2022 30th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) | 978-1-6654-6958-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/PDP55904.2022.00011

Evaluating Micro-batch and Data Frequency for Stream Processing Applications on Multi-cores

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

***Abstract*—In stream processing, data arrives constantly and is often unpredictable. It can show large fluctuations in arrival frequency, size, complexity, and other factors. These fluctuations can strongly impact application latency and throughput, which are critical factors in this domain. Therefore, there is a significant amount of research on self-adaptive techniques involving elasticity or micro-batching as a way to mitigate this impact. However, there is a lack of benchmarks and tools for helping researchers to investigate micro-batching and data stream frequency implications. In this paper, we extend a benchmarking framework to support dynamic micro-batching and data stream frequency management. We used it to create custom benchmarks and compare latency and throughput aspects from two different parallel libraries. We validate our solution through an extensive analysis of the impact of micro-batching and data stream frequency on stream processing applications using Intel TBB and FastFlow, which are two libraries that leverage stream parallelism on multi-core architectures. Our results demonstrated up to 33% throughput gain over latency using micro-batches. Additionally, while TBB ensures lower latency, FastFlow ensures higher throughput in the parallel applications for different data stream frequency configurations.**

I. Introduction

Most stream processing applications need to handle data that arrives with varying intensity. This intensity can be defined as data arrival frequency, data size (including batching), data complexity, and other aspects [1], [2]. For example, in network monitoring applications, the data stream frequency is higher at peak times. Regarding data complexity, a person recognition application might be more computationally intensive in more crowded frames. In addition, batches based on time slots may have varied sizes, which means micro-batching stream applications must handle this minimizing performance losses.

In stream processing, ideally data should be processed as it arrives, in near real-time. Therefore, unexpected spikes, bursty phases, and other abrupt changes in the input streams can cause undesirable effects that impact negatively on throughput, latency, or even lead to a system failure and/or data loss [3]. Due to this intrinsic nature of stream processing, a significant research and development effort is put into mitigating the impact of these fluctuations in input streams and increasing fault tolerance. The characteristics of these streams, such as data frequency and micro-batch size, can be regulated for improving or achieving optimal performance levels. Thus, scientists are constantly developing solutions for applications in this domain, both for the design of new applications and adaptivity [2], [4]–[6]. The use of micro-batch can amortize undesirable effects. However, the batch size significantly affects the performance in stream processing applications [1].

Varying the data stream frequency is important for testing the adaptability of stream processing systems. For example, scaling out to see if a system can avoid backpressure or simulating frequency variations to analyze if the system can sustain a target throughput or latency. To the best of our knowledge, no one has compared Threading Building Blocks [7] with FastFlow [8] in stream processing applications with regards to throughput and latency. Other works from literature assessed these tools mainly using fixed data frequency and rarely investigate the impact of micro-batching. Analyzing data frequency in stream processing with multiple applications is a complex and challenging task. However, we have not found support tools in the literature that allow users to create custom stream processing benchmarks that natively support latency and throughput analysis with dynamic micro-batch sizing and data frequency. Related work is very domain-specific and not easily parameterized.

In the past [9], [10], we proposed a framework for creating benchmarks of stream processing applications that makes it easier to assess multiple parallel programming interfaces (PPIs). In those works, we evaluated the performance of different real-world stream applications under multiple PPIs in terms of latency, throughput, and resource utilization. In contrast, in this paper, we added new features to the framework, such as dynamic micro-batch size and data stream frequency, to enrich the performance analysis of PPIs and other tools that aim at leveraging parallelism. SPBench is free software and is publicly available[1].

This work aims to investigate how stream processing applications and PPIs behave under different data intensity, degrees of parallelism, and workloads. We studied Intel TBB and FastFlow libraries, as they support stream parallelism and few analyses are available under these parameters. Therefore, we are contributing for: a framework that simplifies the benchmarking

This work 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), FAPERGS 10/2020-ARD project SPAR4.0 (Nº 21/2551-0000725-7), Universal MCTIC/CNPq Nº 28/2018 project SPARCLOUD (Nº 437693/2018-0).

[1] https://github.com/GMAP/SPBench

978-1-6654-6958-6/22/$31.00 ©2022 IEEE
DOI 10.1109/PDP55904.2022.00011

of micro-batch sizing and data stream frequency on stream processing applications; and an analysis of the performance impact on micro-batch sizing and data frequency, varying applications and PPIs.

The structure of the remainder of this paper is as follows. Section II gives a background and discusses related work. Our framework is briefly introduced in Section III. Section IV describes the methodology used in the experiments, and the results are presented in Section V. Finally, in Section VI, we draw our conclusions and future work.

II. Background

A. Related Work

In previous work we have discussed in detail benchmarks and frameworks related to SPBench [10]. Here, we discuss related work that investigated micro-batching and data stream frequency in stream processing.

