Autonomic and Latency-Aware Degree of Parallelism Management in SPar

Adriano Vogel[1], Dalvan Griebler[1], Daniele De Sensi[2], Marco Danelutto[2], and Luiz Gustavo Fernandes[1]

[1] Pontifical Catholic University of Rio Grande do Sul (PUCRS)
adriano.vogel@acad.pucrs.br
[2] Department of Computer Science, University of Pisa (UNIPI)

Abstract. Stream processing applications became a representative workload in current computing systems. A significant part of these applications demands parallelism to increase performance. However, programmers are often facing a trade-off between coding productivity and performance when introducing parallelism. SPar was created for balancing this trade-off to the application programmers by using the `C++11` attributes' annotation mechanism. In SPar and other programming frameworks for stream processing applications, the manual definition of the number of replicas to be used for the stream operators is a challenge. In addition to that, low latency is required by several stream processing applications. We noted that explicit latency requirements are poorly considered on the state-of-the-art parallel programming frameworks. Since there is a direct relationship between the number of replicas and the latency of the application, in this work we propose an autonomic and adaptive strategy to choose the proper number of replicas in SPar to address latency constraints. We experimentally evaluated our implemented strategy and demonstrated its effectiveness on a real-world application, demonstrating that our adaptive strategy can provide higher abstraction levels while automatically managing the latency.

Keywords: autonomic computing, stream processing, parallel programming, adaptive degree of parallelism

1 Introduction

Stream processing applications gained even more attention in the recent computing age due to the increasing use of techniques to collect data from different sources (*e.g.,* sensors, cameras, radars). These applications are characterized by a continuous flow of data and high variance of input data rates [3,2]. In addition to that, due to the growing of data generation, parallel programming can be used in stream processing applications as an option for increasing performance. A set of programming frameworks and libraries were developed to allow the stream parallelism exploitation on multi-core systems. Examples are Intel Thread Building Blocks (TBB) [14], FastFlow [7,1], and StreamIt [17]. Despite

the coding abstraction introduced by these programming frameworks, they are still not abstract enough for application programmers, which are the ones focused on developing the stream processing application [10] and which may not be parallel programming experts.

To raise the abstraction level on stream parallel applications, the SPar [9] DSL (domain-specific language) was designed for parallelizing stream processing application in a simpler and more productive way than the state-of-the-art alternatives [10]. SPar maintains the sequential structure of C++ codes and programmers identify regions that can run in parallel. The programmer can annotate these regions by using C++11 attributes, and the SPar compiler will parse such annotations and generate the associated parallel code. Some regions can be executed concurrently by a number of entities called *replicas*. In SPar, as well as in other state-of-the-art frameworks, the number of concurrent entities (i.e., the degree of parallelism) is static and must be manually set by the programmer. Choosing a proper number of replicas is a complex task, since the best choice depends both on the arrival rate of the data but also on the performance requirements for the specific application. For example, while having more replicas can improve the throughput, it could also increase the latency required to process the stream items. Unfortunately at moment being, SPar and other state-of-the-art frameworks (TBB, FastFlow, and StreamIt) do not provide any automatic and latency-aware strategy for selecting the most appropriate number of replicas.

In this work, we propose a strategy to automatically set, without any user intervention, the number of replicas to be used in parallel applications with SPar. The optimal number of replicas will be selected according to the latency requirements of the application. The main contributions of this work are:

- An extension of the SPar DSL [9,10] with a new parallelism abstraction. This abstraction is achieved by a strategy to automatically adapt the number of replicas in SPar that is fully abstracted from the application programmer. The adaptation mechanism is designed based on a feedback loop, through which a specific latency Quality of Service (QoS) is provided. The application is monitored at run-time and the adaptation strategy periodically takes actions to optimize the number of replicas, considering the latency of stream items. Consequently, the adaptation strategy concerns stream processing applications sensitive to latency.
- An experimental evaluation of the effectiveness of the strategy running on a stream processing application.

