Computing
https://doi.org/10.1007/s00607-021-01025-6

SPECIAL ISSUE ARTICLE

SPBench: a framework for creating benchmarks of stream processing applications

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

Received: 15 May 2021 / Accepted: 22 October 2021

Abstract
In a fast-changing data-driven world, real-time data processing systems are becoming ubiquitous in everyday applications. The increasing data we produce, such as audio, video, image, and, text are demanding quickly and efficiently computation. Stream Parallelism allows accelerating this computation for real-time processing. But it is still a challenging task and most reserved for experts. In this paper, we present SPBENCH, a framework for benchmarking stream processing applications. It aims to support users with a set of real-world stream processing applications, which are made accessible through an Application Programming Interface (API) and executable via Command Line Interface (CLI) to create custom benchmarks. We tested SPBENCH by implementing parallel benchmarks with Intel Threading Building Blocks (TBB), FASTFLOW, and SPAR. This evaluation provided useful insights and revealed the feasibility of the proposed framework in terms of usage, customization, and performance analysis. SPBENCH demonstrated to be a high-level, reusable, extensible, and easy of use abstraction to build parallel stream processing benchmarks on multi-core architectures.

Keywords Parallel computing · Stream parallelism · Performance analysis · Computing workloads · Parallel programming

Mathematics Subject Classification 68W27 · 68U01 · 68M20

1 Introduction

The parallelism in stream processing is exploited explicitly through the implementation of operators or stages in a sequence that can be view as a directed acyclic

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

1 School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil

2 Federal University of Pampa (UNIPAMPA), Bagé, Brazil

Published online: 10 January 2022

graph [1]. These graphs can present different patterns of parallelism such as pipeline, data, and task parallelism [2]. Moreover, different Parallel Programming Interfaces (PPIs) such as libraries, frameworks, and Domain-Specific Languages (DSLs) are adopted to express and exploit stream parallelism. Some target distributed systems and Java-based languages, for instance: Apache Flink [3], Apache Storm [4], and Apache Spark [5]. These distributed platforms rely on the Java Virtual Machine (JVM) to abstract the underlying hardware and streamline the development process of such commercial platforms. Unfortunately, JVM cannot provide efficient data access due to processing overheads. The overhead is induced by data serialization (and deserialization), objects scattering in main memory, virtual functions, and garbage collection [6]. Such disadvantages, allied to the advancements in multi-core technologies, turn PPIs targeting shared-memory systems and C++, such as Intel TBB [7], FastFlow [8], SPar [9], GrPPI [10], and WindFlow [11], quite interesting for adoption and investigation.

New PPIs and related solutions are constantly evaluated in the stream processing domain. Some researches focus on evaluating PPIs [11–14] or developing techniques to improve different aspects of them, such as self-adaptive parallelism [15], performance optimizations [16], add new features [17], and support for new parallelism abstractions [13,18] and architectures [14,19,20]. There are some recent initiatives to build stream processing benchmark suites [21–26] that could be useful for such researchers to evaluate their work, and evaluate and compare different PPIs. However, none of them provide a suitable programming abstraction to create custom benchmarks for evaluating new PPIs, parallelism strategies, platforms, architectures, and others. Therefore, researchers end up having to develop their specific-purpose solutions that require a huge effort or use outdated benchmarks [8,11,27–29].

In this paper, we first present SPBench[1], a framework for benchmarking stream processing applications over a variety of options. It continues our previous research work published at [30], which was preliminary and lacks several conceptual ideas of the proposed framework regarding the API (new application) and evaluation (newer computer architecture and other benchmark versions). SPBench aims to provide a set of real-world stream processing applications for the C++ community in a higher-level, reusable, and cleaner abstraction so that users can fast or with minimal effort build custom benchmarks for evaluating different approaches or technologies. SPBench offers an API to program stream processing applications in a standardized way, which automatically supports different performance metrics that can be evaluated across multiple layers. It also envisions automatically support different workload options such as data sources, problem sizes, and others.

One of the main features of SPBench is the way the user interacts with the benchmarks. Figure 1 illustrates how this interaction occurs and how it differs from others. This interaction is usually direct and manual in other benchmarks. Users select an application, compile, and run it with a given workload and performance metric. To add a new parallel implementation, users need to do the integration manually as well. Most parallel benchmark suites do not even provide a sequential version of the parallel benchmarks for users to start a new version.

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

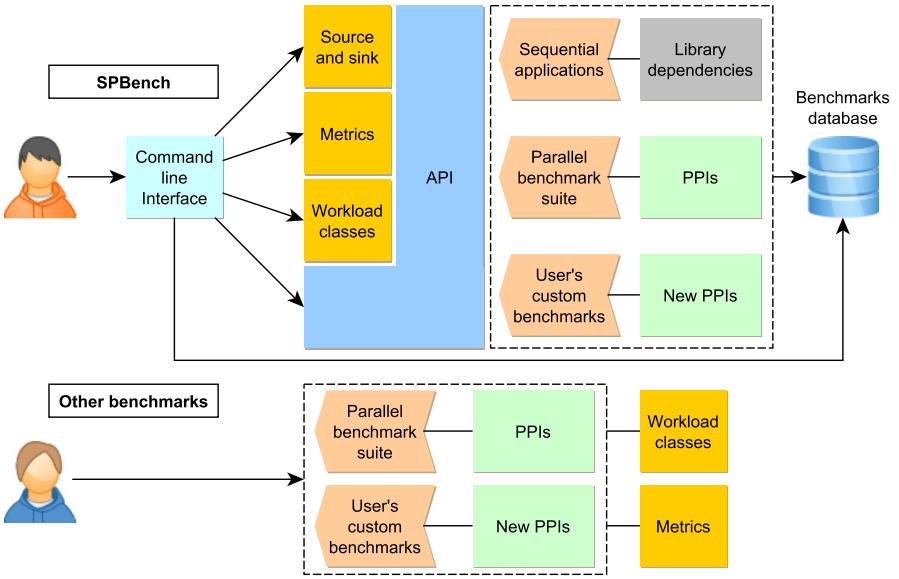


Fig. 1 SPBENCH methodology versus other benchmarks

In SPBENCH, the interaction occurs through a command-line interface (CLI). SPBENCH maintains a database of the suite of benchmarks. With CLI, users can access these benchmarks, modify them, and even add new custom versions. The CLI allows new versions to be easily integrated, reimplemented, and configured. It also manages the compilation and execution process of the benchmarks, allowing users to select different data sources, workloads, and performance metrics. Our scientific contributions are twofold:

- A framework to create benchmarks of stream processing applications. It provides an API and CLI to simplify, reuse code, customize, extend, and evaluate different aspects or properties regarding parallel programming and architectures.
- A performance benchmark of Intel TBB [7], FASTFLOW [8], and SPAR [9] on stream processing by using SPBENCH.

The paper organization is as follows. Section 2 discusses related benchmark suites and frameworks. Section 3 presents the SPBENCH and all its features, including the API, the current applications set, and the CLI. Section 4 presents a use case of SPBENCH. We use the framework to characterize the applications, build custom parallel benchmarks, and evaluate the performance of some PPIs. Finally, Sect. 5 draws our conclusions and discusses possible future works and challenges that come along with them.

2 Related work

First, we searched for benchmark suites of stream processing or benchmark suites that include at least some stream processing applications. We found the following benchmark suites: RIoTBench [23], StreamBench [24], StreamBench [25], SparkBench [26], DSPBench [21], StreamIt [31], and PARSEC [32].

Most of the stream processing benchmark suites focus only on data stream applications [21,23–26]. These are applications that intersect the domains of Big Data and IoT, which are developed using frameworks for distributed platforms. All of them are implemented in JVM languages. The remaining benchmark suites [31,32] include some stream processing applications, but their benchmarks have limitations in terms of programming language, parallelism exploitation, execution metrics, and parametric options [27,33]. The benchmark suite provided by SPBench focuses on the C++ community and aims to include any type of stream application. It also includes most of the representative metrics found in related work [21,26].