Das et al. [1] propose a self-adaptive algorithm to reduce the latency of distributed batched streaming systems through dynamic batching resizing. They used Apache Spark and tested the algorithm by varying the input data rate with waveform and binary (sudden low-high frequency changes) strategies. Zhang et al. [4] targeted the same problem with a different approach, but they also tested their algorithm using a binary strategy for data stream frequency. Stein et al. [2] also have the same goal, but they target compression algorithms and graphics processing units (GPUs). Here the authors tested the algorithm with four workloads presenting different complexity patterns across the dataset to vary the data intensity. Abdelhamid et al. [5] introduce an algorithm for self-adaptive parallelism for micro-batch stream processing and test it with several data stream frequency strategies.

In [11] the authors propose a reconfiguration algorithm for power-aware parallel applications. They implemented the algorithm in the PARSEC [12] suite using FastFlow and tested it changing the size of the items to simulate drops and rises in data intensity. In [6] the authors evaluated scalability of benchmarks implemented with Apache Flink and Apache Kafka Streams by varying the data stream frequency. The strategy they used to increase data frequency was to increase the number of data sources rather than increase the rate of item generation.

RIoTBench [13] is a benchmark suite for IoT stream processing. They evaluated Storm's performance under real workloads that naturally exhibit increasing, decreasing, wave, and binary data stream frequency strategies. [14] evaluated the performance of micro-batching distributed stream processing systems (DSPSs) using two data intensity strategies. Wang et al. [15] presented a Storm-based framework for auto-elasticity and tested it using data size and complexity as a way to tune the data intensity. In [16] is proposed a framework for generating data to evaluate different engines for Linked Stream Data (LSD). This framework allows different parameters to be adjusted, such as data size, the number of sources, and data stream frequency. Karimov et al. [17] propose a benchmarking framework that generates data at a configurable rate and acts as a distributed in-memory data generator for evaluating DSPSs.

NAMB [18] is a framework that automatically generates micro-benchmarks to evaluate DSPSs. It includes a Kafka synthetic workload generator that can be configured to generate data streams at different frequencies. They evaluated the micro-benchmarks using a binary strategy for data stream frequency. Balkesen et al. [19] proposed a framework for adaptive input admission and data management in distributed stream processing. The authors used the distributed engine Borealis and modeled synthetic and real GPS data to meet several specific data stream frequency strategies to test the framework. In [20], some strategies are presented for proactive elasticity and energy awareness in data stream processing. This work target multi-core architectures and use FastFlow [8] to perform the experiments. In addition to the original workload, it implements a strategy where the data intensity increases or decreases in small random steps. Navarro et al. [21] evaluated pipeline parallelism and compared PThreads and TBB using different scheduling policies, optimizations, and parallel compositions.

Although many of these works evaluated different PPIs with micro-batch or data stream frequency, most aim for distributed engines [16]–[19]. Other works have evaluated PPIs that support multi-core architectures, but either the focus was on stream processing on GPUs [2] or they did not compare different PPIs [11], [20]. Regarding performance evaluation, they are commonly either latency-, or throughput-aware [1], [2], [4], [5], [16]. Benchmarks that allow exploring micro-batching transfer this responsibility to DSPSs (Spark usually) [1], [13], [17]. As can be observed, none of these works analyze and compare the impact of micro-batching and data stream frequency on the performance of different PPI for stream processing on multi-cores. To the best of our knowledge, there are no similar approaches for benchmarking stream processing with micro-batch and data frequency configuration.

B. Micro-batch stream processing

Micro-batch (or mini-batch) processing is a variant of traditional batch processing. In micro-batching systems, data is processed in small groups at a higher frequency. In stream processing, micro-batching can be used as an optimization technique that trades throughput for latency [3]. The size of a micro-batch can be defined according to predefined criteria, such as time intervals (e.g., each 2-second interval forms a new micro-batch), data size (e.g., micro-batches with 5 MB of data), number of items, and others [1]. In general, the optimal size is the one that achieves the desired trade-off between throughput and latency [2]. However, this is not a static value since fluctuations in data stream frequency and processing cost of each item are very common in stream processing.

Besides the disadvantage of increased latency, there are many advantages for using micro-batches in stream processing. Enabling batching support in an application implies in adding loops. Therefore, the compiler may optimize these loops with unrolling techniques using software pipelining. It also enables vectorization. Besides, micro-batching may improve throughput by amortizing operator-firing and communication costs. Such amortizable costs can include deeply nested calls, warm-up

costs (e.g., for the instruction cache), and scheduling costs, possibly involving a context switch [3]. For stream processing with GPUs, it requires batching input elements for efficient resource utilization [2]. Micro-batching can ensure system stability and lower latency for a wide range of auto-adaptive algorithms workloads despite significant variations in data rates and operating conditions [1]. Such algorithms can achieve performance levels without demanding extra resources or leading to data losses.