The remainder of this paper is organized as follows: the next section presents the scenario of this study. The need for low latency in stream processing applications is emphasized in Section 3. Section 4 presents the strategy that manages the latency by adapting the number of replicas. In Section 5 we present our experimental evaluation. Then, the related work is discussed in Section 6. Eventually, in Section 7 we draw the conclusion and discuss some possible future directions for this work.

2 An Overview of SPar

SPar[3] is a DSL for stream parallelism that offers high-level C++11 attributes to enable automatic parallelization by means of source code annotations. The parallel code is generated by SPar compiler through source-to-source transformations [9]. SPar relies on the FastFlow runtime, a high-level and pattern-based parallel programming library [7,1]. SPar's compiler generates parallel code using FastFlow library through source-to-source transformations. SPar also allows code parallelism by simply adding annotations in the original sequential code. By doing so, SPar relieves the programmers from the effort in dealing with advanced concepts such as scheduling, load balancing and parallelism strategies. Since SPar is based on the C++ standard interface, application programmers do not need to learn a new language for parallelizing their code, and can just focus on the functional parts of their applications.

SPar provides five attributes, which we describe in the following to exploit key aspects of stream parallelism (Listing 1.1 presents a use case example). The `ToStream` attribute represents the beginning of a stream region with the production of the stream elements. Inside the *ToStream* section, it is possible to add a number of `Stages`, which represents different and subsequent phases of the computation over the stream elements. The data needed by each stage can be indicated by using the `Input` attribute. Similarly, by using the `Output` attribute, the programmer can specify the variables representing the data produced by the stage.

Each stage can be executed by multiple threads. To define how many threads (replicas) should be used for a stage, the `Replicate` attribute can be used. As SPar currently supports stateless stream operators, each replica is independent from the others and they can operate in parallel without any need of synchronization among them. During the source-to-source compilation process, some flags can be specified to customize the behaviour of the generated code. For example, to change the way in which the elements are scheduled to the replicas or to preserve the order of the stream elements among different stages [9].

