Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200082

Seamless Parallelism Management for Video Stream Processing on Multi-Cores

Adriano Vogel [a,1], Dalvan Griebler [a,c], Luiz Gustavo Fernandes [a], Marco Danelutto [b]
[a] *School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil*
[b] *Computer Science Department, University of Pisa, Italy*
[c] *Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio - Brazil*

Abstract. Video streaming applications have critical performance requirements for dealing with fluctuating workloads and providing results in real-time. As a consequence, the majority of these applications demand parallelism for delivering quality of service to users. Although high-level and structured parallel programming aims at facilitating parallelism exploitation, there are still several issues to be addressed for increasing/improving existing parallel programming abstractions. In this paper, we aim at employing self-adaptivity for stream processing in order to seamlessly manage the application parallelism configurations at run-time, where a new strategy alleviates from application programmers the need to set time-consuming and error-prone parallelism parameters. The new strategy was implemented and validated on SPar. The results have shown that the proposed solution increases the level of abstraction and achieved a competitive performance.

Keywords. Parallel Programming, Domain-Specific Language, Stream Processing, Autonomic Computing, Self-adaptive Systems, Seamless Computing.

1. Introduction

A significant amount of applications/systems must gather and analyze data in real-time [2]. Processing continuous stream sequences and responding fast enough requires powerful machines and robust runtimes/languages. Performance optimization for stream processing applications concerns parallelism, which is important because computer architectures have multiple processing units per chip. Therefore, performance gains are usually conditioned to parallel executions.

We have seen the emergence of parallel programming frameworks and libraries for stream processing, such as Intel TBB [11], StreamIt [13] and, FastFlow [4,1]. However, the programming abstractions provided by the parallel programming frameworks remain arguably complex for application programmers[2], which are more concerned with the developing of stream processing algorithms than implementing low-level techniques for

[1]Corresponding Author: adriano.vogel@edu.pucrs.br

[2]The separation of concerns covers the skills and aspects for different types of programmers. Application programmers are software developers focused on the algorithm design and implementation while system programmers are focused on better using computational resources.

exploiting the parallel architecture. Recently, SPar[3] [6] was created for providing additional parallel programming abstractions on stream parallelism targeting multi-core architectures.

Although FastFlow was supported with abstraction concerning parallelism and energy in [12,3], we believe that opportunities exist for novel higher and ready to use parallelism abstractions in the stream parallelism domain. In this work, we aim at providing additional abstractions regarding the definition of the degree of parallelism. In the context of stream processing, manually defining and statically using a degree of parallelism throughout the execution is not suitable. Defining the degree of parallelism tends to be a complicated and time-consuming task because the programmer has to run the same program several times to decide which is the optimal configuration.

Moreover, a significant part of stream processing applications requires recurrent optimizations at run-time. Mainly because stream processing applications have load fluctuations (*e.g.,* performance, environment, or input rates). Consequently, static/unchangeable executions can lead to inefficient resources usage (waste) or poor performance. One way to respond to fluctuations is by adapting the degree of parallelism to improve the performance and/or the efficiency of stream processing applications. Regarding adaptation to load fluctuations, a conventional approach for handling it could be a proactive one, attempting to predict the future load. The challenge is that it is very difficult to predict performance peaks due to the combination of input temporal changes, irregular behavior, and different workload patterns. In this scenario, reactive approaches that are effective by reacting fast, accurately, and run with low computational complexity are a potential solution for enabling suitable adaptations to runtimes.

Abstracting the definition of parallelism configurations is an opportunity for simplifying the process of running parallel applications. In previous work, we presented a new latency-aware self-adaptive strategy [15], where we demonstrated how the degree of parallelism impacts in the latency of stream items. We also provided abstractions [8,14] that enable users/programmers to express service-level objectives (SLO), such as energy bounds, system utilization, and throughput. These implemented strategies require the definition of a target performance or SLOs. Yet, this can be a usability challenge since users/programmers may have no performance/system expertise. However, it is challenging for a completely abstracted strategy to make adaptation decisions without user hints. For instance, approaches that require a definition of a target performance or service objective can decide by comparing such parameters to the actual system/application state.