The primary control variable in micro-batching is the batch size. It can be controlled either statically or dynamically [3]. StreamIt [22], for instance, has a static batching algorithm that aims to trade the data-cache cost of requiring larger buffers for the benefits of using instruction-cache when processing micro-batches. However, systems with statically set micro-batches may exhibit high latency under lower loads or may not cope with sudden increases in data frequency or item processing cost [1]. On the other hand, dynamic batching is commonly used with self-adaptive methods for reacting to load changes and maintaining system stability. There are many works focused on developing algorithms that exploit dynamic batching to improve performance or resource utilization [1], [2], [4], [5], [23], [24]. These works require the researcher to allocate extra time to implement benchmarking support, diverting from the research scope. Therefore, we argue that there is a demand for tools like our framework, which can be very useful for researchers in this area.

C. Data Frequency

Data frequency can have different meanings in stream processing. It can mean the frequency with which the application receives items of the same type. It is also be used under several synonyms in the literature, such as data (or arrival, or item, or stream) rate or frequency, stream pressure, input frequency, and so on. Here, we define data stream frequency as the number of items generated by sources per unit of time. In practice, in this work, we add a time delay for each item in the source. Decreasing the time delay entails in increasing the frequency and vice versa.

It is pretty standard for data not to arrive at constant speeds throughout the execution of a stream processing application. Fluctuations can occur due to workload characteristics, transient network issues, garbage collection in JVM-based engines, etc. [17]. Examples of loads that can present huge and often predicted fluctuations are data from network monitoring, traffic control, GPS, and others. This kind of fluctuation is usually linked to the times people use these services the most throughout the day and can be drawn as a waveform. However, many works in the literature need to do tests with abrupt fluctuations and shorter periods [5], [13], [17]–[19].

In some cases, data stream frequency (and also micro-batching) merges with the concept of data intensity or data complexity [25]. In these cases, the workload behavior is given by the size or computational cost of the items. In [11], for example, the authors use an image processing application and cut by half the resolution of the input images at specific points. They used it to simulate a binary strategy. [2] used real and custom workloads for data compression with stretches of higher and lower processing costs. In this work, we do not artificially modify the workloads in this way because they naturally exhibit patterns of intensities compatible with the wave, binary, and other strategies. Table I presents the data stream frequency (or data intensity) strategies that have been used at least twice in related work (Section II-A). It shows that *increasing*, *wave*, and *binary* are the most commonly used strategies. Through our framework, users can create custom benchmarks, run tests with these strategies, and even create other custom strategies.

Data frequency strategy	Related work
Increasing	[6], [13], [16]
Wave	[1], [5], [13], [19]
Binary	[1], [4], [5], [13], [14], [17]–[19]

TABLE I: Data stream frequency strategies found in R.W.

III. SPBench

SPBench was first introduced on [9], and the first release was fully described and evaluated on [10]. It is open-source and available under the GPLv3 license. The goal of SPBench is to enable users to easily create custom benchmarks from real-world stream processing applications and evaluate multiple PPIs. It provides a C++ API that allows users to access simplified versions of the applications in our suite. Based on the sequential versions, users can implement parallelism, create new benchmarks with different parallelism strategies, or even explore new PPIs. Users can also configure and modify parameters, such as build dependencies and metrics, through a command-line interface (CLI). It is fully modular and parallel code can be quickly replicated across all SPBench applications.

One of the aspects that most differentiates our framework from related work is how users interact with it. Specific low-level details of each application are abstracted and the API presents the user with the core of each application in a few lines of code. Such applications can originally be much longer than a thousand lines. Users can access through the CLI a database containing all applications/benchmarks previously added to our suite (e.g., the ones we used in past and in this work) and their new customized versions. Other secondary parameters can be tuned via the CLI with simple commands. This allows users to entirely focus on writing and tuning the parallelism rather than spending time with the non-relevant low-level aspects of each application. It is helpful for researchers to test their new solutions and technologies for stream parallelism, and also for learning/teaching purposes.

Figure 1 shows the representation of the SPBench framework. The first released version allows users to select receiving data from a disk or memory (we plan adding a network option in the near future and multiple sources support). Users can evaluate latency, throughput, and resource usage at different granularities and depth levels. Users can also choose from multiple workload classes for stressing different characteristics

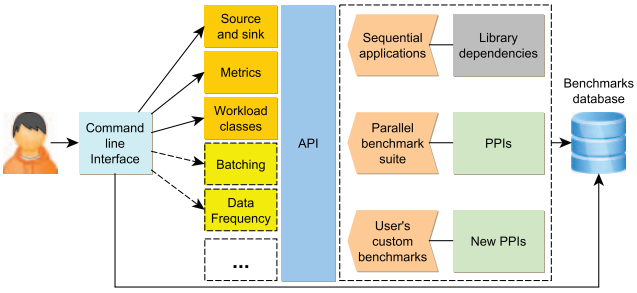


Fig. 1: SPBench framework (new contributions in yellow [10]).

in the application and PPI. In this work, we added the options of dynamic micro-batch sizing and data stream frequency control. These new features are already fully integrated and functional. We marked our new contributions to the framework as light yellow dashed blocks in the Figure 1.