In Listing 1.1, we show a trivial example of a sequential code enhanced by means of SPar annotations. This application generates the stream, applies a function over each stream element, and then outputs the results. In Figure 1, we can visualize the association between the different parts of the code and the execution unit, which will be executed in parallel.

```
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 1.1. SPar example.

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

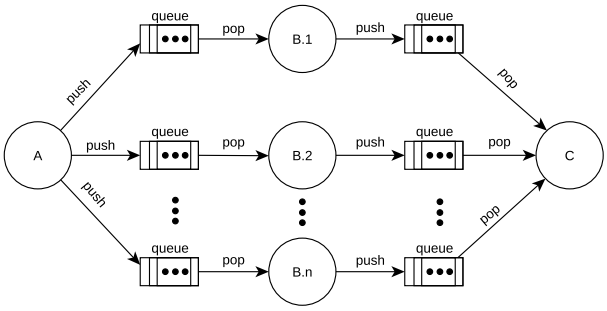


Fig. 1. Farm - communication queues inside SPar runtime.

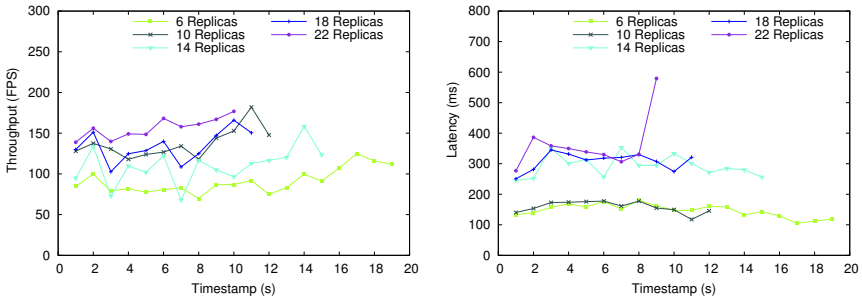
In this specific example, n replicas are activated (which corresponds to B.1 to B.n in Figure 1), each of which receive data from the previous stage and sends produced results to the subsequent stage. Communications between stages occur through shared queues. By default, SPar schedules the stream items to the workers with a round-robin policy. However, other scheduling strategies can be used and this behaviour can be customized during the source-to-source compilation process. For example, to improve load balancing, it is possible to schedule stream items in an *on-demand* fashion so that an element is scheduled to a specific worker when it is not already processing another element.

3 The impact of parallelism on latency

In this section we describe the relationship between the number of replicas and the performance in a stream processing application. We consider the *Lane Detection* application [11], a video processing application used to identify road lanes in videos recorded, for example, by self-driving vehicles. This application has a similar structure to that shown in Figure 1 (3 stages) where one stage is replicated by a number of times. In the experiments, we used as input a video file (5.25MB - 640x360 pixels) to simulate a typical execution of a video streaming application. We execute this application on a multi-core machine composed by 12 cores with 2-way Simultaneous MultiThreading (SMT) for a total of 24 hardware threads.

Firstly, we show in Figure 2(a) the throughput of the application (i.e. how many stream elements per second are processed) for different number of replicas. The number of replicas is statically chosen and never modified during the execution. These results prove that the use of SMT is beneficial for the throughput of this kind of application since the best throughput is obtained by using 22 replicas.

As shown in Figure 2(b), increasing the number of replicas may have detrimental effects on the latency of the application. It is worth noting that a significant increase in the latency (as well as a decrease in the throughput) can be



(a) Throughput. (b) Latency.

Fig. 2. Lane Detection characterization.

observed when more than 10 replicas are used. Moreover, it is possible to note a significant increase in the oscillation of the latency when using more replicas. These effects are caused by the contention between stages running on two SMT cores corresponding to a same physical core.

There can be seen a correlation between throughput and latency. Achieving a high throughput using many replicas tends to increase the latency. On the other hand, using too few replicas decreases the throughput and latency. Consequently, a balance between the two performance goals is required. The challenge is that a high throughput is commonly pursued, and at the same time low latency may also be necessary. In this work, the goal is to manage latency in replicated stages.

4 Autonomous Degree of Parallelism

In the previous section we have seen how the number of replicas affects the latency of stream items. Responding in real-time according to latency constraints and the actual rates cannot be done manually by the programmer. As a consequence, we are abstracting from programmers the aspects related to the number of replicas and latency for latency sensitive applications.

