2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) | 978-1-6654-1455-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/PDP52278.2021.00021

Introducing a Stream Processing Framework for Assessing Parallel Programming Interfaces

Adriano Marques Garcia *, Dalvan Griebler*†, Luiz G. L. Fernandes*, Claudio Schepke‡
* 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.
‡Federal University of Pampa (UNIPAMPA), Alegrete, Brazil.
Email: {adriano.garcia, dalvan.griebler}@edu.pucrs.br, luiz.fernandes@pucrs.br, claudioschepke@unipampa.edu.br

***Abstract*—Stream Processing applications are spread across different sectors of industry and people's daily lives. The increasing data we produce, such as audio, video, image, and text are demanding quickly and efficiently computation. It can be done through Stream Parallelism, which is still a challenging task and most reserved for experts. We introduce a Stream Processing framework for assessing Parallel Programming Interfaces (PPIs). Our framework targets multi-core architectures and C++ stream processing applications, providing an API that abstracts the details of the stream operators of these applications. Therefore, users can easily identify all the basic operators and implement parallelism through different PPIs. In this paper, we present the proposed framework, implement three applications using its API, and show how it works, by using it to parallelize and evaluate the applications with the PPIs Intel TBB, FastFlow, and SPar. The performance results were consistent with the literature.**

I. Introduction

The demand for real-time processing has grown and traditional batch-oriented data processing is known not to be sufficient to keep up with this demand [1]. This way, organizations are increasingly adopting Stream Processing (SP) systems, which can process data in nearly real-time. In SP, data is continuously processed as new data becomes available for analysis, applying a series of small computations as stages in a pipeline, doing this processing incrementally [1]. SP is present in many sectors, such as: surveillance systems, signal processing, fraud detection, stock market, data compression, image/audio/video processing, etc. [2], [3].

SP applications require parallelism exploitation to accelerate the computation and process large volumes of data in a timely manner. This parallelism can be applied through different Parallel Programming Interfaces (PPIs). However, as the SP domain is growing, so is the development of new PPIs. In addition, there are many studies focused on evaluating PPIs [4], [5], [6], [7] or developing techniques to improve different aspects of them, such as self-adaptive parallelism [8], add new features [9], and support for new parallel abstractions [10], [6] and architectures [11], [3], [7].

At the moment, we lack SP benchmarks for developers and researchers to test and evaluate PPIs, techniques, and parallelism strategies. Even with the few existing solutions, evaluating these new technologies with different SP applications is a time-consuming task that shifts the programmer's focus away from the technology itself. In this paper, we introduce a Stream Processing framework for assessing PPIs. The goal is to provide a set of SP applications for the C++ community plus a framework that makes it easy for programmers to implement parallelism and evaluate PPIs and technologies. **The main contributions of this work are**: 1) A high-level framework API for SP applications; 2) A shell to manage the applications, add new parallel implementations, and collect performance metrics, such as latency, throughput, CPU and memory usage; 3) We evaluate our work with three PPIs: Intel TBB [12], FastFlow [13], and SPar [14].

II. Related Work

As related work we consider benchmarks suites for Stream Processing, suites that includes some Stream Processing applications, and other benchmark approaches for SP. We found no similar research idea to ours. NAMB (Not only A Micro-Benchmark) is a platform for a generation of prototype applications based on their high-level description [15]. It can generate a set of synthetic/micro-benchmarks as well as prototypes of Java applications for Apache Flink, Storm, and Heron platforms. The framework also allows users to change data input rates, degree of parallelism, load balancing, etc. RIoTBench [16] is a suite that regroups a large set of IoT micro-benchmarks to cover different patterns and common IoT tasks. Similarly, StreamBench [17] consists of 7 micro-benchmarks with 4 different synthetic work-load suites generated from real-time Web logs and network traffic to evaluate Distributed Stream Processing Systems (DSPSs). There is another suite called StreamBench [18] that also specific targets DSPS. It uses three micro-benchmarks and measures latency and throughput.

SparkBench [19] is a framework-specific suite for Apache Spark that evaluates CPU, memory, disk, and network IO, intending to identify the best configurations to improve Spark's performance. Bordin [2] proposed a benchmark suite to provide a common reference for DSPS evaluation. It includes 14 benchmarks from several domains using Storm and Spark. The author identified the most frequently used metrics in SP: latency, throughput, scalability, tuple loss, and resource usage. Within the suite, there is an API framework that allows users to run, collect metrics, and validate the resulting output.