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 Xeon E5-2620 v3 processors (total of 12 cores and 24 threads). 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 2.4.13.6, Intel TBB 2020 Update 2 (TBB_INTERFACE_VERSION 11102), and FastFlow version 3. Implementations with FastFlow used queue size 1. For TBB we defined $ntokens = nthreads \times 10$ [21].

We built benchmarks using four applications supported by our framework [10]: Bzip2, Lane Detection, Person Recognition, and Ferret (from PARSEC) [12]. To monitor the applications, we used the routines of the API itself, which allows for performance monitoring with microsecond precision. We choose a 250 ms monitoring interval to avoid interfering with the results [20]. Furthermore, for each application, we used the same load, and parallelism strategy described in [10]. Although uur framework allows for dynamic batch sizing, we focused on evaluating micro-batching under multiple parallelism degrees. Thus, we used static micro-batch sizes for space reasons.

For data frequency we used four strategies with different behaviors, which include the main strategies found in the literature: *increasing*, *decreasing*, *wave*, and *binary*. The frequency is controlled by inserting a time delay for each item in the source operator. Our framework allows users to control this delay with microsecond precision, and data can be loaded directly from the main memory (in-memory execution) to reach the maximum frequency. Although the minimum item delay (i.e., maximum frequency) can be set to zero, the memory access delay must be regarded.

In our data frequency strategies, we vary the time delay between 0 and 300 milliseconds. In [9] and [10] it can be seen that latency of 300 ms is a value that closely approximates what was achieved in executions with 24 replicas in most test cases. Therefore, we modeled our strategies based on these bounds. We divided each workload into twenty steps. At each step, the frequency increases or decreases, according to the chosen strategy. Ferret with `native` load [12], for example, processes 225 items by step, against 22 items in Person Recognition.

In the *increasing* strategy, the item delay is decreased from 300 to 0 ms, which implies increasing the data frequency from minimum to maximum through small 15 ms steps. In the *decreasing* strategy, the opposite occurs. In *wave* each wave has a four-step amplitude, which means a 75 ms item-delay jump by step (up or down). In *binary* the frequency is changed abruptly between minimum and maximum twenty times.

V. Experiments

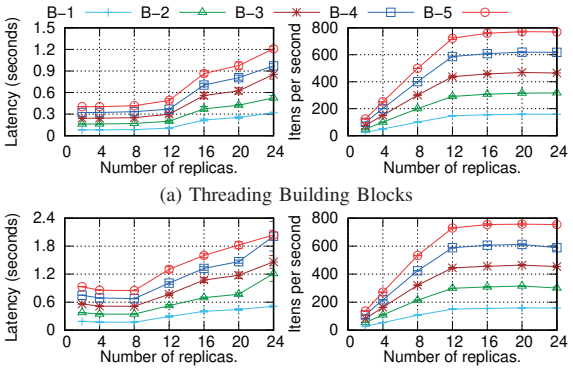
This section presents the experiments for micro-batching and then for data stream frequency. Although the benchmark applications support other parallelism patterns, and they are available to users in our framework, we evaluate only the Farm pattern for comparison and to simplify the analysis and discussion. It consists of three-stage pipeline, where there is a producer stage (source), n worker stages (here all intermediate stages of the pipeline are collapsed into a single one and then replicated for data parallelism), and a final consumer stage (sink), which aggregates the final result [10].

A. Micro-batching Results

We run the experiments with micro-batches using multiple degrees of parallelism, from 2 to 24 maximum parallel workers in the farm. The actual number of threads created by each PPI varies according to its design. However, we will focus the discussion on the results with 12 and 24 replicas, which are the number of physical cores and the total number of threads for this system. Besides replicas, we also vary the micro-batch size from 1 (no batch) up to 5 items per batch (lines in the charts). We have experimented with a more extensive range, but we believe that above 5 items it is already out of the micro-batching concept, and the latency becomes significantly higher in some cases.

The graphs with micro-batching results are in Figures 2-5. Each figure contains the results for one of the applications. The graphs show latency and throughput (items per second) for Intel TBB and FastFlow. Figure 2 shows the experimental results for the Lane Detection application. In this application, TBB with 12 replicas increased latency by 363% when using five-item micro-batch (Figure 2a). On the other hand, throughput increased by 391%. The throughput gain was 7.7% higher than the latency gain. With FastFlow (Figure 2b) this increase was 10.91%, an even higher difference. These differences are more significant at the 24th replica. TBB had a 33.79% increase in throughput over latency, while FastFlow showed a 26.5% increase. Although FastFlow with 12 replicas got a higher advantage of increasing the micro-batch size, the latencies achieved are still more than double the results of TBB. Regarding throughput, the performance of both PPIs is similar.

