Micro-batch and data frequency for stream processing on multi-cores

(Article begins on next page)

20 March 2024

Micro-batch and data frequency for stream processing on multi-cores

Adriano Marques Garcia[1] · Dalvan Griebler[1] · Claudio Schepke[2] · Luiz Gustavo Fernandes[1]

Abstract

Latency or throughput is often critical performance metrics in stream processing. Applications' performance can fluctuate depending on the input stream. This unpredictability is due to the variety in data arrival frequency and size, complexity, and other factors. Researchers are constantly investigating new ways to mitigate the impact of these variations on performance with self-adaptive techniques involving elasticity or micro-batching. However, there is a lack of benchmarks capable of creating test scenarios to further evaluate these techniques. This work extends and improves the SPBench benchmarking framework to support dynamic micro-batching and data stream frequency management. We also propose a set of algorithms that generates the most commonly used frequency patterns for benchmarking stream processing in related work. It allows the creation of a wide variety of test scenarios. To validate our solution, we use SPBench to create custom benchmarks and evaluate the impact of micro-batching and data stream frequency on the performance of Intel TBB and FastFlow. These are two libraries that leverage stream parallelism for multi-core architectures. Our results demonstrated that our test cases did not benefit from micro-batches on multi-cores. For different data stream frequency configurations, TBB ensured the lowest latency, while FastFlow assured higher throughput in shorter pipelines.

Keywords Parallel programming · TBB · FastFlow · Benchmark · Performance analysis · Stream processing applications

✉ Adriano Marques Garcia
  adriano.garcia@edu.pucrs.br

Extended author information available on the last page of the article

1 Introduction

Data in stream processing systems can arrive with varying intensity. Data arrival frequency, data size (including batching), data complexity, and other aspects are examples of definitions for intensity [1, 2]. Applications from different domains, such as network monitoring, fraud detection, surveillance, and social media, need to handle increasing data frequency during peak times. Regarding data complexity, a face recognizer application might be more computationally intensive in more crowded frames. Similarly, batches built based on time windows may vary their size.

Stream processing aims to process data as it arrives, in near real time. Therefore, applications in this domain are susceptible to unexpected spikes, bursty phases, and other abrupt changes in the input streams. It can cause undesirable effects that negatively impact the throughput and latency or even lead to a system failure or data loss [3]. A common strategy to avoid it is scaling the computational resources. Another practice is changing the stream characteristics to improve or achieve optimal performance levels. The use of micro-batches, for example, can amortize undesirable effects. No wonder batch size is considered one of the most important tuning parameters in stream processing systems [1, 4].

A significant research and development effort is mitigating the impact of these fluctuations in input streams and increasing fault tolerance. Scientists are constantly developing solutions for applications in this domain, both for design applications using new technologies and for adaptive systems [2, 5–9]. Therefore, varying the data stream frequency is necessary for testing the adaptability of stream processing systems. For example, increase the frequency to see whether a system can avoid backpressure, load shedding, or simulating frequency variations to analyze if the system can sustain a target throughput or latency. To our knowledge, no one has compared Threading Building Blocks [10] with FastFlow [11] in stream processing applications concerning throughput and latency under these circumstances. Works from the literature that assess these tools mainly use 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 job. In the related work (Sect. 2.1), we have not yet found support tools or frameworks in the literature that allow users to create custom stream processing benchmarks and natively support latency or throughput analysis with dynamic micro-batch sizing and data frequency.

In the past [12, 13], we proposed the SPBench, a framework for creating benchmarks of stream processing applications that makes it easier to assess multiple parallel programming interfaces. In those works, we evaluated the performance of different real-world stream applications under multiple PPIs for latency, throughput, and resource utilization. We evaluated micro-batching and data frequency in a more recent work [14]. But we only adopted one parallel pattern, the frequency patterns were not representative, and the analysis was quite limited by space issues. Also, the metrics used there were not enough to bring out some relevant aspects of the benchmarks.

This paper extends this last previous work from [14]. We use the micro-batching system to evaluate the benchmarks on another architecture and use different parallel patterns. We also improved the data stream frequency system to generate more predictable frequency patterns. In addition, we have added new metrics to SPBench that allow the measurement of instant latency and throughput. It enriches the performance analysis of PPIs and other tools that leverage stream parallelism. SPBench is free software and is publicly available.[1]

Therefore, we are *contributing* for:

- A framework that simplifies the benchmarking of micro-batch sizing and data stream frequency on stream processing applications.
- A set of algorithms for generating the literature's most commonly used data stream frequency patterns.
- An analysis of the performance impact on micro-batch sizing and data frequency, varying applications, PPIs, and parallelism strategies.

The structure of the remainder of this paper is as follows. Section 2 gives the background and discusses related works. The introduction of SPBench is with our contributions to the framework in Sect. 3. Section 4 describes the experimental methodology, including the execution environment, parallelism implementation, data frequency and micro-batching strategies, and performance metrics. The presentation of the results is in Sect. 5. Section 6 summarizes the results and adds some concluding remarks. Finally, in Sect. 7, we draw our conclusions and future work.

2 Background

2.1 Related work

In previous work, we have discussed in detail benchmarks and frameworks related to SPBench [13]. Here, we discuss related work investigating 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. [5] 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. [6] introduce an algorithm for self-adaptive parallelism for micro-batch stream processing and test it with several data stream frequency strategies. TS-BatPro [15] is a

---

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

framework that integrates time- and space-based batching techniques to optimize the energy consumption of latency-constrained applications in multi-core data centers. The technique consists of batching items to keep processors in a low-power state for longer. The authors evaluated the framework using different stream processing benchmarks, including Ferret from the PARSEC suite.

In [16], the authors propose a reconfiguration algorithm for power-aware parallel applications. They implemented the algorithm in the PARSEC [17] suite using FastFlow and tested it by changing the size of the items to simulate drops and rises in data intensity. In [7], the authors evaluated the 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. ESP-Bench [18] is a benchmark for the enterprise stream processing domain. It implements an abstraction layer using Apache Bean, which allows running the benchmark using different DSPSs. It has a configuration file where users can adjust the input stream frequency via Apache Kafka, which is the message broker used by ESP-Bench. Although it is a benchmark that aims to make it simple and easy to evaluate and compare different stream processing systems, it is still limited in terms of workload, metrics (it only measures latency), and supported architecture.

RIoTBench [19] is a benchmark suite for IoT stream processing. They evaluated Storm's performance under real workloads exhibiting increasing, decreasing, wave, and binary data stream frequency strategies. van Dongen and Van den Poel [20] evaluated the performance of micro-batching distributed stream processing systems (DSPSs) using two data intensity strategies. Wang et al. [21] presented a Storm-based framework for auto-elasticity and tested it using data size and complexity to tune the data intensity. In [22] 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. [23] propose a benchmarking framework that generates data at a configurable rate and acts as a distributed in-memory data generator for evaluating DSPSs. The authors evaluated Apache Flink, Apache Storm, and Apache Spark with different workloads. They first ran these systems at maximum intensity and then decreased it until they found the point of sustainable throughput (no back-pressure) for each. Based on this point, they varied the data frequency with a binary strategy to see how well these DSPSs could handle and adapt to spikes and rapid changes in stream speed. In [24], the authors evaluate different stream processing algorithms for anomaly and malicious network traffic detection. In their experiments, they use input streams with spikes in the frequency of incoming items, representing attacks on the network.

NAMB [25] 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. [26] 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 [27], some strategies are presented for proactive elasticity and energy awareness in data stream processing. This work target multi-core architectures and use FastFlow [11] 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. [28] evaluated pipeline parallelism and compared PThreads and TBB using different scheduling policies, optimizations, and parallel compositions.

Bebortta et al. [29] evaluated a face recognition video application with different frame rates in the input stream. The goal was to find a trade-off between computational cost and accuracy. Xu et al. [30] proposes a framework to optimize stream processing for edge computing systems. They evaluate the framework using different data stream applications and vary the frequency of arriving data in the input stream. In [31], the authors propose an algorithm for the optimal placement of stream processing operators in fog computing systems. They evaluate the proposed algorithm with different data frequencies in the input stream. Although these last three mentioned works evaluated stream processing applications with input data frequencies to different degrees, they set the frequency statically for each run.

Although many of these works evaluated different PPIs with micro-batch or data stream frequency, most aim for distributed engines [22, 23, 25, 26]. 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 [16, 27]. Regarding performance evaluation, they are commonly either latency- or through-put-aware [1, 2, 5, 6, 22]. Benchmarks that allow exploring micro-batching transfer this responsibility to DSPSs (Spark usually) [1, 19, 23]. Those that allow modifying the frequency of the input stream, usually through Apache Kafka, only allow static configurations or do not provide simplified ways for the user to manage and create specific frequency patterns. Therefore, none of the related work we found compares the impact of micro-batching and data stream frequency on the performance of different PPIs for stream processing on multi-cores. To our knowledge, there are no similar approaches for benchmarking stream processing with micro-batch and data frequency configuration.

2.2 Micro-batch stream processing

Micro-batch (or mini-batch) processing is a variant of traditional batch processing. Micro-batching systems process data in small groups at a higher frequency. Stream processing systems can use micro-batching as an optimization technique that trades throughput for latency [3, 4]. The size of a micro-batch may vary according to pre-defined criteria, such as time intervals (e.g., each 2-s 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 the data frequency and processing cost of each item are very common in stream processing.

Besides the disadvantage of increased latency, micro-batches have many advantages in stream processing. Enabling batching support in an application implies adding loops. Therefore, the compiler may optimize these loops with unrolling techniques using software pipelining. It also enables vectorization. In addition, 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]. Also, stream processing with GPUs requires batching input elements for efficient resource utilization [2]. Batching techniques can also be used to reduce energy consumption and carbon emissions of data centers [15]. Therefore, 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 batch size. It may be either statically or dynamically set [3]. StreamIt [32], for example, 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 that statically set micro-batch sizes may exhibit high latency under lower loads or may not cope with bursts in data frequency or item processing cost [1]. On the other hand, self-adaptive methods commonly use dynamic batching to react to load changes and maintain system stability. Many works focus on developing algorithms that exploit dynamic batching to improve performance or resource utilization [1, 2, 5, 6, 15, 33, 34]. 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.

Although there is ongoing work to implement support for heterogeneous and distributed systems in the SPBench, its current version only supports homogeneous shared memory architectures. In these architectures, inter-stage communication costs are minimal, and no major performance gains are expected. In a preliminary study, we evaluated the impact of micro-batching on the latency and throughput of multi-core stream processing applications. However, we later discovered implementation problems regarding throughput measurements, which led to invalid results. In this paper, we solve the throughput measurement issue and re-run the experiments. Our goal is to understand if micro-batching does bring any performance gain to stream processing applications in multi-cores and to what extent it does. Possible gains could be obtained by reduced item sorting and scheduling costs, better use of cache memory, and other system optimizations at the software and hardware level.

2.3 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. The literature uses several synonyms, such as data (or arrival, or item, or stream) rate or frequency,

Table 1 Data stream frequency patterns found in the literature

Data frequency strategy	Related work
Increasing	[6, 7, 19, 22, 35, 37–40]
Wave	[1, 6, 19, 26, 35–37, 41, 42]
Binary	[1, 5, 6, 19, 20, 23, 25, 26]
Spike	[24, 37, 42–44]

stream pressure, input frequency, and others. Here, we define *data stream frequency* as the number of items available for the source operators to read per unit of time. In practice, in this work, we add a time delay at the beginning of the source's main loop. Decreasing the time delay entails increasing the data 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. [23]. Examples of loads that can present huge and often predicted fluctuations are data from network monitoring, traffic control, GPS, social media, and others. These fluctuations link to the times people use these services the most throughout the day and draw a waveform. However, many works in the literature need to do tests with abrupt fluctuations and shorter periods [6, 19, 23, 25, 26].

Sometimes, when the data frequency exceeds the sustainable throughput [23, 35] of the application, this can result in an increased allocation of computational resources and trigger amortization techniques such as batching or back pressure mechanisms [3, 23]. These mechanisms are activated to avoid data loss (e.g., load shedding), and frameworks may require several minutes of resource reconfiguration to increase data throughput. Some works evaluate these mechanisms to deal with back pressure and propose improvements to increase the performance of the reconfiguration algorithms [2, 5, 6, 36]. Also, in micro-batch systems, such as Spark-based ones, data frequency can be tied to variations in batch size [20]. Therefore, it is important to have benchmarks that help simulate specific data overload scenarios so that these systems can be deeply evaluated.

