Towards On-the-fly Self-Adaptation of Stream Parallel Patterns

Adriano Vogel*‡, Gabriele Mencagli‡, Dalvan Griebler*†, Marco Danelutto‡, Luiz Gustavo Fernandes*
* School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil.
†Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio, Brazil.
‡ Computer Science Department, University of Pisa (UNIPI), Pisa, Italy.
Corresponding author: adriano.vogel@edu.pucrs.br

***Abstract*—Stream processing applications compute streams of data and provide insightful results in a timely manner, where parallel computing is necessary for accelerating the application executions. Considering that these applications are becoming increasingly dynamic and long-running, a potential solution is to apply dynamic runtime changes. However, it is challenging for humans to continuously monitor and manually self-optimize the executions. In this paper, we propose self-adaptiveness of the parallel patterns used, enabling flexible on-the-fly adaptations. The proposed solution is evaluated with an existing programming framework and running experiments with a synthetic and a real-world application. The results show that the proposed solution is able to dynamically self-adapt to the most suitable parallel pattern configuration and achieve performance competitive with the best static cases. The feasibility of the proposed solution encourages future optimizations and other applicabilities.**

I. Introduction

Large amounts of data are being generated due to the proliferation of devices (e.g., sensors, cameras) able to sense the external world. It is difficult to process fast enough the high amount of data being generated. Hence, parallel computing is a potential solution for accelerating stream processing programs [1]. The continuous data arrival requires stream processing applications to run for long or even infinite periods. These long executions are subject to occasional fluctuations in the environment (e.g., resource availability) and at the application level (input rate, workload trend) [2], [3]. Responsiveness at run-time is a potential way of coping with such a scenario.

A potential way of achieving responsiveness in stream processing applications is by applying autonomic management with self-adaptiveness [4]. This can reduce the burden on application programmers by transparently self-optimizing the executions and entities, such as the degree of parallelism [5], [6], [7], the number of cores and clock frequencies [8], and batch sizes [9], [10]. Moreover, supporting dynamic changes to parallel application structure has been proposed as a more powerful adaptation [11], [12] that can provide the flexibility needed for stream processing applications.

From a programming perspective, supporting parallel execution is still complex, i.e. application programmers are in charge of threads creation, synchronization, and load balancing. These complexities can be alleviated with parallelism abstractions that provide a simplified and high-level view of the features related to parallelism. Structured parallel programming provides relevant methodologies for improving programmability [13], where the concepts are incorporated into parallel programming frameworks like Intel TBB [14] and FastFlow [15].

In structured parallel programming, application programmers can easily create different application structures by instantiating high-level pattern constructors and combining them in compositions. However, it can be complex to find a pattern composition that provides Quality of Services (QoS) under dynamic executions (e.g., stream processing). Supporting dynamic changes in the pattern compositions used is expected to be a potential solution for abstracting complexities from users/application programmers and at the same time providing QoS or efficiency. For instance, alternative pattern compositions could be automatically discovered and instantiated, then the best one can be found and activated transparently.

Dynamically reshaping the pattern compositions can be relevant for several domains. In this work, we focus on stream processing applications where it tends to be more challenging. First, effective and safe mechanisms for applying changes are complex. Second, there is a need for more generic strategies for self-adaptive decision making. Third, applying changes can have detrimental effects on the QoS like application downtime[1]. In this work, we intend to tackle these main challenges. To the best of our knowledge, the novel scientific contributions provided in this paper are:

- A strategy for providing self-adaptiveness and deciding which parallel pattern composition to use. This strategy is expected to be generic enough for different frameworks and applications.
- Integration of the proposed strategy into a C++ programming framework (FastFlow).
- Experimental evaluation of the proposed strategy and runtime mechanisms with stateless applications.

This paper is organized as follows. Section II shows the motivational context of this work as well as related approaches. Then, Section III presents the proposed solution and Section IV provides an experimental evaluation. Finally, Section V highlights the conclusions and perspectives of this paper.