StreamIt [20] is a compiler and programming language focused on SP applications. It comes within a benchmark

978-1-6654-1455-5/21/$31.00 ©2021 IEEE
DOI 10.1109/PDP52278.2021.00021

suite, but it only supports the StreamIt language and architecture. Moreover, it is limited to the dataflow and data stream domains. PARSEC [21] is a suite that includes three representative real-world stream applications (dedup, ferret, and x264), among others, implemented in POSIX Threads.

Most of the related work focus only on data stream applications [15], [16], [17], [18], [19], [2]. These are applications that intersect the domains of Big Data and IoT, which are developed using frameworks for DSPS platforms. Almost all of them are implemented in the Java language. The remaining related work do not focus on SP [20], [21]. They include some traditional SP applications but their benchmarks have several limitations in terms of language, parallelism exploitation, execution metrics, and parametric options. **In our work, we focus on applications for more generic stream processing targeting the C++ community**. However, we also intend to include C++ data stream applications using emerging libraries such as WindFlow [4] in the near future. **Our work also includes most of the metrics identified as important by some of the related work** [19], [2].

III. Framework Proposal

The main feature of our framework (Figure 1) is a set of Stream Processing applications implemented in the form of an API. To build our API, we disassemble all operators from the original application and put them individually into a new source code. This way, the application calls the operators by including a header file. Therefore, the cornerstone of the framework is composed of APIs representing each application and a set of sequential applications that instantiates them. As we parallelize these applications with different PPIs, the parallel implementations are also integrated into the set. In addition to the sequential applications and given parallel implementations, the framework provides tools to add new PPIs or freely modify the existing parallel code examples. We expect that through the API and the framework, users can easily implement one of the sequential applications with a new PPI or simply modify and customize the available examples of parallel implementations given within the framework. All of this is done through the command-line shell, where users can edit, configure, compile, execute, add execution metrics, select the workload, and from where it will be read (disk, memory, or network), among several other features.

Figure 1 describe the big picture of our framework. Parallelize applications from the Stream Processing domain often requires a great learning curve to the programmers unfamiliar with this paradigm. Programmer have to identify different operators in the source code, which is not a simple task. Each operator region may be from a few lines of code up to thousands. Using our framework API, users only have to insert a function call for given stream operator. Figure 1 shows in a code snippet (in the left) how an application is implemented using the API. Between lines 5-7 are the n operators calls