In some cases, data stream frequency (and micro-batching) merges with the concept of data intensity or complexity [45]. In these cases, the workload defines its behavior by the size or computational cost of the items. In [16], for example, the authors use an image processing application and cut the resolution of the input images at specific points by half. They used it to simulate a binary pattern. Stein et al. [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, since they naturally present fluctuations in the processing cost.

Table 1 shows the most frequently used frequency patterns in the related work. We did not search the literature specifically for frequency patterns; here, we only list the patterns found at least twice in the works presented in Sect. 2.1. Other less commonly found patterns, such as *random-walk* [11], may be explored in the future. Users can also add custom patterns to the SPBench (Sect. 3.4). Therefore, Table 1 shows that *wave*, *binary*, *increasing*, and *spike* are the most commonly

used patterns in related work. Based on this, we selected these patterns to add to SPBench. This way, through SPBench, users can create custom benchmarks, run tests with these frequency patterns, and even create other custom strategies.

3 SPBench

SPBench was first introduced on [12], and the first release was fully described and evaluated on [13]. It is open source and available under the GNU General Public License version 3 (GPLv3). 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 add and modify parameters, such as build dependencies and metrics, through a command-line interface (CLI). It is fully modular and the parallel code can be quickly replicated across all SPBench applications. Therefore, through SPBench, users can create from unique scenarios to test specific functionality to more comprehensive scenarios to test different aspects with various benchmarks.

The way users interact with the SPBench is a distinguishing feature of our framework. The SPBench API presents the core of each application in a standard way and with few lines of code. The original source code of such applications can be thousands of lines long. Here, the low-level details of the applications are abstracted and become transparent to the users. Users can access through the CLI a database containing all applications/benchmarks previously added to the SPBench suite (e.g., the ones we used in the past and this work) and their new customized versions. Also, other secondary parameters can be tuned via the CLI with simple commands. Therefore, SPBench 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.

While there are initiatives aimed at adding abstraction layers for writing parallel code [46, 47] or for other aspects of parallelism in stream processing [7–9, 48], few works aim to parameterize and add abstractions for the benchmark applications in this area. This way, SPBench is helpful for researchers to test their new solutions and technologies for stream parallelism and learning/teaching purposes.

The operation of SPBench is straightforward. All users need to do to run a benchmark after downloading the framework is to execute the following commands:

- `./spbench install` (install the application dependencies)
- `./spbench download-inputs` (download all the inputs)
- `./spbench compile -bench all` (compile all the benchmarks)
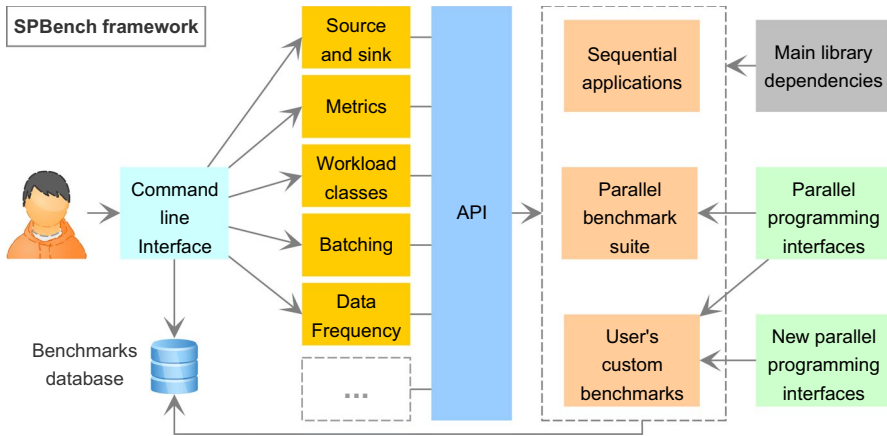- `./spbench exec -bench<benchmark> -input<class>` (run it)

Fig. 1 SPBench framework [13]

The following command, for example, can be used to run a Bzip2-TBB benchmark with the large input class and eight threads, measuring the average latency and throughput and enabling in-memory mode:

```
./spbench exec -bench bzip2_tbb_farm -input large -nthreads 8
                -latency -throughput -in-memory
```

Figure 1 shows the SPBench framework. The current version allows users to select receiving data from a disk or memory. Users can evaluate latency, throughput, and resource usage at different granularity and depth levels. Users can also choose from distinct workload classes for stressing characteristics in the application and PPI. In this work, we extended and improved the dynamic micro-batch sizing and data stream frequency mechanisms. These new features are already fully integrated to the framework and functional.

3.1 Using micro-batches in SPBench

To implement batch support in SPBench, instead of each item containing a single piece of data, it now contains an array of data and carries information about its batch size. We have added loops to all the operators in the application to process these arrays. This way, the size of the batches can be changed statically or dynamically. The following command is an example of how to use batch statically:

```
./spbench exec ... -batch-size 5 -batch-interval 2
```

The command above will run the application creating micro-batches with 10 items or a 2-s interval, whichever occurs first. Therefore users can set a single batch

size limiting parameter or both combined. To change batch sizes dynamically during execution, users can use the methods `SPBench::setBatchSize()` or `SPBench::setBatchInterval()` inside the code.

3.2 Frequency manager

The frequency in SPBench is set according to a time delay applied at the beginning of the source operator. A longer time delay implies a lower frequency and vice versa. A frequency of 100 means that at most 100 items will be available to be consumed by the source per second. It does not imply that the source will send 100 items per second to the workers because the source of each application has its delay, which can be caused either by the processing of the source itself or by delays introduced in the stream by subsequent stages. To set a static frequency throughout the execution, users can simply use the `-frequency` parameter in the run command, as below:

```
./spbench exec ... -frequency <number of items per second>
```

Algorithm 1: SPBench's algorithm for frequency management

---

1 **Function** `freqManager`($lastItemTimestamp$) **do**
2 $\quad frequency \leftarrow$ `frequencyPatternGenerator()`
3 $\quad expectedDelay \leftarrow 1000000/frequency \qquad \triangleright$ In microseconds
4 $\quad itemProcTime \leftarrow$ `currentTime()` $- lastItemTimestamp$
5 $\quad actualDelay \leftarrow expectedDelay - itemProcTime$
6 $\quad$ **if** $actualDelay > 0$ **then**
7 $\quad\quad | \quad usleep(actualDelay)$
8 $\quad$ **end**
9 **end**

Algorithm 1 presents the methodology used to manage the frequency and compute the delay. The function `freqManager` is called at the beginning of the source operator. It takes the source's last processed item's timestamp as an argument. The timestamp of each item is also assigned at the beginning of the source. The first thing `freqManager` does is to adjust the frequency according to the set frequency pattern. If no pattern has been set, `frequencyPatternGenerator` returns the frequency itself. Then, on line 3, the expected time delay in microseconds is computed according to the predefined frequency. Next, it computes the time elapsed since the source received the last item. This elapsed time is then subtracted from the expected delay (line 5). If the result of the subtraction is higher than zero, the source still has to wait to receive the next item. Otherwise, the item is released immediately. So, for example, if the user has set the frequency to 2 items per second (500 ms delay), and the source took 150 ms to process the last item, then `freqManager` will add a delay of 350 ms to release the next item.

3.3 Frequency patterns

The previous version of SPBench [14] implemented the frequency patterns in a static way. It was item oriented, which means that building up the frequency patterns would require previous knowledge about the number of items to be processed. Then it sliced this number into twenty steps and the frequency would increase or decrease according to the chosen frequency pattern at each step. It led to unwanted behaviors, such as the inability to keep the input at a high-frequency state for more than a small fraction of time in high-throughput applications. In this work, we have improved the frequency induction algorithms. Now, the computation is time based rather than item based. This way, we can draw much more predictable and accurate frequency waveforms.

The frequency patterns currently supported by the SPBench are sine wave, binary, increasing, decreasing, and spike. All these patterns are set via four parameters: pattern type, cycle period, minimum frequency, and maximum frequency. The exception is the spike pattern that can also get an additional parameter to define the spike duration interval as a percentage of the period (default is 10%). Users can use the frequency patterns through the `-freq-patt` exec parameter:

```
./spbench exec ... -freq-patt <pattern,period,minFreq,maxFreq>
```

Another alternative is to use the `SPBench::setFrequencyPattern()` function available for use inside the benchmark. This function can change the patterns and their behavior anytime during the run. Therefore, the data frequency in the input stream is highly configurable in SPBench. The following subsections describe how we implemented these frequency patterns within SPBench.

3.3.1 Sine wave frequency pattern

The SPBench implements the wave pattern based on a sine wave given by Eq. (1) [49].

$$A \cdot sin(2\pi f t + \varphi) + k \tag{1}$$

where:

- A is the amplitude, the peak deviation of the function from zero.
- f is the ordinary frequency, the number of cycles that occur each second of time.
- t is the elapsed time.
- φ is the phase shift, where in its cycle the wave is at $t = 0$.
- k is the vertical shift from zero.

Algorithm 2: Sine wave frequency pattern

---
1   $A \leftarrow (maxFrequency - minFrequency)/2$
2   $f \leftarrow 1/period$
3   $k \leftarrow A + minFrequency$
4   **Function** `wavePatternGenerator`(A, f, k) **do**
5      $newFrequency \leftarrow A \times sin(2 \times \pi \times f \times$ `elapsedTime()`$) + k$
6   `setFrequency`$(newFrequency)$
7   **end**

So, based on the input parameters and the sine wave equation, we implement the input frequency as described in Algorithm 2. In this algorithm, lines 1–3 represent the computation of some presets for the sine wave function. These variables are computed only once or every time the pattern is modified since users can dynamically change the parameters or select a different pattern during the execution. The `wavePatternGenerator` function receives these parameters and uses them and the application's execution time to compute the sine, updating the frequency.

3.3.2 Binary frequency pattern

The binary frequency pattern interchanges between the maximum and minimum frequency set by users with an instantaneous transition, unlike the sine wave, where the transition is smooth. Therefore, the frequency under this pattern draws a kind of square wave. This pattern can simulate scenarios where sudden changes occur and remain for some time in the input stream.

Algorithm 3: Binary frequency pattern

---
1   $halfCycle \leftarrow period/2$
2   $isAtMaxState \leftarrow false$
3   $halfCycleStartTime \leftarrow$ `currentTime()`
4   **Function** `binaryPatternGenerator`$(spikeInterval)$ **do**
5      $halfCycleElapsedTime \leftarrow$ `currentTime()`$ - halfCycleStartTime$
6   **if** $halfCycleElapsedTime > halfCycle$ **then**
7         $halfCycleStartTime \leftarrow$ `currentTime()`
8   **if** $isAtMaxState == false$ **then**
9   `setFrequency`$(maxFrequency)$
10            $isAtMaxState \leftarrow true$
11   **else**
12   `setFrequency`$(minFrequency)$
13            $isAtMaxState \leftarrow false$
14   **end**
15   **end**
16   **end**

The frequency in the binary pattern has two possible states: minimum or maximum. These states alternate once in a period (it changes from minimum to

maximum frequency) and then repeat this pattern cyclically. Algorithm 3 shows the methodology we used to implement it. It uses a variable to store the current state (lines 2, 10, and 13) and changes between the states each half cycle (line 6).

3.3.3 Increasing and decreasing frequency patterns

The increasing and decreasing patterns are not cyclic. Here, the frequency increases or decreases linearly and once, from minimum to maximum, or vice versa, over the user-defined period. After the period, the frequency no longer changes and remains constant at the user-defined maximum frequency (increasing case) or minimum frequency (decreasing case). These frequency patterns can help test adaptive resource provision scenarios over long intervals, for example.