[1] A time period where a given application does not produce output

II. Context

A. Motivation

Attempting to simplify the parallelism exploitation, parallel patterns are usually composed and combined by users/programmers creating pattern compositions (a.k.a. stream graph, graph topology). In Figure 1 we show pattern composition configurations [2], where a number of functions (f1, f2, f3) are decomposed in stages. A representative for stream processing applications, there is a data source and at least a Sink stage that will collect the results for producing an output. In the middle part, different compositions can be used according to specific application characteristics and user goals.

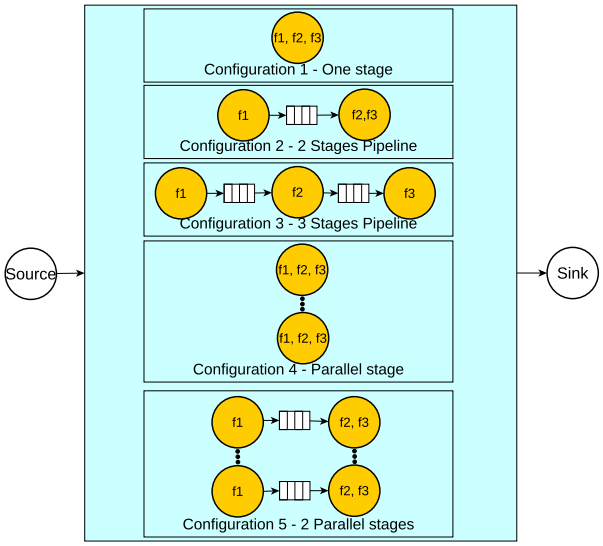


Fig. 1: Example of compositions for stream processing.

Configuration 1 represents a sequential stage (1S.) running the three functions. *Configuration 2* separates the functions into two stages (Pipe-2S.), whereas *Configuration 3* runs with one more stage(Pipe-3S.). Considering that some applications or performance goals are not suitable for sequential stages, *Configuration 4* shows an example of a pipeline with a parallel stage (P.E.1) running all functions. Considering that functions can be decomposed into multiple parallel stages, *Configuration 5* provides a variation of *Configuration 4* where 2 parallel stages (P.E.2) are employed, which can be useful for applications that are not embarrassingly parallel. For instance, there can be an internal state that prevents the easy replication of independent computations, i.e., first performing (in parallel) a filtering step, then computing the filtered data.

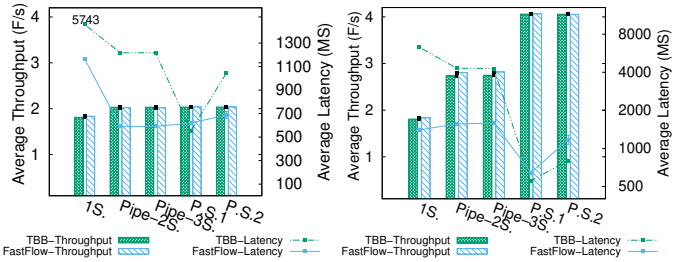
A stage can be represented as a node in the programming framework, where an important characteristic is the mapping of nodes to threads. In some cases, the nodes are mapped and executed by software threads. There is also the concept of tasks that are logical entities executed as independent operations, where software threads compute the tasks, i.e., a given computation in a stage can be processed as a task. Importantly, the mapping of nodes to threads and the pattern compositions shape the application structure, which can have a high impact on the application's performance and resource consumption. For instance, in a mapping of each node to a thread (one to one), there is only one software thread in the middle part of *Configuration 1*, such a configuration is suitable only for applications with a low-performance demand.

[2]Here the terms composition and configuration are used interchangeably.

Figure 2 shows the performance of a stream processing application (setup described in Section IV) with the configurations from Figure 1 using two programming frameworks: Intel TBB [14] and FastFlow [15]. The data arrives at a fixed input rate (IR) of 2 frames per second (F/s), where 2 is a suitable throughput for sustaining the IR. Latency is another relevant metric that corresponds to the time taken to compute a given item, low latency is a constraint for many applications.

Figure 2a evinces that in FastFlow *Configuration 2* was the best one by sustaining the input rate, providing lower latency, and using fewer nodes that consume fewer resources. In case other entities were being adapted this best pipeline configuration with a given number of stages would not be achievable. For instance, adapting the number of workers (parallelism degree) would only be suitable for configurations using parallel stages. In TBB only the configuration with one parallel stage achieved a competitive latency. Under a higher input rate, Figure 2b shows that only the configurations with one parallel stage sustained the input rate with low latency.