We implemented a strategy in the SPar's runtime that monitors and manages the latency of stream items by adjusting the number of replicas. Figure 3 shows the architecture we use to adapt the number of replicas considering the monitored latency of stream items. This adaptive mechanism is based on a feedback loop [13] that at each *control step*, monitors the application and takes decision so to optimize the execution of the application at the next step. By doing so, it is possible to be reactive to select the best number of replicas even in presence of workload fluctuations, which is common in data streaming applications. The implemented strategy works on a single replicated stage. However, this strategy can also work in more complex compositions formed by several replicated stages, although different strategies would need to be handled by other means.

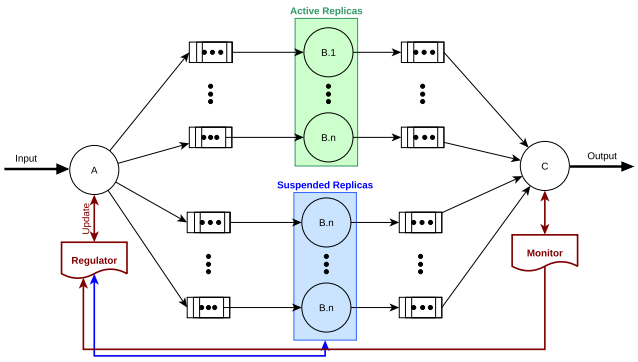


Fig. 3. Latency - Regulator and Monitor.

A *monitor* routine is attached to the last stage of the application. It monitors the latency of each stream element and calculates the average latency of the elements processed in each iteration of the feedback loop. The latency calculated by the monitor is read by a *regulator* connected to the first stage of the application. The regulator, by using the information collected by the monitor, decides which actions to take at the next step of the feedback loop in order to enforce the latency required by the user.

In Algorithm 1 we show the regulator used in this work. It calls the monitor for the current latency and when it is higher than the target one, the number of replicas is reduced. On the other hand, the regulator increases the number of replicas if the latency is significantly lower than the constraint. The part that dynamically regulates the parallelism was implemented using low-level calls to the FastFlow runtime library for changing the status of the replicas (active, suspended). The regulator changes the number of replicas at run-time without restarting the application. In order to avoid oscillation in the number of replicas, a threshold value is used so that the number of replicas is not increased when the latency is lower but close to the constraint. This strategy of the regulator tries to maximize throughput while the latency constraint is met, pursuing a balance between throughput and latency requirements.

Algorithm 1 Parallelism Regulator

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

By considering the example in Figure 3, if executed on a machine with N cores, we would activate at most $N - 2$ replicas. Indeed, as we shown in Figure 2(b), when the replicas share the computing resources with other stages of the application, this could lead to detrimental effects for both latency and throughput of the application. The regulator we shown in Algorithm 1 assumes that at most one stage is replicated. If more stages are replicated, the strategy should find the best number of replicas for each of them. We will consider this scenario in our future work. Moreover, the implemented strategy works on stateless computations. In case of a stateful scenario, the internal state would need to be handled manually.

An important part of the configuration is the scaling factor (SF), which is how many threads/replicas are added or remove when adjusting the degree of parallelism. In the literature, the most common SF value is 1 threads/replicas. Our implementation is tested with SF of 1 and 2, thus in lines 5 and 7 of Algorithm 1, 1 or 2 replicas can be suspended or awaken on each iteration.

Another relevant aspect is how often the algorithm should consider the possibility of adding/removing replicas. The most common approach is time-driven that, at fixed time intervals, it decides if changing the number of active replicas. The choice of the time interval is critical and depends from the application. In general, a shorter time interval allows to react quickly to changes in the application. In [8], [4], [15], the authors used time intervals ranges from 0.1 to 5 seconds. For our scenario, we consider 1 second as the default time interval. We experimentally saw that this configuration avoids too many changes in the number of replicas, but also maintain a correct level of sensitivity to application fluctuations. The impact on latency caused by the different choices of the time interval is left to be evaluated in the future.

5 Results