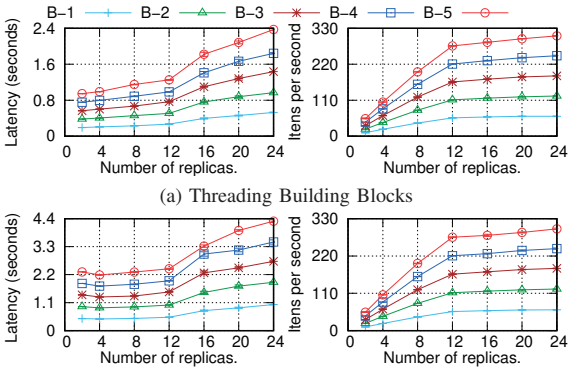
Experimental results with Bzip2 also achieved a higher throughput increase over latency. With 24 replicas this increase



(a) Threading Building Blocks

(b) FastFlow

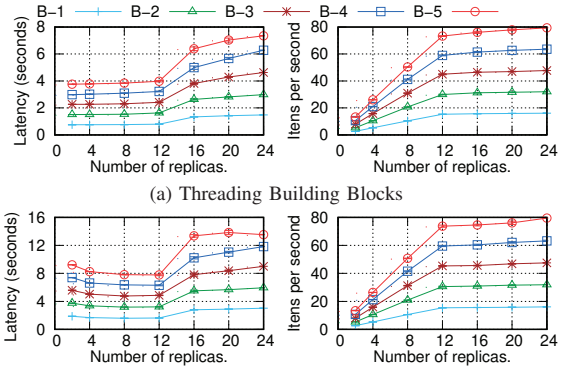
Fig. 2: Latency and throughput results for Lane Detection with multiple parallelism degrees and micro-batch sizes.



(a) Threading Building Blocks

(b) FastFlow

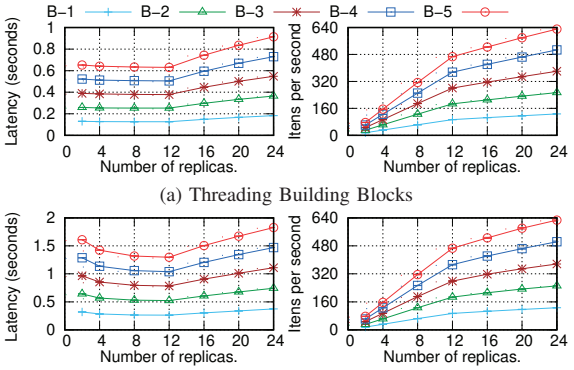
Fig. 3: Latency and throughput results for Bzip2 with multiple parallelism degrees and micro-batch sizes.



(a) Threading Building Blocks

(b) FastFlow

Fig. 4: Latency and throughput results for Person Recognition with multiple parallelism degrees and micro-batch sizes.



(a) Threading Building Blocks

(b) FastFlow

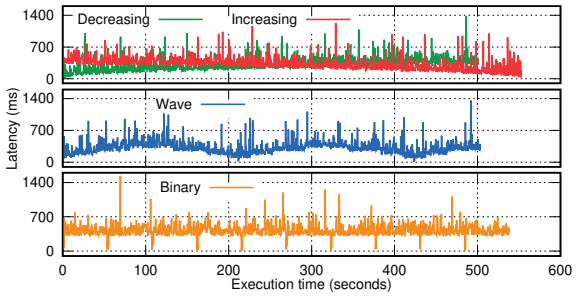
Fig. 5: Latency and throughput results for Ferret with multiple parallelism degrees and batch micro-sizes.

in TBB was 13.4% (Figure 3a). Here, FastFlow performed better, showing a 21.4% throughput gain with 24 replicas (Figure 3b). Similar to Lane Detection, Bzip2 achieved positive results using micro-batching, considering throughput and latency as inversely proportional metrics (which may not be true for all scenarios).

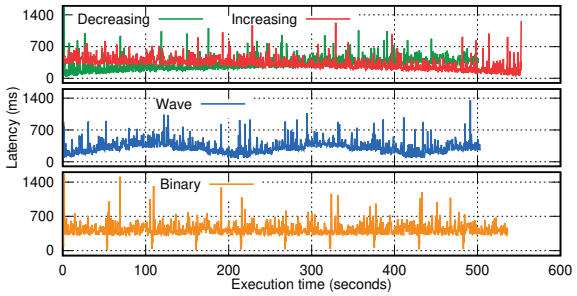
The Person Recognition in Figure 4 showed different behavior. In the scenario with 12 replicas, TBB had a 6.3% higher latency increase over throughput, going opposite to Lane Detection and Bzip2. FastFlow increased both latency and throughput by 380%, a balanced result. With 24 replicas, the throughput gain occurs again, but this time, FastFlow achieved a 13.5% increase, while TBB showed no difference. Finally, Figure 5 shows the results for the Ferret application. This application did not show any difference above 1% between the two PPIs when comparing the increase between latency and throughput. In all the scenarios evaluated, these two metrics increased by approximately 400% for a five-item micro-batch.