Algorithm 4: Increasing frequency pattern

---

1 $increStep \leftarrow (maxFrequency - minFrequency)/period$
2 setFrequency($minFrequency$)
3 $stepStartTime \leftarrow$ currentTime()
4 **Function** increasingPatternGenerator() **do**
5   **if** currentFreq() $< maxFrequency$ **then**
6         $stepElapsedTime \leftarrow$ currentTime() $- stepStartTime$
7         $newFreq \leftarrow$ currentFreq() $+ (stepElapsedTime \times increStep)$
8   setFrequency($newFreq$)
9         $stepStartTime \leftarrow$ currentTime()
10   **end**
11 **end**

Algorithm 4 presents our strategy to implement the increasing pattern in SPBench. Generating this pattern is not as simple as it may seem. First, we divide the range between maximum and minimum frequency by the period (line 1). This value represents how much we need to increase the frequency per second to draw a linear increase over the period. However, this algorithm does not run independently, in a separate thread, for example. It runs every time an item arrives at the source. However, items may arrive at irregular intervals because of the workload's intrinsic characteristics, among other factors. Therefore, SPBench always needs to recalculate the step size at an increasing frequency. To do this, we take the elapsed time in seconds from the last step and multiply the step size that should be taken per second (*increStep*) by this value. After recalculating the step size, the algorithm adds this value to the current frequency (line 7 of the algorithm). Thus, if the source operator took three seconds to process an item, the SPBench will take three steps in incrementing the frequency for the next item. Similarly, if an item took 500 ms, the algorithm will only take half an increasing step for the following item. The strategy that generates the *decreasing* pattern is the opposite logic.

3.3.4 Spike frequency pattern

The spike frequency pattern is cyclical and creates one spike per period. This pattern can represent scenarios where the frequency changes abruptly for short intervals. Here, users can modify the spike's duration in addition to the maximum and minimum frequency and period. This duration is defined as a percentage of the period, and the default value is 10%. In 10 s, during the last second, the SPBench would incrementally increase the frequency to the maximum and drop it to the minimum frequency again, creating a one-second spike with a sawtooth shape, for example.

Algorithm 5: Spike frequency pattern

1 setFrequency($minFreq$)
2 $spikeInterval \leftarrow period \times (spikeSize/100)$ ▷ Spike size is given as percentage of period
3 $spikeIncreFactor \leftarrow (maxFreq - minFreq)/spikeInterval$
4 $lowInterval \leftarrow period - spikeInterval$
5 $periodStartTime \leftarrow$ currentTime()
6 **Function** spikePatternGenerator() **do**
7    $periodElapsedT \leftarrow$ currentTime() $- periodStartTime$
8   **if** $periodElapsedT > lowInterval$ **then**
9       $step \leftarrow (periodElapsedT - lowInterval) \times spikeIncreFactor$
10       $newFreq \leftarrow minFreq + step$
11   setFrequency($newFreq$)
12   **if** $periodElapsedT > period$ **then**
13   setFrequency($minFreq$)
14          $periodStartTime \leftarrow$ currentTime()
15   **end**
16   **end**
17 **end**

The strategy used to create this pattern is present in the Algorithm 5. Like the other algorithms, there is an initialization phase that runs only once. First, we set the frequency to the minimum. Then we compute the duration of the spike based on the percentage of the period the user has chosen. With this value, on line 3, we calculate the spike factor, which is how much we need to increase the frequency per second during the spike phase. Here we also calculate how long the low-frequency phase, which is the off-spike phase in each cycle, should last (line 4). Therefore, whenever the elapsed time of the cycle is longer than the low phase (line 8), the algorithm creates a spike. The logic we use to increase the frequency in the spike, in lines 9 and 10, is the same logic we use in the increasing pattern. For the last, every time the elapsed time of the cycle is greater than the user-defined period (line 12), the spike ends, and the cycle starts again.

3.4 Adding new frequency patterns to SPBench

The frequency manager presented in Sect. 3.2 also enables users to use their own frequency pattern in the SPBench benchmarks. There are different ways to do this, but most involve calling the method `spb::setFrequency()`. So users can call this method on-the-fly to change the frequency according to a particular pattern. In this case, users still need to implement the computation that results in the desired pattern. The code snippet in Listing 1 shows an example of how this could be achieved. In this example, the frequency pattern consists of initializing the frequency with ten items per second and increasing this rate by 0.5 with each new item.

```
1  float myCustomPattern(){
2      return spb::getFrequency() + 0.5;
3  }
4  int main (int argc, char* argv[]){
5    spb::setFrequency(10);
6    /* Stream region */
7    while(1){
8      spb::setFrequency(myCustomPattern());
9      /* Application operators */
10   }
11 }
```

Listing 1: Example of benchmark implementation using a custom frequency pattern in SPBench.

4 Experimental methodology

This section presents the methodology we used for the experiments, which includes a description of the execution environment, the parallelism strategies we used, how the SPBench was set up, and how we evaluated performance.

4.1 Experimental setup

In this section, we discuss the methodology that was used for the experiments in the next sections. In the experiments, we used two computers with Intel processors and a computer with an AMD processor. The first Intel one has 144 GB of RAM and two Intel® Xeon® Silver 4210 CPU @ 2.20 GHz processors (a total of 20 cores and 40 threads). The other Intel computer has two Intel® Xeon® E5-2620 v3 @ 2.40 GHz processors (total of 12 cores and 24 threads) and 32 GB of RAM. The AMD one has 16 GB of RAM and an AMD Ryzen™ 5 5600X CPU @ 3.70 GHz (a total of 6 cores and 12 threads). We used GCC 9.4.0 with `-O3` flag, and `performance` governor was enabled in all of them. The Xeon E5-2620 has Ubuntu 18.04.5 LTS operating system with Linux kernel 4.15.0-156-generic, and the others have Ubuntu 20.04.4 LTS, with Linux kernel 5.4.0-105-generic. Other

Fig. 2 Farm parallel pattern, where *E* stands for Emitter, *W* for Workers, and *C* for Collector
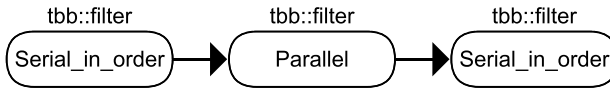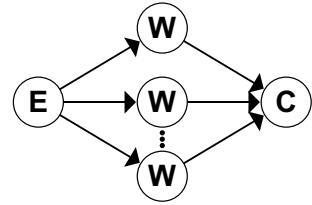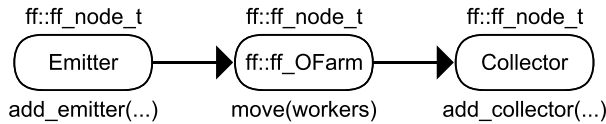

Fig. 3 TBB farm implementation

Fig. 4 Farm implementation in FastFlow

libraries used were OpenCV 2.4.13.6 for the video processing benchmarks, Intel TBB 2020 Update 2 (TBB_INTERFACE_VERSION 11102), and FastFlow version 3.

4.2 Parallelism strategies

We built benchmarks using four applications supported by SPBench [13]: Bzip2, Lane Detection, Face Recognizer, and Ferret (from PARSEC) [17]. We parallelize these applications with Intel TBB and FastFlow. We implemented a farm parallel pattern for Bzip2, Face Recognizer, and Lane Detection. The basis of a farm is a three-stage pipeline consisting of an Emitter stage (source), a Worker stage, and a Collector stage (sink). However, the worker stage can be replicated *n* times to increase the application's performance, as shown in Fig. 2. The Ferret application was originally implemented as a pipeline of farms in the PARSEC suite [17], i.e., its pipeline has four intermediate stages (workers), and each worker can be replicated *n* times. So there are four consecutive farms. We could merge them into a single farm for optimization. However, since the other benchmarks already implement a single farm, we have chosen to keep Ferret's original parallel pattern. In the future, we can explore this and other optimizations, like implementing it as a farm of pipelines instead of a pipeline of farms.

The farm implementation with TBB and FastFlow is simple. In the TBB version, we build a three-stage pipeline and implement the worker stage as a parallel filter, as illustrated in Fig. 3. In FastFlow, we used the ff_node_t directive to implement
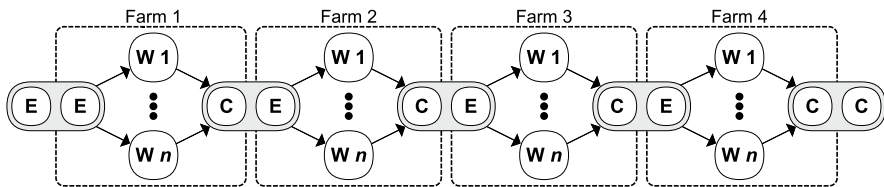
Fig. 5 Pipeline-farm implementation in FastFlow

the pipeline stages and the `ff_Farm` (or `ff_OFarm` for ordered farm) directive to create a farm. Then, we added the emitter and collector, as shown in Fig. 4. For the pipeline-farm (pipeline of farms) pattern with TBB, we basically added more parallel filters into the pipeline. With FastFlow, however, we had to put more work into implementing an optimized pipeline farm.

Unlike TBB, where we manually implemented a farm by building it on top of the pipeline, FastFlow natively provides a farm structure. Therefore, each farm in FastFlow has its own emitter and collector stages, and both run in a new thread. These stages continue to exist when the farms are composed into a pipeline. However, FastFlow, in newer versions, can automatically merge the Emitter of one farm and the Collector of the next farm into a single stage. Then we manually merged the first stage (source) with the Emitter of the first farm. We did the same with the sink of the pipeline and the Collector of the last farm, just as we did with a single farm (Fig. 4). It reduces the number of threads used by these stages by half and eliminates the communication queue that would otherwise exist between them. The final structure of the pipeline-farm Ferret with FastFlow is illustrated in Fig. 5. Nevertheless, in a pipeline-farm implementation where the sum of the maximum number of workers between TBB and FastFlow is equivalent, the number of threads used by FastFlow is higher. That is, in this implementation, Fastflow uses five extra threads just to run the Emitters and Collectors, represented by the gray nodes in Fig. 5.

The difference in thread usage makes it very difficult to compare both PPIs fairly. This problem is exacerbated by load imbalance, where TBB has the advantage due to its task scheduling policy [10]. For FastFlow, equally dividing the maximum number of threads the architecture supports among the farms is unfair since TBB automatically assigns more threads to process tasks where there is a bottleneck. FastFlow, on the other hand, allocates threads statically among workers. Therefore, if the threads are equally distributed among the farms, low-load stages become idle.

To circumvent these issues, we evaluated the pipeline-farm performance of FastFlow in several ways, using the fine-tuning options that this PPI allows. The flexibility of tuning is one of the great differentiators of FastFlow [11]. However, finding a configuration that benefits both throughput and latency is challenging, especially in the presence of load unbalance. In this work, we choose a strategy that prioritizes throughput. For this, we put FastFlow in blocking mode for the idle threads to free up computational resources. Additionally, instead of splitting the maximum number of workers between the farms, we run each farm with the maximum number of workers. This way, when less loaded workers/threads free up resources, the

system can reallocate those resources to more loaded workers/threads. This strategy has great potential to improve throughput performance in an application like Ferret, where only one of the fourth intermediate stages is more computationally intensive.

We also configure other attributes of these PPIs. For TBB, we defined *ntokens* = *nthreads* × 10 [28]. In FastFlow, in addition to blocking mode (compiling with `-DBLOCKING_MODE`), we use communication queues of size one between stages (`-DFF_BOUNDED_BUFFER -DDEFAULT_BUFFER_CAPACITY=1`). In our experiments, this queue size does not significantly impact throughput but greatly reduces latency. We also compiled the benchmarks with `-DNO_DEFAULT_MAP-PING` for disabling FastFlow's thread pinning as recommended when using hyperthreading. We have also enabled SPBench in-memory mode to achieve higher frequencies and avoid I/O bottlenecks.

4.3 Performance evaluation methodology

We measure application performance in two different ways. The first one is the overall average latency and throughput. Here, the average latency is processing time latency, that is, the elapsed time from the moment the source reads an item until the moment the last stage of the pipeline finishes processing it. Throughput means items processed per second. Items within a batch are counted individually. The results are an average of 10 runs of each benchmark, including the standard deviation as error bars. Since the standard deviation was very low in most cases, the error bars may be almost invisible on the graphs.

The second way we measured performance was by monitoring each benchmark throughout its execution. We used the routines of the SPBench, which allows for performance monitoring with microsecond precision. We measure the instantaneous latency and the instantaneous throughput, which is the average of these metrics over a short time interval. The interval we set for the workload characterization, presented in Sect. 5.1, was 5 s since fewer items are processed by second. However, the parallel benchmarks have higher throughputs. Therefore, we have reduced the average interval to 250 ms. We also choose a minimum of 250 ms monitoring interval to avoid interfering with the results [27].