The results from Figure 2 emphasize that different configurations can be necessary to be used at run-time because the input rate can change due to network fluctuations or variations in the number of devices producing data [3]. Resource availability can also fluctuate in shared/dynamic environments like Clouds. Consequently, stream processing applications are expected to support dynamic adaptations at run-time. Considering that there are several aspects that correlate in a nonlinear manner, the user/programmer should not be expected to on-the-fly hand-tune the configurations. A solution is to support users/programmers to set only high-level goals like throughput or latency and rely on expert strategies that enforce a suitable QoS by finding and enforcing the best parallel pattern configuration at run-time.



(a) Input Rate 2 F/s. (b) Input Rate 4 F/s.

Fig. 2: Example on a Video Processing App.

B. Related Work

Several entities can be optimized at run-time [1]. For instance, although being complex to manage with applications' metrics like throughput and latency, batching can be used as an optimization in some application scenarios [9], [10].

The number of cores and their frequency can be changed at run-time for reducing energy consumption [8]. Another optimization consists of dynamic tuning the communication queues' concurrency modes [16]. More related to the parallelism exploitation, there are works changing dynamically the degree of parallelism of parallel stages [5], [6], [17], [17], [7].

However, the aforementioned optimizations are not as flexible as they target specific scenarios with difficulties for covering different pattern compositions that real-world applications may need. Changing the pattern compositions at run-time has been proposed as a more powerful adaptation strategy [12].

Dynamic compilation [11] was proposed as a solution for changing at run-time to alternative configurations. However, dynamic recompilation for stream processing applications is still limited because it requires suspending and restarting the application affecting the QoS (latency glitch, throughput spikes, or downtime) [1].

In this vein, concurrent recompilation has been proposed for reducing application downtime [12]. However, the techniques needed for controlling downtime are intrusive, which can affect the task's processing (ordering, throughput) and consume additional resources. Additionally, we have seen in practice that this approach is hard to generalize to other applications and programming frameworks.

Considering the QoS cost of reconfigurations, [2] proposed network-aware optimizations specific for route-maps in Storm's topologies. However, the approach of Rapolu *et al.* [2] optimizes lower-level aspects from the network instead of pattern compositions.

Contrasting with the related approaches, we propose a decision making strategy that is expected to be more generic by avoiding the need for overwhelming technicalities (e.g., input duplication, resource throttling) for changing the pattern compositions. Moreover, the implementation of the proposed solution avoids the need for recompilation by creating multiple pattern compositions and finding at run-time the best one to satisfy the user-defined QoS.

III. Proposed Solution

A. Design Goals and Requirements

An effective approach of dynamic compositions for stream processing have different goals and requirements. 1) control loop with periodic monitoring applying changes only when necessary. 2) the adaptation should not cause application downtime. 3) smooth transition between configurations. 4) a suitable approach is lightweight without demanding a significant extra amount of resources and optimizes resource consumption while achieving the user goals.

B. Decision Making Strategy

Proposing a flexible and generalizable decision making strategy demands several assumptions to abstract specific implementation technicalities. Runtime mechanisms should be available for applying changes in programming frameworks. Moreover, the strategy receives a number of configurations to be tested i.e., by a user or system. Finally, external entities fed the strategy with information and alerts for making decisions.

The designed self-adaptive decision making for the pattern compositions is described at a high-level abstracting the formalism. Such a description is expected to be sufficient for the reproducibility of the proposed solution that has three steps:

1) Training step: it activates each configuration for a time interval (e.g., 1 second) and collects execution statistics.
2) Optimal configuration: is the one that sustains QoS and needs fewer nodes. If the user goal is not achievable, enforces the configuration with the closest value.
3) Steady-state: returns to step one if the monitoring detects changes or if the user-defined goal is violated.