Many factors may explain the different behavior of micro-batching in these applications, such as the characteristics of the applications, PPIs, and the loads used. For example, Ferret processes small items and does not need sorting in the last stage. So batching did not play a significant role here. Person Recognition, on the other hand, requires the items to arrive in a specific ordering. However, the increase in replicas causes more item to clutter because more concurrent threads are writing to the output queue. The time required to sort the items can impact latency. However, Person Recognition has a naturally high latency because items are more computationally intensive. Therefore, the use of micro-batch does not alleviate the cost of sorting items. On the other hand, FastFlow managed to achieve throughput gains with 24 replicas.

Regarding Bzip2 and Lane Detection, both process items at a higher speed than Person Recognition and bigger items than Ferret. So there is room for performance improvement when using a micro-batching strategy because this reduces the number of items, implying less clutter and reduced communicating costs. Considering the results achieved by both applications, we believe that using micro-batching is a useful optimization. The latency increases at a slower pace than the throughput up to five-item batch. However, this positive balance may be questionable in different applications. For example, in a realistic scenario, it may not be interesting to increase the latency of a lane detection application, as this data is may be required by



(a) Threading Building Blocks



(b) FastFlow

Fig. 6: Parallel Ferret under different data stream frequency strategies.

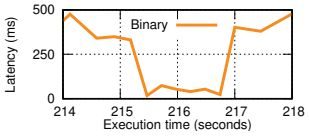


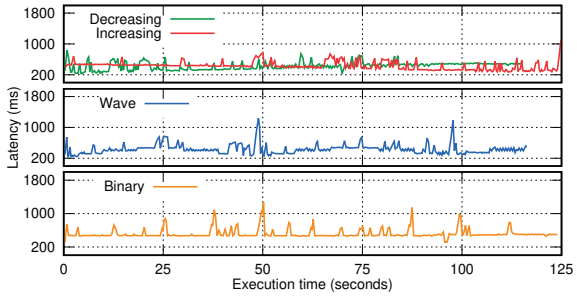
Fig. 7: Snapshot of a frequency switching cycle from Ferret using the binary pattern.

fast decision-making systems such as a self-driving car. Data compression applications, on the other hand, usually do not have this requirement. Also, although load imbalance can still occur, it may be negligible when using the Farm pattern [21].

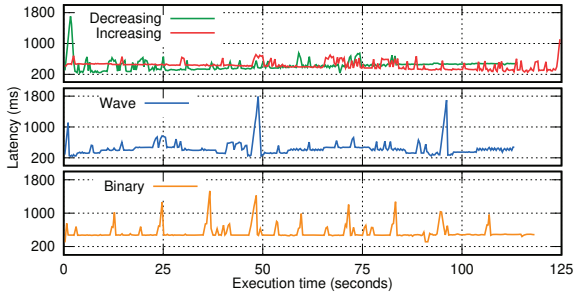
B. Data Frequency Results

This section presents the data stream frequency experiments. We use four strategies widely used in related work: *decreasing*, *increasing*, *wave*, and *binary*. We discuss these strategies in Sections II-C and IV. Figures 6, and 8-10 present the results for each application, with TBB on top and FastFlow on the bottom. Here, instead of considering the average latency, as was done in the micro-batching experiments, we perform latency monitoring throughout the execution (using 24 replicas). The X-axis on graphs has the execution time for each application.

The slower data access speed on the hard disk is a limiter to achieving higher frequencies. Therefore, the runs in the following experiments were done in-memory, as described in Section IV. This makes it possible to achieve high data frequencies, with a delay of a few microseconds between items in the source. Although we have executed these experiments with multiple degrees of parallelism, we present the results



(a) Threading Building Blocks



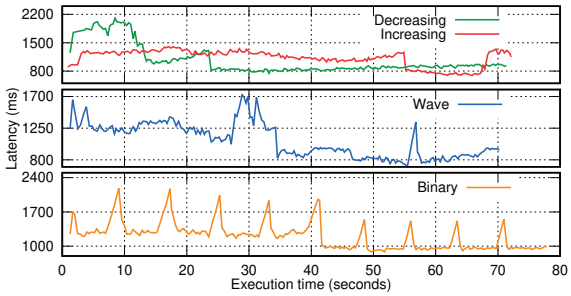
(b) FastFlow

Fig. 8: Parallel Bzip2 under different data stream frequency strategies.

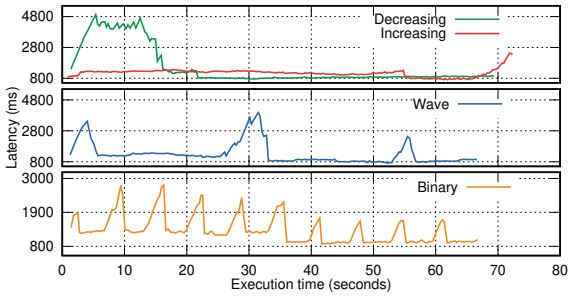
with 24 replicas, which is the number of cores available on the processor and best shows the impact of varying the frequency.