4.4 Micro-batching and data frequency strategies

To model the frequency patterns through SPBench, we adopted the same criteria for all benchmarks. We first run the benchmark with the same parameters but without setting a specific frequency. Then we measure this sample's average throughput and execution time and use these data to model the frequency patterns and re-run the benchmarks.

For all the patterns, we set the minimum frequency to be 10% of the average throughput of the sample. The maximum frequency was equal to 110% of the average throughput. For the periodic patterns (wave, binary, and spike), the size of each period was set to 1/3 of the sample execution time for at least three cycles to occur. The period was equal to the execution time for the non-cyclical patterns (increasing

and decreasing). We also set each spike duration as 10% of the period for the spike pattern.

For micro-batch experiments, we first evaluate it under multiple parallelism degrees. Thus, we used static micro-batch sizes for these experiments. We also evaluate it by dynamically changing the size of the batches during the execution of the benchmarks. We increased the batch size from 1 to 10 in this case. SPBench also supports limiting the size of batches by the number of items or time windows. But analyzing micro-batches by time window adds too many other variables and would open up the scope of this work. Therefore, we use micro-batches limited by the number of items.

5 Experimental results

This section aims to understand the impact of micro-batching and data frequency on performance. Also, it demonstrates the load behavior of the different benchmark configurations for micro-batching and data frequency. We first characterize the workloads and then present the performance, micro-batching, and data stream frequency experimental results.

5.1 Workload characterization

The SPBench workloads are quite diverse and allow for the representation of different scenarios. SPBench provides at least five classes of workloads, and in this work, we use the Huge class in all cases. The Huge class of workloads means one 120-s video for Lane Detection and one 30-s video for Face Recognizer, both with 360p resolution. For Bzip2, we use a 4.2 GB Wikipedia dump file. For Ferret, it is the same as the *Native* workload from the PARSEC suite [17]. More details are available in the SPBench documentation.[2]

In Fig. 6 we present performance results from sequential benchmarks of each SPBench application. The metrics were obtained through the SPBench monitoring mode and measured every five seconds during the execution of the benchmarks. All results were obtained simply by executing the following SPBench command once:

```
./spbench exec -ppi sequential -input huge -monitor 5000
```

The above command runs all sequential applications with the Huge workload class and monitors performance every 5000 ms. Each column in Fig. 6 represents one of the applications. The four applications have the same *y*-axis scale for each metric for easy comparison. In the first row of graphs, we evaluate CPU usage. All benchmarks show high CPU usage, quickly reaching 100%, indicating great potential for improving performance through parallelism.

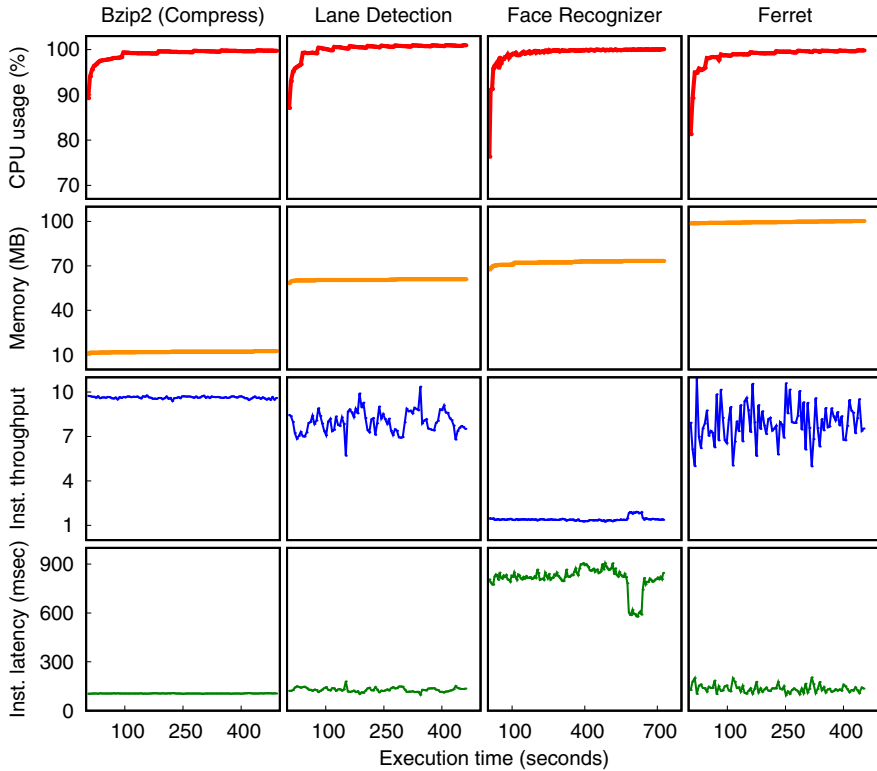---

[2] https://spbench-doc.rtfd.io.

Fig. 6 Characterization results (on Intel Xeon Silver 4210)

The second row of graphs presents memory usage. The benchmarks show relatively diverse behavior in this respect. Bzip2, for example, is the SPBench application that uses the least memory, and Ferret is the one that uses the most. Lane Detection and Face Recognizer have quite similar memory usage. Although there is an increase in memory consumption throughout the execution of these applications, this increase is relatively low.

The third metric is instantaneous throughput, i.e., average throughput measured over a short time interval. In these cases, the time interval we used was 5 s (the same for latency). Comparing the throughput of the applications, we can see that Bzip2 varies the least throughout the execution. Its throughput varied less than 0.4 items per second throughout the execution, a steady workload. The one that varies the most is Ferret, with up to 6 items per second variation, similar behavior for Lane Detection.

The last row of graphs in Fig. 6 shows the instantaneous latency (5-s average). As expected, latency pretty much mirrors throughput. Latency spikes indicate regions of the input stream that demand more computation and vice versa. For example, in Lane Detection, when a car changes lanes or during intersections on the road, there will be more lanes to detect. In Face Recognizer, there are moments in the input
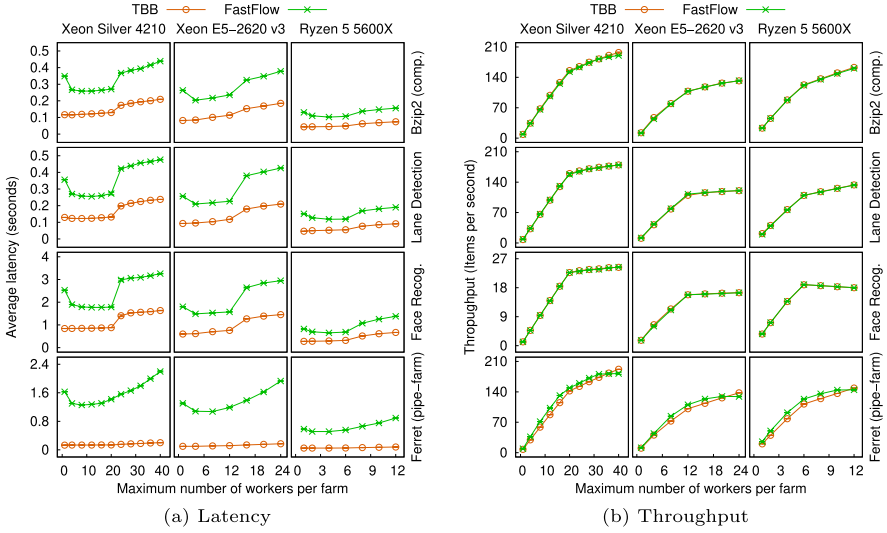
Fig. 7 Latency and throughput results of the FastFlow and TBB benchmarks in different architectures, varying the parallelism degrees

where several faces are in the frame, creating latency spikes, and moments with no faces at all, which is the case of that big drop in latency in the graph. Therefore, the workloads used are pretty diverse and represent multiple scenarios.

5.2 General performance results

We have run experiments to evaluate SPBench performance on two Intel and one AMD architecture. It is essential to evaluate the performance of the benchmarks, along with the workload characterization, to draw baselines that will help better understand the micro-batch and frequency results. In addition, the benchmarks had not yet been evaluated on non-Intel architectures. Since we are using Intel TBB, it is important to see whether or not it has any advantage on Intel architectures in these situations.

Figure 7 shows the average latency and throughput results for each benchmark with TBB and FastFlow, varying the degree of parallelism and the underlying architecture. Regarding latency (Fig. 7a), we can see that all applications implementing a single farm have similar behavior. They keep the latency lower until they use all the physical cores of the processors, and then there is a jump. In pipeline farm (Ferret), the results between TBB and FastFlow are more distant, which is explained by more queues among the stages in FastFlow. The latency maintains the same behavior across the architectures but at different scales. Possibly the most impacting variable here is the processor clock frequency, which is higher in AMD. Also, the Ferret latency on the Xeon Silver 4210 starts to reduce again at about 40 workers. Better load balancing seems to occur in this case.

Regarding throughput (Fig. 7b), it is interesting to see that the AMD processor can achieve higher throughputs than the two Xeon E5-2620, even though it has half the available processing cores. Besides having a higher clock, the AMD Ryzen 5 5600X is a much newer architecture, which explains these results. However, the highest throughputs were achieved by the Xeon Silver 4210. Although it has the lowest clock frequency (2.20 GHz, against AMD's 3.7 GHz), its large number of cores makes up for this. TBB and FastFlow showed equivalent performance in most cases. The exception is in Ferret, where FastFlow achieves higher throughputs, thanks to the resource over-allocation used by our implementation strategy (Sect. 4.2).

The performance difference between the farm and pipeline-farm parallel patterns is not evident in the TBB results. However, it impacts FastFlow to a greater extent, mainly latency, and many factors may explain it. Adding more stages to a pipeline can cause load imbalance and add more buffers/queues to communicate the stages. It also incurs ramp-up and ramp-down problems, which happens when the application starts and the last stages are idle and when the stream ends and the first stages become idle while waiting for the computation to finish. With TBB and its work-stealing scheduling policy, each thread tries to execute all operations on the same item through the pipeline. This tasking model alleviates the ramp-up or ramp-down problem while also providing better load balancing by allocating more tasks to different stages dynamically [10]. Therefore the communication delay among the pipeline stages is decreased. Besides, in FastFlow, each thread executes the same operator/stage on several items throughout the execution. This model adds static communication queues between the stages and, if there is no load balancing among the stages (which is the case with Ferret), this can impact the application performance [14]. It, in addition to the blocking mode, explains the higher latency of Fast-Flow compared to TBB. However, the combination of blocking mode + overprovisioning was a simple strategy for pipeline farm that managed to deliver throughput competitive with TBB. In the end, in the evaluated cases, TBB and FastFlow presented good performance portability for the evaluated architectures, and TBB has no apparent advantage on Intel processors.

5.3 Micro-batching results

In previous work [14], we evaluated the impact of micro-batch size on the performance of stream processing applications on multi-core systems. In the experiments we performed in that work, increasing the batch size could incur throughput increases, besides the expected increase in latency. However, we later identified inconsistencies between the SPBench metrics system and the new batching features that were added in that work. This problem was causing the throughput to be overestimated. Here, in this work, we fix the metrics problem for batching. Therefore, we performed more experiments to check to what extent the advantages of batching discussed in Sect. 2.2 apply in multi-core systems.