The proposed strategy is expected to enable non-functional requirements for stream processing. The users/programmers set an objective to be pursued by the self-adaptive strategy. Detecting fluctuations is an important part where values are considered significantly different when they have a contrast equal to or higher than a threshold of 20%, the suitability of such a value was ascertained in [18]. Moreover, pursuing stability and avoiding response to minor fluctuations, adaptation is triggered when three successive values indicate a change.

C. Implementation

Frameworks and libraries available were considered for implementing the proposed solution. There are available industry and academic solutions such as Intel TBB [14], FastFlow [15], and SPar [19]. Considering the support for performing adaptations at run-time, TBB supports only dynamic task distribution and load balancing for stream processing applications, where other adaptations have to be implemented at the lower level. Considering that we are interested in higher-level abstractions, FastFlow is more flexible by supporting dynamic adaptation on several aspects like the parallelism degree and communication queues' concurrency modes [16]. Thus, FastFlow was used for implementing the proposed solution.

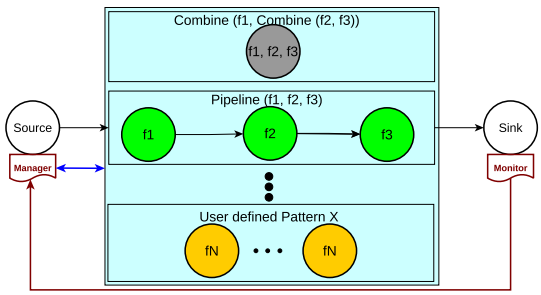


Fig. 3: Proposed solution implemented in FastFlow.

Abstracting specific implementation technicalities, the proposed self-adaptive strategy was implemented in FastFlow in the form of a ready-to-use C++ header-only library. The solution works by default in FastFlow's blocking mode. Figure 3 provides a representation of the implementation, where one entity is the $Manager$ is embedded in the data source and uses runtime mechanisms for applying changes in an autonomous mode. Another entity is $Monitor$ implemented as another embedded entity within the Sink stage that periodically collects data and feds the $Manager$. Figure 3 shows some possible configurations from Figure 1, where a pipeline with 3 stages is active and the other is inactive. Moreover, the lower part of Figure 3 demonstrates the achievable flexibility because several other configurations can be composed by the user/programmer.

In our solution, the business logic code inside the application functions is reused instead of duplicated. The user declares a parallel pattern using the FastFlow skeleton library. For instance, the three staged pipeline used in Figure 3 can be declared and added with two C++ code lines, other compositions can be declared and included with similar coding productivity. In addition to including headers and patterns instantiation, the self-adaptive strategy only requires two extra code lines for calling the $Manager$ and $Monitor$.

The implementation covers a smooth transitioning from one configuration to another. It is important to prevent two configurations to be active at the same time because it causes unpredictable performance variability or losses, such as throughput peaks and latency glitches [12]. This is tackled in our solution with a draining phase that estimates the amount of time to wait for the active configuration to finish its computations before sending tasks to the next configuration.

IV. Evaluation

A. Experimental Setup

A multicore machine equipped with an Intel Xeon processor 2.40 GHz (12 cores- 24 threads) and 32 GB of memory was used for running experiments. The operating system is Ubuntu Server 16.04 and G++ compiler (7.5.0) with -O3 flag. The runtime buffer sizes were set to 1. The configurations illustrated in Figure 1 were used for evaluating if the proposed strategy is able to find the best configuration. Testing five configurations in different applications can be considered a pessimistic scenario that increases the training step. However, each application can have specific configurations and our solution provides flexibility for programmers to instantiate the configurations to be tested.

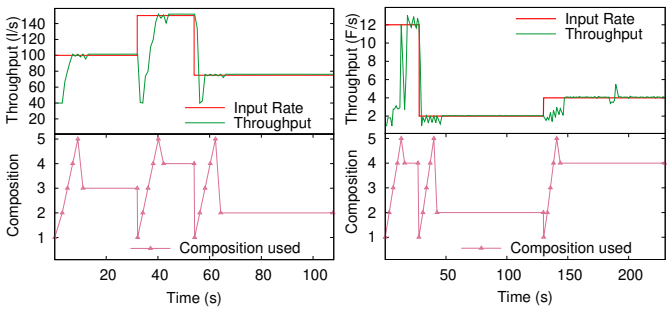
The strategy is characterized in a scenario simulating unexpected input rate changes. The performance is also evaluated with static executions using the same configurations as a baseline, where parallel stages configurations used a parallelism degree equal to the number of cores. Moreover, the $Adaptive$ (abbrev. $Adapt.$) executions are the ones relative to the proposed strategy and $Adaptive - A.T.$ (abbrev. $Adapt. - A.T.$) refers to the performance collected after the training step. Executions correctness was ascertained by hashing the outputs.

