiler flag in the SPar compiler (`-spar_ondemand`). When this flag is present, the queues size is one (stream item). Thus, as soon as one stream item is popped by the current stage, another will be pushed from the previous stage.

In the SPar runtime, the default configuration is the stages actively trying to push or pop stream items from the queues. If the queue is full or empty, the stage thread remains in a loop, trying to perform push or pop until it finally succeeds. Every time that a given stage fails in perform push or pop, the stage generates a push or pop lost event. This may generate an extra overhead for coarse-grain computations. Therefore, users may set the SPar runtime to behave in a blocking mode through the *spar_blocking* compiler flag. In this case, the stage thread will not stay in a loop, it will wait until it can perform push or pop in the shared queue.

Another important concern is to preserve the order of stream items, the ordering is implemented in stage after the replicated one. SPar is able to automatically handle out-of-order stream items in the last stage when `-spar_ordered` compiler flag is defined.

2.3 | SPar compiler

A relevant part of SPar is its code generation, which is provided by a compiler tool. The SPar's compiler is powered by CINCLE (Compiler Infrastructure for New C/C++ Language Extensions)[2]. In short, CINCLE applies a parser step

[2]Technical implementation details can be found in Reference 35.

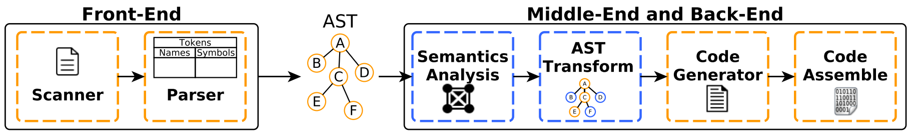


FIGURE 2 The SPar's compiler compilation flow. Extracted from Reference 9 [Color figure can be viewed at wileyonlinelibrary.com]

following the standard C++ grammar with an interface that creates the Abstract Syntax Tree (AST). As an illustrative example, Figure 2 shows the generic compilation flow, where blue boxes correspond to SPar implementations and the orange boxes relate to the CINCLE modules provided to generate SPar compiler. Note that semantic analysis and the AST transformations are those implemented to support SPar language and parallel code generation.

In this work, new transformation rules were designed. Nothing has been changed at the language level. Therefore, only the *AST Transform* module was changed to implement the new transformation rules, which generates our self-adaptive strategies to configure the number of replicas autonomously. In Section 4.2, we present the new transformation rules and the corresponded code generation.

3 | SELF-ADAPTIVE DEGREE OF PARALLELISM

This section presents the design goals, strategies, and their characterization for providing a ready to use and self-adaptive degree of parallelism.

3.1 | Design considerations

Heinze et al.[27] proposed a set of **requirements** for autoscaling strategies for elastic data stream operators. The requirements for scaling data stream processing applications can be used to design the requirements in our specific context.

Workload independence is a characteristic of a strategy that adapts the degree of parallelism, which should be independent from the workload characteristics. A strategy should be *adaptive* by continuously adapting on-the-fly according to the workload and the fluctuations of different processing phases. Moreover, *transparency* is a characteristic derived from Reference 27 and *configurability* (be easy to configure for users). However, as we are working with parallelism abstraction, a strategy is expected to encompass transparency by adapting at run-time. Moreover, we envision to achieve configurability with strategies that work without manual user intervention.

Computational feasibility is also needed concerning the internal algorithm used by an adaptive strategy, such algorithm has to be simple enough and effective to run and perform on-the-fly decisions. Finally, a *low overhead* is aimed as the strategy should not be too complex, consume too many resources, or cause significant overhead. The overhead is expected to be negligible compared with static executions.

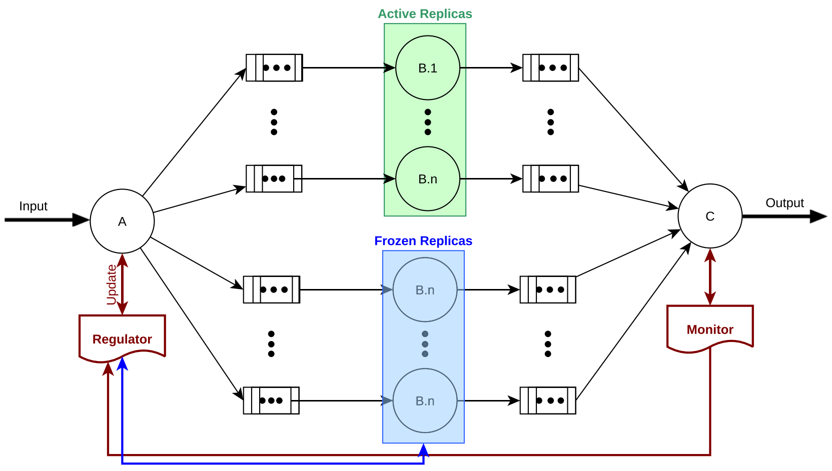
A set of **properties** and interests can be included when designing adaptive strategies. In this work, we consider **SASO** (Stability, Accuracy, Settling time, and Overshooting) properties.[19] Related works[23,24] have also used SASO properties when designing their solutions.

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".[19] *Accuracy* is used in related studies to search for the degree of parallelism that optimizes performance.

A system is expected to present *short settling times* by quickly responding to changes and reaching an optimal state.[30] When the load fluctuates, the response to an event should be rapid and maintain a service level objective. A strategy should also avoid *overshooting* by only using the necessary amount of resources. Overshooting is prevented in our solution by reducing the number of replicas when the performance goal is satisfied. One may argue that the maximum replicas should be used for increasing the performance. However, more replicas do not always increase performance. In addition, in stream processing, more performance does not necessarily improve the user experience.[21] Avoiding overshooting improves the system efficiency, which can reduce energy consumption, lower costs (e.g., in pay-per-use models like Cloud Computing), and leave more computational power available to other users or applications.

Considering the aforementioned design principles, a solution was proposed for adapting the number of replicas at run-time. The workflow model was based on a closed-loop system from control theory,[19] where in Figure 3 is presented a

FIGURE 3 Throughput—regulator and monitor [Color figure can be viewed at wileyonlinelibrary.com]



representation of the designed solution covering the relevant entities called regulator and monitor. In this workflow, the monitor is an entity hosted in the last stages that collects quality of service indicators (e.g., throughput). The first stage can be broadly viewed as the task scheduler, which hosts the parallelism regulator entity. The regulator is designed for periodically executing in the form of iterations and deciding whether or not to change the number of replicas. Moreover, the regulator has a maximum value for the number of replicas, which is defined according to the machine's CPUs resources capacity.

It is important to highlight that the adaptivity scope of the strategies addressed in this study is limited on some aspects. The adaptive strategies proposed here are meant to work on stateless replicated (a.k.a fissioned) stages. In addition, we based them on the on-demand scheduling in order to deal with a finer granularity, rapid response to load fluctuations, and to achieve better load balancing. Regarding the threads creation and CPUs affinity optimizations, the SPar's runtime library (FastFlow) already optimizes the placement of threads. Our solution assumes that the runtime library will do its best in terms of finding CPUs affinity.

3.2 | A strategy for abstracting user-defined parameters

The workflow model showed in Figure 3 was used for implementing a strategy for dynamically adapting the number of replicas. *Transparency* and *configurability* requirements are supported as the strategy proposed works without user-defined parameters (WUDP). The execution starts using a close to optimal (considering the majority of stream processing applications) configuration for the number of replicas equal to the physical cores value, then during the execution, this configuration self-adapts when necessary.

The core of the proposed strategy is the decision making performed by the regulator, represented in Algorithm 1. A simple heuristic was implemented for taking adaptation decisions, where the current and previous iterations' performance are analyzed. The throughput is continuously monitored and, if the throughput is lower than one of the previous

Algorithm 1. 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
 8:         end if
 9:     end while
10: end procedure
```

Algorithm 2. Throughput Monitor