In this paper, the main contributions can be summarized as the following: 1) We provide a new fully abstracted self-adaptive strategy for the autonomic management of the parallelism in stream processing applications, this new strategy seamlessly manages the parallelism by detecting workload fluctuations; 2) A characterization and comparison of the decision making of the new strategy with respect to other solutions; 3) A validation of the proposed solution with video stream processing applications in terms of performance and resources consumption.

This paper is organized as follows. The background scenario is presented in Section 2. The proposed solution is shown in Section 3. Then, Section 4 shows the experimental results of this paper and Section 5 discusses aspects related to the proposed solution. Finally, the conclusion is presented in Section 6.

[3] SPar home page: https://gmap.pucrs.br/spar

2. Context

The scenario of this study is related to extending SPar DSL features, SPar is briefly described in Subsection 2.1. Moreover, Subsection 2.2 presents relevant related approaches.

2.1. SPar Overview

SPar [6] provides a standard C++ annotation interface, fully compatible with the host language and compiler. In Spar, programmers are invited to simply add annotations on their source code with C++ attributes that represent stream parallelism properties. Then, the compiler interprets the annotations added and generates parallel code with source-to-source transformations.

SPar provides five attributes to exploit key aspects of stream parallelism. The *ToStream* attribute represents the beginning of a stream region, the code block between the *ToStream* and the first *Stage* will run as the first processing stage. More *Stages* can be created inside the *ToStream*. The *Input* attribute allows programmers to define the data to be processed inside a stream region. In contrast, the *Output* attribute is used to define the processing results produced. *Replicate*[4] is the attribute used to define the degree of parallelism. In the code example shown in Listing 1, the data type is a "string" and the input stream comes from a file (read in line 3). This code block is a loop with iterations and a new stream item is read and computed (line 6) on each iteration. In line 5, the attribute *Replicate* defines the degree of parallelism with 4 replicas, which is the static number of replicas used during the entire execution. Finally, in line 8 an output is produced. Figure 1 represents the activity graph with 3 stages of the parallel execution implemented in the runtime according to the annotations introduced in Listing 1.

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

Listing 1 SPar code example.

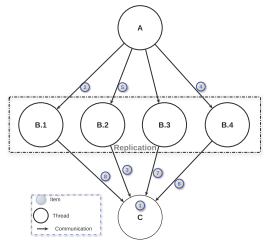

Figure 1. Parallel activity graph.

2.2. Related Approaches

In the related literature exist studies for adaptivity on stream processing. Noteworthy, Sensi *et al.*[12] present a programming interface and runtime called *NORNIR*, which aims at predicting performance and power consumption. *NORNIR* manages the system for maintaining a given power consumption and/or a performance goal. The execution

[4]The term replicate refers to the degree of parallelism in SPar, here the number of replicas and degree of parallelism are used interchangeably.

is managed by adapting system configurations (*e.g.,* number of cores, clock frequency) at run-time. In addition, Matteis and Mencagli [10] presented elastic properties for data stream processing, their goal was to improve performance and energy consumption. The proposed model was implemented along with the FastFlow runtime using one controller thread for monitoring the environment as well as for triggering changes.

Gedik *et al.* [5] and Heinze *et al.* [9] address distributed stream systems. Our approach in contrast targets parallelism abstraction for stream parallelism in multi-core systems. The algorithm implementations provided by related works arguably do not provide sufficient abstractions for application programmers. Differently, our goal is to provide new parallelism abstractions for parallel stream processing applications that is ready-to-use. Our solution require no additional configuration nor drivers installation. Additionally, we propose an improved evaluation of the overhead caused by the adaptivity as we measured the performance and memory consumption. The solution is also compared to the regular static executions.

3. Seamless Parallelism Management

Defining a performance goal is presumably easier for application programmers than defining a low-level parameter of the runtime library. Therefore, in previous works [14,8] we presented strategies that abstracted from users the need to set parallelism parameters related to the number of replicas. The parallelism abstraction was achieved by monitoring the actual application performance and responding to performance violation by continuously adapting the number of replicas. In listing 2 is shown a SPar example with the solution proposed in [14], where the difference compared to the Listing 1 is that the definition of the number of replicas inside the *Replicate* attribute was no longer required. Regarding the adaptation at run-time, in Figure 2 is shown the solution that creates a pool of replicas and dynamically changes the status of the replicas (active, suspended).