B. Experimental Results

The first application is a synthetic where 10,000 stream items are processed and each one has a service time of 24 milliseconds (ms). Figure 4a shows results from the execution of the self-adaptive pattern composition configurations, where the user-defined goal is throughput in items per second (I/s) equal to the input rate. The execution starts with a training step of the self-adaptive strategy that tests each configuration. After the training, the decision making enters a stable phase with configuration 3, which is a 3 staged pipeline configuration that sustained the input rate demanding the fewest amount of resources. Then, around the second 30, another training step was necessary because the input rate changed, where the strategy stabilizes with configuration 4. Another training step was performed when the input rate changed again to 75 I/s, where the execution stabilized with configuration 2. From a QoS perspective, it is possible to note that the self-adaptive strategy was able to effectively change and find the best configuration for achieving a suitable throughput.

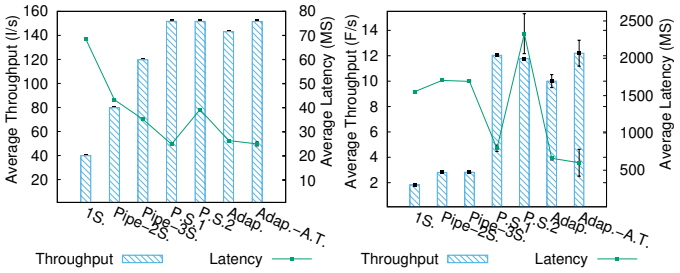
Results from real-world applications are also provided with a customized version of the Person Recognition application [20], which has multiple functions to recognize people in video streams. The input used was a 30 seconds long video with frames resolution of 260 pixels. Figure 4b shows the results where the goal of the self-adaptive strategy was to achieve an application throughput that would sustain the input rate. Importantly, the input rate varies from 2 to 12 tasks per second, wherein in this application, each task is a video frame. Although the throughput fluctuated when testing suboptimal configurations, the proposed solution effectively reconfigured to find the best configuration.

Figure 5 provides performance results with the metrics of throughput and latency. The collected metrics are an average of all processed tasks in a given execution. Moreover, each execution was repeated ten times (except in $Adaptive - A.T.$). $Adaptive - A.T.$ is a relevant outcome of the performance without the overhead of the training step. Figure 5a shows a representative execution of the synthetic application with an input rate of 150 I/s, where the best static configuration was with a parallel stage. Noteworthy, this was the configuration used by the self-adaptive strategy showing its effectiveness.



(a) Synthetic Application. (b) Person Recognition.

Fig. 4: Self-adaptiness Characterization.



(a) Synthetic Application. (b) Person Recognition.

Fig. 5: Performance Evaluation.

Figure 5b shows a representative result of the Person Recognition application with an IR of 12 F/s. The self-adaptive strategy was effective by choosing a parallel stage after the training step, which is the best static configuration under the high input rate of 12. However, the average throughput is lower than the static one, because the input used was not large with a short execution time of around 75 seconds. Consequently, the adaptive execution spent a significant amount of time in training (around 15 seconds) that reduced the throughput average. In long-running executions, the training impact could be lower with a performance similar to the $Adaptive - A.T.$.

V. Concluding Remarks

In this paper, we presented a solution for supporting self-adaptive pattern compositions, which was validated in stream processing applications. There are implications of the achieved results as well as some limitations that are relevant to emphasize. The results show that the proposed solution is technically feasible and that the proposed strategy is effective for adapting pattern compositions at run-time. Importantly, the adaptation is possible with competitive performance. A relevant implication of these results is that new abstractions can be provided for users/programmers. The dynamic adaptations and self-adaptive executions can provide additional flexibility for improving QoS (throughput, latency) and/or system efficiency.