Compared to the other three applications, Ferret can achieve the lowest latencies [10]. Therefore, this application is bounded by input source [21] and was expected to handle the increase in stream frequency with the least impact on latency. Considering the *increasing* and *decreasing* strategies on top of Figures 6a and 6b, we can observe that the behavior of TBB and FastFlow was similar. Both PPIs showed an increase in latency spikes in the higher frequency periods. Although both strategies act inversely, the increase in execution time for the *increasing* strategy can be explained by the workload characteristic. This application has no overhead caused by item sorting. However, the load incurs in lower latency at the beginning and a higher latency at the end [10]. So this natural latency of the workload matches the *decreasing* strategy. On the other hand, with *increasing* the high-latency items negate Ferret's natural advantage in processing data at a high frequency.

Using the *wave* strategy, TBB and FastFlow also showed similar behavior. The exception occurs at maximum frequency points (e.g., at 210 seconds), while TBB generates a small latency spike down to almost zero, FastFlow presents a high spike. In the other applications, both TBB and FastFlow show a latency spike with the *wave* strategy. However, the FastFlow spikes reach higher latencies (notice that the latency is scaled differently between the PPIs in Figures 9 and 10). Regarding the *binary* strategy, FastFlow was able to achieve low latency peaks when the item delay tends to zero (high frequency), as was TBB. As we defined the high and low-frequency periods based on the number of items, it is difficult to observe this

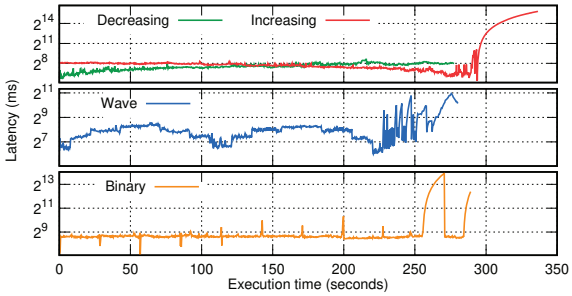


(a) Threading Building Blocks

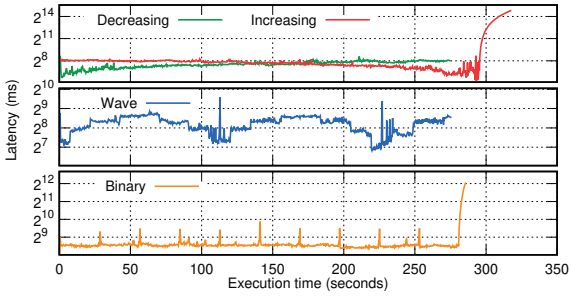


(b) FastFlow

Fig. 9: Parallel Person Recognition under different data stream frequency strategies.



(a) Threading Building Blocks



(b) FastFlow

Fig. 10: Parallel Lane Detection under different data stream frequency strategies.

low-high cycle here because the computing time of each item is very short in Ferret. To illustrate how this low-high cycle occurs, in Figure 7 we show a snapshot of one such spike for this application. This stretch where the latency tends to zero represents a high-frequency period and lasts less than two seconds. Ferret processed 225 items in this short interval, the same amount processed in the low-frequency period that lasts over 50 seconds.

Regarding Bzip2 (Figure 8), the average latency with 24 replicas in this workload is higher: above 1000 ms with FastFlow as was seen in Figure 3. Therefore, it is expected that this application will not gain much performance by increasing data stream frequency. On the contrary, sudden increases, as with the *wave* and *binary* strategy, cause an increase in latency because these parallel implementations could not consume data that fast. The graphs confirm this, by showing large latency spikes (higher with FastFlow) in high-frequency phases. Concerning the execution time overhead with the *increasing* strategy, we hypothesize that it might be caused by item clutter or load unbalance.

The Person Recognition results in Figure 9 show a different pattern from the previous ones. In the *wave* and *binary* strategies, it can be seen that approximately halfway through the execution there is an overall reduction in latency. In the *decreasing* strategy, this reduction in latency occurs at the very beginning of the execution, while with *increasing* this occurs at 55 seconds. This behavior is explained by the characteristic of the workload. In the second half of the input video, there are no more recognizable persons. Therefore the computational effort is lower in this part. As the computation time per item is longer here, the spikes in latency with *wave* and *binary* last longer and are more pronounced. If we compare the different y-scales between TBB and FastFlow, it can be seen that the increase in frequency had more impact on FastFlow also for this application.

For the last, Figure 10 shows the performance of the Lane Detection application. Similar to Person Recognition, this application processes video frames. The computation cost of each frame also fluctuates a lot. There are times in the input video where there is no lane to be detected, and at other times there are intersections or the car changes lanes, resulting in multiple lanes detected per frame. Although the average latency of the sequential application is lower in the latter part of the video, it has the highest latency spikes, as studied in [10]. This high spike coincides precisely with the parts where the latency jump occurs in the graphs in Figure 10. This spike is also observed in the *increasing* strategy, but not in *decreasing*, which shows that this problem is indeed linked to the high-frequency of the stream. Added to this is the issue of item ordering and possibly load unbalancing. This application processes items much faster than Person Recognition so that these factors may be more impactful. Comparing TBB to FastFlow (Figures 10a and 10b), TBB was able to keep latency lower in most high frequency moments, such as in *wave* and the first half of *binary*. FastFlow showed high latency spikes in all these high-frequency stretches. On the other hand, TBB was more impacted at the end of the computation than FastFlow. Nevertheless, since the workload naturally fluctuates throughout the execution, it is not possible to draw precise conclusions for these cases.