```
1: procedure MONITOR( )
2:     while true do
3:         Sleep(timeInterval)                              ⊳ Wait until the next iteration
4:         $numOfProcTasks = currentTask - lastProcTask$
5:         $Throughput = numOfProcTasks/timeInterval$
6:     end while
7: end procedure
```

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. The decision of the strategy relies on the previous 3 iterations to be sensitive enough to the workload fluctuations and because when the execution starts there is no prior information available to be analyzed. Thus, the outcome of Algorithm 1 is one of the three options: increase the number of replicas, decrease the number of replicas, or maintain the same number of replicas. The decision is made periodically, where the number of replicas can be changed transparently while a given program is running. On each adaptation step, several replicas can be activated or suspended. Here, the number of replicas changed on each step is called a scaling factor (SF). For instance, a SF 2 means that two replicas can be added or removed at once.

The minimum and maximum number of replicas that a given parallel stage can use must also be determined. 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. We expect that the parallelism regulator can detect the CPUs capacity from the running machine using mechanisms provided by the runtime library used.

The throughput value (tasks/second) is calculated in the last stage. Listing 2 shows a high-level representation of the monitor implementation from the entity showed in Figure 3. The monitor calculates the number of tasks processed in the current iteration 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 time taken. The last stage gathers the tasks and also measures the throughput. The throughput rates are then stored and accessed by the regulator. This flow feeds the regulator with information to decide if an adaptation is required for deciding if the current parallelism configuration must be adapted.

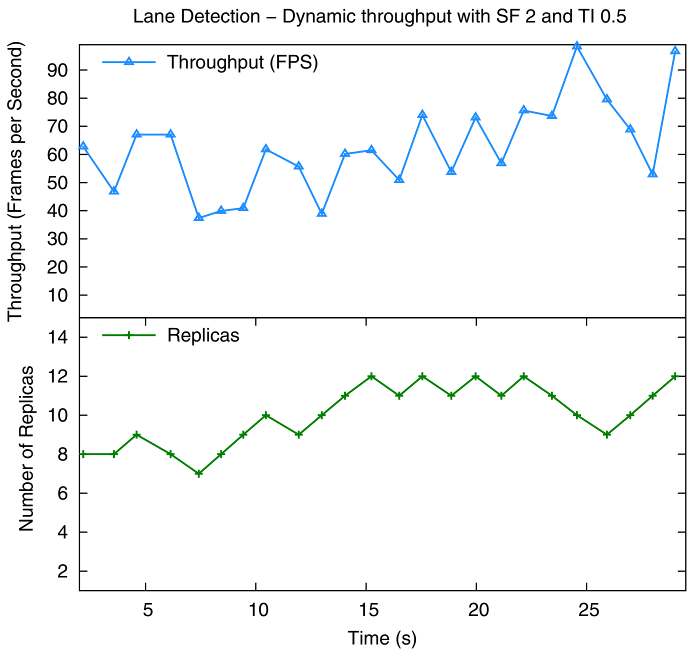
The decision making of the proposed strategy was characterized by running realistic experiments. The self-adaptive part was manually included in a parallel version (using SPar) of the Lane Detection video application, additional information about the application and tested Machine 1 is presented in Section 5. The main goal of this experiment was to evaluate the effectiveness of the adaptive strategy in adapting the degree of parallelism. 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. In our primary tests, we used a video file as an input to simulate a typical execution of a video streaming application. Moreover, 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. In this video application, a task is a video frame.

Figure 4 presents the results of the throughput and the number of replicas used during execution of the strategy WUDP. This experiment was configured with a representative SF of 1 replica and monitoring time interval (TI) of 1 s. It is notable that the throughout oscillates during the execution, which is mainly caused by workload trend of the application's input. Consequently, the number of replicas is not always stable since it varies during the execution. 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. Figure 4 shows that 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 Section 5.

3.3 | A strategy for user defined target throughput

In the previous section, we presented a strategy for adapting the number of replicas WUDP, where it was demonstrated the possibility to adapt the parallelism degree and optimize the application execution at run-time. Some application

FIGURE 4 Adaptive strategy without user-defined parameters [Color figure can be viewed at wileyonlinelibrary.com]



Algorithm 3. 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
8:          end if
9:      end while
10: end procedure
```

programmers may prefer not to specify the performance goal (performance concerns are abstracted) while others may prefer to set the specific desired performance goal for a given application. It is expected to be a trade-off between the flexibility for defining parameters and a complete transparent/abstracted execution. Defining performance parameters according to specific scenarios increases the flexibility at the price of additional complexities, where performance expertise is required from the application programmers. On the other hand, a completely abstracted execution is expected to increase the abstraction level by reducing complexities but tends to result in less flexibility as well as in lower performance in our approach. From a practical perspective, the performance (e.g., throughput) is usually higher when the precise value is set. This allows to respond quicker to reach the optimal configuration as the self-adaptive strategy can compare the actual performance to the goal and apply adaptations when necessary.

Here, we propose another strategy that aims at providing the flexibility for the application programmer to define parameters and performance indicators. The goal is to empower the application programmers allowing them to choose the level of abstraction that they need. Hence, we implemented a strategy that adapts the degree of parallelism based on the user defined throughput goal. The design follows the model described in the previous section. Here, the core difference lays in the regulator entity, represented in Algorithm 3. As shown in lines 4–7, the decision whether to change the degree of parallelism is based on the target (expected) and measured (actual) throughput. In addition, this strategy uses a throughput monitor in the last stage that calculates the actual throughput, shown in Algorithm 2.

The number of replicas is changed by suspending active replicas if the throughput is higher than the target or activating suspended replicas. Furthermore, the throughput tends to oscillate and unnecessary adaptations should be avoided. In the related works (Section 6), each approach is specific and implements the strategy considering its scenario, no consensus is found regarding to when the degree of parallelism is expected to be reduced. 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.[28] The threshold is used in line 6 of algorithm 3.

In order to characterize this strategy, the same steps from the previous described strategy were followed. Moreover, 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 addition, it is relevant to emphasize that this strategy allows one to test different conditions.

Figure 5(A) shows a simulated scenario where the target throughput (TT) is dynamic during the execution, starting with an arbitrary value of 40 tasks as the TT. This test was performed in order to evaluate the strategy under a real-world condition where the performance goal changes. In this simulation, when the execution time reached 8 s, the TT was changed to 18. At 16 s, the performance goal was increased to 50. When the execution reached 30 s, the TT was changed to 70. In Figure 5(A), the SF was 2, adding or removing 2 replicas on each adaptation. SF 2 reacts faster to changes, which could results in a short settling time at the price of less stability. Regarding replicas adaptation, Figure 5(A) shows that the self-adaptation strategy was able to respond to changes and pursue the different throughput goals.

While some stream processing applications can present changes in the target performance, others have a continuous demand for a specific performance. However, it does not mean that the execution will be without fluctuations since the input workload and the environment can oscillate. In addition, the demand for resources can vary following the workload trend. One possible customization to respond to load fluctuations is to adapt the number of replicas. In the experiments with a static TT, we are only presenting results with 0.5 as a TI between each iteration. This decision is due to paper space constraints and in previous results is shown that a short TI caused no additional overhead.

In Figure 5(B), we present an experiment with a static TT of 60 frames per second and SF of 2, using the same input video from previous experiments. In the tested machine, the number of replicas used varies from 8 to 14. As the number of replicas impacts the actual throughput, it was decided to start the execution using half of the total available cores (with hyperthreads). In previous experiments, it was possible to note a need to increase the number of replicas right after the execution starts. The load from the input file caused most of the throughput fluctuations. It is noteworthy that in some specific instances, even using the maximum number of replicas, it was not possible to achieve the TT. However, it was not caused by the adaptive strategy, but is a consequence of the machine's limited processing capability.

The strategy presented here is intended for providing flexibility to users with performance expertise. This strategy assumes that the throughput goal set by the application programmer is achievable provided that the application has

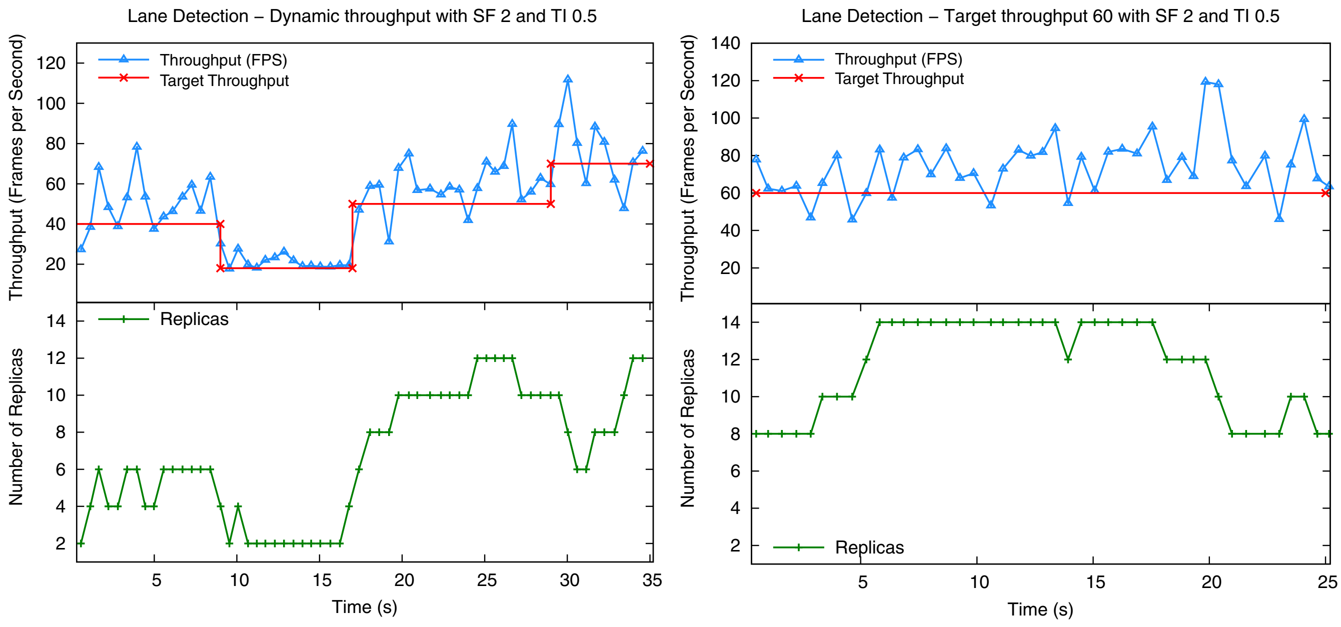


FIGURE 5 Target throughput strategy [Color figure can be viewed at wileyonlinelibrary.com]

enough data to be processed, which characterizes a suitable input task rate.[20] However, in case the application programmer sets a wrong throughput goal that is higher than the input rate, the input rate would limit the maximum achievable performance. In such a scenario, the regulator will increase the number of replicas as an attempt to increase the performance, which would consume more resources without necessarily achieving performance gains. This suboptimal scenario caused by wrong parameters defined by the user can be mitigated by including an additional monitor in the task scheduler for measuring the input rate. Then, in case the TT is higher than the input rate, the regulator could change the TT to the value of the input rate. This additional monitoring entity integrated with the regulator would potentially impact in the performance. Consequently, implementing and validating such a mechanism is left as a future feature to be included and validated.

3.4 | Strategies' remarks

Here, we discussed the motivations and requirements for an adaptive degree of parallelism and the implemented strategies were characterized using the Lane Detection application. The first strategy that we implemented adapts the number of replicas without the need to set a target performance. Because many application programmers do not have performance expertise, this strategy aims to maximize the throughput without a user-defined parameter (Section 3.2). The advantage of more abstraction comes at the price of less flexibility.

Furthermore, Section 3.3 presented a solution that requires from the user the definition of a TT, which is used as a configuration to optimize the execution. The experiments revealed that the target performance might vary during execution as well as the demand for resources.