Past work already discussed the related work regarding benchmark suites in detail [30]. Therefore, this paper focuses the discussion on related frameworks instead. These are frameworks that ease somehow the parallelism or the use of stream applications. NAMB (Not only A Micro-Benchmark) is a platform for the generation of prototype applications based on their high-level description [22]. It consists of a framework based on fundamental data stream characteristics that supports a configurable topology description. It aims is to avoid the user having to edit the application code. It can generate a set of synthetic/micro-benchmarks as well as prototypes of Java applications for Apache Flink, Storm, and Heron platforms. The framework also allows users to change input data frequency, parallelism degree, tuple size, etc.

Theodolite is a framework for evaluating the scalability of distributed platforms [34]. This framework comprises a seven-dimension workload generator for microservice architectures. The middleware is an Apache Kafka system that allows changing the characteristics of the stream, such as item frequency, and add multiple sources. It includes four benchmarks with Apache Flink and Apache Kafka Streams.

SpinStreams [16] is a framework for predicting the performance of a given stream application and statically restructure its data flow topology to improve performance. It tries to apply operations such as join and fission to stages to correct backpressure and bottleneck problems in specific way. It supports Java applications and generates parallelism for the Akka Streams engine.

Similar to [34], there are several others focused on generating workloads to test different aspects of distributed PPIs. In [35] their framework runs in a separated system and generates realistic workloads to evaluate latency and throughput of DSPSs, supporting stateful performance analysis. [36] provides network security monitoring for data streams to test the ability of the most common distributed PPIs to process this type of data. [37] proposes a framework to generate representative workloads for social media applications to evaluate DSPSs.

Table 1 summarizes and compares the main features of each related framework, including SPBench in the last row. Although all of these frameworks are designed for distributed platforms, they provide workloads only for domain-specific applications, as shown in the second column of the table. SPBench targets any stream processing.

Table 1 Comparison of main characteristics of related work

R.W.	Target applications	Platform	Stream modifiers	Supported metrics	Goal	PPI extensibility
[22]	Data stream	Distributed	–	Data frequency* Tuple size	Application generation Automatic parallelism Synthetic data generation	Diffilcult
[16]	Data stream	Distributed	Data frequency	Throughput	Automatic parallelism Performance optimization Performance analysis	Diffilcult
[34]	Data stream (microservices)	Distributed	Data frequency Multiple sources Mixed batches	Scalabillity	Synthetic benchmark suite Synthetic data generation Performance analysis	Not easy
[37]	Data stream (linked data)	Multicore	Data frequency Dataset size Multiple sources	Throughput	Input data generation Performance analysis	Not easy
[36]	NetFlow processing	Distributed	Multiple sources	Throughput	Micro-benchmark suite Performance analysis	Not easy
[35]	Data stream (social media)	Distributed	Data frequency Multiple sources	Throughput Event-time latency	Synthetic data generation Performance analysis	Easy
SPBENCH	General Stream Processing	Multicore	In-memory processing	Throughput CPU/Mem. usage Latency: Processing-time Per-operator	Benchmark suite Benchmark generation Performance analysis	Easy

* It can not set an inter-tuple interval under 1ms due to Java constraints

In addition, most of these frameworks focus on distributed platforms. SPBENCH tries to meet the growing trend of stream processing on multicore systems, but we also look at supporting distributed architectures in the future.

In addition, most of these frameworks focus on distributed platforms. SPBENCH tries to meet the growing trend of stream processing on multicore systems, but we also look at supporting distributed architectures in the future. In addition, none of the related frameworks we found support C++, all of them target JVM languages.

The fourth column of the Table 1 lists any modifications to the input stream that the frameworks support. The most common modification allows the user to change the data arrival frequency, use multiple data sources, or select tuple/batch/dataset size. For now, SPBENCH only supports in-memory processing as a stream modifier. However, data frequency (microsecond precision), multiple sources, and batch size control will be included in the following stages of implementation.

Regarding metrics (fifth column), throughput is the most common. [34] is the only framework that supports a specific composite metric called *scalability*. Latency can be evaluated in at least three dimensions: event-time, processing-time, and per-operator [35]. SPBENCH can evaluate latency in these three dimensions. In addition, SPBENCH also evaluates memory and CPU consumption. Our framework can evaluate and monitor these metrics in different **time dimensions** as well: end-to-end (global average), per item, and per custom time intervals. Hence, SPBENCH allows users to evaluate benchmarks comprehensively at different layers.

The second-to-last column of Table 1 tries to summarize the goals of each framework. Most of them include some small benchmark suite with performance evaluation. Other goals revolve around the generation of applications/benchmarks, input data, or parallel code. However, this code/data generation part is usually tied to specific PPIs, which limits the extensibility of them. Frameworks marked as difficult to extend (in the last column) are those that require a lot of programming effort to support a new PPI. That being a task most reserved for the framework's developers. By "Not easy" to extend we mean that the framework's code still requires some modification, but experienced users can do this. And easy to extend means that no change is required on the framework's source code to add an implementation with a new PPI. SPBENCH is in this category, where the user only needs to write the parallelism and describe the building dependencies.

3 SPBench

In [29], the author proposed SPAR [9] for expressing high-level stream parallelism. The "high-level" term, in that context, is for parallelism abstractions that prevent the user from the deal with details related to parallel architecture optimizations, avoid code rewriting, and reduce the programming effort to support parallelism. This PPI requires the programmer to only annotate the parallelism without having to rewrite the original code. Programmers basically must identify the operators and their respective data dependencies. Hence, SPAR is one of the PPIs for stream parallelism that offers the highest level of parallelism abstractions for programmers.

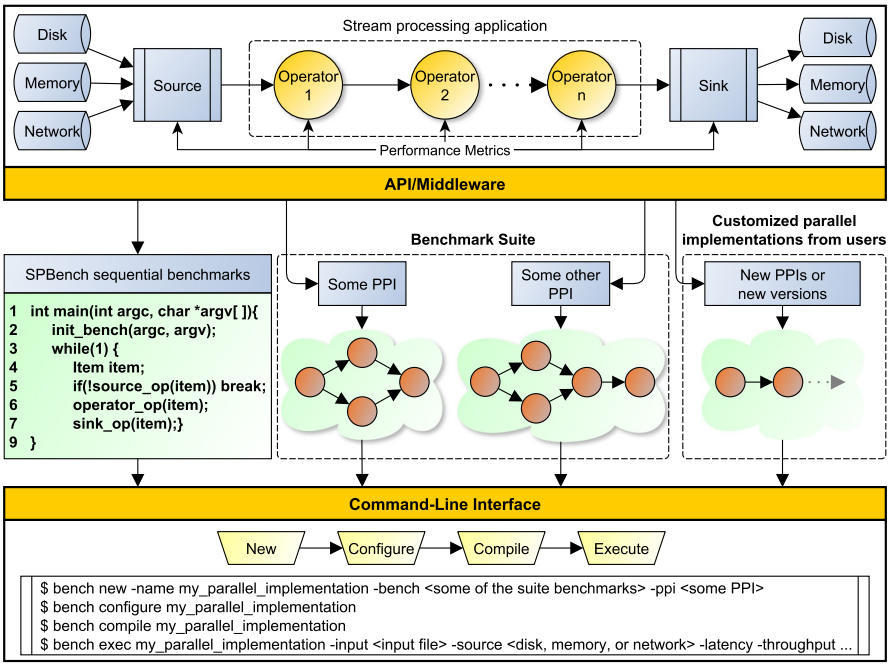


Fig. 2 SPBENCH framework

There are stream processing applications with thousands of lines of code, where it can be still very difficult to identify the beginning and end of each operator, and also identify all the data dependencies across them. Therefore, exploring stream parallelism can be difficult even using high-level abstractions and PPIs that provide structured parallel patterns such as FastFlow [8] and TBB [7].