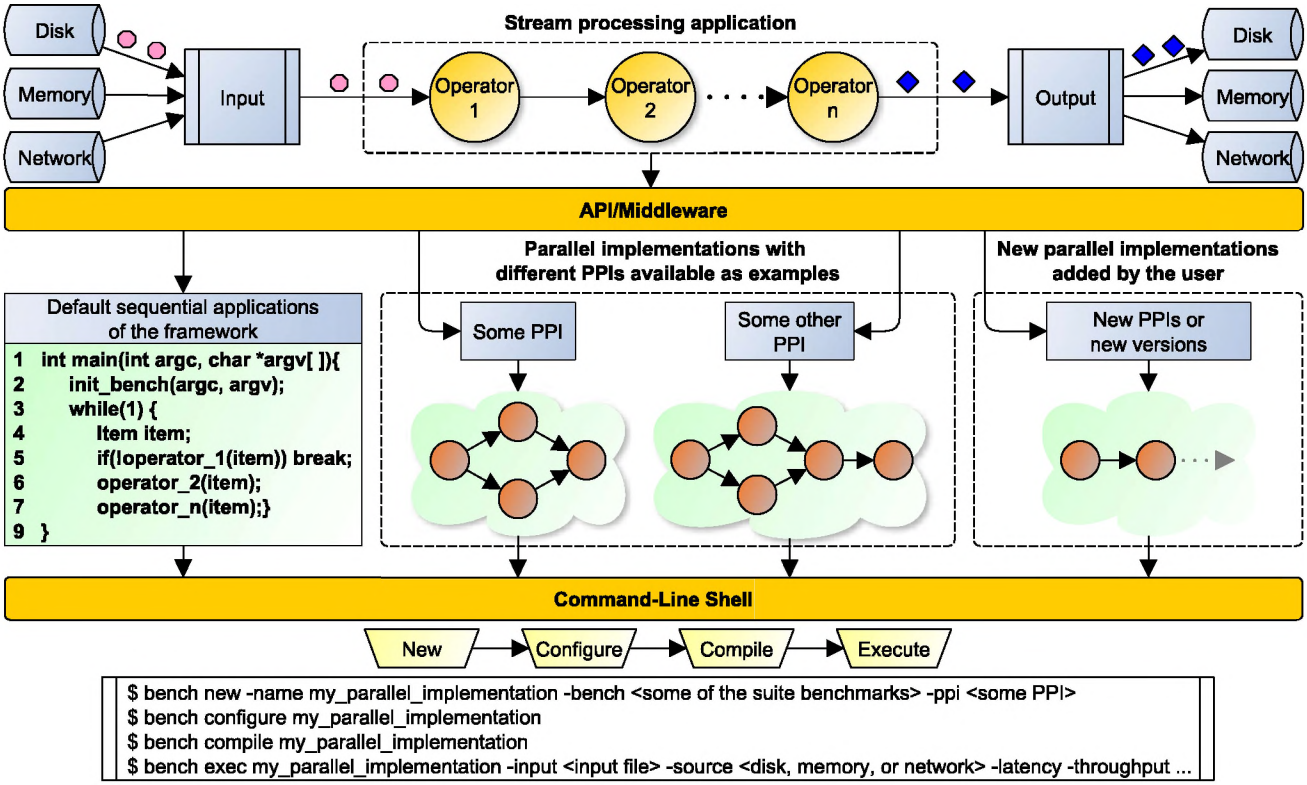

Fig. 1: Proposed framework for assessing PPIs.

(in the original application this may be hundreds of lines). The API also inserts execution metrics and manages the data source. This data source can be the disk, the memory (in-memory execution), or the network. For the last one, we plan to develop an independent system that generates data for the applications simulating a real system, with data arriving through the network with dynamic input rates.

In addition to the operators, the API also provides abstraction for the data that is communicated among them. Usually, these applications have smaller items that can change their data structure in different operators. Identifying all the variables that run through the operators as well as their dependencies, is a complex and time-consuming task. For this reason, our API encapsulates this data in a way that each operator uses the same data structure. In the API this generic class is called "Item" and this structure is the only thing that is communicated among all the operators of an application. This `Item` is created in line 4 of the code snippet in Figure 1. It is passed as a parameter to the operators. All programmers have to do to parallelize one of these applications is to structure the parallelism of a given PPI around these operators, considering the loop (lines 3-7). Thus, the API abstracts all main aspects regarding the application, where programmers can exclusively focus on parallelism aspects.

The central part of Figure 1 shows how the applications set is organized within the framework. Besides the default applications and the parallel examples, users can also use the command-line shell to re-code, or add new versions of the parallel implementations, or add implementations with other PPIs. At the bottom of the Figure 1, there is an example of how the shell can be used to add a new implementation. The "new" command creates a copy of the sequential application for users to edit. The "configure" opens a JSON configuration file to insert the PPI dependencies. Then "compile" and "exec" compile and execute the new program (to execute the default applications these two commands are enough). In the "exec" command, users can select the workload, the data source, the execution metrics, among other options. Among the metrics, users can select execution time, latency, throughput (itens/second), and CPU and memory usage for the whole application or individual operators. These metrics were chosen because they were considered the most relevant for SP by related work [19], [2].

IV. Methodology

In this section we discuss the methodology that was used for the experiments in the next sections. All experiments were performed in a computer that has 32 GB of RAM and two Intel(R) Xeon(R) CPU E5-2620 v3 2.40GHz (total of 12 physical cores and 24 threads with Hyper-Threading) processors. The operating system was Ubuntu Server 18.04, 64 bits, kernel 4.15.0-88-generic, and GCC 7.5.0 using `-O3` flag. Other libraries used were OpenCV version 2.4.13.6, TBB 2017 (INTERFACE_VERSION 9107), and FastFlow (revision 2.2.0-45).

To monitor the applications, we used the routines of the API itself. These routines allow us to monitor runtime with microsecond precision and get CPU/memory usage information obtained from the `/proc/[pid]/stat` pseudo-file. Although each new item can be monitored, we choose to monitor every 250 ms to avoid interfering with the results. For parallel executions, we used from 1 to 24 replicas. 0 replicas indicates the result of sequential applications (do not confuse replicas with threads, as the amount of threads varies at each PPI). Each result represents the average of 10 executions, with standard deviation properly included in the graphs with error-bars. For the Bzip2 application, we used an ISO Image of 702 MB as an input file. For Lane Detection and Person Recognition, we used 640 × 360 resolution MPEG-4 videos with 1858 and 450 frames, respectively. The validation of the results was done through the `md5sum` tool, comparing our solution with the ones given by the original applications.