Stream processing applications may run only pursuing the maximum throughput without considering the latency. However, it is not suitable for those latency sensitive applications that need to rapidly return their results. At the same time, using a minimal number of replicas for reducing the latency tends to result in a low throughput as well as inefficient usage of computational resources. Therefore, our regulator tries to improve the throughput by increasing the number of replicas when the latency is below the constraint. We tested our strategy for latency with the same application and input used in Section 3. In this experiments, the scaling factor (SF) of the parallelism regulator was 1 or 2, meaning that on each reconfiguration one or two replicas can be activated or suspended. Also, we used a control step of 1 second, which is a time interval sensitive enough to react without compromising the overall execution. Another aspect tested is related to the thresholds of the latency constraint presented in Section 4. In our scenario, the best thresholds were 10% and 20%.

In Figure 4, we show on the left side the throughput and latency of the application, while on the right side we plot the number of replicas used during

the execution. In this experiment, we set a latency constraint of 180 milliseconds with a 10% threshold. As we can see from the Figure 4, the number of replicas is reduced when the latency increases, and the number of replicas is changed several times due to oscillations in the input video. Comparing the configurations, we observed that SF of 2 reacts faster to changes and increases the throughput at the price of more latency violations.

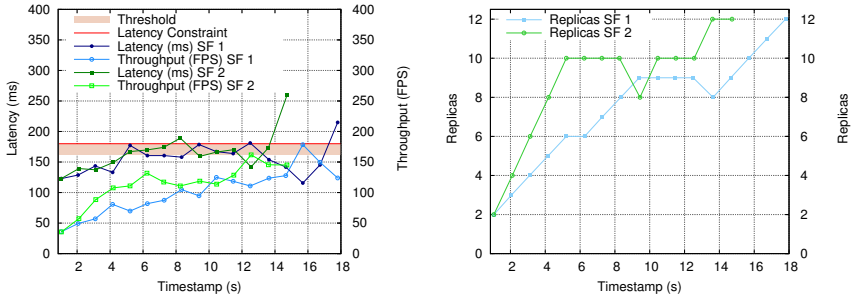


Fig. 4. Threshold 10% - Latency constraint of 180 ms (Left) and replicas used (Right).

In Figure 5 is presented an experiment with the same latency constraint but using a threshold of 20%. In this experiment, fewer latency violations occurred because the threshold of 20% is more conservative, which avoids adding more replicas when the latency is close to the constraint. Comparing thresholds 10% and 20%, we noted that the effectiveness of threshold 20% in managing the latency did not decrease the application throughput significantly. Moreover, SF of 1 is more stable by avoiding to overreact in the face of latency oscillations caused by the application workload fluctuations.

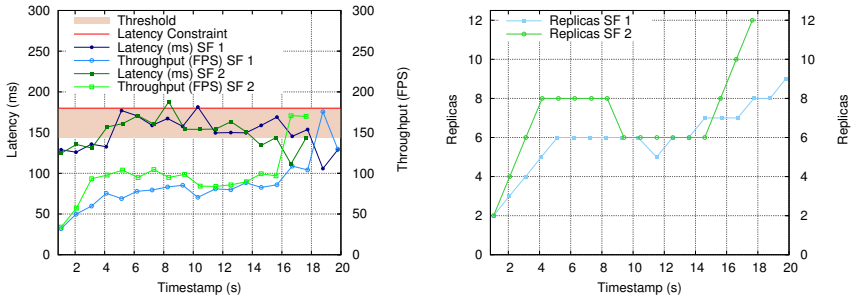


Fig. 5. Threshold 20% - Latency constraint of 180 ms (Left) and replicas used (Right).

An experiment tolerating higher latency (200ms) is shown in Figures 6 and 7. Despite the different constraint, the performance trend from the configurations is

similar. The threshold of 10% resulted in too many re-configurations that caused latency violation by using too many replicas. Thanks to fewer latency violations, SF of 1 was most suited than SF of 2. Considering the SF of 1, that yielded the best trade-off between latency and throughput, the results revealed a similar throughput regarding the thresholds 10% and 20%. Using the threshold of 20%, it only violated the latency constraint in the last seconds of the execution. This event is not caused by the adaptive mechanism but by the application, and it also occurred with a static number of replicas as seen in Figure 2(b). Consequently, the adaptive mechanism was unable to respond because the latency violations occurred right before the application termination. Considering these results, we

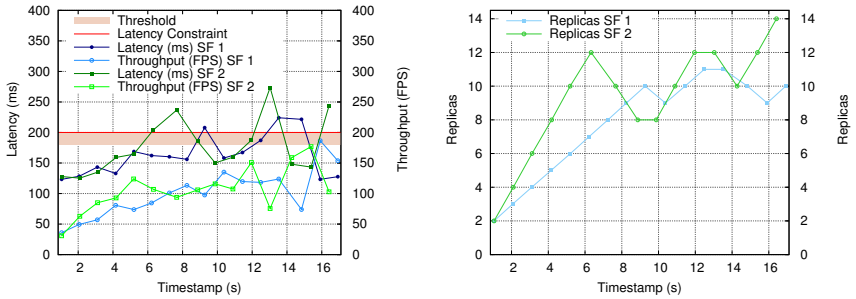


Fig. 6. Threshold 10% - Latency constraint of 200 ms (Left) and replicas used (Right).

can highlight that latency-sensitive stream processing applications with fluctuations perform better using SF of 1 and higher thresholds. In fact, an acceptable performance depends on the constraints and on the user requirements. Often in stream processing applications, a high throughput does not mean that users will actually have a better experience [6]. Consequently, it is important to support custom configurations (*e.g.,* throughput, latency) and to adapt the application at run-time while maintaining high-level parallelism abstractions.

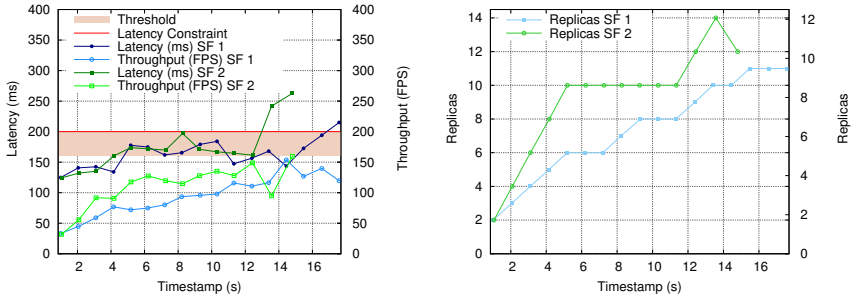


Fig. 7. Threshold 20% - Latency Constraint of 200 ms (Left) and Replicas used (Right).

6 Related Work