In the micro-batch experiments, we used two strategies. In the first one, we vary the batch size dynamically at execution time from 1 to 10. The goal was to validate this feature we added to SPBench and observe the impact of the batch size on the
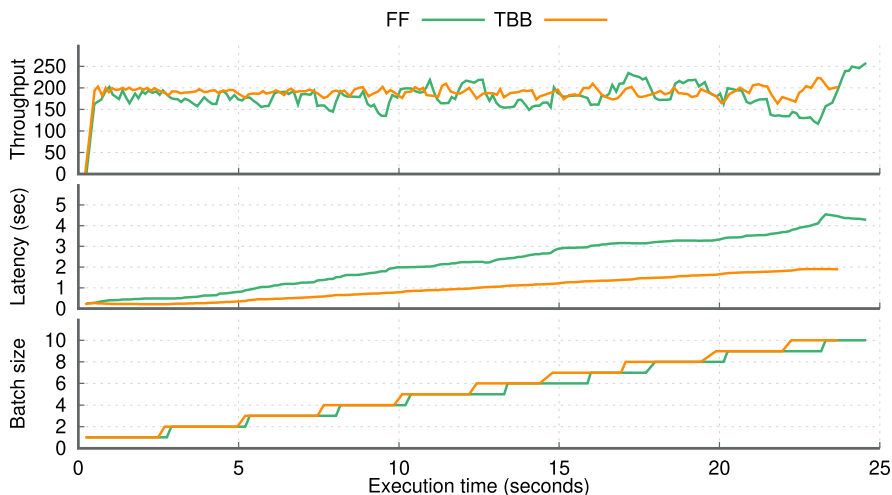
Fig. 8 Throughput and latency results of Bzip2 benchmark implemented as a farm (40 workers) with TBB and FastFlow, increasing the batch size dynamically from 1 to 10 along the execution
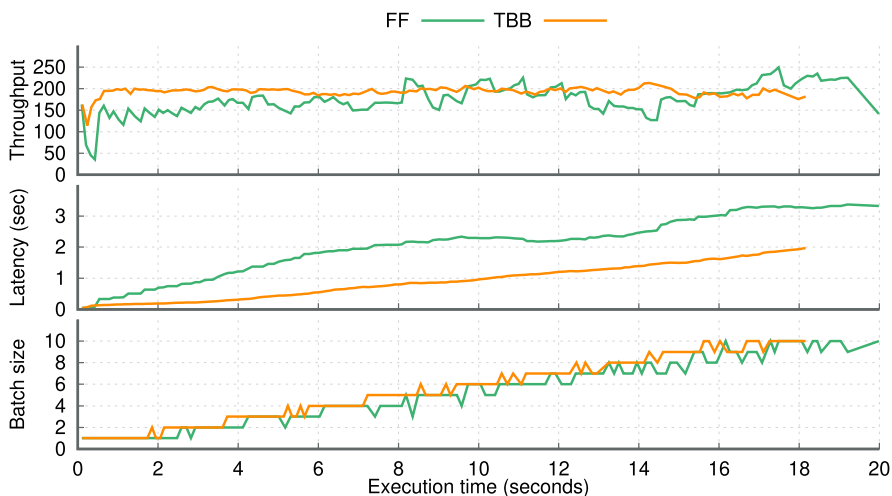

Fig. 9 Throughput and latency results of Ferret implemented as a pipeline of farms (maximum of 40 workers per farm) with TBB and FastFlow, increasing the batch size dynamically from 1 to 10 along the execution

performance of the benchmarks at execution time. Figure 8 shows the throughput and latency results of the Bzip2 application on Intel Xeon Silver 4210. We can see that the FastFlow implementation took longer to run. Also, FastFlow has a less stable instantaneous throughput than TBB, and the increase in batch size expels the latency difference between PPIs. The increase in batch size apparently did not influence the application's throughput.
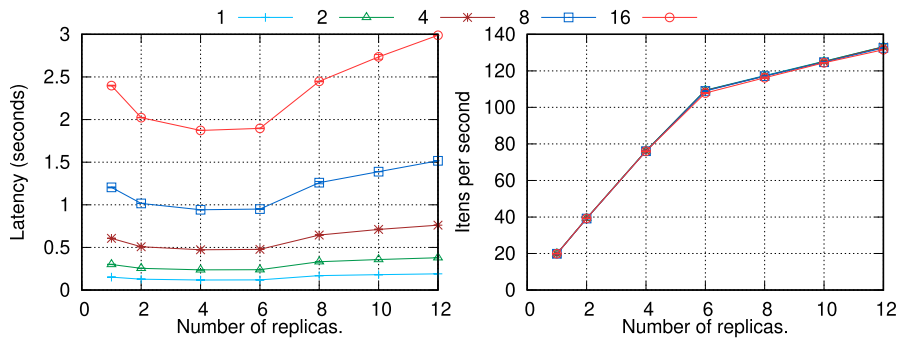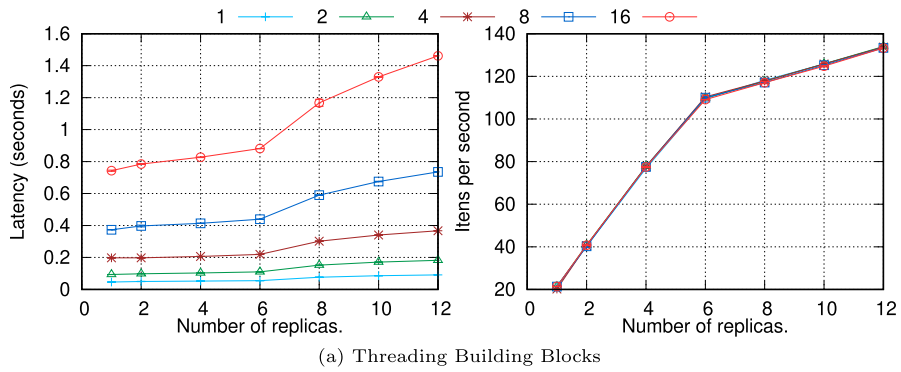
(a) Threading Building Blocks

Fig. 10 Latency and throughput results for Lane Detection with multiple parallelism degrees and statically set micro-batch sizes (AMD Ryzen 5 5600X)

Figure 9 presents the results of the Ferret benchmark. The same discussion made in the case of Bzip2 applies here. FastFlow has a more unstable throughput and increases latency at a higher rate than TBB as the batch size increases. In the third graph in Fig. 9, regarding batch size, there is a difference from Bzip2. Ferret is an application that does not require item sorting. The batch size monitoring in SPBench is performed in the last stage of the pipeline. Therefore, the batch size spikes represent items arriving out of order in the sink. It shows that our FastFlow implementation causes more items to arrive unordered at the sink than TBB. This factor can also impact the latency of FastFlow in applications that require sorting.

We also investigate how batch size impacts application performance when varying degrees of parallelism. Here, instead of changing the batch size dynamically, we set it statically at the beginning of the execution. Therefore, besides the number of farm workers, we also vary the micro-batch size from 1 (no batch) to 16 items per batch. One of our goals with these experiments is to see if using batches could alleviate the cost of item ordering in applications with ordering constraints. After all, it is known that this cost can impact latency and throughput [50]. Therefore, if we decrease the number of items in the stream using batches of ordered items, there may be some performance gain in applications with issues with item cluttering.
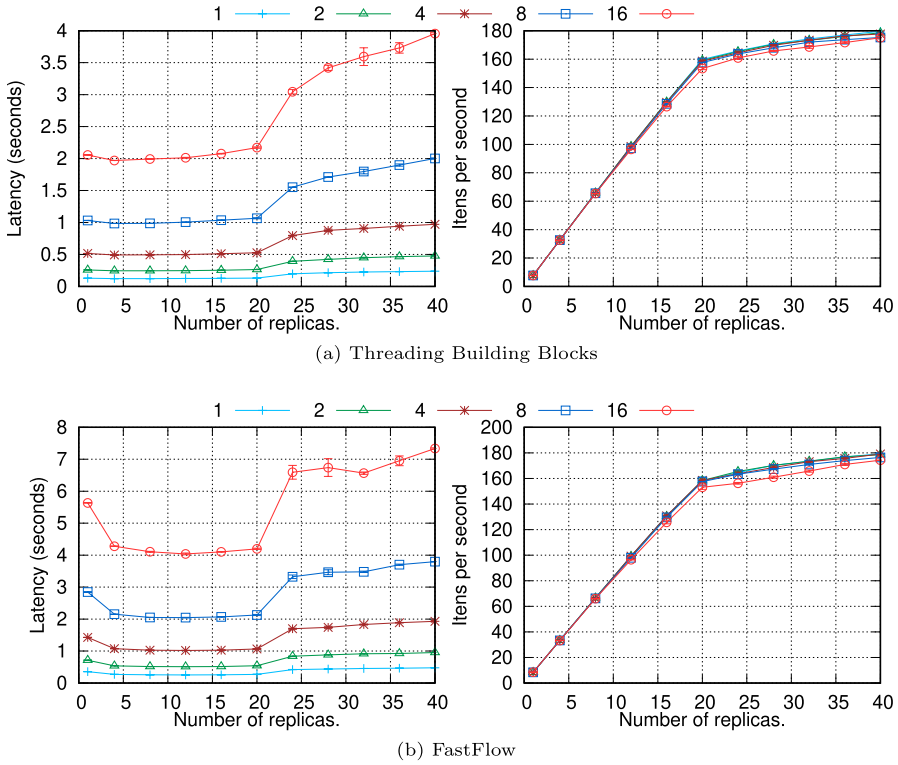
(a) Threading Building Blocks

(b) FastFlow

Fig. 11 Latency and throughput results for Lane Detection with multiple parallelism degrees and statically set micro-batch sizes (Intel Xeon Silver 4210)

Figure 10 presents the results of the micro-batch experiments varying the degree of parallelism in the Lane Detection benchmarks using the AMD computer. Lane Detection is a good test case for us to evaluate since it has a high throughput (more chances of items arriving at sink out of order), and the computational cost of each frame varies greater than Bzip2 or Face Recognizer, as seen in Fig. 6. This further increases the clutter of items in the stream, and this application requires the frames to be ordered in the output video. The results show that for both TBB and FastFlow the latency increases proportionally to the batch size. However, there is no significant throughput gain at all. We can conclude that batch played no role here.

We repeated the experiment as shown in Fig. 10, but we used the computer with the Intel Xeon Silver 4210 this time. This computer has a total of 40 cores, and we can use more workers. The increase in workers causes more items to clutter because more concurrent threads are writing to the output queue. These results are shown in Fig. 11. The latency behavior remained the same in this case. However, the throughput got worse when increasing the batch size at a higher degree of parallelism. The time required to fill each batch may prevent the source from

meeting the demand of the subsequent stages. There is a minimal increase in Fast-Flow throughput with batch size 2, but it is minimal and should be disregarded. We performed the same experiments with the other benchmarks. However, the results were similar and did not lead to different conclusions, so we omitted them in this paper. So we can conclude that despite the possible advantages of using micro-batch in multi-core (Sect. 2.2), we could not identify this in our experiments. In the future, we intend to test the SPBench in scenarios where batching is more impactful, such as distributed and heterogeneous systems.

5.4 Data frequency results

This section presents the data stream frequency experimental results. We use frequency patterns widely used in related work, such as *increasing*, *wave*, and *binary*. We also run the benchmarks with *spike* and *decreasing* patterns provided by SPBench (Sect. 3.2). We explain how the frequency patterns work in Sect. 2.3, and in Sect. 4 we discuss the parameters we used for these experiments.

The combinations of frequency patterns and applications we tested resulted in many data. Therefore, we present the most representative results for each frequency pattern. The graphs show the frequency set for the input stream as dashed lines. Since the frequency patterns were built based on data given by pre-runs of the benchmarks, the resulting frequency patterns for TBB and FastFlow may vary. Although the throughput performance of these interfaces is very similar, minor differences can accumulate and change the target frequency as the run progresses, and this is why the graphs show the set target frequency for each PPI. The top graph of each figure presents the instantaneous throughput, and the bottom graph shows the instantaneous latency. All the next experiments were performed on the computer with Intel Xeon Silver 4210 processors. It has more processing cores and achieved the highest throughputs in our experiments (Sect. 5.2). Although we have executed these experiments with multiple degrees of parallelism, we present the results with a maximum of 40 workers per farm, which is the number of available threads on the processor and best shows the impact of varying the frequency.