Stream parallelism is still a challenging task, even disregarding the complexity of writing parallel code. The goal of the SPBENCH is to ease for programmers to write their parallel code and create a specific custom version for benchmarking stream processing. The structure of the framework is illustrated in Fig. 2. The framework has three main parts: the API (on top), the applications/benchmarks area (middle), and the command-line interface (bottom).

3.1 SPBench API

The kernel of SPBENCH is an API that offers three main advantages to users: (1) it allows the implementation of stream processing applications in a modular, reconfigurable, and standardized way; (2) it automatically and transparently adds all the main benchmarking metrics used in this domain, which can evaluate the benchmarks in different layers and time dimensions; (3) and it can offer various workload customization, such as data input rate, batch size, different and multiple sources, etc.

To build this API, we disassemble all operators from the original application and put them individually into a new source code. This way, the application calls the operators

by including a C++ header file with their implementations. Beside operators, the API also encapsulates the data communicated among them and manages the input/output streams (sources and sinks). It must also handle the management of command-line arguments, variables initialization, metrics, and some other application requirement. Therefore, an application inside the framework integrates all of this and makes it totally transparent for the user.

In the SPBENCH API, all operators of an application are encapsulated individually, from source to sink. For this, we first optimized and restructured the application code so that the different operators are highlighted and data dependencies identified. Then, we rewrite them as individual functions, containing the sequential code of the respective operator. Here, code for time measurements are added at the beginning and end of the operator for later performance computations. Operators can be accessed by functions calls, for instance: `OperatorName_op()`.

The API also encapsulates the dependent data into a class called `Item`. Although this is standardized for all applications, the data within this item may vary for each case. Therefore, each application have its own type of data item with a variable structure, which must load all the data communicated across the operators. These items are also optimized to avoid over-sizing. In the application implemented with the SPBENCH API, these items must be used as argument when calling the operators, such as `OperatorName_op(item)`.

3.1.1 Input data management

Besides the internal operators, each application must have at least a `source` and a `sink` operator. The source operator receives the task for initializing the data item with the data received from external sources. Sink works similarly, but in reverse mode. The framework must support three alternatives for external source or sink: disk, memory, or network. In the "disk" option, the application receives a pointer to a file on disk and the source operator reads micro-batches of it, which can be a video frame, a set of frames, a block of bytes, etc. The items are processed and then received by the sink, which builds the resulting file on disk.

The "memory" option consists of an in-memory execution. Specialized in-memory stream processing architectures are emerging ([38,39]) and our framework must also address such architectures. To do so, the input file is first loaded into memory. Then, the source operator receives only the memory address of the data and makes the assignment of the memory blocks to each item.

The third external source/sink option planned is "network". This option is important to represent some real-world scenarios. For instance, a Person Recognition application could receive video frames from a monitoring camera over the network. This option is the most complex functionality to implement in SPBENCH and we are still working on it. This system must run independently, simulating real external data stream sources. So, it has to run in a parallel thread alongside the application and use communication protocols.

To simulate realistic scenarios, it is necessary to induce other different behaviors. SPBENCH has to allow users to vary the frequency of arriving items and batch size anytime during the execution. In addition, it must support input streams from multiple

sources. All of these features need to be generic enough to support all applications workloads (when required) and be easily configurable. SPBench will also allow users to register custom workload classes.

3.2 SPBench benchmarks

In the middle part of Fig. 2 (under the API region) is the benchmark applications area. SPBench have three distinct sets of applications: default sequential applications (left), applications from the benchmark suite (center), and custom benchmarks added by users (right).

The piece of code in Fig. 2 (left) is an implementation example of a sequential application with its operators using SPBench's API, where applications share similar program structure. The only thing that changes among applications are the name and amount of internal operators (line 6 in the example). In real applications, the amount of internal operators can vary from a single one to over a dozen. Besides the internal operators, all other elements are identical in all applications for simplicity. All this standardization of elements and operators names and code structure is planned to facilitate the understanding of applications. Once users understand the structure of an application, they can automatically understand the structure of all of them. Thus, maintaining this structure is important to increase parallel code reuse among applications.

Listing 1 Example of custom benchmark using the SPar PPI.

```
/*beginning of the stream region*/
[[spar::ToStream]] while(1){
    Item item;
    //Read a piece of data from a external source
    if (!source_op(item)) break;
    [[spar::Stage, spar::Input(item), spar::Output(item), spar::
    Replicate()]]{
        //A single operator in the pipeline stage
        operatorA_op(item);
    }
    [[spar::Stage, spar::Input(item), spar::Output(item), spar::
                                Replicate()]]{
        //Two operators combined in the same pipeline stage
        operatorB_op(item);
        operatorC_op(item);
    }
    [[spar::Stage, spar::Input(item)]]{
        sink_op(item); //sink operator
    }
}
/*end of the stream region*/
```

Based on the sequential applications, programmers can write the parallel code using some PPI and build their own custom benchmarks. These benchmarks automatically include all performance metrics and workload management. This way, users can focus only on the parallel code and parallelism strategies.

Listing 1 presents an application example with three intermediate operators, besides source and sink. It shows how it looks like a customized implementation. This example

uses SPar to implement a four-stage Pipeline parallelism strategy with replicated stages. As can be noted, no single rewriting of the original code is needed. The first Pipeline stage runs the source operator. The second stage runs *n* replicas of operator A. The third stage combines the operators B and C and also runs *n* replicas of them, and last stage runs the sink operator. In this way, users can set up different parallelism strategies just by moving the sequential operators around, without having to deal with large blocks of code or worry about data dependency. For stateful operators there are some limitations and synchronization mechanisms may be needed.

These custom parallel implementations are represented by the right part of Fig. 2. However, SPBench also provides users with several ready-to-run parallel implementations, using different PPIs and parallelism strategies, for all applications. These parallel implementations that come with the framework are the SPBench benchmark suite (shown in the center of Fig. 2) and later presented in this work.

3.3 Current applications set

The current applications set of SPBench comprises four real-world applications: `Bzip2`, `Lane Detection`, `Person Recognition`, and `Ferret` (PARSEC [32]). These applications have already been studied and used as benchmarks in prior work [12,13]. Although, they were reimplemented to fit in SPBench API. Our plan is to include more stream processing applications in the near future.

- **Bzip2** [40] is a free and open-source data compression application. It compresses data more effectively than the older LZW (.Z) and Deflate (.zip and .gz) algorithms, but it is considerably slower than them. Therefore, parallel implementations that increase compression speed are useful. This application can be divided into a three-stage Pipeline (source, compress/decompress, sink), as shown in Fig. 3a, and has two operation modes: compress and decompress.
- **Person Recognition** [41] tries to match human faces from a video frame against a database of faces. It could be used to recognize faces in a crowd and compare against a police database to identify wanted criminals, for instance. For each video frame it applies a detection algorithm to detect all the faces in it. Then, it uses a set of face images and compares each of the detected faces in the frame with the faces on that set. The recognized faces are marked with a circle, and then the frames are written to the output file. Therefore, this application can be divided into a four-stage Pipeline, represented by Fig. 3b. This application is described with details in Reference [13].
- **Lane Detection** is the task of detecting lanes of a road from a camera device. It can be used to keep autonomous cars on track to avoid collisions, for instance. This application captures each frame of a input video file and applies three computer vision algorithms. It can be divided into a nine-stage Pipeline, as shown in Fig. 3c. Through these stages, the detected lanes are marked with straight lines in a new frame. This new frame with the marked lanes is then overlaid on the original, and the resulting frame is written to the output file. More details about this application can be found in Reference [13].