Considering the requirements for an adaptive degree of parallelism (Section 3.1), the strategies also met the goals for adaptive and transparent executions. The overhead and computational feasibility as well as the workload independence will be evaluated in Section 5. The overhead and feasibility are tested by evaluating the performance of running applications, and workload independence is shown by using different applications and their inputs.

The desired SASO properties were also met by the strategies. The strategies are also stabilized by using TIs between iterations and the short settling times are pursued by reducing the TIs and using faster scaling configurations. In addition, overshooting is avoided by decreasing the degree of parallelism when suitable.

4 | SELF-ADAPTIVE NUMBER OF REPLICAS IN SPAR

In previous section, we described and characterized two strategies for different user profiles. Here, we present how these strategies can be integrated into an existing DSL (SPar) for generating a self-adaptive parallel code. SPar was studied to find out techniques and mechanisms to change the number of replicas at run-time. Considering that stream processing requires an effective mechanism to change a program's execution without needing to recompile or rerun, the strategies were implemented using low-level calls to the runtime library that change the status of the replicas (active or suspended). These low-level mechanisms for adapting the status of the replicas are available in FastFlow that is the runtime library used by SPar, described in Section 2.2.

Considering the design considerations and requirements (Section 3.1), the adaptive entities were designed to execute as functions embedded to the runtime library for reducing the potential overheads. For instance, the regulator entity is implemented in the first stage of the SPar runtime (see Figure 3) to avoid the need to restart the application when changing the number of replicas as well as to avoid the need of creating an additional thread. Regarding the practical use of the monitor entity, it is embedded within the last stage of the execution workflow.

4.1 | Existing SPar rules

Transformation rules are used within SPar to generate parallel code. These rules are related to the SPar runtime described in Section 2.2 and according to the chosen parallel patterns. As described in Section 2, SPar transforms an annotation sequence into parallel code by composing parallel patterns,[36,37] which are represented using functional semantics.

According to Reference 9, 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 tasks to the workers, the farm worker (*B*) that computes the tasks, and the farm collector (*C*) that gathers results. 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)]]{
  f2();
                  ⇒ farm(A(f1), B(f2), C(f3));

 }
 [[spar::Stage]]{
  f3();
 }
}
```

Listing 2: Example of SPar's source-to-source transformation rule for generating a Farm

Rule 1 shows an existing rule relevant to this work, representing 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 emitter task scheduler (*A*(…)). Furthermore, `f2` (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*. In addition, Rule 1 has another `Stage` annotation, which is another sequence of commands represented by `f3`, which is assigned to *C* that gathers the tasks.

4.2 | Self-adaptive code generation

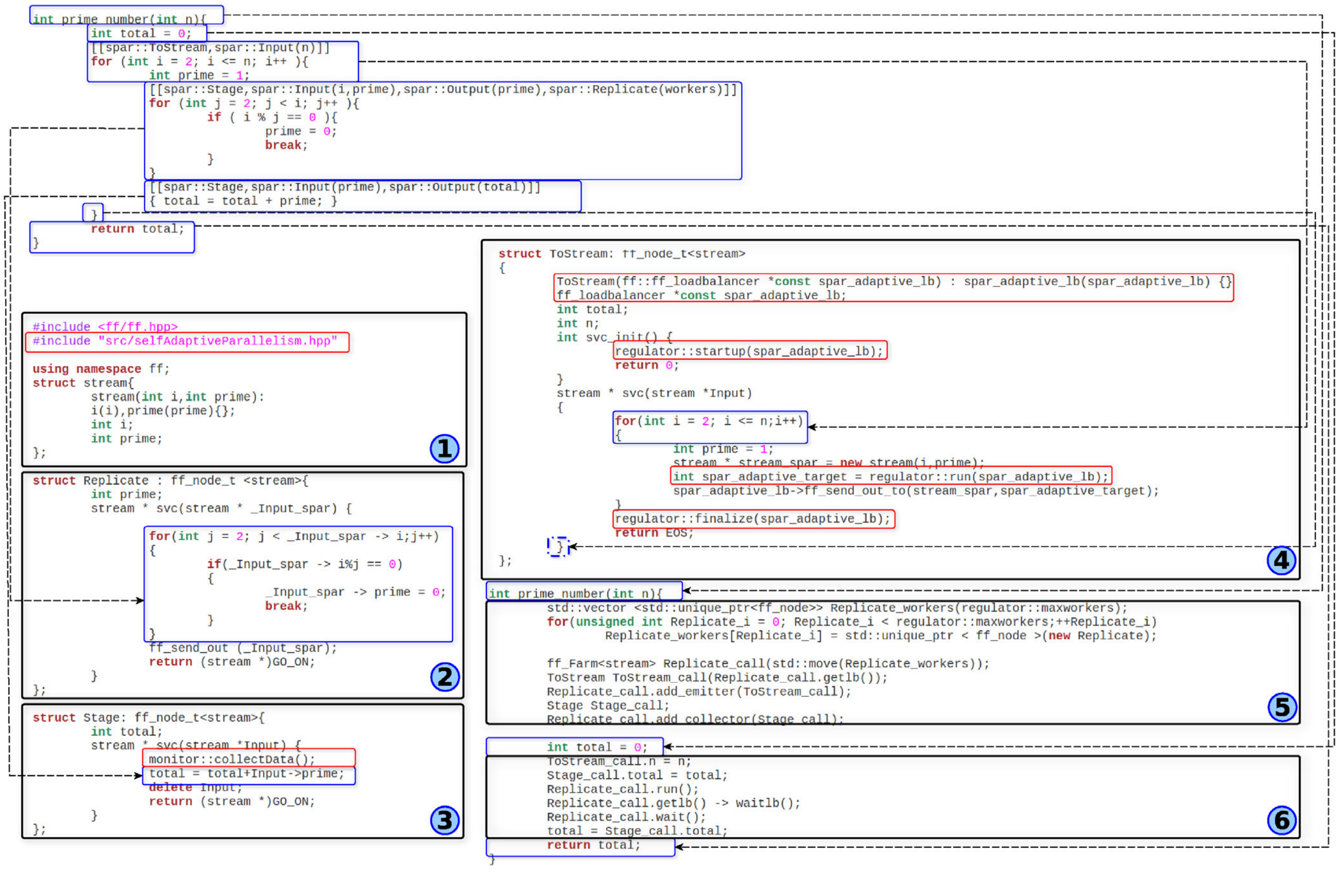
From the SPar's language standpoint, the support for self-adaptive executions relates to generating additional code blocks, which is supported by the SPar compiler (Section 2.3). Moreover, the proposed strategies were implemented in the form of a C++ header file, which enables SPar to include the files and only generate the code for the instrumentation (call the functions).

We use transformation rules for representing the code generation of the self-adaptive strategies along with the SPar regular code. In addition to the existing SPar rules, we designed a new rule to cover the adaptive degree of parallelism for the SPar DSL. Rule 2 is similar to Rule 1 representing a code generation for a Farm. However, in rule 2 the code generated for *A* and *C* also has the parts of the adaptive strategies. The first part of an adaptive strategy is the regulator method in such a way that *regulator*() is implemented as a new method of class *A*. For instance, this method can cover the decision making for the application's TT strategy (showed in Listing 3). The *regulator*() method is called to manage in *A* the number of replicas. *C* has the code implementing the performance monitoring through the method *monitor*() represented in Listing 2.

Importantly, the *regulator*() and *monitor*() have interactions where the *regulator*() periodically accesses a shared queue that contains data collected by the *monitor*(). Consequently, this workflow enables the *regulator*() to decide whether the number of replicas needs to be adapted. Considering that the regulator and monitor run only periodically as sequential and asynchronous entities, potential synchronization or race condition issues are avoided. Nevertheless, the only minor issue that could occur by the combination of unpredictable and rare events is that the regulator may need one more iteration to get the most updated statistics collected by the monitor.

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

Listing 3: SPar's source-to-source transformation rule for generating a Farm with an adaptive number of replicas



```
int prime_number(int n){
    int total = 0;
    [[spar::ToStream,spar::Input(n)]]
    for (int i = 2; i <= n; i++ ){
        int prime = 1;
        [[spar::Stage,spar::Input(i,prime),spar::Output(prime),spar::Replicate(workers)]]
        for (int j = 2; j < i; j++ ){
            if ( i % j == 0 ){
                prime = 0;
                break;
            }
        }
        [[spar::Stage,spar::Input(prime),spar::Output(total)]]
        { total = total + prime; }
    }
    return total;
}
```

```
#include <ff/ff.hpp>
#include "src/selfAdaptiveParallelism.hpp"

using namespace ff;
struct stream{
        stream(int i,int prime):
        i(i),prime(prime){};
        int i;
        int prime;
};
```

```
struct Replicate : ff_node_t <stream>{
        int prime;
        stream * svc(stream * _Input_spar) {

                for(int j = 2; j < _Input_spar -> i;j++)
                {
                        if(_Input_spar -> i%j == 0)
                        {
                                _Input_spar -> prime = 0;
                                break;
                        }
                }
                ff_send_out (_Input_spar);
                return (stream *)GO_ON;
        }
};
```

```
struct Stage: ff_node_t<stream>{
        int total;
        stream * svc(stream *Input) {
                monitor::collectData();
                total = total+Input->prime;
                delete Input;
                return (stream *)GO_ON;
        }
};
```

```
struct ToStream: ff_node_t<stream>
{
        ToStream(ff::ff_loadbalancer *const spar_adaptive_lb) : spar_adaptive_lb(spar_adaptive_lb) {}
        ff_loadbalancer *const spar_adaptive_lb;
        int total;
        int n;
        int svc_init() {
                regulator::startup(spar_adaptive_lb);
                return 0;
        }
        stream * svc(stream *Input)
        {
                for(int i = 2; i <= n;i++)
                {
                        int prime = 1;
                        stream * stream_spar = new stream(i,prime);
                        int spar_adaptive_target = regulator::run(spar_adaptive_lb);
                        spar_adaptive_lb->ff_send_out_to(stream_spar,spar_adaptive_target);
                }
                regulator::finalize(spar_adaptive_lb);
                return EOS;
        }
};
```