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

Listing 2 SPar code example.

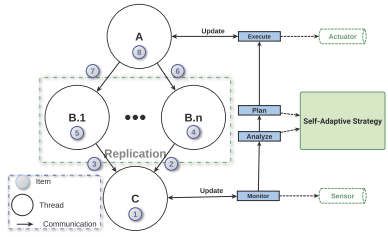

Figure 2. Autonomous Parallelism.

The previous proposed strategies [14,8] require from users the input of performance hints for adapting the number of replicas. However, low-level performance aspects tend to be complex for application programmers. Additionally, stream processing applications are usually long running and with significant load fluctuations, where temporal changes could require different performance objectives. Consequently, we propose a new strategy to manage the execution in an autonomous and seamless way. This new strategy abstracts from users the parameters set. This solution enables a fully seamless execution, which is

achieved by a new decision strategy that monitors the application, detects changes in the workload and performs optimizations in the number of replicas used.

The new decision strategy workflow implements a sensor with a *monitor* running inside the last stage, while the parallelism is adapted by the actuator running in the first stage. The decision (D) whether the number of replicas should be adapted is performed by the Analyze and Plan phases with the following steps: 1) stores data regarding the application performance collected by the monitor; 2) After the execution starts, when it has a minimum of three (a number defined from empirical tests for having a balance between fast and accurate decisions) performance results from monitor iterations, compares this previously collected data with the current performance; 3) If the current performance is significantly lower than the previous one, a new replica (R) is activated (D1); 4) If the current performance is significantly higher than previous results, an active replica is suspended (D2); 5) After the monitor executed 10 iterations with performance results, the regulator enters a new phase where it has more performance data for deciding, which tends to improve the decisions accuracy. Then, for the sake of stability, the average of the previous three throughput collected is compared to the average throughput from all historical data.

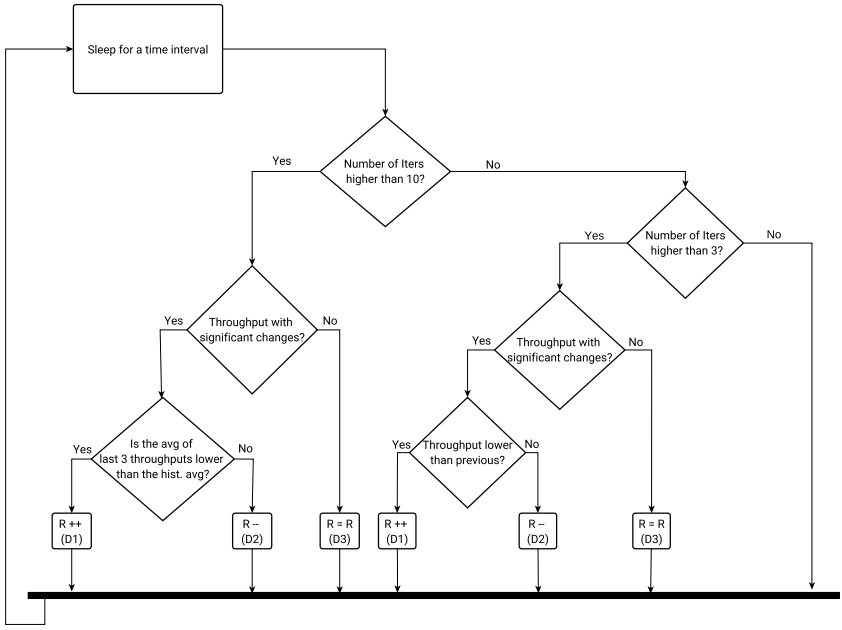

Figure 3. Overview of the Analyze and Plan phases.

