The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-05934-z

Performance and programmability of GrPPI for parallel stream processing on multi-cores

Adriano Marques Garcia[1] · Dalvan Griebler[2] · Claudio Schepke[3] · José Daniel García[4] · Javier Fernández Muñoz[4] · Luiz Gustavo Fernandes[2]

Accepted: 27 January 2024

Abstract
GrPPI library aims to simplify the burdening task of parallel programming. It provides a unified, abstract, and generic layer while promising minimal overhead on performance. Although it supports stream parallelism, GrPPI lacks an evaluation regarding representative performance metrics for this domain, such as throughput and latency. This work evaluates GrPPI focused on parallel stream processing. We compare the throughput and latency performance, memory usage, and programmability of GrPPI against handwritten parallel code. For this, we use the benchmarking framework SPBench to build custom GrPPI benchmarks and benchmarks with handwritten parallel code using the same backends supported by GrPPI. The basis of the benchmarks is real applications, such as Lane Detection, Bzip2, Face Recognizer, and Ferret. Experiments show that while performance is often competitive with handwritten parallel code, the infeasibility of fine-tuning GrPPI is a crucial drawback for emerging applications. Despite this, programmability experiments estimate that GrPPI can potentially reduce the development time of parallel applications by about three times.

Keywords Stream parallelism · GrPPI · SPBench · OpenMP · Intel TBB · FastFlow

1 Introduction

Implementing parallelism for stream processing is not easy. Some strategies can mitigate this difficulty, such as structured parallel patterns [1]. Pipeline and Farm are examples of parallel patterns for stream processing. Some parallel programming interfaces (PPIs), such as FastFlow [2] and Intel® Threading Building Blocks (TBB) [3], natively implement concurrent patterns. However, even using these PPIs, implementing stream parallelism while achieving performance improvement is still a task for experts.

Extended author information available on the last page of the article

Published online: 26 February 2024

Based on that, some PPIs aim to ease the programmability of these parallelism libraries by adding extra abstraction layers. This way, users can implement parallelism with simpler syntax and reduced lines of code. However, such abstraction may end up causing some mechanisms specific to backend PPIs not to be supported. Fine-tuning is often required to achieve desirable performance and efficient resource utilization. Therefore, evaluating such solutions with diverse setups can help identify how well they can express parallelism for different scenarios and how efficient they are.

In this context, GrPPI is a generic and reusable parallel pattern interface for both stream processing and data-intensive C++ applications [4]. It allows users to compile their own programs with FastFlow, TBB, OpenMP, ISO C++ threads, and other backends from a single generic implementation. Even though they also implement parallel patterns for data stream processing applications [5], which usually have low-latency requirements, and support for distributed platforms [6, 7], which increases the communication delay between nodes of the pipeline, we found no performance evaluation of GrPPI regarding latency in the literature. Most previous work evaluated it regarding execution time or speedup.

This work extends the last previous work from [8]. The goal is to evaluate the performance of GrPPI's backends for multi-cores in terms of throughput, latency, memory usage, and programmability. In the previous work, GrPPI was not evaluated against handwritten OpenMP and ISO C++ threads. Furthermore, the experiments were incomplete, and the analysis and discussion were rather limited. To help achieve the goal, we implemented four benchmarks using GrPPI and compared the performance of each backend against handwritten parallel code using the same backends: OpenMP, Intel TBB, FastFlow, and ISO C++ threads. We use SPBench [9] to create the handwritten and GrPPI benchmarks. It is a framework that simplifies the development and management of custom benchmarks for stream processing. Its main goal is to enable users to evaluate and compare the performance of PPIs, which is the purpose of this paper.

The main contributions of this work can be summarized as follows:

- An extension of the SPBench benchmarking framework with support for benchmark generation using GrPPI.
- An analysis of GrPPI performance from a latency perspective using four real-world stream processing applications and comparing it against handwritten parallel implementations with TBB, OpenMP, FastFlow, and ISO C++ Threads.
- An investigation of the programmability of GrPPI using Halstead's method adapted for parallel applications.
- A critical analysis and suggestions for improvements to GrPPI.

The organization of our paper is as follows. Next, in Sect. 2, we provide a background for this work, introducing concepts of parallel stream processing, the GrPPI library, and the SPBench framework. Section 3 discusses related work. Section 4 describes the methodology used in the experiments, and the results and experimental analysis are in Sect. 5. In Sect. 6, we evaluate the experimental results with a more critical view. Section 7 presents a list of suggestions for improvement and

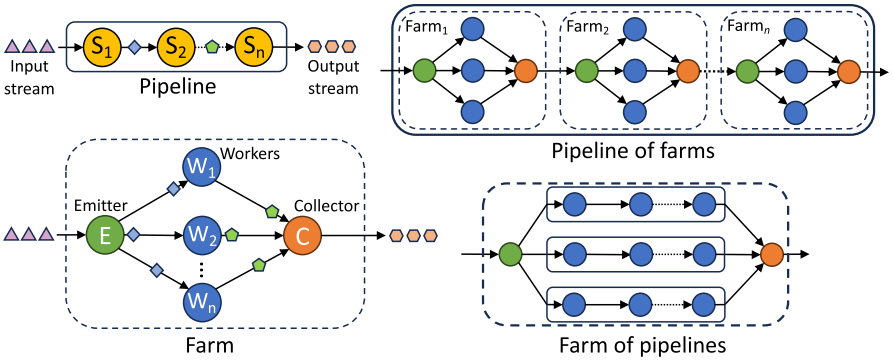


Fig. 1 High-level representation of the pipeline and the farm parallel patterns and two variations/combinations also used in this work: pipeline of farms (pipe-farm) and farm of pipelines (farm-pipe)

future directions of GrPPI. Finally, in Sect. 8, we draw our final conclusions and future work.