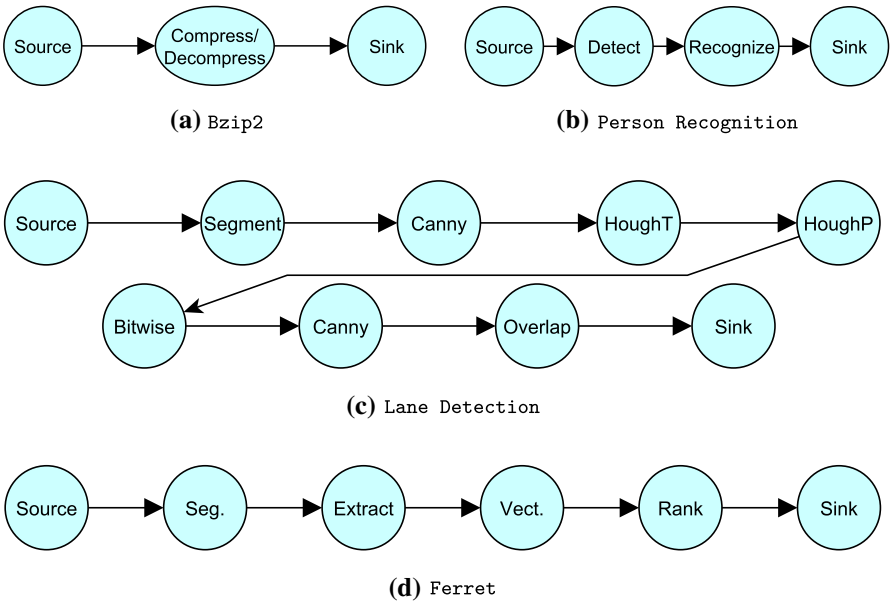


Fig. 3 Applications flow graphs

– **Ferret** is a PARSEC [32] application intended for a content similarity search in data such as video, audio, and images. In PARSEC the parallelism is implemented with POSIX Threads using a six-stage Pipeline (Fig. 3d). The first and last ones are source and sink. The second stage performs the segmentation, a process that organizes the regions of each image into sets. The third stage extracts 14 features from the segmented regions of each image. The fourth stage (vectorization) applies an indexing method, which selects possible similar images. The fifth stage performs a refined search of the images selected in the fourth stage, ranking the most similar images.

3.4 Performance metrics

The goal of SPBENCH is to ease the creation of custom benchmarks for stream parallelism from real-world stream applications. These benchmarks need to be representative for the scenario being evaluated. Therefore, it needs to provide evaluation metrics that are also representative. [21] conducted a broader survey and identified some metrics that were more representative in stream processing benchmarks. The result of the survey is similar to what we identified in our related work. Therefore, SPBENCH also includes such metrics, which are: latency, throughput, and resource usage, such as CPU and memory.

– **Latency**: It is basically the time need to complete a task [2]. It can be the time a operator takes to process a single item, or the the time the application takes to process this item from end-to-end. Lower latency is better.

- **Throughput**: It is the rate of processed items per unity of time. Higher throughput is better.
- **CPU and Mem. usage**: Here this metric refers to CPU and memory usage. It can be used to monitor an application and analyze the behavior of different workloads at discrete intervals of time.

The framework applications implemented with the API offer these metrics by default to developers. That is, custom parallel implementation have these metrics automatically available and users can enable or disable them at runtime. The performance can be evaluated at different levels of accuracy and detail. About accuracy, it consists of the granularity of the measurement. Here users can choose a fine-grain evaluation, such as once per item, at minimum, or they can also select coarse granularities dynamically. Here, it is necessary to define a time interval in which the selected metrics will be measured, as once per second.

Regarding detailing, in SPBENCH users can select three levels. The first level provides a global average of the selected metrics, considering the execution as a whole and measuring latency and throughput from end-to-end of the Pipeline. This level allows the evaluation of global performance of the application. The second level presents such results as an average per operator, individually. This is important for the user to evaluate aspects such as bottlenecks and load balancing between operators, allowing the optimization of the parallelism strategy.

The third level monitors the metrics at time intervals defined by users and generates a report. This greater level of detail allows the application to be monitored throughout the execution and the workload characteristics to be evaluated. All measurements are calculated and stored in-memory, to reduce disk I/O overhead and printed into a file when computation ends.

3.5 Command-line interface

The bottom of Fig. 2 illustrates the SPBENCH Command-Line Interface (CLI). This interface allows users to run all applications, modify the provided benchmark codes, implement new versions, add implementations with new PPIs, select different workload configurations, select and configure performance metrics, and among others. (see Fig. 1). Bellow are the main commands:

- **'new'** - create a new copy of an application for users to customize and implement their own benchmark with one of the PPIs available in the framework.
- **'edit'** - open the source code of a application/benchmark for the user to write/-modify the parallel code.
- **'configure'** - open for editing a JSON file that allows the user to insert specific compilation commands, library dependencies, compiling and PPI flags, etc.
- **'compile'** - compile a specific benchmark.
- **'execute'** - execute a specific benchmark (here users can select workload options, execution metrics, and add custom arguments).

4 SPBench use case

In this section we present a use case for SPBench. We first run the sequential applications to characterize their behavior and the workloads. Here we demonstrate how the metrics work in monitoring mode. Next, we use the SPBench interface to create parallel versions of the applications with different PPIs. Then we use these parallel versions as benchmarks to compare and evaluate the performance of these PPIs. At the end, we summarize and discuss the results (Sect. 4.5).

4.1 Experimental setup

The experiments were performed on two computers. The one we used for most experiments in this paper has 64 GB of RAM and two processors Intel® Xeon® Silver 4210 (total of 20 physical cores and 40 threads with Hyper-Threading). We will refer to it as "`computer A`" in the following sections. The operating system was Ubuntu Server 20.04, 64 bits, kernel 5.4.0-59-generic, and GCC 9.3.0 using `-O3` optimization flag. Other libraries used were OpenCV version 2.4.13.6, Intel® TBB 2020 Update 2 (TBB_INTERFACE_VERSION 11102), and FastFlow version 3.

The second computer (`computer B`) has 32 GB of RAM and two processors Intel® Xeon® E5-2620 v3 (total of 12 physical cores and 24 threads). Benchmarks for the Bzip2, `Lane Detection`, and `Person Recognition` applications have already been run on this machine using SPBench in [30]. Hence, here we only evaluate the parallel benchmarks for `Ferret`, which is the application that has not yet been evaluated on this architecture.

To monitor the sequential applications, we used the routines of the SPBench API itself. These routines allow us to monitor runtime with microsecond precision and get CPU/memory usage information obtained from the `/proc/[pid]/stat` pseudo-file. Although each new item can be monitored, we choose to monitor every 250 ms and all performance metrics are stored in a vector and calculated after the stream region, to avoid interfering with the results. For parallel executions, we used 1–40 and 1–24 replicas. Zero replica indicates the result of sequential applications. Each result represents the average of 10 executions, with standard deviation properly included in the graphs with error-bars.

Due to the space constraints, we evaluated the applications using a single workload and input type. For `Bzip2` benchmarks, we used an ISO Image of 702 MB as an input file, using 900 KB micro-batches. We choose 640 × 360 resolution MPEG-4 videos with 450 frames for `Person Recognition` and 1858 frames for `Lane Detection`. Finally, for `Ferret` we used the *native* workload from PARSEC [32]. The validation of the results was done through the `md5sum` tool, comparing our solution with the ones given by the original applications.