Figure 3 shows a high-level representation of decision phases and iterations performed. It is important to note that in addition to decisions 1 and 2 (D1 and D2), there is also the D3 that is performed when the decision is for maintaining the same number of replicas. Moreover, the self-adaptive strategy runs continuously and decides if the number of replicas should be adapted. Although the strategy runs several times and changes the configuration, the adaptations do not affect the regular computations of the appli-

cation. In fact, while the application is running, the strategy periodically runs and then sleeps for a time interval. In this study, we consider 1 second as the default sampling time interval, which allows the strategy to achieve a suitable level of sensitivity to workload fluctuations. Too frequent adaptations can cause instability, while too high sampling times can result in unresponsiveness to changes. Also, two is the minimum number of replicas in a replicated stage, which is a value for minimum parallelism. The maximum number of replicas is defined by the self-adaptive strategy by detecting the machine configuration. The maximum number of replicas is set to at most one application thread per hardware thread, also counting threads from other sequential stages (*e.g.,* Read, write).

4. Evaluation

This section characterizes the new strategy comparing to other solution and to parallel static executions. The new strategy is also evaluated in terms of performance and memory utilization.

4.1. Methodology

The proposed solution was evaluated by implementing it to existing parallel stream processing applications. 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. In this work, two applications were tested. The first is **Lane Detection** that is an application used on autonomous vehicles to detect road lanes, which is using for maintaining the car on the road. This is performed by reading a video feed from a camera. The road lanes are detected through a sequence of operations where the parallel implementation is like an assembly line composed of three stages, where the second stage is stateless and therefore replicated [7].

Person Recognition is the other tested application that 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 detected matches the database one, the face is marked with a green circle. Person recognition's performance was evaluated with a MPEG-4 video (1.36MB - 640x360 pixels) using a training set of 10 images with faces to be recognized in the video [7].

4.2. Characterization

The new seamless strategy is characterized and compared to an existing one [14] that requires a manual definition of a target performance, which was defined to a throughput of 50. The experiments shown here and in the next section were carried out on a multicore machine equipped with 32 GB of memory, a dual-socket Intel Xeon CPU 2.40GHz (12 cores- 24 threads). The operating system used was Ubuntu Server, G++ v. 5.4.0 with the -O3 compilation flag. The parallel version used the on-demand scheduling policy that is suitable for stream processing, which improves the load balancing by distributing one item to each replica. Moreover, in order to avoid overhead, the emitter and collector stages were placed on dedicated physical cores.

The seamless strategy behavior is characterized in Figure 4 using the Lane Detection application and the input workload was a file of 260 MB [14]. The experiment demon-

strates the throughput and the number of replicas used by each strategy in parallel executions. Moreover, the self-adaptive strategies are compared to static executions running with a fixed number of replicas. For the sake of visual clarity, we only show representative results of static executions with 10 and 20 replicas.

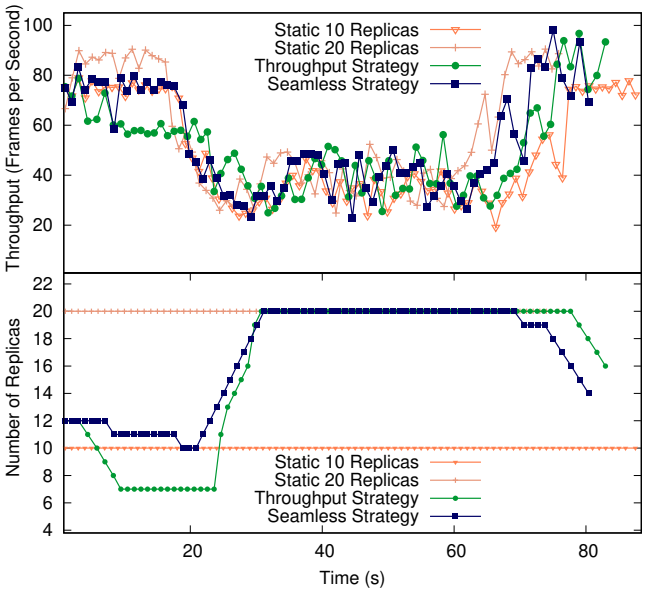

Figure 4. Characterization - Parallel Executions.

In Figure 4, we can observe throughput fluctuations caused by the input workload [14]. The executions with a static number of replicas also presented throughput fluctuations, which emphasizes that the oscillations were caused by input workload instead of the self-adaptive strategies. Regarding the proposed seamless performance strategy, it is important to note that after the first iterations, the throughput increased because of the workload fluctuation. As a consequence, the parallelism actuator changed the number of replicas from 12 to 11. Noteworthy, considering the workload fluctuations around the middle of the execution, the actuator responded to this fluctuation by increasing the number of replicas between the seconds 21 and 36. Another event that highlights the correct sensitivity of this strategy is that the number of replicas was reduced when the execution entered a new phase that increased the throughput (near the second 70).