Figure 12 shows the latency and throughput of Bzip2, Lane Detection, and Ferret benchmarks running under a wave frequency pattern. In Bzip2 and Lane Detection, TBB benchmarks take a little longer to execute than FastFlow. Based on that, we can assume that the throughput with FastFlow is slightly higher on average since it processes the same number of items in a shorter time. On the other hand, at high frequencies, FastFlow could not sustain a low latency, presenting high latency spikes in Bzip2 and Lane Detection. If we look at Ferret's results in Fig. 12c, however, we can see that the difference in latency between TBB and FastFlow decreases. Also, Fast-Flow's execution time is longer in these cases, which implies lower throughput. It was expected that FastFlow would not perform as well as TBB in this case. After all, Ferret implements the pipeline-farm parallel pattern, and load unbalancing is a big issue for Ferret. So FastFlow is at a considerable disadvantage in these scenarios. On the other hand, in applications that do not require ordering, as with Ferret, as long as there is computational resource available, a TBB task can process an item from the
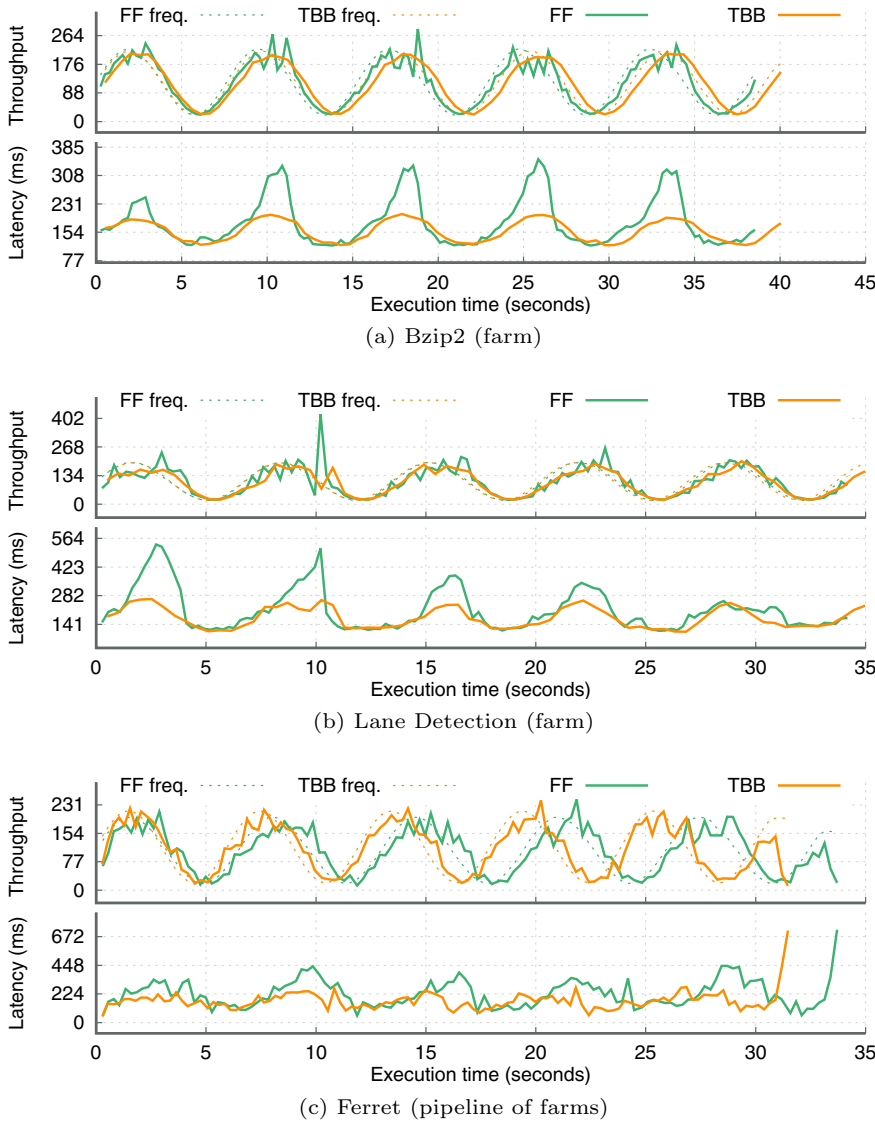
(a) Bzip2 (farm)

(b) Lane Detection (farm)

(c) Ferret (pipeline of farms)

Fig. 12 Wave frequency pattern with Bzip2, Lane Detection, and Ferret

beginning to the end of the pipeline without interruption. So this is a scenario where TBB has a significant advantage, as opposed to FastFlow.

Figure 13 presents the results for the binary pattern. For example, a binary pattern in a Lane Detection application can represent scenarios where there are sensor cameras with dynamic FPS, and the frame rate rapidly decreases or increases if the vehicle stops or starts moving. Comparing the results for Bzip2 with the wave pattern, it is possible to observe that the latency is higher at the maximum state in the binary
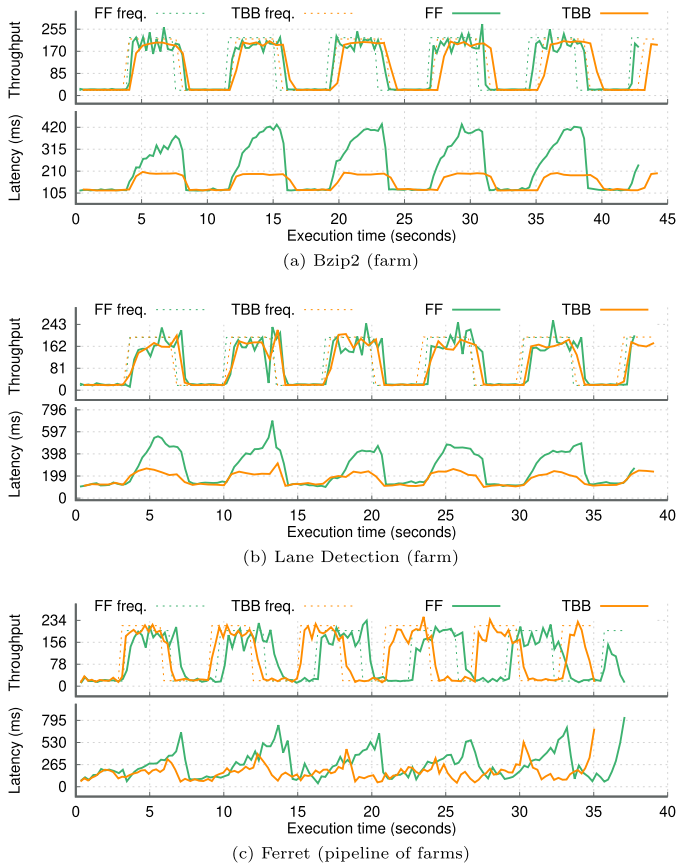
Fig. 13 Binary frequency pattern with Bzip2, Lane Detection, and Ferret

case. It is an expected difference since, in a sinusoidal wave, the frequency stays at a high state for a smaller time interval.

Lane Detection's latency results with wave and binary frequencies are a bit different. With the wave pattern, FastFlow reduces the latency in the latter part of the execution, even at high-frequency times. However, this behavior is not observed in the binary pattern. We hypothesize that this is related to the natural frequency of the workload in this application. In Fig. 6, we can see that the latency already shows a kind of wave pattern itself in the final part. So we assume that this wave pattern from the workload matches the wave frequency generated by the SPBench. It probably occurs in the binary pattern as well. After all, the latency peaks are also smaller in the final part of the execution, but the effects are less noticeable due to the sudden state changes. Besides the higher latency presented by FastFlow, in Lane Detection, its execution time was almost 5% lower than TBB. Regarding Ferret, with a binary frequency, it behaved similarly to the wave pattern, and the same considerations apply here.
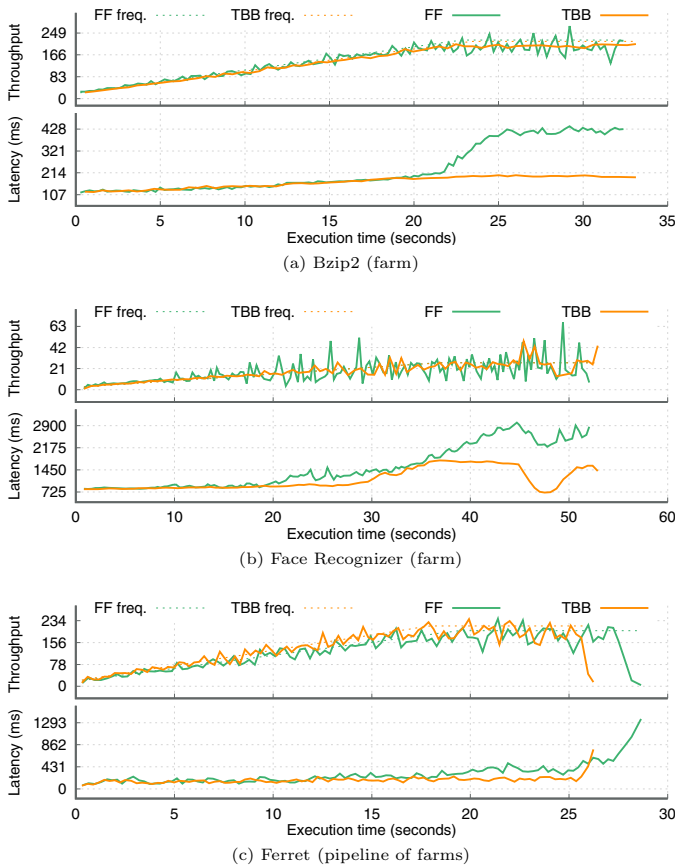
(a) Bzip2 (farm)

(b) Face Recognizer (farm)

(c) Ferret (pipeline of farms)

Fig. 14 Increasing frequency pattern with Bzip2, Face Recognizer, and Ferret

The results for the increasing frequency pattern are in Fig. 14. Here we evaluate the benchmarks for Bzip2, Face Recognizer, and Ferret applications. Bzip2 and Face Recognizer presented a distinct behavior. As previously discussed, the maximum target frequency in this experiment is 110% of the average throughput the application can sustain at maximum frequency. Since Bzip2 has a more stable input, presenting minor fluctuations along with the execution (see Sect. 5.1), once the target frequency reaches the threshold given by the sustainable frequency, the throughput stabilizes, and there is a latency jump with FastFlow (Fig. 14a). We believe that the main reason for this is that, in FastFlow, the source reads an item from the input stream even if there is no free worker to process it or free space in their queues [11]. So, this item remains enqueued waiting to be processed, increasing its processing time, thus the latency. In the TBB state-based pipeline, when an item (task) is assigned to a thread, this thread can run this item through all the stages of the pipeline until it is no more possible (when it encounters an ordered stage and the item is not in the correct order, for instance) [51]. However, cluttering is less present in

more stable workloads, mainly when items are more computational costly (Bzip's case). Therefore, this method can avoid queuing up items between stages, implying reduced latency. The steady throughput increase of TBB can confirm it.

On the other hand, the natural latency of the Face Recognizer workload is not as stable as that of Bzip2. Therefore, items are processed at different speeds by each worker. So as we increase the frequency, the item clutter gets worse. It dramatically impacts FastFlow, increasing latency before the frequency reaches the sustainable throughput level (Fig. 14b). However, it also impacts TBB, which shows a rapid increase in latency at the moment of maximum frequency. In this scenario, a single TBB thread cannot process an item from end to end of the pipeline. If the item arrives in a disordered state at the last stage, the thread puts that item in a buffer and takes another task to execute, increasing the latency for that item.

Regarding Ferret (Fig. 14c), FastFlow's latency is almost equal to TBB when the frequency is slightly below the maximum sustainable throughput. So, in scenarios like increasing, where most of the time the frequency is below this limit, FastFlow can keep the latency closer to TBB for longer. That is because, under low frequencies, there are more chances of having resources to process the items that the source takes from the input stream. With TBB and its task scheduler [51], in our experiments, there will be resources to keep processing further an item taken from the input stream most of the time. It contributes to keeping the latency low. However, like in the other frequency patterns, Ferret has this throughput drop and latency spike in the last few seconds, which occurs in both FastFlow and TBB. We do not precisely know what causes this behavior, but we suspect it is a consequence of load unbalancing.

Figure 15 shows the behavior of Face Recognizer, Lane Detection, and Ferret applications when run under the decreasing frequency pattern. Unlike the increasing pattern, at the end of the down-ramp, the decreasing pattern remains at the minimum frequency (10% of maximum sustainable throughput), so it is expected that the applications would take longer running after the ramp. It could be different if we set a more extended ramp period, but we prefer to use the same period we used in the increasing pattern for comparison purposes. Even though the throughput remains stable at the minimum, in Face Recognizer, there is still a low latency moment after 100 s that represents the same pattern seen in Fig. 6. Since the frequency manager only limits the maximum throughput and discounts the processing time of the item at the source from the added delay, it is expected that it will not affect the native low-latency moments of the workload.