```
int prime_number(int n){
        std::vector <std::unique_ptr<ff_node>> Replicate_workers(regulator::maxworkers);
        for(unsigned int Replicate_i = 0; Replicate_i < regulator::maxworkers;++Replicate_i)
                Replicate_workers[Replicate_i] = std::unique_ptr < ff_node >(new Replicate);

        ff_Farm<stream> Replicate_call(std::move(Replicate_workers));
        ToStream ToStream_call(Replicate_call.getlb());
        Replicate_call.add_emitter(ToStream_call);
        Stage Stage_call;
        Replicate_call.add_collector(Stage_call);

        int total = 0;
        ToStream_call.n = n;
        Stage_call.total = total;
        Replicate_call.run();
        Replicate_call.getlb() -> waitlb();
        Replicate_call.wait();
        total = Stage_call.total;
        return total;
}
```

FIGURE 6 Example of SPar's self-adaptive code generation [Color figure can be viewed at wileyonlinelibrary.com]

In addition, the compiler and runtime library can decide between the two designed strategies by the compilation flag exemplified below:

```
-spar_adaptive-throughput [target-throughput]
```

This flag accepts the `target-throughput` in tasks/second. The argument is intended to provide flexibility with relevant customization for the user/application programmer. In case a TT is specified, the number of replicas will be regulated using the TT strategy. On the other hand, if the parameters were not defined, the strategy WUDP is used.

In order to illustrate how the adaptive code generation works with SPar's source-to-source transformations, we consider a trivial application that calculates the prime numbers as a code example. In Figure 6, we show six steps of the code generated by the SPar compiler, where at the top is showed the code with SPar's annotation scheme. SPar's code generation follows some specific steps.[9] First, the compiler detects input and output dependencies for generation the step block ①. Then, the stages are built within blocks ② and ③ we must properly manage data, taking care of the associated input and output dependencies. The next step is ④ that creates the emitter (task scheduler) for the stream region. Finally, an example of a creation of a farm pattern is showed in block ⑤ and in ⑥ the variable values are updated as well as the Farm starts running.

In this example, SPar generates a parallel code supported by the FastFlow runtime [3]. Importantly for supporting the generation of self-adaptive code, the code parts highlighted in the color red in Figure 6 are changes or additional parts for enabling self-adaptive executions using the proposed strategies. The self-adaptive code is generated using the already described Rule 2. First, it is necessary to include the header files, which is where the monitor and regulator entities are actually implemented. Second, in the code generated corresponding to the emitter (A), the first step is to declare the *ff_loadbalancer* that is the runtime mechanism that controls the status of replicas. Third, it is necessary to call the regulator's function that prepares the system for starting the execution of the applications. Then, the regulator's *run*()

[3]Additional details concerning the SPar's default code generation can be found in Reference 9.

function returns an ID of the replica to send the task and periodically verifies if the number of replicas has to be changed or not. In case a change is needed, the *run*() function transparently interacts with the runtime library and updates the configuration. Finally, when there are no more tasks to be processed, the regulator's *finalize*() function is called for a cleaning up step that suspends the replicas to enable the runtime library to end the execution.

Moreover, in the stage represented in step ③ of Figure 6 that corresponds to *C*, a single extra code line is necessary for calling the monitor entity, which collects relevant statistics and feeds the regulator. Although here we are describing how the solution for a self-adaptive number of replicas is integrated in SPar and its runtime library, these additional lines can be also manually instrumented/added to the other parallel codes.

5 | PERFORMANCE EVALUATION

Here, we introduce a set of experiments aimed to evaluate the strategies and their mechanisms for an adaptive degree of parallelism. The strategies are tested under different supported configurations and these results are compared with the hand-coded static parallel applications. We ran a set of experiments to evaluate the effectiveness of the adaptive strategies using the throughput as a performance metric. Throughput was calculated based on the number of processed items at a time. We also present the final throughput, which was derived by taking the total number of processed items in a given execution divided by the total time. Each execution was repeated 10 times and the results presented are the arithmetic means. The standard deviation is also shown in the graphs. Section 5.1 describes the tested applications and Section 5.2 shows the test environment. Then, in Sections 5.3 and 5.4 the performance is evaluated with real-world applications.

5.1 | Real-world stream processing applications

Our strategies were evaluated by implementing them in existing parallel stream processing applications generated by SPar. In fact, real-world applicability was the key criterion used to select the applications. We also selected them based on different characteristics and QoS requirements. Therefore, some applications were chosen perform simple processing and outputting results under a regular workload trend (e.g., Lane Detection), while other applications can perform more complex processing under an irregular (variant) workload trend (e.g., Person Recognizer). The following applications were tested:

- **Lane detection:**is an application used on autonomous vehicles to detect road lanes maintaining the car on the road. This is performed by reading a video feed from a camera. The lane is detected through a sequence of operations, which are described in Reference 38. The parallel implementation of this application is a pipeline composed of three stages where the second stage, which is stateless, is replicated. Lane Detection was tested using two input files in order to test whether different workload trends result in different performance in the adaptive strategies.
- **Person recognition:** This application is used to recognize people in video streams. It starts by receiving a video feed and detecting the faces. The faces that are detected are then marked with a red circle, and then compared with the training set of faces. When the face comparison matches, the face is marked with a green circle.
- **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.[39,40] The compression has a workflow based on entities. The first step is completed by reading the input, followed by a compression stage, and finally writing to the output channel. The compression mode can be viewed as a pipeline with the read stage, the parallel compression stage, and the sequential writing stage. The parallel version of Bzip2 (Pbzip2) using SPar is shown in Reference 40, and this parallel version with the static *Replicate* attribute. We implemented the strategies for an adaptive degree of parallelism in this parallel version.

5.2 | Environment of tests

We used two machines to test the performance and decision making of the self-adaptive strategies in different applications. The first machine used (**Machine 1**) was a multicore system with 2 Sockets Intel(R) Xeon(R) CPU 2.40 GHz (8 cores-16 threads), with a memory of 16 GB - DDR3 1066 MHz. **Machine 2** where the same experiments were repeated is a more robust one equipped with memory 32 GB - 2133 MHz and a dual socket Intel(R) Xeon(R) CPU 2.40 GHz

(12 cores- 24 threads). In both machines the OS used was Ubuntu Server 16.04 and the programs were compiled with G++ (5.4.0) with the -O3 flag. In addition, this environment was dedicated for these experiments, thus all other workloads were not running at the same time.

5.3 | Performance of strategies' configurations

Flexibility can be supported in the adaptive strategies by enabling the definition of several configurations and parameters. A relevant aspect of the configuration is the SF, which is how many threads/replicas are added or removed when adjusting the degree of parallelism. In the literature, the most commonly used SF value is 1 (threads/replicas). Our adaptive strategies are tested with SFs 1 and 2.

A challenge related to an adaptive degree of parallelism is *when* to change the execution. The most common approach is time-driven, where a periodical execution is performed every TI, which can be viewed as a sampling time running after a given sleep TI. Authors of References 21,23,24,41 used TIs ranging from 0.1 to 5 s. Based on our empirical tests, it was determined that 0.1 s is a too low interval and 5 s are too high. Our adaptive strategies are therefore tested with TIs of the sleep time between each run of 0.5 and 1 s. This configuration was chosen aiming to avoid excessive iterations, while at the same time maintaining a correct level of sensitivity for being responsive to workload fluctuations. Consequently, we tested the impact of the different TIs in the adaptive strategies.