Comparing the strategies, it is possible to note a similar performance trend caused by the input workload. The strategy based on a manual target performance presented a short settling time, which is notable in the adaptation of the number of replicas after the second 20. The seamless performance strategy required more time to respond to workload fluctuations, which can impact negatively on those applications that demand very fast adaptations. Moreover, it is possible to note in Figure 4 that the seamless performance strategy had a slightly lower execution time, which occurred because this execution had a higher throughput in the first seconds by using more parallel replicas.

4.3. Performance and Overhead

A relevant evaluation of the proposed solution concerns the performance achieved and the resources consumption. The static executions have a simpler runtime that does not perform any adaptation. The advantage tends to be in theory a higher performance. On the other hand, static executions are unresponsive to workload or resources changes. In some cases, with a specific number of replicas, we have seen that static executions achieved the highest performance. However, manually finding the best performing number of replicas configuration is a time-consuming and sometimes counter-productive task. In this section, we present the performance of the adaptive solution compared to static execution. The performance metric is the average throughput, which is a result considering the number of processed items divided by the total time taken in an entire execution. Observe that this is different from the previous performance characterization, where the throughput was collected during different time-steps.

In Table 1 is shown the throughput and memory usage of adaptive and static executions in the Lane Detection application. It is notable that the throughput and memory utilization increases with more replicas. The Seamless performance strategy achieved a slightly higher throughput than the throughput strategy. Comparing to the static executions, the static using more than 16 replicas achieved a higher performance, but these executions also consumed more memory space. The performance of the Seamless strategy is less than 5% lower than the best static execution.

Execution	Average Throughput (FPS)	Memory Usage (MBytes)
Static 10 Replicas	47	807
Static 12 Replicas	48.26	1368
Static 14 Replicas	49.14	1276
Static 16 Replicas	50.31	1648
Static 18 Replicas	50.89	1799
Static 20 Replicas	52.11	2228
Throughput Strategy (50)	48.57	1272
Seamless Strategy	49.67	1327

Table 1. Lane Detection Application

In Table 2 is presented the throughput and memory usage of the Person Recognizer application. In this case, a different performance trend can be seen. The Throughput strategy achieved higher performance, while the Seamless strategy again achieved a throughput similar to the best static executions. Regarding memory usage, the self-adaptive strategies used more memory space on the Person Recognizer application.

5. Discussion

When evaluating higher level abstractions, they often tend to present less performance. However, the best static configuration varies from machines, applications, and workloads. Therefore, tuning all these parameters can be error-prone, time consuming, and may become instantly suboptimal in phase changes or fluctuations. Consequently, a seamless strategy that reacts to workload changes can be a suitable solution that achieves a compromise between abstractions and performance.

Execution	Average Throughput (FPS)	Memory Usage (MBytes)
Static 10 Replicas	12.64	193.6
Static 12 Replicas	12.72	212.60
Static 14 Replicas	12.96	222.10
Static 16 Replicas	12.94	232.10
Static 18 Replicas	13.29	262
Static 20 Replicas	13.22	293.8
Throughput Strategy (15)	13.78	487.7
Seamless Strategy	12.99	448.4

Table 2. Person Recognition Application

There may be overheads as seen in Section 3. A self-adaptive strategy has additional monitoring and actuators entities. For instance, monitoring has a computational cost, but it occurs concurrently while worker replicas are computing tasks. Thereby, the parallel execution is not suspended for monitoring because the monitor runs inside the last stage, which periodically and asynchronously collects statistics. For instance, considering the machine used in the experiments, it took in average only 523 nanoseconds for the monitor implemented in C++ to measure the application throughput. In this case, such a minor amount of time is negligible. The design choices combined with effective mechanisms implemented in the runtime library resulted in a low overhead regarding performance without significantly consuming memory resources.