In this section, we present and contextualize the related studies that present efforts for autonomous properties in the stream processing scenario. De Sensi et al. [6], [5] propose `Nornir`, a simple programming interface and runtime support for dynamically and automatically control the resources allocated to the application according to the user needs. `Nornir` enables the application to change number of cores, clock frequency and placement of threads during run-time. It also aims to satisfy bounds regarding power consumption and throughput, even in presence of changes in the input rate, in the application phases, or external interference. `Nornir` is validated using simulations and real-world benchmarks from the PARSEC suite. In SPar, we do not focus on power-aware computing. With respect to `Nornir`, we provide the possibility to express latency constraints by adapting the number of replicas.

De Matteis et al. [4] present elastic properties for data stream processing regarding performance (latency) and energy efficiency (number of cores and frequency). Elasticity support is stated as a solution for an efficient usage according to QoS requirements and so reducing the operating cost. The proposed model was implemented in the FastFlow runtime, which is a framework for stream processing targeting shared-memory multi-core architectures and also used by our target SPar runtime. In this work, the authors use a controller thread to monitor the application and to change the number of replicas and the clock frequency of the CPUs when needed.

In Gedik et al. [8], the authors show aspects related to parallelism in pipeline stages and they presented the motivation and challenges for elastic degree of parallelism during run-time. They proposed an elastic auto-parallelization solution, which adjusts the number of replicas aiming to achieve high throughput without wasting computational resources. Elasticity is implemented by requiring the programmer to define a threshold and a congestion index in order to decide whether to add or not more replicas.

Heinze et al. [12] emphasizes the complexity involved in determining the right point to increase or decrease the degree of parallelism. The authors investigated issues and requirements related to elasticity in the data stream for auto-scaling (scaling in or out) and they manage latency in a distributed system by keeping the system utilization in a range (min, max).

Selva et al. [16] show an approach related to the adaptation in run-time for streaming languages. The StreamIt language is extended to allow the programmer to specify the desired throughput and the runtime controls the execution. Moreover, it was implemented an application and system monitor that checks the throughput and system bottleneck, respectively. Using the implemented strategy, the system can adapt the execution based on previous observations.

Our research differs from existing works because we provide autonomous degree of parallelism and latency-aware management for the SPar DSL, shown in Section 2. De Sensi et al. [6] and De Matteis et al. [4] used the FastFlow framework to implement autonomic management of energy consumption on parallel applications. Besides providing a new strategy for implementing the latency-

aware degree of parallelism, we integrated our strategy in the SPar's runtime system, adding therefore a new parallelism abstraction for its users.

We have a different scenario and target architecture compared to Gedik et al. [8] and Heinze et al. [12] because they focused on distributed systems while SPar targets multi-core environments. While Selva et al. [16] optimize the placement and throughput in StreamIt, we abstract parallelism complexities and focus on latency constraints for the SPar DSL. Moreover, the available solutions do not focus on parallelism abstractions and can be complicated to be used even for experts in parallel programming.

There is a demand to relieve end-users from the need to set a degree of parallelism and to enable their applications to run transparently without the manual intervention. We aim to free programmers from defining the degree of parallelism by implementing a strategy that supports an adaptive degree of parallelism in any application sensitive to latency parallelized by using SPar.

7 Conclusion