Figure 7 shows the average throughput from 10 executions using the different configurations of the adaptive strategies in Machine 1. In general, the configurations achieved a similar performance. In some specific cases the SF 2 (goes up and down faster) and a TI of 0.5 s configuration is more sensitive to application fluctuations and for adapting faster the degree of parallelism. The configuration with TI 0.5 s and SF 1 is also responsive to fluctuations.

The performance difference between the configurations tends to be small. In case a transparent execution is desired, the SF of 2 and TI 0.5 s are more responsive. Figure 8 provides additional results with the same configurations tested in a more robust machine. It is notable in Lane Detection application that in both strategies the best configuration is TI 0.5 and SF 2, such an outcome is justified by the fact that this configuration has a lower settling time. In this case, reducing the settling time allowed to reach earlier the maximum throughput with the best degree of parallelism. Despite some minor differences between the configuration in Person Recognizer and Pbzip2, the overall performance trend of the configurations in Machine 2 is similar to the results seen in Figure 7 from Machine 1. Importantly, it is possible to conclude that the configurations have an impact on the responsiveness and settling time of the self-adaptive strategies, which motivates the support of these customizations for users expert on performance.

It is important to note that in the results presented in Figures 7 and 8 we are not comparing which configuration yields the best performance. Instead, we are evaluating how the adaptive strategies behave under these different configurations and how responsive they are in adapting the degree of parallelism. Noteworthy, these results are compared with baseline static executions in Figures 9–11, which are discussed in the following section.

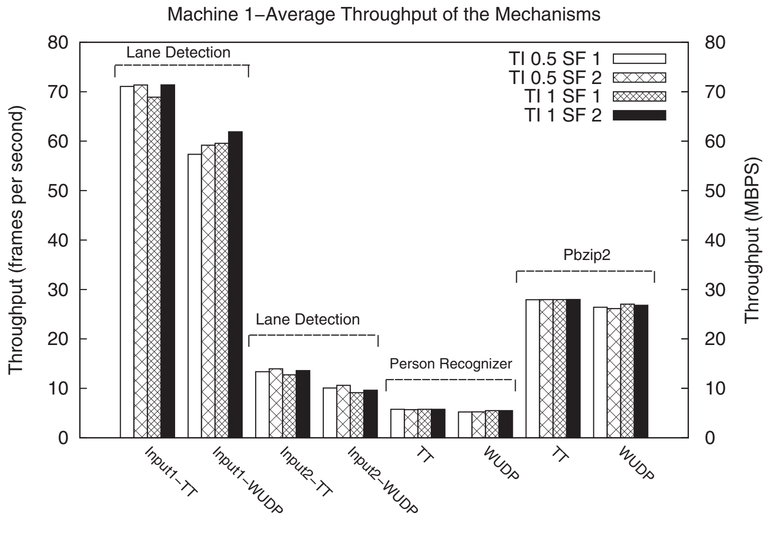


FIGURE 7 Average throughput—configuration of strategies in Machine 1

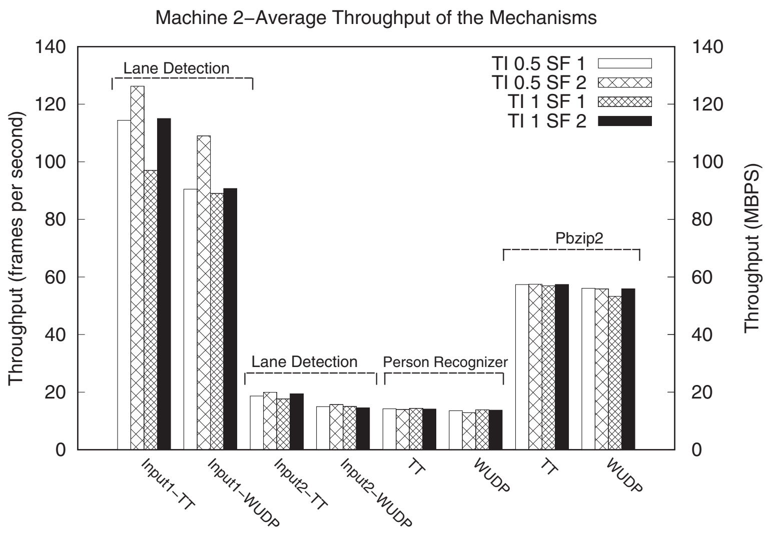


FIGURE 8 Average throughput—configuration of strategies in Machine 2

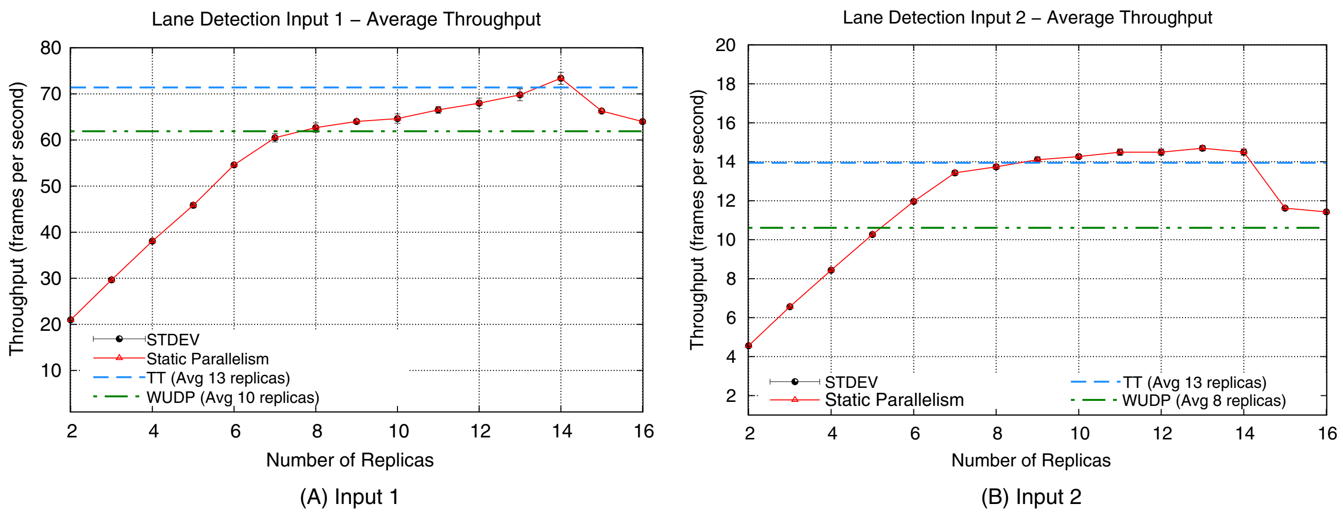


FIGURE 9 Average throughput of lane detection in Machine 1 [Color figure can be viewed at wileyonlinelibrary.com]

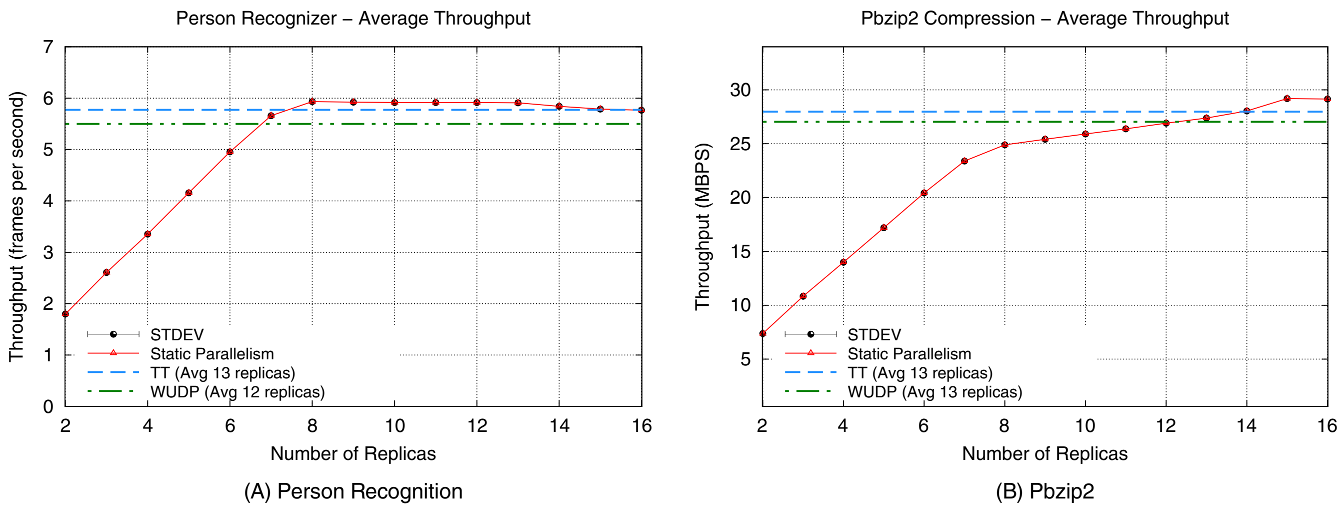


FIGURE 10 Average throughput in Machine 1 [Color figure can be viewed at wileyonlinelibrary.com]

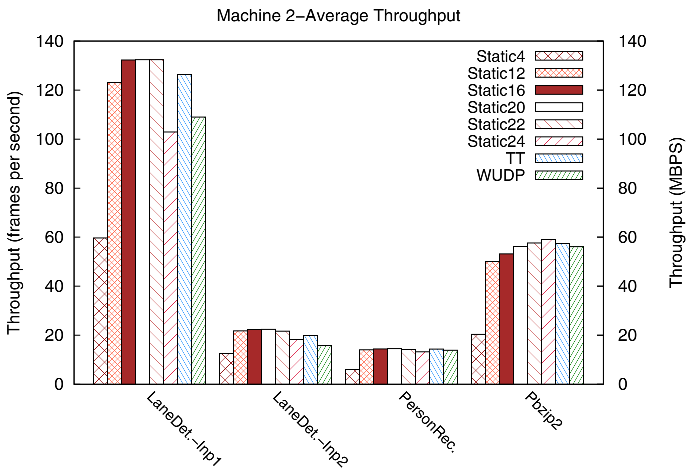


FIGURE 11 Machine 2—summary of average throughput [Color figure can be viewed at wileyonlinelibrary.com]

5.4 | Performance compared with static degree of parallelism

This section evaluates the adaptive strategies comparing them with regular parallel executions that use a static (a.k.a. fixed) number of replicas, ranging in the Machine 1 from 2 to 16 replicas. It is important to note that in this evaluation we are considering only the performance, we aim to evaluate the impact of the abstractions for adaptive and transparent degrees of parallelism. The adaptive strategies tend to have a more elaborate execution with monitoring and adaptations, which can reduce the performance. Moreover, the results from the self-adaptive strategies presented in Figures 9 and 10 are the ones using the representative configurations from Figures 7 to 8.

To evaluate if the differences are significant' from the statistic standpoint of view, we provide the hypotheses tests using 95% for the confidence interval in Table 1 and 2. We take in to account the recommendations of References 42,43. We used R language to build a script that tests whether the differences among the samples are "statistically significant".[44] Our baseline is the static parallelism configuration, which is manually set by the programmer to compare with TT and

TABLE 1 *p*-value for Machine 1

		Lane Detection Input 1		Lane Detection Input 2		Person Recognition		Pbzip	
		TT	WUDP	TT	WUDP	TT	WUDP	TT	WUDP
Static Parallelism	2	0.001	0.001	0.001	0.001	0.001	0.001	0.000	0.000
	3	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
	4	0.001	0.001	0.001	0.037	0.001	0.001	0.001	0.001
	5	0.001	0.001	0.001	0.275	0.001	0.001	0.001	0.005
	6	0.001	0.003	0.001	0.001	0.001	0.001	0.001	0.001
	7	0.001	0.275	0.193	0.001	0.001	0.003	0.000	0.000
	8	0.001	0.625	0.064	0.001	0.001	0.001	0.000	0.000
	9	0.001	0.064	0.001	0.001	0.001	0.001	0.000	0.000
	10	0.001	0.037	0.001	0.001	0.001	0.001	0.001	0.001
	11	0.001	0.001	0.001	0.001	0.001	0.001	0.000	0.000
	12	0.001	0.001	0.001	0.001	0.001	0.001	0.000	0.035
	13	0.019	0.001	0.001	0.001	0.001	0.001	0.001	0.001
	14	0.027	0.001	0.001	0.001	0.003	0.001	0.048	0.001
	15	0.005	0.001	0.001	0.001	0.431	0.001	0.000	0.000
	16	0.001	0.130	0.001	0.003	0.625	0.001	0.000	0.000

Abbreviations: TT, target throughput; WUDP, without user-defined parameters.

TABLE 2 *p*-value for Machine 2

		Lane Detection Input 1		Lane Detection Input 2		Person Recognition		Pbzip	
		TT	WUDP	TT	WUDP	TT	WUDP	TT	WUDP
Static Parallelism	2	0.001	0.001	0.001	0.001	0.001	0.001	0.000	0.000
	3	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
	4	0.001	0.001	0.001	0.009	0.001	0.001	0.001	0.001
	5	0.001	0.001	0.001	0.556	0.001	0.001	0.001	0.001
	6	0.001	0.001	0.001	0.048	0.001	0.001	0.001	0.001
	7	0.001	0.002	0.556	0.001	0.001	0.001	0.001	0.001
	8	0.001	0.232	0.001	0.001	0.001	0.001	0.001	0.001
	9	0.001	0.845	0.001	0.001	0.001	0.000	0.001	0.001
	10	0.193	0.019	0.001	0.000	0.001	0.001	0.000	0.000
	11	0.130	0.009	0.001	0.000	0.001	0.556	0.001	0.001
	12	0.160	0.009	0.001	0.001	0.003	0.362	0.001	0.001
	13	0.322	0.003	0.001	0.001	0.003	0.232	0.001	0.001
	14	0.236	0.001	0.001	0.001	0.003	0.385	0.000	0.000
	15	0.003	0.001	0.001	0.000	0.556	0.013	0.000	0.000
	16	0.001	0.001	0.001	0.001	0.375	0.001	0.000	0.000
	17	0.003	0.001	0.001	0.000	0.492	0.001	0.000	0.000
	18	0.001	0.001	0.001	0.000	0.105	0.001	0.000	0.000
	19	0.013	0.001	0.001	0.000	0.105	0.001	0.001	0.001
	20	0.001	0.000	0.001	0.000	0.074	0.001	0.000	0.605
	21	0.003	0.001	0.001	0.000	0.083	0.001	0.000	0.004
	22	0.005	0.000	0.001	0.000	0.083	0.064	0.130	0.001
	23	0.001	0.845	0.001	0.019	0.001	0.037	0.001	0.001
	24	0.001	0.149	0.001	0.027	0.001	0.000	0.000	0.000

Abbreviations: TT, target throughput; WUDP, without user-defined parameters.

WUDP results. Before running the actual hypothesis test, we ran the Shapiro–Wilk test, which is suitable for samples of less than 30, to determine if the sample is in a normal distribution. If both the samples that are being compared have a normal distribution, we applied the paired T-test. Otherwise, we applied the paired Wilcoxon test. Only a few samples were not normally distributed. It is important to know how samples are distributed for choosing the appropriate test. Both hypotheses tests return the *p*-value that is presented in the tables. The null hypotheses (H_0) are `Static≠TT` and `Static≠WUDP` while the respective alternative hypotheses (H_1) are `Static=TT`and `Static=WUDP`for all benchmarks. The null hypothesis is false when the *p*-value is higher than .05, otherwise, it remains true. We highlighted in italic the specific comparisons where H_0 is false. In these cases, it is not possible to assume that there is a statistical difference in the throughput when comparing the samples.

Figure 9(A) 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 were increased until it reached the scalability limit of the application. We plotted the results from the adaptive strategies shown in the previous sections. In Machine 1, the adaptive executions started using 8 replicas, since it is the half of the total number of available cores collected by the adaptive algorithm. The throughput results are an average of the final throughput, and the error bars are plotted to represent the standard deviation. We also included the average number of replicas used by the self-adaptive strategies, which emphasizes that the strategy with a TT achieved a higher performance by using more replicas than the WUDP one.

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 can be used during the execution. Figure 9(B) shows the throughput of the different executions using Input-1. The static parallelism show the highest throughput 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 result demonstrates that even with the additional parts implemented, the adaptive strategy can achieve performance similar to the best static in this application.

When comparing the tested adaptive strategies, it is notable that the strategy with a target performance achieved a slightly better throughput than that without the performance definition (WUDP). It is caused because the target performance approach scales faster and under a high target performance it uses more replicas. Moreover, the poorer performance of the strategy WUDP occurred because in this strategy is not possible to enforce the maximum performance goal. The strategy WUDP only changes the number of replicas when the load fluctuates and the maximum number of replicas where only used at some specific points. Due to the fact that Lane Detection achieved the best performance with more replicas, the average throughput of the strategy WUDP was smaller than the executions that used more the maximum number of replicas.