Moreover, adapting the parallelism of applications at run-time brings additional concerns about safety, which relates to the state and ordering of stream processing applications. Safety is important to ensure that an application can be changed at run-time while preserving its correctness. In SPar, stateless stages can be replicated by default, while stateful executions would require synchronizing a shared internal state. If the ordering of data items is required, the last stage orders the items in SPar. Consequently, self-adapting the parallelism of a stateless stage easily maintains stream items ordered because the last stage is still sequential. Moreover, another aspect of safety is that a worker replica is only suspended after it finishes its computations.

6. Conclusion and Future Work

In this study, we have seen aspects related to the complexities of abstracting parallelism and autonomously managing parallelism configurations at run-time. The new proposed strategy that abstracts the need to set the parallelism and performance configuration shown to be effective. However, the strategy that uses a target performance was able to react faster by comparing the actual performance to the target one.

The alternative that required the definition of a target performance increases the flexibility at the price of additional complexities. On the other hand, running an application transparently increases the abstraction level, but tends to provide less flexibility and lower performance. Some users/programmers may have performance expertise, in which case they may customize their execution by setting system parameters and target performance. However, the provided strategy for seamless execution is designed for users/programmers with no performance and system expertise. Regarding the experimental results, it is important to note that the performance slightly varied among the tested applications, but the trend was similar: the self-adaptive Seamless strategy achieved a com-

petitive performance. Consequently, an implication from the experimental results is that self-adaptivity is suitable for seamlessly managing parallelism configurations.

This study is also limited in some aspects, the implemented strategies control applications with only one replicated stage, use parallel applications with a more complex structure is a future goal. Additionally, our proposed Seamless strategy was validated only with video stream processing applications. Although the applications are representative of stream processing, a different performance trend may be seen under other application characteristics. In the future, we aim at porting related solutions from the literature to our context for comparing them to our strategies.

Acknowledgment This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, Univ. of Pisa PRA_2018_66 "DECLware: Declarative methodologies for designing and deploying applications", the FAPERGS 01/2017-ARD project called PARAELASTIC (No. 17/2551-0000871-5), and the Universal MCTIC/CNPq N 28/2018 project called SParCloud (No. 437693/2018-0).

References

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. Wiley-Blackwell, 2014.

[2] H. Andrade, B. Gedik, and D. Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[3] D. De Sensi, T. De Matteis, and M. Danelutto. Simplifying Self-Adaptive and Power-Aware Computing with Nornir. *Future Generation Computer Systems*, 87:136–151, 2018.

[4] FastFlow. FastFlow (FF) Website, 2019. last access in Fev, 2019. URL: http://calvados.di.unipi.it/.

[5] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, Jun 2014.

[6] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):1740005, March 2017.

[7] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes. Higher-Level Parallelism Abstractions for Video Applications with SPar. In *Proceedings of the International Conference on Parallel Computing*, ParCo'17, pages 698–707, Bologna, Italy, September 2017. IOS Press.

[8] D. Griebler, A. Vogel, D. De Sensi, M. Danelutto, and L. G. Fernandes. Simplifying and implementing service level objectives for stream parallelism. *The Journal of Supercomputing*, Jun 2019.

[9] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer. Online Parameter Optimization for Elastic Data Stream Processing. In *Proceedings of ACM Symposium on Cloud Computing*, pages 276–287. ACM, 2015.

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

[11] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.

[12] D. D. Sensi, M. Torquati, and M. Danelutto. A Reconfiguration Algorithm for Power-Aware Parallel Applications. *ACM Transactions on Architecture and Code Optimization*, 13(4):43:1–43:25, Dec 2016.

[13] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction*, pages 179–196, 2002.

[14] A. Vogel, D. Griebler, M. Danelutto, and L. G. Fernandes. Minimizing Self-Adaptation Overhead in Parallel Stream Processing for Multi-Cores. In *Euro-Par 2019: Parallel Processing Workshops*, Lecture Notes in Computer Science, page 12, Göttingen, Germany, August 2019. Springer.

[15] A. Vogel, D. Griebler, D. D. Sensi, M. Danelutto, and L. G. Fernandes. Autonomic and Latency-Aware Degree of Parallelism Management in SPar. In *Euro-Par 2018: Parallel Processing Workshops*, Lecture Notes in Computer Science, pages 28–39, Turin, Italy, August 2018. Springer.