In this study, we extended SPar with a new parallelism abstraction. This was accomplished by implementing a strategy that adapts, without any programmer intervention, the number of replicas in order to have a latency lower than that specified by the application programmer. This is particularly useful for stream processing applications, which are characterized by fluctuations in the input rates. Our strategy monitors the execution and adapts the degree of parallelism. The manual, complex, and time-consuming definition of the degree of parallelism is no longer required in SPar. Experimental results demonstrated the effectiveness of our solutions when adjusting the number of replicas at runtime. Although the result trends are expected to occur in different scenarios, the presented results are limited to the tested application and environment.

In this study we proposed a strategy to control applications where only one stage is replicated. In the future, we plan to extend this work to consider applications with a more complex structure. Moreover, we aim to evaluate our latency-aware approach in other latency sensitive applications, specially those running for long time periods. Eventually, we will improve the adaptive strategy, for example, by using proactive rather than reactive approaches, to minimize the number of times the number of replicas is changed at run-time.

Acknowledgement

This work has been partially supported by the EU H2020-ICT-2014-1 project RePhrase (No. 644235), FAPERGS 01/2017-ARD project ParaElastic, and CAPES scholarships.

References

1. Aldinucci, M., Meneghin, M., Torquati, M.: Efficient Smith-Waterman on Multi-Core with FastFlow. In: Euromicro Conference on Parallel, Distributed and Network-based Processing. pp. 195–199 (2010)

2. Andrade, H., Gedik, B., Turaga, D.: Fundamentals of Stream Processing: Application Design, Systems, and Analytics. 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. Advances in Database Systems, Springer US (2009)
4. De Matteis, T., Mencagli, G.: Keep Calm and React with Foresight: Strategies for Low-latency and Energy-efficient Elastic Data Stream Processing. SIGPLAN Not. 51(8), 13:1–13:12 (Feb 2016)
5. De Sensi, D., De Matteis, T., Danelutto, M.: Simplifying Self-Adaptive and Power-Aware Computing with Nornir. Future Generation Computer Systems pp. – (2018)
6. De Sensi, D., Torquati, M., Danelutto, M.: A Reconfiguration Algorithm for Power-Aware Parallel Applications. ACM Transactions on Architecture and Code Optimization 13(4), 43 (2016)
7. FastFlow : FastFlow (FF) Website (2017), `http://mc-fastflow.sourceforge.net/`, last access in Dec, 2017
8. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic Scaling for Data Stream Processing. IEEE Transactions on Parallel and Distributed Systems 25(6), 1447–1463 (2014)
9. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.G.: SPar: A DSL for High-Level and Productive Stream Parallelism. Parallel Processing Letters 27(01), 20 (2017)
10. Griebler, D., Filho, R.B.H., Danelutto, M., Fernandes, L.G.: High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. International Journal of Parallel Programming pp. 1–19 (2018)
11. Griebler, D., Hoffmann, R.B., Danelutto, M., Fernandes, L.G.: Higher-Level Parallelism Abstractions for Video Applications with SPar. In: Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing. pp. 698–707. ParCo'17, IOS Press, Bologna, Italy (2017)
12. Heinze, T., Pappalardo, V., Jerzak, Z., Fetzer, C.: Auto-Scaling Techniques for Elastic Data Stream Processing. In: IEEE International Conference on Data Engineering Workshops. pp. 296–302 (2014)
13. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems. John Wiley & Sons (2004)
14. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media (2007)
15. Schneider, S., Hirzel, M., Gedik, B., Wu, K.L.: Auto-Parallelizing Stateful Distributed Streaming Applications. In: Proceedings of the international conference on Parallel architectures and compilation techniques. pp. 53–64 (2012)
16. Selva, M., Morel, L., Marquet, K., Frenot, S.: A Monitoring System for Runtime Adaptations of Streaming Applications. In: Euromicro International Conference on Parallel, Distributed and Network-Based Processing. pp. 27–34 (2015)
17. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for Streaming Applications. In: International Conference on Compiler Construction. pp. 179–196. Springer (2002)