4.2 Applications characterization

To show the behavior of the applications and how they differ from each other, in Fig. 4 we present the characterization results for the sequential applications. We monitor

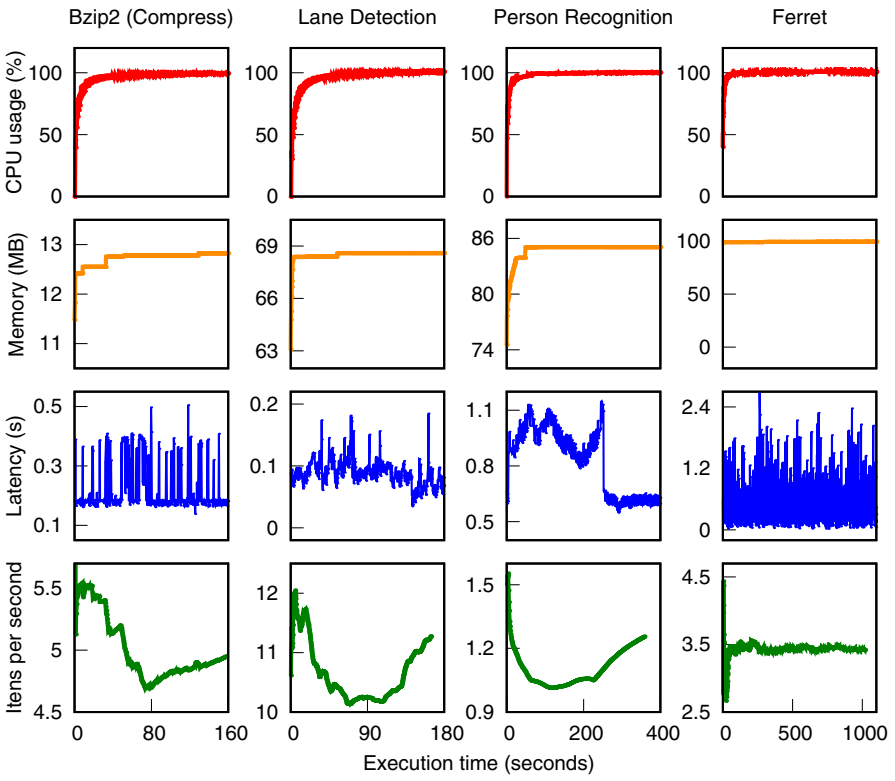


Fig. 4 Characterization results for the sequential applications

latency, throughput, and CPU and memory usage, which are the metrics currently supported by SPBENCH. Analyzing CPU usage, we can see that all three applications used 100% CPU almost all the time. It shows great potential to achieve performance through parallelism. Regarding memory, they present distinct behaviors. `Bzip2` is the application that used less, and `Person Recognition` presents a lower curve until reaching its maximum level. `Ferret` presented a stable memory usage, a high variation in latency, and a consistent throughput throughout the execution.

The variation in latency and throughput is explained by the workload characteristics. The sharp drop in latency in the second half of the `Person Recognition` execution exactly matches the moments when there is no front face identified in the input video and, therefore, no face to be recognized. These moments of more or less activity also apply to the other applications. The throughput at each time point represents a partial average of the execution as a whole. Therefore, it is not immediately impacted by changes in the workload, unlike latency, which is not affected by previous results.

4.3 Building custom benchmarks

To evaluate the main features of SPBench, we use it to build custom benchmarks that will be made available for users along with the sequential versions. We parallelize them using Intel TBB, FastFlow, and SPar, and evaluate their performance only in terms of latency and throughput due to space constraints. These PPIs were chosen because they are widely used for stream processing in C++. TBB offers few customization options and is by default very well optimized for these scenarios that we will test. On the other hand, FastFlow provides more optimization options, and it is up to the programmer to understand the characteristics of each application and workload to extract the maximum performance. In contrast, SPar provides a small set of annotations to express stream parallelism. Its compiler generates parallel code, calling FastFlow's pattern routines.

For FastFlow and SPar, we implemented an `on-demand` + `blocking` queue configuration [8], as recommended by [13]. We also built benchmarks version using the custom FastFlow's thread mapping (physical cores first) and without it (default mapping). For all benchmarks a Farm pattern is implemented with the PPIs. This parallelism strategy was based on prior works [12,13]. Our focus is not to provide a very optimized version, because there are several options that could be combined in each PPI, instead we provide the basic strategies suggested in the state-of-the-art works for these applications. Other researcher interested in deeply evaluate each PPI may extend and continue this study using SPBench.

`Bzip2` has three operators: `source()`, `compress()` (or `decompress()`), and `sink()`, which can be seen in Fig. 3a. The operator in the middle is stateless, which means that it does not need to keep the state of previous iterations and can work independently on each iteration. Therefore, we implement the parallelism instantiating a ordered Farm pattern in FastFlow (`ff_OFarm`). In TBB, we built a three-stage Pipeline, setting the middle stage to perform with a parallel filter. We annotated the code with SPar, adding replication for the middle stage.

The `Person Recognition` application has four operators (Fig. 3b). Besides the basic `source()` and `sink()`, there is `detect()` and `recognize()`. The latter two are also stateless operators, just like the `compress()` operator of `Bzip2`. It means that we could replicate both independently. However, previous studies have shown that unifying both operators into one avoids load imbalance and improves performance [13]. Therefore we follow this same strategy. Thus, both internal operators are joined into a single one and then replicated n times.

In the applications with many operators, such as `Lane Detection` and `Ferret` (Fig. 3c and 3.d), there are several ways to combine and replicate them. For space reasons, with `Lane Detection` we follow the same Farm strategy and combine all the intermediate operators into a single stage/worker. For `Ferret`, besides the Farm pattern, we explore other compositions as well.

4.3.1 Ferret custom compositions

The `Ferret` application in PARSEC is originally parallelized using a Pipeline-Farm pattern: `pipe(seq(source)`, `Farm(seg,` n`)`, `Farm(extract,` n`)`, `Farm(vect,`

n), `Farm(rank, n)`, `seq(sink)`), where n is the number of replicas/workers in each Farm and $n = 1$ represents a simple Pipeline. Although some operators are stateful in `Ferret`, the concurrent access to the state is lock-protected. Therefore, other compositions can also be explored.

SPBENCH allows the application operators to be evaluated individually. We previously evaluated the sequential `Ferret` and realized that the `rank` operator does most of the computation. So, this application is unbalanced, making the Pipeline-farm implemented in PARSEC not efficient in some scenarios, since it takes more resources than necessary. Therefore, in addition to the Farm and Pipeline-farm, in this work we also used the SPBENCH to implement other `Ferret` compositions. TBB does not support many compositions besides Farm, Pipeline and Pipeline-farm, so we only use FASTFLOW for these other compositions. Other compositions for `Ferret` have already been explored in P^{3}ARSEC [33], but there the authors evaluated speedup.

Pipeline compositions in FasFlow increase the latency, because the items wait a long time idle in the queue of each stage. This occurs even using an `on-demand` setup with queue size one in `Ferret`, because the most intensive stage (`rank`) is at the end part of the Pipeline. Therefore, to try to improve both latency and throughput, we implemented this composition: `pipe(seq(source), Farm((seg, extract, vect),` n_1`), Farm(rank,` n_2`), seq(sink))` compositions. The goal is to reduce the number of Pipeline stages while building a dedicated Farm for the `rank` operator. SPBENCH allowed to add and run all these `Ferret`'s parallel versions easily and quickly, because it is just a matter of moving operators across building blocks (classes or functions).

4.4 Parallelism performance

The experiments were conducted as described in Sect. 4.1. We performed experiments evaluating memory as source/sink in a previous work [30]. Here we use disk as source/sink option (the network option is still in developing phase). Performance in stream processing is usually defined in terms of latency or throughput. In this case, each value in the Farm charts in Fig. 5 represents the average latency (top) and throughput (bottom) calculated among all the processed items.