Figure 9(A) shows the result of another experiment with the lane detection application using a different input. This input uses a video with a higher resolution (MPEG-1920 × 1080), which requires more computation and decreases the frame processing rate. When 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 10 to 14 replicas. The adaptive strategy based on the TT performed better. By contrast, the strategy WUDP presented additional performance degradation because it scaled slower and it had more configuration changes with fluctuations from the workload trend. These results show potential areas for future work for optimized strategies, as well as techniques and models to improve performance.

Using different inputs in the lane detection application achieved a similar performance outcome. In Input 1, the best static performance was 2.72% better than the adaptive strategy with TT, while in Input 2 the same aspect had a contrast of 5.08%.

The person recognition application behaves differently than lane detection. On the one hand, lane detection only detects and marks lines. On the other hand, person recognition processes the video detecting faces and comparing them to a database of images. Person recognition's performance was evaluated through a MPEG-4 video with 1.36 MB (640 × 360 pixels) using a training set of 10 images with faces to be recognized in the video.

Figure 10(A) shows the average throughput using the person recognition application. Noteworthy results are that 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, that with a TT outperformed the one WUDP. Moreover, it is possible to note the strategy WUDP performed better in the person recognizer compared with lane detection, because the person recognizer did not achieve the best throughput when using the maximum number of replicas. The adaptive executions with TT achieved a throughput similar to the static using 14 replicas and had a throughput only 2.67% lower than the best static execution.

Figure 10(B) shows the Pbzip2 throughput of the adaptive strategy as well as the static execution under a varied number of replicas. The static executions show the impact of the number of replicas on performance, increasing the throughput from around 7 MBPS with two replicas to around 29 MBPS with 15 and 16 replicas. Noteworthy in this application, the best execution time was achieved with 16 static replicas. This number of replicas showed significant performance losses in the previous applications.

Moreover, the standard deviation was plotted along with the executions. However, it is not possible to identify it since the deviation was minimal. In this experiment, the best performing adaptive strategy was the TT that achieved a performance close the one using 14 static replicas. Moreover, the best static execution with 15 replicas was only 4.15% better than the adaptive strategy using a TT. On the other hand, the strategy WUDP had a performance 7.38% degraded compared with best static execution. It is important to note that in this specific application and machine the best performance was achieved with 15 static replicas, while the adaptive strategy used 14 as the maximum number of replicas.

The experimental evaluation also analyzed the statistical properties of the results. Table 1 shows the outcome of *p*-value encompassing the static and adaptive executions. Results marked with the italic values are those that the null hypothesis is false with 95% of reliability, which means that in these cases there is no significant statistical difference in the performance of adaptive and static executions, because we assume then the H_1. Notably, in most cases as shown in Table 1, the null hypothesis is true. The hypothesis tests were also run comparing the TT and the WUDP, which showed a *p*-value of: .001 for `Lane Detection Input 1`; .001 for `Lane Detection Input 2`; .001 for `Person Recognition`; and .000 for `Pbzip2`.

Figure 11 shows an overview of results from Machine 2 that has more available resources, where representative adaptive executions from Figure 8 are compared with static ones. The throughput of the static executions with all degrees of parallelism is not shown for the sake of space and clarity. As Machine 2 has more physical resources available, the degree of parallelism can be higher. In this scenario, the static execution with the best performance was 22 replicas in Lane Detection and slightly better Person Recognizer with 19 replicas, while 24 static replicas had the best performance in Pbzip2. In general, the adaptive strategy with a TT outperformed the WUDP adaptive strategy.

The experimental results reveal that the performance trends of the applications tend to be similar, even with different inputs. In addition, the adaptive strategies showed only a lower performance when compared with the best static cases, specially the strategy requiring a TT. In fact, the tolerated performance losses depends on the programmers' performance objectives and their skills in parallel programming. The level of performance that can be sacrificed in order to have adaptive and high-level parallelism abstractions depends on the specific demands and scenarios.

We also present a statistical analysis of results from Machine 2. Table 2 shows the outcome of *p*-values considering the static and adaptive executions using all possible number of replicas in static executions. In the majority of cases, the null hypothesis is true, characterizing the results with statistically different. However, in some cases, the adaptive executions are statistically considered as equal to static configurations, which vary from applications and input workloads. The performance contrasts evinced in Figure 11 are significant. For instance, although the two self-adaptive visually seem to have similar performance (e.g., person recognition application), the low visual differences in Figure 11 are statistically distinct in favor of the TT strategy with respect to WUDP. Although Table 2 is not showing the results of this comparison, we ran the hypotheses test, which showed a *p*-value of: .009 for `Lane Detection Input 1`; .003 for `Lane Detection Input 2`; .005 for `Person Recognition`; and .000 for `Pbzip2`.

6 | RELATED WORK

Several parallelism optimizations can be applied at run-time.[1] Two example of optimizations are the degree of parallelism and job batching. Another potential optimization is to automatically tune the communication queues' concurrency modes,[34] but this a complementary optimization specific to particular runtime systems. Although the mechanisms for exploiting dynamic adaptation of the degree of parallelism can be complex to implement, it is a powerful and effective approach for achieving several user objectives. Jobs batching is also relevant, for instance, Metzger et al.[45] proposed a solution for enforcing deadlines with an analytical framework, which optimizes the resources consumption and the applications' performance by batching jobs. Batching jobs is a suitable alternative for a set of applications, but the optimization space can be low for some scenarios (e.g., latency-sensitive applications). Moreover, the impact of batch sizes on applications' metrics like throughput and latency can be nonlinear and complex to optimize effectively.[46] Thus, in this work, we focus on optimizations with a dynamic degree of parallelism, where we envision that in the future batching optimizations can be combined with the degree of parallelism.

A number of researchers have assessed how to best determine the optimal degree of parallelism in parallel applications, of note are References 17,47 and 48. However, these approaches deal with self-adaptive aspects in nonstream processing applications. Thus, they are not feasible for streaming applications due to the specific behaviors and constraints that they present, such as the nature of load fluctuations, unpredictable input rates, and nonlinear behavior. Self-adapt parallelism configurations is a potential solution for improving the executions to different performance goals and scenarios. However, stream processing applications require specifically targeted solutions. Some authors have addressed the degree of parallelism problem in stream processing applications. We will describe these approaches below.

Sensi et al.[21,49,50] explore 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. In order to do so, they proposed and implemented a simple programming interface and runtime called *NORNIR*, which enabled the application to change (the number of cores and clock frequency) during run-time. The *NORNIR* system was aimed to satisfy power consumption or performance bounds, which have to be defined by the user. It then triggers actions when it detects changes in the input rate or application. *NORNIR* interacts with the OS and with FastFlow's runtime.

In addition, Matteis and Mencagli[23] present elastic properties for data stream processing to improve performance and energy efficiency (number of cores and frequency). Their proposed model 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. Matteis and Mencagli[23] use 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.[51]

Bringing the term "self-aware" to stream processing applications, Su et al.[26] 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. Su et al.'s[26] mechanism to support adaptive features in stream processing applications has four parts: detecting, gathering, analyzing, and acting. It adjusts stream parallelism by adding or removing worker threads.

On the other hand, Gedik et al.[24] use the term elastic autoparallelization 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. Moreover, Gedik et al.[24] argue that the parallelism profitability problem depends on workload changes (variation) and resource availability. They propose an elastic autoparallelization 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.