V. Performance Characterization

The applications set of the framework is initially composed of three real-world applications: Bzip2, Lane Detection, and Person Recognition. We chose these applications because they have been explored and used as benchmarks in prior work [5], [6]. We plan to include more applications in the future.

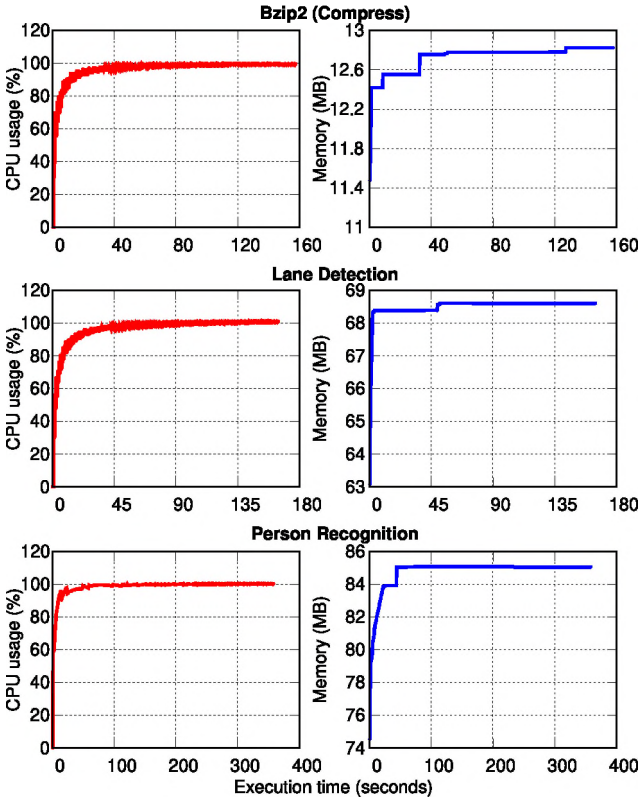

Fig. 2: Characterization results for the sequential applications.

Bzip2 [22] is a free and open-source data compression application. This application can be divided into a three-stage

pipeline (read, compress/decompress, write). In this work, we present the results only for the compression operation, for the sake of space. **Lane Detection** application is the task of detecting lanes of a road from a camera device. It captures each frame of a input video file and applies three computer vision algorithms. It can be divided into a nine-stage pipeline. Through these stages, the detected lanes are marked with straight lines in a new frame. This new frame with the marked lanes is then overlaid on the original, and the resulting frame is written to the output file.

The **Person Recognition** application matches human faces from a video frame against a database of faces. For each frame it applies a detection algorithm to detect all the faces in it. Then, it uses a set of face images and compares each of the detected faces in the frame with the faces on that set. The recognized faces are marked with a circle, and then the frames are written to the output file. Therefore, this application can be divided into a four-stage pipeline.

To show the behavior of these applications and how they differ from each other, in Figure 2 we present the characterization results. We monitor CPU and memory usage, Analyzing CPU usage, we can see that all three applications use 100% CPU almost all the time. It shows great potential to achieve performance through parallelism. Regarding memory, they present distinct behavior, Bzip2 is the application that uses less, and Person Recognition presents a lower curve until reaching its maximum level.

VI. Framework Use Case

To evaluate our proposal, we parallelized the applications using Intel TBB, FastFlow, and SPar, and evaluate their performance in terms of latency and throughput. The parallelism strategy was based on prior works [5], [6], where there the Farm-like pattern is implemented using PPIs.

The experiments were conducted as described in Section IV. We chose in-memory executions as it is a new feature and previous studies have performed similar experiments with the default mode (reading directly from disk) [5], [6]. The three chosen PPIs are widely used for stream processing in C++. TBB offers few customization options and it is by default very well optimized for these specific scenarios that we will test. On the other hand, FastFlow provides more optimization options and it is up to the programmers to understand the characteristics of each application and workload in order to extract the maximum performance. In contrast, SPar provide a small set of annotation to express stream parallelism. Its compiler generates parallel code, calling FastFlow's pattern routines. For FastFLow and SPar, we used an on-demand + blocking [13] queue configuration, as recommended by [6]. We also tested both with the custom FastFlow's thread mapping (physical cores first) and without it (no mapping).