All cases with Farm share a reduction in performance scalability when using over 20 replicas. From this point, the processor needs to allocate threads in both physical and virtual cores. This is the expected behavior for these applications that have high CPU usage. Regarding latency, lower is better. With less than 20 replicas the latency with TBB is less than half as high as FASTFLOW and SPAR for all cases. Above 20 replicas, the latency quickly increases and gets more than twice as high with 40 replicas in most scenarios.

The difference between TBB and FASTFLOW/SPAR can be explained by the characteristic of the communication queues between the workers and the source (emitter), and sink (collector). TBB has a work stealing scheduler. This means that a thread process an item and can take over any stage. In FASTFLOW the threads always run the same stage. Therefore, the one that runs the `source_op()` operator is done faster.

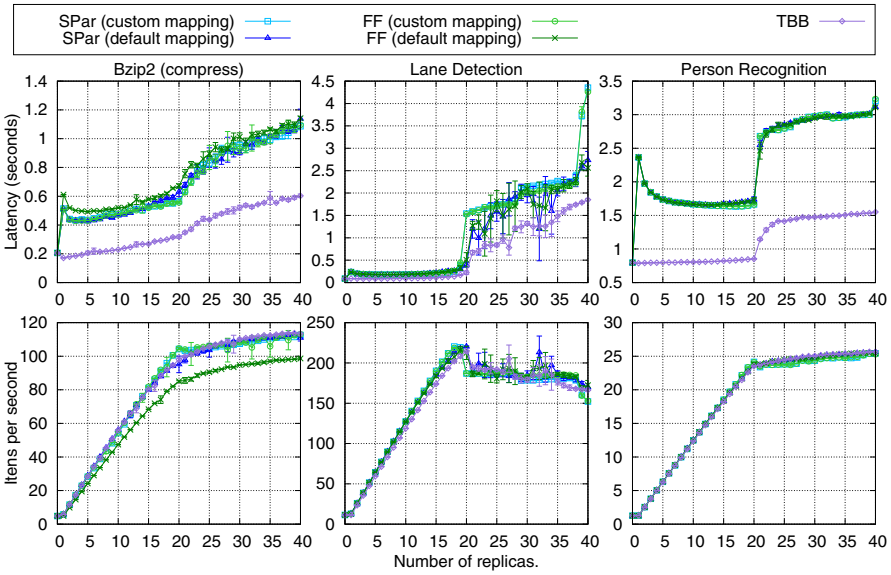


Fig. 5 Latency and throughput for the Farm implementations

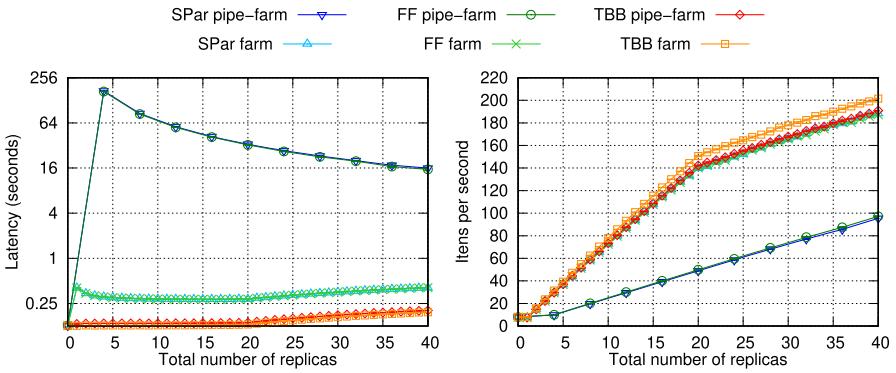
This adds an extra delay when calculating from the moment a item is read to the moment it is finally processed by the last stage, increasing the latency.

Regarding throughput, higher is better. All PPIs and test cases presented good scalability. The higher latency of FastFlow/SPar did not negatively impact the throughput and these PPIs showed even better performance than TBB with 20 replicas. The exception is `Bzip2` with FastFlow, when not using its custom mapping policy, where performance above 20 replicas was about 20% lower than the other versions. The default mapping in the other scenarios showed equivalent performance, being only slightly better in `Person Recognition` above 20 replicas.

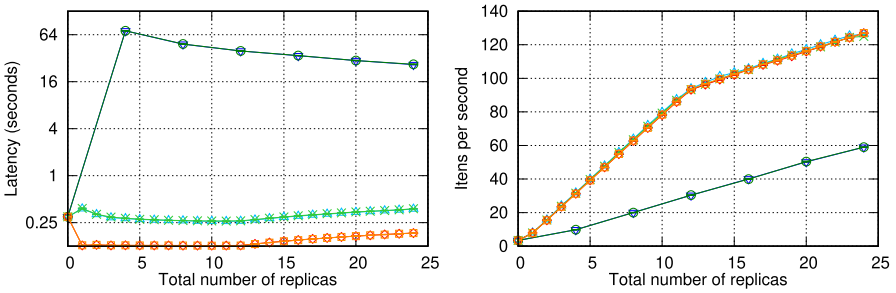
The `Lane Detection` benchmarks on Fig. 5 (middle column) presented a distinct behavior for both latency and throughput. Over 20 replicas the latency increases faster and throughput can not scale-up. This result can be explained by the workload characteristic (Sect. 4.1). `Bzip2` processes 783 data items, while `Person Recognition` processes 450 items. On the other hand, `Lane Detection` receives 1858 data items to process and do it in a similar time to other applications (Fig. 4). It shows the presence of a disk I/O bottleneck, which impacted the scalability of this application. This shows also why persistent memory architectures are becoming more popular and why it is important that SPBench provides this option for users to test these realistic scenarios.

4.4.1 Ferret results

Here we present the results for the `Ferret` only. We implemented three types of compositions, as described in Sect. 4.3.1: Farm, Pipeline-farm, and a customized Pipeline-farm. With this customized version we aim to find an alternative version that



(a) `Ferret`: `Farm` and `Pipeline-farm` versions (`computer A`)

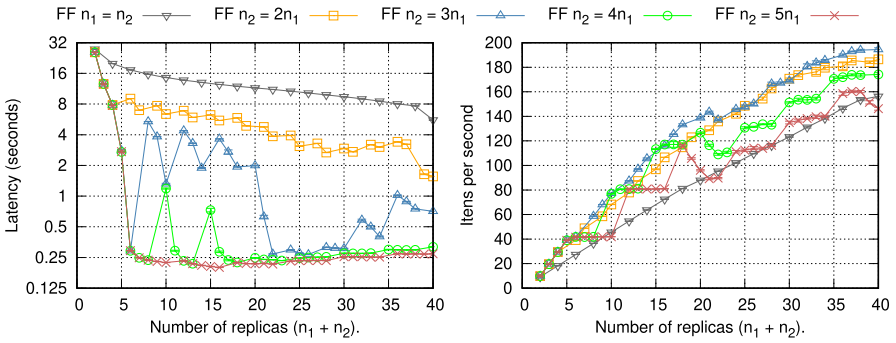


(b) `Ferret`: `Farm` and `Pipeline-farm` versions (`computer B`)

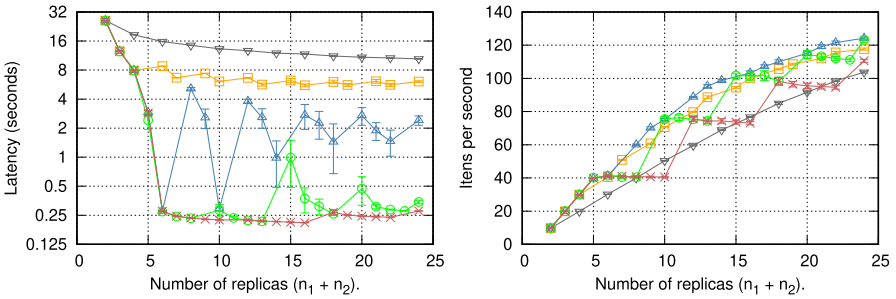
Fig. 6 Latency and throughput for the `Farm` and `Pipeline of Farms` compositions of `Ferret`. The `pipe-farm` versions have four `Farms`, so each increment of n increases four replicas at once

is able to achieve a better trade-off between latency and throughput. Figure 6 shows the result for the first two compositions and Fig. 7 shows variations of the customized version. Here, the y-axis of the latency graphs (left) is presented in logarithmic scale for better visualization of the results. The x-axis in all graphs presents the total sum of replicas (n) of the workers used in each Farm. Since Pipeline-farm implements four Farms, in this version each increment of n is multiplied by a factor of four. In the customized versions in Fig. 7 this factor varies for the values of n_1 and n_2. For all implementations with SPar/FastFlow, we enabled the blocking mode and, to be able to implement the `on-demand` policy, the versions were compiled with the macros `FF_BOUNDED_BUFFER` and `DEFAULT_BUFFER_CAPACITY=1`.