Yet another approach to stream processing is provided by Heinze et al..[27] 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 autoscaling (scaling in or out), workload independence, adaptive, configurability, and computational feasibility. They also explore latency aspects in a distributed system, using metrics with fixed minimum, maximum, and target utilization.

Selva et al.[25] show 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.

Adaptability was also proposed by Floratou et al.[52] for real-time stream processing analytics, where the notion of self-regulation in Twitter's Heron framework was introduced with the proposed system called Dhalion. In this solution, the user sets a TT and Dhalion transparently configures the number of processes and cloud instances for achieving the user goal. Dhalion was validated with real-world applications, showing that the system can dynamically self-regulate to meet SLOs.

Table 3 shows an overview of the related works. The main aspects considered were the adaptation strategy and the scenario of the applications evaluated in each study. Each approach has specific goals, and implemented strategies. The applications evaluated vary among the approaches, but we selected only related works that focus on dynamic degree of parallelism for stream processing applications.

TABLE 3 Overview of related works that adapt the parallelism

Approach	Goal	System	Applications	External actuator
21	Performance and power consumption	NORNIR	PARSEC	Yes
23	Performance and power consumption	Elasticity on FastFlow	High-frequency trading	Yes
26	Performance per Watt	StreamAware	PARSEC	Yes
24	Congestion and throughput	Algorithm implementation	Finance, Twitter, PageRank, and Network	Yes
27	Maximize utilization and control the latency	System prototype	Financial	Yes
25	Quality-of-services	StreamIt extension	Dataflow applications	Yes
52	Maximize throughput	Algorithm implementation	Word count	Yes
This approach	Parallelism abstraction (throughput)	Extension of SPar DSL	Lane detection, person recognition, and Pbzip	No

Our research differs from existing papers because we provide an adaptive degree of parallelism support to a parallel runtime system (named SPar). Sensi et al.[21] and Matteis and Mencagli[23] aim to predict energy consumption. Though all share the same application domain, we focused on stream parallelism abstractions regarding reactive adaptations of the degree of parallelism. For instance, proactive approaches such as Matteis and Mencagli[23] attempt to predict a future trend. The challenge is that it is almost impossible to predict performance peaks on stream processing due to the combination of load fluctuations, irregular behavior, and dynamic execution environments. By contrast, we use a reactive approach that is effective and run with low computational complexity to respond fast and accurately.

Moreover, the research problem presented in Gedik et al.[24] is related to this work, since both approaches present an adaptive degree of parallelism. However, we have a different scenario and target architecture. While Gedik et al.,[24] Heinze et al.,[27] Su et al.,[26] Selva et al.,[25] and Floratou et al.[52] address distributed stream systems, our approach targets stream parallelism in shared-memory multicore systems for parallelism abstraction.

Concerning SPar and ready to use parallelism abstractions via self-adaptiveness, we argue that the related solutions are complex even for experts in parallel programming without focusing on application programmers. The algorithm implementations of related works are not sufficiently abstracted for application programmers. Moreover, the related works used an external actuator/thread for monitoring the execution and triggering actions. In this work, we implemented the actuator inside existing parallel activities, which reduces complexity and overhead. In summary, the advantages and differences of our solution compared with existing ones are the following: (1) We are proving higher level abstractions with the WUDP strategy that self-configures the degree of parallelism and does not require any configuration parameter; (2) Our solution uses an internal actuator, this aspect has the potential to reduce the consumption of resources resulting in less overhead; (3) Our strategies are ready-to-use by not requiring the installation/inclusion of external libraries or code blocks, any application using SPar supports now the self-adaptive strategies.

7 | CONCLUSION AND PERSPECTIVES

In this article, we provided high-level self-adaptive abstractions for stream parallelism with SPar. Therefore, we proposed and evaluated two new self-adaptive strategies. Our strategies considered different types of information for deciding if optimization is required based on the application's throughput. We also implemented a heuristic to further simplify the work for programmers. The proposed strategies can be automatically generated along with the SPar parallel code using the designed transformation rules. Compared with related approaches seen in Section 6, our solution provides additional parallelism abstractions along with an autonomous runtime system. Our solution can potentially make computer systems more autonomous and efficient without requiring expertise from application programmers. Our solution could be encompassed by other parallel programming frameworks, and by computational environments and infrastructures. Our insights from implementation and experimental results may be used as a theoretical contribution for applying self-adaptiveness to other scenarios.

The implication of the experimental results is that self-adaptiveness is suitable for providing parallelism abstraction. We have demonstrated the effectiveness of our solutions when adjusting the number of replicas at run-time. It is possible to conclude that the self-adaptive strategies increased the level of abstraction without compromising the performance. The strategy based on a TT achieved the best performance. The option WUDP (a simple heuristic) was easier to use but had inferior performance. If necessary, the programmer can fine-tune the application's performance by using our strategy based on TT.

In this study, the implemented strategies were tested with three real-world applications and two machines. Although the applications are representative, slightly different performance may be achieved in different machine's architectures and under other application characteristics. Furthermore, we only tested our solutions in application scenarios with one replicated stage. The strategies tend to work when more stages are replicated, in such a way that each parallel region has one self-adaptive strategy that configures the degree of parallelism independently. Though testing a composition with more than one replicated stage was not our goal here, we do intend to tackle this question in the future. In complex stream compositions, our solution can potentially maintain reasonable performance because it uses an embedded actuator for reducing the overhead when being self-adaptive.

Apart from the CPUs affinity already provided by FastFlow runtime library, the strategies do not cover additional CPUs affinity optimization. This can be provided for our self-adaptive strategies in the future. In addition, the proposed solution assumes that the application has enough data to be processed, in the future we intend to implement and validate additional features for applications that can have inconstant and very low input rates.[20]

We aim to extend this work in many ways. Initially, the strategies can be extended to run in distributed machine environments (clusters), including cloud and fog, increasing the flexibility with an elastic infrastructure. Then, we aim to test our solution in applications with complex structures, such as those with several replicated stages. Moreover, we hope to continue improving the adaptive strategies by providing optimizations. For instance, improving the decision-making stability in order to increase the performance of applications. Another relevant optimization is to design an adaptive SF to improve the performance. Finally, we aim to evaluate the possibility of implementing online learning in our strategies.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable comments.

ORCID

Adriano Vogel https://orcid.org/0000-0003-3299-2641
Dalvan Griebler https://orcid.org/0000-0002-4690-3964
Luiz Gustavo Fernandes https://orcid.org/0000-0002-7506-3685

REFERENCES

1. Hirzel M, Soulé R, Schneider S, Gedik B, Grimm R. A catalog of stream processing optimizations. *ACM Comput Surv*. 2014 Apr;46(4):1-34.
2. Andrade HCM, Gedik B, Turaga DS. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge, MA: Cambridge University Press; 2014.
3. Chakravarthy S, Qingchun J. *Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing*. New York, NY: Springer; 2009.
4. Cardellini V, Presti FL, Nardelli M, Russo GR. Decentralized self-adaptation for elastic data stream processing. *Futur Gener Comput Syst*. 2018;87:171-185.
5. Reinders J. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. Sebastopol, CA: O'Reilly Media Inc; 2007.
6. Thies W, Karczmarek M, Amarasinghe S. StreamIt: a language for streaming applications. Paper presented at: Proceedings of the International Conference on Compiler Construction, Grenoble, France; 2002:179-196.