This study is limited in some aspects. The solution was designed to be generic, but mechanisms are necessary for the programming framework for achieving self-adaptive pattern compositions on each scenario. FastFlow framework provides these mechanisms, but we argue that other tools can be extended to support flexible adaptations at run-time.

Moreover, the adaptation space can be limited by the alternative configurations provided, but this potential limitation can be mitigated. One way is to increase the adaptation space by combining the dynamic compositions with other less flexible optimizations such as batching and parallelism degree.

In future works, we intend to provide techniques for optimizing the training step of the decision making, i.e. black-listing inappropriate configurations. Moreover, we intend to validate our solution on stateful stream processing applications and with other workloads.

Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, FAPERGS 05/2019-PQG ParAS (Nº 19/2551-0001895-9), and Universal MCTIC/C-NPq Nº 28/2018 SParCloud (No. 437693/2018-0), and by Univ. of Pisa PRA_2018_66 "DECLware: Declarative methodologies for designing and deploying applications".

References

[1] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM CSUR*, vol. 46, p. 46, 2014.

[2] N. Rapolu, S. Chakradhar, and A. Grama, "Vayu: Accelerating stream processing applications through dynamic network-aware topology re-optimization," *JPDC*, vol. 111, pp. 13–23, 2018.

[3] 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 Workshops*, vol. 11997. Springer, 2019, p. 12.

[4] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury, *Feedback Control of Computing Systems*, ser. Wiley - IEEE. Wiley, 2004.

[5] T. D. Matteis and G. Mencagli, "Keep Calm and React with Foresight: Strategies for Low-latency and Energy-efficient Elastic Data Stream Processing," *SIGPLAN Not.*, vol. 51, no. 8, pp. 13:1–13:12, Feb. 2016.

[6] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE TPDS*, vol. 25, pp. 1447–1463, 2014.

[7] A. Vogel, D. Griebler, M. Danelutto, and L. G. Fernandes, "Seamless Parallelism Management for Multi-core Stream Processing," in *Intl. Conference on Parallel Computing (ParCo)*, vol. 36. Prague, Czech Republic: IOS Press, September 2019, pp. 533–542.

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

[9] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive Stream Processing using Dynamic Batch Sizing," in *ACM Symp. on Cloud Computing*. ACM, 2014, pp. 1–13.

[10] P. Metzger, M. Cole, C. Fensch, M. Aldinucci, and E. Bini, "Enforcing deadlines for skeleton-based parallel programming," in *IEEE Real-Time and Embedded Technology and Applications Symp.*, 2020, pp. 188–199.

[11] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, "Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures," in *Intl. Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2009, pp. 214–223.

[12] S. Rajadurai, J. Bosboom, W.-F. Wong, and S. Amarasinghe, "Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 98–112, 2018.

[13] M. Danelutto, G. Mencagli, M. Torquati, H. González-Vélez, and P. Kilpatrick, "Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming," *IJPP*, pp. 1–22, 2020.

[14] M. Voss, R. Asenjo, and J. Reinders, *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. Apress, 2019.

[15] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: High-Level and Efficient Streaming on Multicore," *Programming multi-core and many-core computing systems, parallel and distributed computing*, pp. 261–280, 2017.

[16] M. Torquati, D. D. Sensi, G. Mencagli, M. Aldinucci, and M. Danelutto, "Power-aware Pipelining with Automatic Concurrency Control," *CCPE*, vol. 31, no. 5, p. e4652, 2019.

[17] A. Vogel, D. Griebler, and L. G. Fernandes, "Providing high-level self-adaptive abstractions for stream parallelism on multicores," *Software: Practice and Experience*, pp. 1–24, 2021.

[18] 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 Workshops*. Springer, 2018, pp. 28–39.

[19] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes, "SPar: A DSL for High-Level and Productive Stream Parallelism," *Parallel Processing Letters*, vol. 27, no. 01, p. 1740005, March 2017.

[20] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes, "Higher-Level Parallelism Abstractions for Video Applications with SPar," in *Intl. Conference on Parallel Computing*. Bologna, Italy: IOS Press, September 2017, pp. 698–707.