For all cases, the scalability is reduced above 12 replicas. From this point, the processor needs to start allocating threads in both physical and virtual cores. This is an expected behavior of these applications that have high CPU usage. Regarding latency, lower is better. The difference between TBB and FastFlow/SPar can be explained by the characteristic of the communication queues. TBB has a work stealing scheduler. This means that a thread can take over any operator along the pipeline. In FastFlow the threads always run the same operator, Therefore, the one that runs the `read_op()` operator is done faster. This adds an extra delay when calculating from the moment a item is read to the moment it is finally processed by the last stage, increasing the latency.

Figure 3 (higher is better) has plotted the throughput. All PPIs and test cases presented good scalability. The higher latency of FastFlow/SPar did not negatively impact the throughput and these PPIs showed even better performance than TBB in Lane Detection with 12 replicas, for instance. The exception occurs in Bzip2 and Person Recognition above 12 replicas, where FastFlow/SPar using the default mapping showed a slight drop in throughput. The throughput obtained by FastFlow and SPar was equivalent to the TBB, which is by default highly optimized for this type of application. The applications were easily parallelized using the framework and we were able to achieve results similar to those in the literature [9]. This first version of the framework allowed fast development and easy reconfiguration of parallel applications for stream processing.

VII. Conclusion

In this paper, we introduced a Stream Processing framework for assessing Parallel Programming Interfaces (PPIs). The goal was to create a way to facilitate the implementation of stream parallelism on the Stream Processing domain. We showed the main features of the proposed framework by using it to evaluate TBB, FastFlow, and SPar. The results were similar to the literature and the API worked perfectly with the three evaluated PPIs. This way, our work has reached its initial goal. In the next steps, we will include more applications to the framework and parallelize them with more PPIs, add new workload classes, add the possibility to change the receiving data input rate and also add the feature that allows reading data from the network.

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 project ParaS (Nº 19/2551-0001895-9), and Universal MCTIC/CNPq Nº 28/2018 project SParCloud (No. 437693/2018-0).

References

[1] H. C. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press, 2014.

[2] M. V. Bordin, D. Griebler, G. Mencagli, C. F. R. Geyer, and L. G. Fernandes, "DSPBench: a Suite of Benchmark Applications for Distributed Data Stream Processing Systems," *IEEE Access*, vol. na, no. na, p. na, December 2020.

[3] C. M. Stein, D. A. Rockenbach, D. Griebler, M. Torquati, G. Mencagli, M. Danelutto, and L. G. Fernandes, "Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units," *Concurrency and Computation: Practice and Experience*, vol. na, no. na, p. e5786, May 2020.