In the Farm implementations of `Ferret` in Fig. 6, the increasing in latency after the number of replicas reach the physical cores limit is lower than the first three applications (Fig. 5). It is due to the fact that this application does not require the resulting items to be written in a specific order at sink. Therefore, input order does not need to be preserved in the sink, which alleviates latency for reordering items. The same reason is applied for the increasing scalability after 20 replicas. This same behavior occurs on both architectures tested (`computers A` and `B`). However, the results presented some differences in these architectures. `Ferret` achieved lower



(a) Custom `Ferret Pipeline-farm` versions (`computer A`)



(b) Custom `Ferret Pipeline-farm` versions (`computer B`)

Fig. 7 Latency and throughput for the custom `pipe-farm` versions of `Ferret`. These versions are defined as `pipe(seq(source), farm((seg,ext,vect),`n_1`), farm(rank,` n_2`), seq(sink))`, where n_1 and n_2 are the number of workers. The five versions represent different proportions of the number of workers in each `Farm`. E.g., $n_2 = 3n_1$ means that every time n_2 is increased by 3, n_1 is increased by 1

latencies on `computer B`, while the throughput of the `Farm` versions of SPar and FastFlow were equivalent to TBB on this same computer.

Regarding the `Pipeline-farm` and `Farm` patterns in Fig. 6, versions implemented with TBB presented no difference with respect to each other. In [42] the authors also implemented `Farm` and `Pipeline-farm` for `Ferret` using TBB and came up with similar results. Here, the same happened in both tested architectures. The way the application executes with these two patterns is similar due to the work-stealing runtime model. However, the same is not true for FastFlow/SPar. The increase in the number of queues for each pair of stages of the `Pipeline` with these PPIs causes the latency to increase dozens of times. Also, stages that require less processing use resources that could be dedicated to the slowest one (`rank`).

Finally, we have the performance of the custom `Pipeline-farm` versions running in the two tested computers in Fig. 7. This versions consist of a four-stage `Pipeline` with a `Farm` in the second and a `Farm` in the third stage. We implemented five variations, each representing a different ratio between n_1 (first Farm) and n_2 (second Farm): 1:1, 1:2, 1:3, 1:4, and 1:5. For instance, for the $1:3$ ratio ($n_2 = 3n_1$), 40 replicas in the charts equals to 10 replicas for the first `Farm` plus 30 replicas for the second `Farm`.

Although the results are somewhat different between the two tested architectures, the conclusions we can draw from these results are the same for both scenarios. The 1:1 version was the worst for both latency and throughput metrics. Although 1:2 and 1:3 achieved good throughput, they did not achieved low latency. On the other hand, version 1:5 achieved the lowest latency of all, but pays the price for throughput. We can observe that the 1:4 ratio is the most balanced for this custom `Ferret` implementation. This version achieved a throughput equivalent to the simple FastFlow `Farm`, but with a latency about 15% lower with 40 replicas.

These experiments highlighted the importance of SPBench to build custom benchmarks and its easy of use to tune FastFlow, SPar, and TBB. We successfully evaluated PPIs for stream parallelism while these applications were quickly implemented when using SPBench. We were able to achieve similar overall results to those in the literature [17,33,42], which also demonstrates the reliability.

4.5 Summary of the results and discussion

In this paper the main features of the SPBench were presented and tested. We first used SPBench to characterize the workloads, through monitoring metrics (Fig. 4). Here, the sequential applications presented distinct results among them. They all presented high CPU usage, indicating high potential for parallelism exploitation. Such feature will help users to better understand the behavior of the SPBench applications to parallelize it without the need for instrumenting the code and extracting these metrics.

Next, we used SPBench to implement 16 custom benchmarks using three different PPIs (Sect. 4.3). It was easy and fast to implement the parallel benchmarks. Once the parallel code is written for a single application, that same code can be reused to other SPBench applications in a few steps. Users only need to change parallel code itself using the target PPI. Reusing existing code for building new parallel applications will increase user productivity to fast obtain results.

With these custom benchmarks, the first experiments (Fig. 5) were useful to evaluate and compare the performance of each PPI. These experiments were a plus to the ones presented in [30], which were performed on the `computer B`. Our benchmarks achieved good performance scalability. In `Lane Detection`, we observed the impact of the I/O bottleneck due to excessive read/write operations on disk. We also evaluated two different scheduling policies in this part. Therefore, these experiments added important insights and demonstrates that the same code can run on different multi-core architectures.

Finally, the `Ferret` application are unique, therefore, results are for two different architectures. As `Ferret` is an application originally implemented with a `Pipeline-farm` pattern in PARSEC, we used the same strategy for testing the SPBench's advantages for creating, managing, and evaluating multiple versions of benchmarks quickly and easily. It was possible to compose different number of stages and number of threads per stage. The custom `Pipeline-farm` (Fig. 7) versions showed how a fine-tuning number of threads per stage can improve the performance. These versions allow you to prioritize latency or throughput, or even a balanced configuration between the two. This shows that SPBench is flexible, which is good

for adapting the configurations that best fits to the architectures characteristics and resources available.

In summary, the SPBENCH framework worked for evaluating the performance with multiple metrics and time dimensions for different PPIs, stream applications, parallel patterns, parallel versions, and architectures. Even though the framework is in an early stage of development, it can already be used for research, as it is public available[2].

5 Conclusions and future work

As discussed in Sects. 1 and 3, researchers face several challenges when they need to evaluate PPIs and similar solutions for stream processing, which becomes a time-consuming task. In the related work section (Sect. 2), we have demonstrated that none of them solve this problem. This way, in this paper we contributed with SPBENCH, a framework for creating benchmarks of stream processing applications. We demonstrated its ease of use for creating custom benchmarks when aiming at evaluating and implementing stream parallelism. Through SPBENCH, we built custom benchmarks with different PPIs, parallel patterns, compositions, and other variations, including `Ferret`, a well-known benchmark from the PARSEC suite [32]. We evaluated all implemented versions with different performance metrics to compare TBB, FASTFLOW, and SPAR. All those versions are available to SPBENCH's user as a parallel benchmark suite.

We have provided a few subsets of experiments. It was enough to show that SPBENCH can be easily used to implement different types of stream parallelism. In the future, we expect SPBENCH for evaluating parallel programming in general and for educational or teaching purposes. There is also room to include more applications to cover and represent other scenarios, which requires an extensive study to understand what is representative in this domain. Then, we must find suitable applications and implement them as part of the SPBENCH API. In addition, the framework does not currently include applications with stateful operations. Adding this type of application while keeping the SPBENCH metrics system consistent is challenging. We will face this same challenge to add the option to instantiate multiple sources.