In this decreasing pattern, the TBB Face Recognizer exhibits the opposite behavior of FastFlow in the increasing pattern with Bzip2 (Fig. 14a). Here the TBB can bring down the latency as soon as the frequency falls below sustainable throughput. Regarding FastFlow, Face Recognizer has a pretty steady throughput, and the hump present in the workload (Fig. 6) makes the average throughput of this application higher. Therefore, our frequency strategy overestimated a little the sustainable throughput of this application. According to the previous experiments, Fast-Flow latency seems more sensitive concerning sustainable throughput. This may be why it takes longer to keep up with the TBB latency. A similar effect occurs with Lane Detection, where the natural throughput of the workload at the beginning of
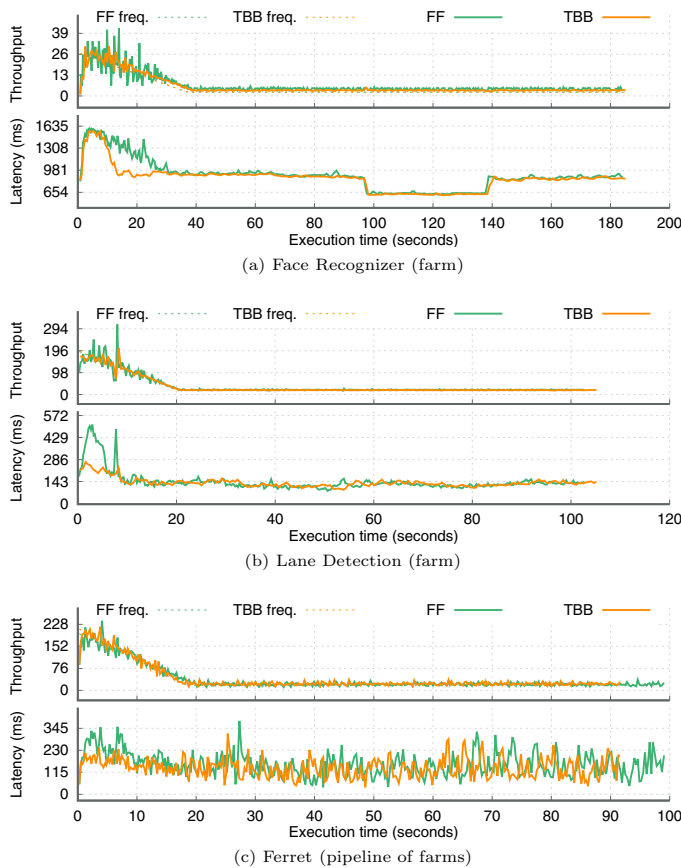
(a) Face Recognizer (farm)

(b) Lane Detection (farm)

(c) Ferret (pipeline of farms)

Fig. 15 Decreasing frequency pattern with Face Recognizer, Lane Detection, and Ferret

the execution is lower than the average. So it is another scenario of overestimated sustainable throughput, but here it occurs only during a short interval. This way, we believe this can contribute to FastFlow reducing latency to a minimum before the end of the decreasing ramp.

In Ferret, TBB and FastFlow showed similar behavior, although FastFlow has a longer execution time. In this application, the workload varies a lot and quite frequently. So overestimation of sustainable throughput is not a problem in this case. Regarding Ferret's large latency fluctuation (Fig. 15c), Fig. 6 shows that Ferret is the application that has the most extensive variation in latency caused by the input workload itself. This behavior has a direct impact on the results presented here.

Figure 16 displays the experimental results using the spike pattern. As expected, applications take longer to run with the spike frequency strategy. After all, in spike, the frequency is at a minimum most of the time. It explains the increase in the number of cycles that occur (one spike per cycle) compared to the other cyclic patterns. In Fig. 16a are the results of the Bzip2 benchmark. TBB and FastFlow showed
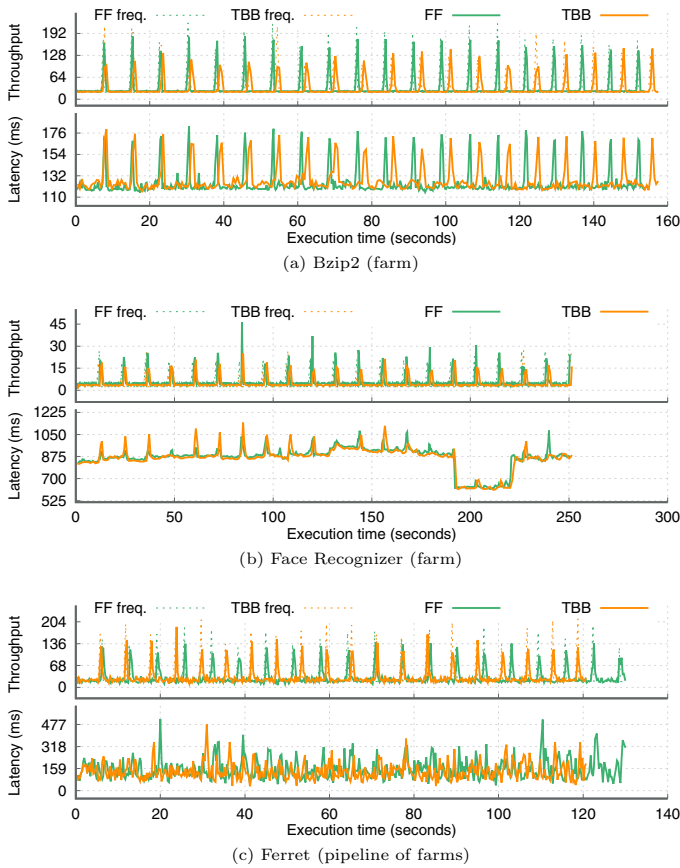
Fig. 16 Spike frequency pattern with Bzip2, Face Recognizer, and Ferret

similar values regarding throughput, latency, and execution time. It is noticeable from the results with the other frequency patterns that FastFlow increases latency over TBB when the frequency approaches the maximum. However, if we look at Fig. 13a, we can see that it takes a while after the frequency increase for the Fast-Flow's latency to rise if compared to TBB. This way, with the spike pattern, it does not happen in Bzip2, probably because of the short spike duration (about 0.8 s). Also, FastFlow has higher throughput spikes than TBB since it ran faster, as in most cases implementing the farm pattern. Although the latency spikes of the TBB are somewhat smaller, it is possible to observe some moments of higher latency in the off-spike intervals.

Face Recognizer has a less stable workload in terms of latency and throughput than Bzip2, as seen in Fig. 6. Therefore, in this application, the spikes in the graphics (Fig. 16b) appear more shapeless than in Bzip2. Moreover, its sequential throughput is almost ten times lower, just over 1 item per second in the sequential version in our experiments. In addition, the application takes longer to execute, implying more

extended periods and, consequently, slightly longer spikes. Although the Face Recognizer behaves similarly to Bzip2 in throughput, the latency results change quite a bit. This application's native latency changes a lot, and many items are processed during spikes. This way, a slight shift in execution time causes the TBB and Fast-Flow to end up processing distinct parts of the input stream during the spikes. Thus, its natural latency may mix with the latency increase caused by the spike pattern and cause large latency spikes, sometimes with TBB and others with FastFlow. A similar effect occurs with Ferret (Fig. 16c), where spikes occur more erratically because of the input stream's significant variations in throughput and latency.

6 Results discussion and final remarks

In this section, we summarize the results and add some general remarks. First, we have improved and updated the workload characterization in this work (Sect. 5.1). It was necessary because SPBench added a new and larger class of workloads called "huge," and we ran the benchmarks with this new class in this work. Also, the SPBench, in its latest version, added instantaneous latency and throughput metrics, which are more representative metrics for monitoring the performance of stream processing applications. Therefore, we use these new metrics in the experiments. We also benchmark performance on different architectures to serve as a baseline and to assess performance portability. TBB and FastFlow showed similar throughput, although FastFlow has a higher latency. The blocking mode may have caused part of the higher latency of FastFlow. However, this configuration, plus the overprovisioning strategy, allowed higher throughputs.

One of the goals of this work was to evaluate how micro-batches use impacts the performance of stream processing applications in multi-core environments. The same had been attempted in previous work [14]. However, we found some flaws in performance measurement that invalidated those results. The flaws have been fixed, and we repeat the experiments in this paper. Several possible advantages could result in some performance gain from using micro-batches, as discussed in Sect. 2.2. We hypothesized that at least high throughput applications with ordering constraints could benefit from batching. However, in our experiments, micro-batching did not show any performance advantage. Latency increases as expected, but any increase in throughput is negligible.

For the last, in Sect. 5.4, we evaluated the impact of data frequency on the benchmarks. FastFlow showed more erratic behavior in all cases than TBB under high frequencies. We believe that the differences in the implementation of the buffers/queues may cause this behavior. In FastFlow, even using queues of size one (on-demand policy), each worker has its input and output queue. So even with short queues, many items are in the pipeline throughout the execution. Each 40-worker farm can hold up to 80 items in FastFlow (40 queued plus 40 with the workers). Therefore, those items already in the pipeline are no longer under frequency constraint, and performance fluctuates more. In TBB, this could also occur in specific highly cluttered input scenarios. However, our micro-batch experiments in Sect. 5.3 have shown that TBB tends to generate less item clutter in the stream.

On the other hand, FastFlow showed slightly higher throughput in most farm-parallel benchmarks. Some factors may prevent TBB from running faster than Fast-Flow in most farm implementations we tested. The benefits of TBB's work-stealing scheduler add some costs. Whenever a thread operates on an item, it creates a new object for the next stage, including its instance variables. Instantiating objects in a multi-thread environment can be slow and cause contention for the heap and the memory allocator data structures, inhibiting concurrency [52]. Also, we used the same value to set the number of replicas of FastFlow workers and the maximum degree of parallelism in the TBB parallel stages. However, in TBB's task model, this degree of parallelism means the maximum number of threads that will execute. The TBB "workers" need to share the work with the source (emitter) and sink (collector) stages. In FastFlow, the workers each run in their dedicated thread. Therefore, the number of threads used in FastFlow is always greater than the number of replicas because it creates one more thread for the source and one more for the sink. Still, these two stages in our benchmarks consume fewer resources than the worker stages. So, if running FastFlow in blocking mode (the case in this paper), these two extra threads that FastFlow creates may give it a short performance advantage over TBB in the parallel farm pattern.

FastFlow's performance is inferior to TBB in the Ferret application with varying data stream frequencies. However, in FastFlow, we could exploit pipeline and stream parallelism in several other ways. For example, with FastFlow on Ferret, we could implement combined nodes or a farm of pipelines, resulting in better performance. But we preferred to keep the original Ferret structure which is a pipeline of farms. The other benchmarks we used already address a similar structure to the combined nodes in FastFlow. Also, reducing the degree of parallelism of the less intensive stages would use fewer resources, which could be allocated to the more intensive ones. Fewer queues would be available to hold items, implying lower latency in this case. However, optimal configurations in stream processing applications take work to achieve. It depends on the type of application, the workload, the architecture, and the available resources, among other factors [8]. Also, the optimizations can benefit the throughput or latency more, as shown in previous work [13]. On the other hand, programmers have more freedom to fine-tune FastFlow applications compared to TBB. Therefore, both TBB and FastFlow have advantages and disadvantages in different aspects, from performance to the possibility of optimization, fine-tuning, and parallelism modeling.

7 Conclusions and future work

In this work, we extended the SPBench benchmarking framework to support micro-batching and data stream frequency. Additionally, we implemented a set of algorithms that allow the creation of specific patterns in the input stream frequency. The experimental results showed that our algorithms successfully generated the most representative frequency patterns used in the literature. With the help of the extended framework, we analyzed the impact of micro-batch and data intensity on stream processing applications with different PPIs. We were able

to create several workloads with some strategies that could change dynamically at execution time. We also tested micro-batching configurations under different levels of parallelism. Micro-batching has the potential to improve performance in multi-cores by enabling software pipelining, vectorization, and amortizing costs such as operator-firing, communication, data sorting, warm-up (e.g., for the instruction cache), data scheduling, and others. However, the experiments showed no benefit from micro-batches on the performance of the benchmarks we used. Latency increased as expected, but it had no positive impact on throughput.

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.