FastFlow showed in almost all test cases higher latency peaks

under high-frequency data and a higher resistance to reach low latencies, as can be seen in the high-frequency periods in Figures 10b and 6b. The inter-stage communication cost and scheduling items of FastFlow possibly impact this aspect a lot, as these factors did alleviate in TBB due to its work stealing policy. Nevertheless, even with higher latency, FastFlow achieved reduced execution time in most cases, indicating a higher throughput than TBB in these experiments. Therefore, we argue that even choosing the right PPI plays an essential role in finding the best trade-off between latency and throughput, or other metrics, in stream processing.

VI. Conclusions and Future Work

With the help of the extended framework in this paper, we analyzed the impact of micro-batch and data intensity on stream processing applications with different PPIs. We were also able to create several workloads with some strategies that could change dynamically at execution time. We tested micro-batching configurations under different levels of parallelism, which revealed throughput gains over latency up 33% with TBB and 26.5% with FastFlow. Regarding data stream frequency, we simulated the most widely used strategies in the literature. It was possible to observe how this impacts the performance of each PPI and how each one trades latency for throughput in each test case. Each workload's different data streaming characteristics also allowed us to understand better how each PPI performs in these scenarios.

For reasons of space, increased complexity, or scope, our work did not address some aspects in the experiments. For example, the definition of the item-delay for the data stream frequency experiments did not use the sustainable throughput [17] of each application as a baseline. Moreover, we did not explore dynamic micro-batching or micro-batching under different data frequencies, although these are features supported by our framework. Nor did we explore different parallelism strategies. It would also be interesting to have a mechanism for stressing item clutter. We aim to address such aspects in future work.

References

[1] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–13.

[2] 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*, p. e5786, May 2020.

[3] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, Mar. 2014.

[4] Q. Zhang, Y. Song, R. R. Routray, and W. Shi, "Adaptive block and batch sizing for batched stream processing system," in *International Conference on Autonomic Computing (ICAC)*. IEEE, Jul. 2016.

[5] A. S. Abdelhamid, A. R. Mahmood, A. Daghistani, and W. G. Aref, "Prompt: Dynamic data-partitioning for distributed micro-batch stream processing systems," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: ACM, 2020, p. 2455–2469.

[6] S. Henning and W. Hasselbring, "Theodolite: Scalability benchmarking of distributed stream processing engines in microservice architectures," *Big Data Research*, vol. 25, p. 100209, 2021.

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

[8] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *Fastflow: High-Level and Efficient Streaming on Multicore*. John Wiley & Sons, Ltd, 2017, ch. 13, pp. 261–280.

[9] A. M. Garcia, D. Griebler, C. Schepke, and L. G. Fernandes, "Introducing a Stream Processing Framework for Assessing Parallel Programming Interfaces," in *29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, ser. PDP'21. Valladolid, Spain: IEEE, March 2021.

[10] ——, "SPBench: a framework for creating benchmarks of stream processing applications," *Computing*, vol. 104, no. 1, Jan. 2022.

[11] D. De Sensi, M. Torquati, and M. Danelutto, "A reconfiguration algorithm for power-aware parallel applications," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, Dec. 2016.

[12] 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.

[13] 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.

[14] G. van Dongen and D. Van den Poel, "Evaluation of stream processing frameworks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1845–1858, 2020.

[15] L. Wang, T. Z. J. Fu, R. T. B. Ma, M. Winslett, and Z. Zhang, "Elasticutor: Rapid elasticity for realtime stateful stream processing," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. Association for Computing Machinery, 2019, p. 573–588.

[16] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink, "Linked stream data processing engines: Facts and figures," in *The Semantic Web – ISWC 2012*. Springer, 2012, pp. 300–312.

[17] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1507–1518.

[18] 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.

[19] C. Balkesen, N. Tatbul, and M. T. Özsu, "Adaptive input admission and management for parallel stream processing," in *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '13. Association for Computing Machinery, 2013, p. 15–26.

[20] T. De 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, Feb. 2016.

[21] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009, pp. 281–290.

[22] 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.

[23] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, p. 230–243.

[24] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker, "Operator scheduling in a data stream manager," in *Proceedings 2003 VLDB Conference*, J.-C. Freytag, P. Lockemann, S. Abiteboul, M. Carey, P. Selinger, and A. Heuer, Eds. San Francisco: Morgan Kaufmann, 2003, pp. 838–849.

[25] S. Henning and W. Hasselbring, "How to measure scalability of distributed stream processing engines?" in *Companion of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '21. New York, NY, USA: ACM, 2021, p. 85–88.