7. Aldinucci M, Meneghin M, Torquati M. Efficient smith-waterman on multi-core with fastflow. Paper presented at: Proceedings of the Euromicro Conference on Parallel, Distributed and Network-based Processing, Pisa, Italy; 2010:195-199.
8. del Rio Astorga D, Dolz MF, Fernandez J, Garcia JD. A generic parallel pattern interface for stream and data processing. *Concurr Comput Pract Exp*. 2017;29(24):e4175.
9. Griebler D, Danelutto M, Torquati M, Fernandes LG. SPar: a DSL for high-level and productive stream parallelism. *Parall Process Lett*. 2017;27(01):1740005. https://doi.org/10.1142/S0129626417400059.
10. Toshniwal A, Taneja S, Shukla A, et al. Storm twitter. Paper presented at: Proceedings of the ACM SIGMOD International Conference on Management of Data; 2014:147-156; ACM, New York, NY.
11. Zaharia M, Xin R, Wendelland P, Das T, Armbrust M, et al. Apache spark: a unified engine for big data processing. *Commun ACM*. 2016;59(11):56-65.
12. Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache flink: stream and batch processing in single engine. *Bull IEEE Comput Soc Techn Commit Data Eng*. 2015;36(4):28–38.
13. Zeuch S, Monte BD, Karimov J, et al. Analyzing efficient stream processing on modern hardware. *Proc VLDB Endow*. 2019;12(5):516-530.
14. Misale C, Drocco M, Tremblay G, Martinelli A, Aldinucci M. PiCo: high-performance data analytics pipelines in modern C++. *Futur Gener Comput Syst*. 2018;87:392-403.
15. Mencagli G, Torquati M, Griebler D, Danelutto M, Fernandes LG. Raising the parallel abstraction level for streaming analytics applications. *IEEE Access*. 2019;7:131944-131961.
16. Rajadurai S, Bosboom J, Wong WF, Amarasinghe S. Gloss: seamless live reconfiguration and reoptimization of stream programs. *ACM SIGPLAN Not*. 2018;53(2):98-112.
17. Pusukuri KK, Gupta R, Bhuyan LN. Thread reinforcer: dynamically determining number of threads via OS level monitoring. Paper presented at: Proceedings of the IEEE International Symposium on Workload Characterization, Austin; 2011:116-125.
18. Macías-Escrivá FD, Haber R, Del Toro R, Hernandez V. Self-adaptive systems: a survey of current approaches, research challenges and applications. *Expert Syst Appl*. 2013;40(18):7267-7279.
19. Hellerstein JL, Diao Y, Parekh S, Tilbury DM. *Feedback Control of Computing Systems*. Hoboken, NJ: John Wiley & Sons; 2004.
20. Aldinucci M, Danelutto M, Kilpatrick P. Autonomic management of non-functional concerns in distributed & parallel application programming. Paper presented at: Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy: IEEE; 2009:1-12.
21. Sensi DD, Torquati M, Danelutto MA. Reconfiguration algorithm for power-aware parallel applications. *ACM Trans Arch Code Optim*. 2016;13(4):43:1-43:25.

22. Sensi DD, Matteis TD, Danelutto M. Simplifying self-adaptive and power-aware computing with Nornir. *Futur Gener Comput Syst*. 2018;87:136-151.
23. Matteis TD, Mencagli G. Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing. Paper presented at: Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, Barcelona, Spain; 2016:13:1-13:12.
24. Gedik B, Schneider S, Hirzel M, Wu KL. Elastic scaling for data stream processing. *IEEE Trans Parall Distrib Syst*. 2014;25(6):1447-1463.
25. Selva M, Morel L, Marquet K, Frenot S. A monitoring system for runtime adaptations of streaming applications. Paper presented at: Proceedings of the Euromicro Conference on Parallel, Distributed and Network-based Processing, Turku, Finland; 2015:27-34.
26. Su Y, Shi F, Talpur S, Wang Y, Hu S, Wei J. Achieving self-aware parallelism in stream programs. *Clust Comput*. 2015;18(2):949-962.
27. Heinze T, Pappalardo V, Jerzak Z, Fetzer C. Auto-scaling techniques for elastic data stream processing. Paper presented at: Proceedings of the International Conference on Data Engineering Workshops, Chicago; 2014:296-302.
28. Vogel A, Griebler D, Sensi DD, Danelutto M, Fernandes LG. Autonomic and latency-aware degree of parallelism management in SPar. Paper presented at: Proceedings of the Euro-Par 2018: Parallel Processing Workshops Lecture Notes in Computer Science; 2018:28-39; Turin, Italy: Springer. https://doi.org/10.1007/978-3-030-10549-5_3.
29. Vogel A, Griebler D, Danelutto M, Fernandes LG. Seamless parallelism management for multi-core stream processing. Paper presented at: Proceedings of the International Conference on Parallel Computing (ParCo), vol. 36 of ParCo'19 Advances in Parallel Computing; 2019:533-542; Prague, Czech Republic: IOS Press. https://doi.org/10.3233/APC200082.
30. Vogel A, Griebler D, Danelutto M, Fernandes LG. Minimizing self-adaptation overhead in parallel stream processing for multi-cores. Paper presented at: Parallel Processing Workshops, vol. 11997 of Lecture Notes in Computer Science Euro-Par 2019; 2019:12; Göttingen, Germany: Springer. https://doi.org/10.1007/978-3-030-48340-1_3.
31. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M. Fastflow high-level and efficient streaming on multicore. *Programming Multi-Core and Many-Core Computing Systems, Parallel and Distributed Computing*. Hoboken, NJ: John Wiley & Sons; 2017:261-280.
32. Danelutto M, Garcia JD, Sanchez LM, Sotomayor R, Torquati M. Introducing parallelism by using REPARA C++11 attributes. Paper presented at: Proceedings of the Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Heraklion, Greece: IEEE; 2016:354-358.
33. Moir M, Shavit N. *Concurrent Data Structures. Computer and Information Science Series*. Boca Raton, FL: Chapman & Hall/CRC Press; 2004.
34. Torquati M, Sensi DD, Mencagli G, Aldinucci M, Danelutto M. Power-aware pipelining with automatic concurrency control. *Concurr Comput Pract Exp*. 2019;31(5):e4652.
35. Griebler D. Domain-Specific Language & Support Tool for High-Level Stream Parallelism [PhD thesis]. Faculdade de Informática - PPGCC - PUCRS; 2016.
36. Mattson TG, Sanders BA, Massingill BL. *Patterns for Parallel Programming*. Boston, MA: Addison-Wesley; 2005.
37. González-Vélez H, Leyton M. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw Pract Exp*. 2010;40(12):1135-1160.
38. Griebler D, Hoffmann RB, Danelutto M, Fernandes LG. Higher-level parallelism abstractions for video applications with SPar. Paper presented at: Proceedings of the International Conference on Parallel Computing ParCo'17 Parallel Computing is Everywhere; 2017:698-707; Bologna, Italy: IOS Press. https://doi.org/10.3233/978-1-61499-843-3-698.
39. Seward J, Bzip2 and Libbzip2, version 1.0. 5: a program and library for data compression; 2007. http://www.bzip.org. December 2017.
40. Griebler D, Hoffmann RB, Danelutto M, Fernandes LG. High-level and productive stream parallelism for Dedup, Ferret and Bzip2. *Int J Parall Program*. 2018;47(1):253-271. https://doi.org/10.1007/s10766-018-0558-x.
41. Schneider S, Hirzel M, Gedik B, Wu KL. Auto-parallelizing stateful distributed streaming applications. Paper presented at: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques; 2012:53-64; ACM, New York, NY.
42. Reyes RP, Dieste O, Fonseca ER, Juristo N. Statistical errors in software engineering experiments: a preliminary literature review. Paper presented at: Proceedings of the 40th International Conference on Software Engineering ICSE '18; 2018:1195-1206; New York, NY, Association for Computing Machinery.
43. Berger ED, Hollenbeck C, Maj P, Vitek O, Vitek J. On the impact of programming languages on code quality: a reproduction study. *ACM Trans Program Lang Syst*. 2019;41(4):1–24.
44. Faraway JJ. *Extending the Linear Model with R: Generalized Linear Mixed Effects and Nonparametric Regression Models*. Texts in Statistical Science. 2nd ed. Boca Raton, FL: Chapman & Hall/CRC Press; 2016.
45. Metzger P, Cole M, Fensch C, Aldinucci M, Bini E. Enforcing deadlines for skeleton-based parallel programming. Paper presented at: Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Sydney, Australia: IEEE; 2020:188-199.
46. Das T, Zhong Y, Stoica I, Shenker S. Adaptive stream processing using dynamic batch sizing. Paper presented at: Proceedings of the ACM Symposium on Cloud Computing; 2014:1-13; ACM, New York, NY.
47. Raman A, Kim H, Oh T, Lee JW, August DI. Parallelism orchestration using DoPE: the degree of parallelism executive. *ACM SIGPLAN Not*. 2011;46:26-37.
48. Sridharan S, Gupta G, Sohi GS. Holistic run-time parallelism management for time and energy efficiency. Paper presented at: Proceedings of the ACM International Conference on Supercomputing, Eugene; 2013:337-348.
49. Danelutto M, Sensi DD, Torquati M. A power-aware self-adaptive macro data flow framework. *Parall Process Lett*. 2017;27(01):1-20.

50. Sensi DD, Matteis TD, Danelutto M. Nornir: a customisable framework for autonomic and power-aware applications. Paper presented at: Proceedings of Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing (Auto-DaSP) - Euro-Par: Parallel Processing Workshops, Santiago de Compostela, Spain; 2017:42-54.
51. Sensi DD, Torquati M, Danelutto M. Mammut: high-level management of system knobs and sensors. *SoftwareX*. 2017;6:150-154.
52. Floratou A, Agrawal A, Graham B, Rao S, Ramasamy K. Dhalion: self-regulating stream processing in Heron. *Proc Very Large Data Base Endow*. 2017;10(12):1825-1836.

How to cite this article: Vogel A, Griebler D, Fernandes LG. Providing high-level self-adaptive abstractions for stream parallelism on multicores. *Softw Pract Exper*. 2021;51:1194–1217. https://doi.org/10.1002/spe.2948