Our work did not address some aspects of the experiments. We did not explore micro-batch size by time window or micro-batching under different data frequencies, although all these are features now supported by SPBench. We also did not evaluate FastFlow with other parallelism strategies in the Ferret benchmark. As feature ideas for the SPBench in the future, it might be possible to add support so that any non-SPBench application can use the stream frequency manager. In theory, the instantiation of the SPBench library could be by adding a single line of code to the source of such an application. It would also be interesting to have a mechanism for stressing item clutter. We aim to address such aspects in future work.

References

1. Das T, Zhong Y, Stoica I, Shenker S (2014) Adaptive stream processing using dynamic batch sizing. In: Proceedings of the ACM Symposium on Cloud Computing. SOCC'14. Association for Computing Machinery, New York, NY, USA, pp 1–13

2. Stein CM, Rockenbach DA, Griebler D, Torquati M, Mencagli G, Danelutto M, Fernandes LG (2020) Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units. Concurr Comput Pract Exp. https://doi.org/10.1002/cpe.5786

3. Hirzel M, Soulé R, Schneider S, Gedik B, Grimm R (2014) A catalog of stream processing optimizations. ACM Comput Surv. https://doi.org/10.1145/2528412

4. Herodotou H, Odysseos L, Lu J (2022) Automatic performance tuning for distributed data stream processing systems. In: 38TH IEEE International Conference on Data Engineering

5. Zhang Q, Song Y, Routray RR, Shi W (2016) Adaptive block and batch sizing for batched stream processing system. In: 2016 IEEE International Conference on Autonomic Computing (ICAC), pp 35–44. https://doi.org/10.1109/ICAC.2016.27

6. Abdelhamid AS, Mahmood AR, Daghistani A, Aref WG (2020) Prompt: dynamic data-partitioning for distributed micro-batch stream processing systems. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. SIGMOD'20. ACM, New York, pp 2455–2469

7. Henning S, Hasselbring W (2021) Theodolite: scalability benchmarking of distributed stream processing engines in microservice architectures. Big Data Res 25:100209

8. Li W, Zhang Z, Shu Y, Liu H, Liu T (2022) Toward optimal operator parallelism for stream processing topology with limited buffers. J Supercomput 1–22

9. Vogel A, Griebler D, Danelutto M, Fernandes LG (2022) Self-adaptation on parallel stream processing: a systematic review. Concurr Comput Pract Exp 34(6):6759. https://doi.org/10.1002/cpe.6759

10. Voss M, Asenjo R, Reinders J (2019) Pro TBB: C++ parallel programming with threading building blocks. Apress, New York

11. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M (2017) Fastflow: high-level and efficient streaming on multicore, vol Chap. 13. John Wiley & Sons Ltd, Hoboken, pp 261–280

12. Garcia AM, Griebler D, Schepke C, Fernandes L.G (2021) Introducing a stream processing framework for assessing parallel programming interfaces. In: 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). PDP'21. IEEE, Valladolid, pp 84–88. https://doi.org/10.1109/PDP52278.2021.00021

13. Garcia AM, Griebler D, Schepke C, Fernandes LG (2022) SPBench: a framework for creating benchmarks of stream processing applications. Computing. https://doi.org/10.1007/s00607-021-01025-6

14. Garcia AM, Griebler D, Schepke C, Fernandes L.G (2022) Evaluating micro-batch and data frequency for stream processing applications on multi-cores. In: 30th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). PDP'22. IEEE, Valladolid, pp 10–17. https://doi.org/10.1109/PDP55904.2022.00011

15. Yao F, Wu J, Venkataramani G, Subramaniam S (2019) Ts-batpro: Improving energy efficiency in data centers by leveraging temporal-spatial batching. IEEE Trans Green Commun Netw 3(1):236–249. https://doi.org/10.1109/TGCN.2018.2871025

16. De Sensi D, Torquati M, Danelutto M (2016) A reconfiguration algorithm for power-aware parallel applications. ACM Trans Archit Code Optim. https://doi.org/10.1145/3004054

17. Bienia C, Kumar S, Singh J.P, Li K (2008) The parsec benchmark suite: characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp 72–81

18. Hesse G, Matthies C, Perscheid M, Uflacker M, Plattner H (2021) Espbench: the enterprise stream processing benchmark. In: ICPE '21: Proceedings of the ACM/SPEC International Conference on Performance Engineering. ICPE '21. Association for Computing Machinery, New York, pp 201–212. https://doi.org/10.1145/3427921.3450242

19. Shukla A, Chaturvedi S, Simmhan Y (2017) Riotbench: an IoT benchmark for distributed stream processing systems. Concurr Comput Pract Exp 29(21):4257

20. van Dongen G, Van den Poel D (2020) Evaluation of stream processing frameworks. IEEE Trans Parallel Distrib Syst 31(8):1845–1858

21. Wang L, Fu TZJ, Ma RTB, Winslett M, Zhang Z (2019) Elasticutor: rapid elasticity for realtime stateful stream processing. In: Proceedings of the 2019 International Conference on Management of Data. SIGMOD '19. Association for Computing Machinery, Amsterdam, pp 573–588

22. Le-Phuoc D, Dao-Tran M, Pham M-D, Boncz P, Eiter T, Fink M (2012) Linked stream data processing engines: facts and figures. In: The Semantic Web—ISWC 2012. Springer, Berlin, pp 300–312

23. Karimov J, Rabl T, Katsifodimos A, Samarev R, Heiskanen H, Markl V (2018) Benchmarking distributed stream data processing systems. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp 1507–1518

24. Lobato AGP, Andreoni Lopez M, Cardenas AA, Duarte OCMB, Pujolle G (2022) A fast and accurate threat detection and prevention architecture using stream processing. Concurr Comput Pract Exp 34(3):6561. https://doi.org/10.1002/cpe.6561

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

26. Balkesen C, Tatbul N, Özsu MT (2013) Adaptive input admission and management for parallel stream processing. In: Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems. DEBS '13. Association for Computing Machinery, Arlington, pp 15–26

27. De Matteis T, Mencagli G (2016) Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing. SIGPLAN Not. https://doi.org/10.1145/3016078.2851148

28. Navarro A, Asenjo R, Tabik S, Cascaval C (2009) Analytical modeling of pipeline parallelism. In: 2009 18th International Conference on Parallel Architectures and Compilation Techniques, pp 281–290. https://doi.org/10.1109/PACT.2009.28

29. Bebortta S, Dalabehera AR, Pati B, Panigrahi CR, Nanda GR, Sahu B, Senapati D (2022) An intelligent spatial stream processing framework for digital forensics amid the COVID-19 outbreak. Smart Health 26:100308. https://doi.org/10.1016/j.smhl.2022.100308

30. Xu J, Palanisamy B, Wang Q, Ludwig H, Gopisetty S (2022) Amnis: optimized stream processing for edge computing. J Parallel Distrib Comput 160:49–64. https://doi.org/10.1016/j.jpdc.2021.10.001

31. Ntumba P, Georgantas N, Christophides V (2022) Scheduling continuous operators for IoT edge analytics with time constraints. In: 2022 IEEE International Conference on Smart Computing (SMARTCOMP), pp 78–85. https://doi.org/10.1109/SMARTCOMP55677.2022.00026

32. Thies W, Amarasinghe S (2010) 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), pp 365–376

33. Welsh M, Culler D, Brewer E (2001) Seda: an architecture for well-conditioned, scalable internet services. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles. SOSP'01. ACM, New York, pp 230–243

34. Carney D, Çetintemel U, Rasin A, Zdonik S, Cherniack M, Stonebraker M (2003) Operator scheduling in a data stream manager. In: Freytag J-C, Lockemann P, Abiteboul S, Carey M, Selinger P, Heuer A (eds) Proceedings 2003 VLDB Conference. Morgan Kaufmann, San Francisco, pp 838–849

35. Imai S, Patterson S, Varela CA (2017) Maximum sustainable throughput prediction for data stream processing over public clouds. In: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp 504–513. https://doi.org/10.1109/CCGRID.2017.105

36. Arkian H, Pierre G, Tordsson J, Elmroth E (2021) Model-based stream processing auto-scaling in geo-distributed environments. In: 2021 International Conference on Computer Communications and Networks (ICCCN), pp 1–10. https://doi.org/10.1109/ICCCN52240.2021.9522236

37. Imai S, Patterson S, Varela C.A.(2018) Uncertainty-aware elastic virtual machine scheduling for stream processing systems. In: 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp 62–71. https://doi.org/10.1109/CCGRID.2018.00021

38. Chu Z, Yu J, Hamdulla A (2021) Throughput prediction based on extratree for stream processing tasks. Comput Sci Inf Syst 18(1):1–22

39. Palyvos-Giannas D, Mencagli G, Papatriantafilou M, Gulisano V (2021) Lachesis: a middleware for customizing OS scheduling of stream processing queries. In: Proceedings of the 22nd International

Middleware Conference. Middleware'21. Association for Computing Machinery, New York, pp 365–378. https://doi.org/10.1145/3464298.3493407

40. Sun D, Cui Y, Wu M, Gao S, Buyya R (2022) An energy efficient and runtime-aware framework for distributed stream computing systems. Futur Gener Comput Syst. https://doi.org/10.1016/j.future.2022.06.007

41. Gedik B, Schneider S, Hirzel M, Wu K-L (2014) Elastic scaling for data stream processing. IEEE Trans Parallel Distrib Syst 25(6):1447–1463. https://doi.org/10.1109/TPDS.2013.295

42. Russo G, Nardelli M, Cardellini V, Lo Presti F (2018) Multi-level elasticity for wide-area data streaming systems: a reinforcement learning approach. Algorithms. https://doi.org/10.3390/a11090134

43. Heinze T, Jerzak Z, Hackenbroich G, Fetzer C (2014) Latency-aware elastic scaling for distributed data stream processing systems. In: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems. DEBS'14. Association for Computing Machinery, New York, pp 13–22. https://doi.org/10.1145/2611286.2611294

44. Mei Y, Cheng L, Talwar V, Levin MY, Jacques-Silva G, Simha N, Banerjee A, Smith B, Williamson T, Yilmaz S, Chen W, Chen GJ (2020) Turbine: Facebook's service management platform for stream processing. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp 1591–1602. https://doi.org/10.1109/ICDE48307.2020.00141

45. Henning S, Hasselbring W (2021) How to measure scalability of distributed stream processing engines? In: Companion of the ACM/SPEC International Conference on Performance Engineering. ICPE'21. ACM, New York, pp 85–88

46. Griebler D, Danelutto M, Torquati M, Fernandes LG (2017) SPar: a DSL for high-level and productive stream parallelism. Parallel Process Lett 27(01):1740005. https://doi.org/10.1142/S0129626417400059

47. Danelutto M, De Matteis T, Mencagli G, Torquati M (2018) Data stream processing via code annotations. J Supercomput 74(11):5659–5673

48. Griebler D, Vogel A, De Sensi D, Danelutto M, Fernandes LG (2019) Simplifying and implementing service level objectives for stream parallelism. J Supercomput 76:4603–4628. https://doi.org/10.1007/s11227-019-02914-6

49. Fleisch D, Kinnaman L (2015) A student's guide to waves. Student's guides. Cambridge University Press, Cambridge

50. Griebler D, Hoffmann RB, Danelutto M, Fernandes LG (2018) Stream parallelism with ordered data constraints on multi-core systems. J Supercomput 75(8):4042–4061. https://doi.org/10.1007/s11227-018-2482-7

51. MacDonald S, Szafron D, Schaeffer J (2004) Rethinking the pipeline as object-oriented states with transformations. In: Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings, pp 12–21. https://doi.org/10.1109/HIPS.2004.1299186

52. Reinders J (2007) Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media Inc, Sebastopol

Authors and Affiliations

Adriano Marques Garcia[1] · Dalvan Griebler[1] · Claudio Schepke[2] · Luiz Gustavo Fernandes[1]

Dalvan Griebler
dalvan.griebler@pucrs.br

Claudio Schepke
claudioschepke@unipampa.edu.br

Luiz Gustavo Fernandes
luiz.fernandes@pucrs.br

[1]   School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Av. Ipiranga 6681 Partenon, Porto Alegre, RS 90619-900, Brazil

[2]   Laboratory of Advances Studies in Computation (LEA), Federal University of Pampa (UNIPAMPA), Av. Tiarajú 810, Alegrete, RS 97546-550, Brazil