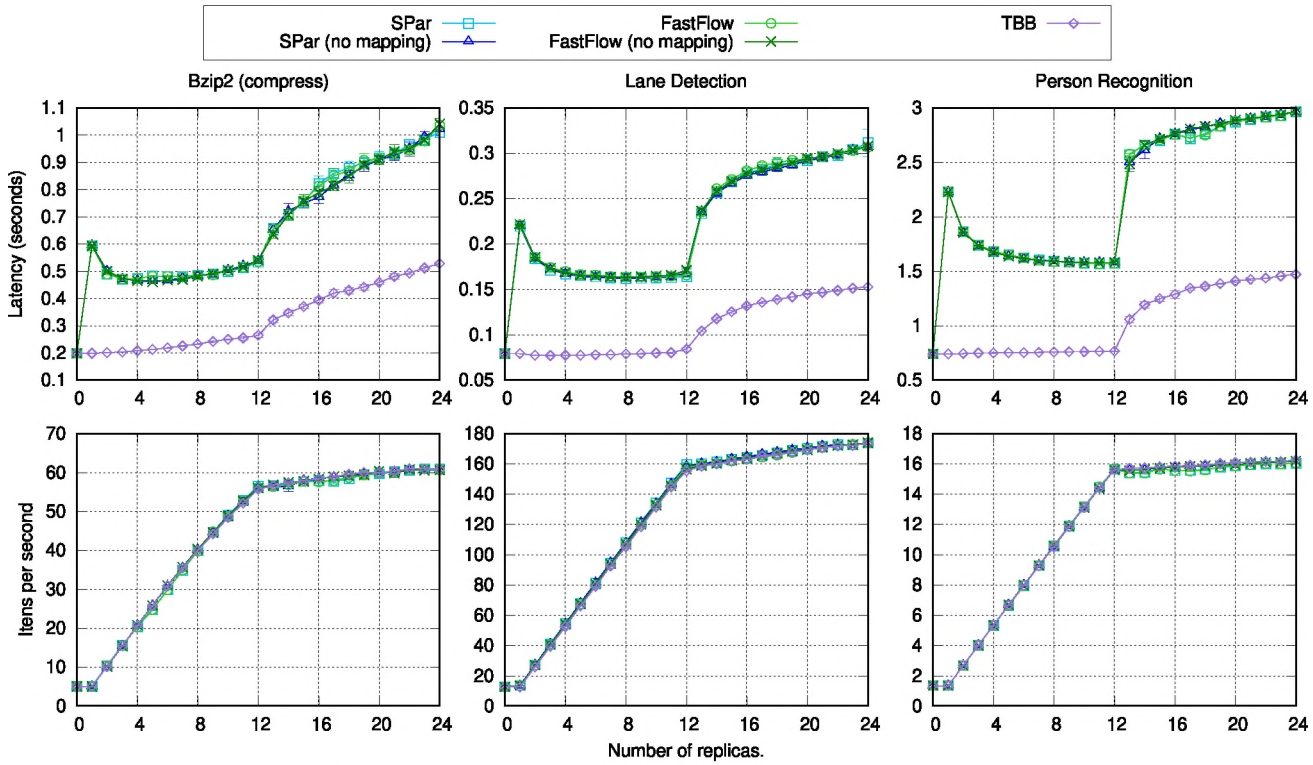

Fig. 3: Performance results using three PPIs and up to 24 replicas.

[4] G. Mencagli, M. Torquati, D. Griebler, M. Danelutto, and L. G. L. Fernandes, "Raising the Parallel Abstraction Level for Streaming Analytics Applications," *IEEE Access*, vol. 7, pp. 131 944 – 131 961, 2019.

[5] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes, "High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2," *International Journal of Parallel Programming*, vol. 47, no. 1, pp. 253–271, February 2018.

[6] ——, "Higher-Level Parallelism Abstractions for Video Applications with SPar," in *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing*, ser. ParCo'17. Bologna, Italy: IOS Press, September 2017, pp. 698–707.

[7] D. A. Rockenbach, C. M. Stein, D. Griebler, G. Mencagli, M. Torquati, M. Danelutto, and L. G. Fernandes, "Stream Processing on Multi-cores with GPUs: Parallel Programming Models' Challenges," in *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, ser. IPDPSW'19. Rio de Janeiro, Brazil: IEEE, May 2019, pp. 834–841.

[8] 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*, ser. Lecture Notes in Computer Science, vol. 11997. Göttingen, Germany: Springer, August 2019, p. 12.

[9] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes, "Stream Parallelism with Ordered Data Constraints on Multi-Core Systems," *Journal of Supercomputing*, vol. 75, no. 8, pp. 4042–4061, July 2018.

[10] R. B. Hoffmann, D. Griebler, M. Danelutto, and L. G. Fernandes, "Stream Parallelism Annotations for Multi-Core Frameworks," in *XXIV Brazilian Symposium on Programming Languages (SBLP)*, ser. SBLP'20. Natal, Brazil: ACM, October 2020, pp. 48–55.

[11] D. A. Rockenbach, D. Griebler, M. Danelutto, and L. G. Fernandes, "High-Level Stream Parallelism Abstractions with SPar Targeting GPUs," in *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing (ParCo)*, ser. ParCo'19, vol. 36. Prague, Czech Republic: IOS Press, 2019, pp. 543–552.

[12] M. Voss, R. Asenjo, and J. Reinders, *Pro TBB: C++ parallel programming with threading building blocks*. Apress, 2019.

[13] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2017.

[14] 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.

[15] A. Pagliari, F. Huet, and G. Urvoy-Keller, "Namb: A quick and flexible stream processing application prototype generator," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 61–70.

[16] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4257, 2017.

[17] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *7th International Conference on Utility and Cloud Computing*, 2014, pp. 69–78.

[18] Y. Wang, "Stream processing systems benchmark: Streambench," Master's thesis, Aalto University, 2016. [Online]. Available: http://urn.fi/URN:NBN:fi:aalto-201606172599

[19] D. Agrawal, A. Butt, K. Doshi, J.-L. Larriba-Pey, M. Li, F. R. Reiss, F. Raab, B. Schiefer, T. Suzumura, and Y. Xia, "Sparkbench – a spark performance testing suite," in *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things*, R. Nambiar and M. Poess, Eds. Cham: Springer International, 2016, pp. 26–44.

[20] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 365–376.

[21] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[22] Seward, Julian, "A Program and Library for Data Compression," 2017. [Online]. Available: http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html