Other forthcoming features are the support to receive data incoming from the network and to dynamically control batch size and data input rate. Here, it is also challenging to present all this in a simple and abstract way to users. Another minor upcoming feature is the possibility for users to register custom workload classes. Besides all this, as future work we will also continue adding new PPIs and different parallelism strategies to SPBENCH.

Acknowledgements This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, FAPERGS 05/2019-PQG project PARAS (Nº 19/2551-0001895-9), FAPERGS 10/2020-ARD project SPAR4.0 (Nº 21/2551-0000725-7), Universal MCTIC/CNPq Nº 28/2018 project SPARCLOUD (Nº 437693/2018-0). The authors acknowledge LAD-IDEIA/PUCRS for computing resources.

[2] https://github.com/gmap/spbench.

References

1. Andrade HC, Gedik B, Turaga DS (2014) Fundamentals of stream processing: application design, systems, and analytics. Cambridge University Press, Cambridge
2. McCool M, Reinders J, Robison A (2012) Structured parallel programming: patterns for efficient computation. Elsevier, Amsterdam
3. Friedman E, Tzoumas K (2016) Introduction to Apache Flink: stream processing for real time and beyond. O'Reilly Media, Inc., Sebastopol
4. Jain A (2017) Mastering apache storm: Real-time big data streaming using kafka, hbase and redis. Packt Publishing Ltd, Birmingham
5. Nabi Z (2016) Pro Spark Streaming: The Zen of Real-Time Analytics Using Apache Spark. Apress, New York
6. Zeuch S, Monte BD, Karimov J, Lutz C, Renz M, Traub J, Breß S, Rabl T, Markl V (2019) Analyzing efficient stream processing on modern hardware. Proc VLDB Endow 12(5):516–530
7. Voss M, Asenjo R, Reinders J (2019) Pro TBB: C++ parallel programming with threading building blocks. Apress, New York
8. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M (2017) Fastflow: high-level and efficient streaming on multicore, chap. 13, pp. 261–280. John Wiley & Sons Ltd, Hoboken
9. 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
10. del Rio Astorga D, Dolz MF, Fernández J, García JD (2017) A generic parallel pattern interface for stream and data processing. Concurr Comput Pract Exp 29(24):e4175
11. Mencagli G, Torquati M, Griebler D, Danelutto M, Fernandes LGL (2019) Raising the parallel abstraction level for streaming analytics applications. IEEE Access 7:131944–131961
12. Griebler D, Hoffmann RB, Danelutto M, Fernandes LG (2018) High-level and productive stream parallelism for Dedup, Ferret, and Bzip2. Int J Parallel Program 47(1):253–271
13. Griebler D, Hoffmann RB, Danelutto M, Fernandes LG (2017) Higher-Level Parallelism Abstractions for Video Applications with SPar. Parallel Computing is Everywhere. In: Proceedings of the international conference on parallel computing, ParCo'17. IOS Press, Bologna, Italy, pp 698–707
14. Rockenbach DA, Stein CM, Griebler D, Mencagli G, Torquati M, Danelutto M, Fernandes LG (2019) Stream Processing on Multi-cores with GPUs: Parallel Programming Models' Challenges. In: International parallel and distributed processing symposium workshops (IPDPSW). IPDPSW'19. IEEE, Rio de Janeiro, Brazil, pp 834–841
15. Vogel A, Griebler D, Fernandes LG (2021) Providing high- level self- adaptive abstractions for stream parallelism on multicores. Softw Pract Exp 51:1194–1217
16. Mencagli G, Dazzi P, Tonci N (2018) Spinstreams: A static optimization tool for data stream processing applications. In: Middleware '18: Proceedings of the 19th international middleware conference, middleware '18, Association for Computing Machinery, New York, NY, USA, pp 66–79
17. 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
18. Hoffmann RB, Griebler D, Danelutto M, Fernandes LG (2020) Stream Parallelism Annotations for Multi-Core Frameworks. In: XXIV Brazilian Symposium on Programming Languages (SBLP). SBLP'20. ACM, Natal, Brazil, pp 48–55
19. Rockenbach DA, Griebler D, Danelutto M, Fernandes LG (2019) High-level stream parallelism abstractions with SPar targeting GPUs. Parallel computing is everywhere. In: Proceedings of the International Conference on Parallel Computing (ParCo), ParCo'19. vol 36, IOS Press, Prague, Czech Republic, pp 543–552
20. 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 33:e5786
21. Bordin MV, Griebler D, Mencagli G, Geyer CFR, Fernandes LG (2020) DSPBench: a suite of benchmark applications for distributed data stream processing systems. IEEE Access 8:222900–222917
22. 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
23. Shukla A, Chaturvedi S, Simmhan Y (2017) Riotbench: An iot benchmark for distributed stream processing systems. Concurr Comput Pract Exp 29(21):e4257

24. Lu R, Wu G, Xie B, Hu J (2014) Stream bench: towards benchmarking modern distributed stream computing frameworks. In: 7th International conference on utility and cloud computing, pp 69–78
25. Wang Y (2016) Stream processing systems benchmark: Streambench. Master's thesis, Aalto University
26. Agrawal D, Butt A, Doshi K, Larriba-Pey JL, Li M, Reiss FR, Raab F, Schiefer B, Suzumura T, Xia Y (2016) Sparkbench - a spark performance testing suite. In: Nambiar R, Poess M (eds) Performance evaluation and benchmarking: traditional to big data to internet of things. Springer International, Cham, pp 26–44
27. Maron CAF, Vogel A, Griebler D, Fernandes LG (2019) Should PARSEC benchmarks be more parametric? a case study with Dedup. In: 27th Euromicro international conference on parallel. Distributed and network-based processing (PDP), PDP'19. IEEE, Pavia, Italy, pp 217–221
28. Zhang S, He B, Dahlmeier D, Zhou AC, Heinze T (2017) Revisiting the design of data stream processing systems on multi-core processors. In: 2017 IEEE 33rd International conference on data engineering (ICDE), pp 659–670
29. Griebler D (2016) Domain-Specific Language & Support Tool for High-Level Stream Parallelism. Ph.D. thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil
30. Garcia AM, Griebler D, Schepke C, Fernandes LG (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, Spain
31. 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
32. Bienia C, Kumar S, Singh JP, 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
33. De Sensi D, De Matteis T, Torquati M, Mencagli G, Danelutto M (2017) Bringing parallel patterns out of the corner: The p3 arsec benchmark suite. ACM Trans Archit Code Optim 14(4):1–26
34. Henning S, Hasselbring W (2021) Theodolite: Scalability benchmarking of distributed stream processing engines in microservice architectures. Big Data Res 25:100209
35. 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
36. Čermák M, Tovarňák D, Laštovička M, Čeleda P (2016) A performance benchmark for netflow data analysis on distributed stream processing systems. In: NOMS 2016 - 2016 IEEE/IFIP network operations and management symposium, pp 919–924
37. Le-Phuoc D, Dao-Tran M, Pham MD, Boncz P, Eiter T, Fink M (2012) Linked stream data processing engines: Facts and figures. In: The Semantic Web – ISWC 2012. Springer, Berlin, Heidelberg, pp 300–312
38. Amanullah MA, Habeeb RAA, Nasaruddin FH, Gani A, Ahmed E, Nainar ASM, Akim NM, Imran M (2020) Deep learning and big data technologies for iot security. Comput Commun 151:495–517
39. Hazelcast: Hazelcast in-memory computing platform (2020). https://hazelcast.com/products/in-memory-computing-platform/#in-memory-solutions
40. Seward J (2017) A Program and Library for Data Compression. http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html
41. Arubas E (2013) Face detection and recognition (theory and practice) . http://eyalarubas.com/face-detection-and-recognition.html
42. 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