2 Background

2.1 Stream parallelism

Pipeline and farm are two of the most widely used parallel patterns in parallel stream processing [1]. Figure 1 shows a representation of both on the left side. In a pipeline, stages receive a stream of data items as input, apply some computation to it, and can generate an output stream with the resulting data. Subsequent stages of the pipeline then compute this resulting data stream, and this process repeats until the end of the pipeline.

Each pipeline stage can be executed concurrently on a different processing unit to increase performance. In this context, throughput and latency are two important metrics to measure performance. In this work, we define throughput as the data the pipeline can process per unit of time. Regarding latency, we use processing-time latency, which is the time that elapses from when a data item is generated at the Source operator (first pipeline stage) to when it is computed in the Sink (last stage). Both metrics are limited by the processing time of the slowest stage. In such cases, exploiting data parallelism by replicating stages can increase pipeline performance. However, improving latency performance usually requires fine-tuning other parameters as well. For example, reducing the size of the buffers between stages implies shorter queues and lower data waiting time inside the pipeline.

Stage replication can be done as represented by the Farm parallel pattern, shown in the bottom-left of Fig. 1. If there is no data dependency, it can be freely replicated. In the presence of data dependencies, such as a shared state, users must handle it to avoid data race and ensure coherency. In this paper, all the benchmark

applications have only stateless operators/stages. Therefore, workers on the farms do not share any data.

In the Farm, an initial stage, often called Emitter, sends the data to the workers, who process it and send it to the Collector. The Emitter sends data using a round-robin strategy by default, but some PPIs also provide distinct data-partitioning strategies. Most PPIs implement data buffers represented by FIFO queues between stages. The FastFlow PPI implements a buffer for each worker or pipeline stage in the Farm. Our handwritten parallel code with ISO C++ threads and OpenMP uses a single blocking shared queue.

On the other hand, TBB implements the work-stealing strategy as a task-scheduling policy [3]. In this case, there is a pool of tasks and worker threads. TBB uses a deque (double-ended queue) to store tasks, and each worker thread has its deque to hold its own tasks. When a worker thread completes its tasks, and its deque becomes empty, it can "steal" tasks from the deques of other worker threads to keep itself busy. Therefore, work-stealing helps in load balancing across threads. Threads with more tasks can give away some of their tasks to other threads, preventing any one thread from being overloaded while others remain idle.

In the other PPIs we evaluate in this work, threads execute the same pipeline stage statically. It reduces the costs related to thread creation, object allocation, and others but adds to the problem of thread idleness in unbalanced pipelines. Load balancing must be done manually, which can be very complex in pipelines with many parallel stages. There are strategies to make load balancing more dynamic through self-adaptive parallelism, for example, but these are not features natively supported by these PPIs [10]. These patterns can be combined in different ways. In this work, we also use the combination pipeline of farms (PF) and Farm of pipelines (FP), shown on the right in Fig. 1.

2.2 GrPPI

Some solutions aim to alleviate the burden of parallel programming, which is a time-consuming and error-prone task. One popular option is to apply algorithmic encapsulation techniques using pattern-based programming models. Parallel patterns allow the implementation of robust, readable, and portable parallel code while abstracting away the complexity of concurrency control mechanisms such as thread management, synchronizations, or data sharing. Intel TBB and FastFlow are two examples of PPIs that support parallel patterns. However, such PPIs do not share the same programming interface and require code rewrites to port a parallel application to other platforms. The GrPPI library [4] was developed to overcome these drawbacks and be a unified, generic abstraction layer between PPIs. It proposes to act as a switch between different parallel programming interfaces. It uses modern C++ features and metaprogramming concepts to provide a compact and generic parallel interface that seeks to hide the complexity of concurrency mechanisms from users. It is also highly modular, allowing easy composition of parallel patterns. Therefore, GrPPI offers users an interface to implement data and stream parallelism with minimal effort for multiple platforms while adding negligible overhead in performance.

Its goal is to make applications independent of the parallel programming framework used underneath, thus providing portable and readable codes [11].

In its latest public release, GrPPI allows running applications with four backends: native (ISO C++ threads), FastFlow, OpenMP, and Intel TBB. Users can switch among backends effortlessly if the application with GrPPI implementation uses dynamic directives. It means that a backend can be chosen after compilation as an execution parameter. However, in stream processing, PPIs often offer specific mechanisms for fine-tuning performance. These mechanisms improve load balancing, reduce latency, apply data order to the stream, optimize resource consumption, and other aspects. In addition, some PPIs have unique mechanisms, such as controlling the number of tokens in TBB. Most of these parameters do not have that much impact on throughput. However, latency is a more sensitive metric and can be excessively high if the application is not adequately tuned. Although GrPPI includes directives for managing many of these parameters, their extent and functionality have not yet further evaluated. We argue that assessing GrPPI with a latency-oriented perspective can show how much it can express parallelism for different stream processing scenarios while maintaining a simple and generic interface.

2.3 SPBench benchmarks

In this work, we create GrPPI benchmarks using the SPBench framework [9] and compare their performance against hand-written stream-parallel implementations of using FastFlow and Intel TBB. SPBench is a framework that allows users to easily prototype benchmarks from real-world applications and assess various parallel programming interfaces for stream processing in C++. Users can create custom benchmarks using different PPIs through a simple and intuitive interface. These benchmarks natively provide most of the performance metrics used in the literature.

Applying stream parallelism to applications in this domain can be very challenging. Commonly, users face sequential applications where operator boundaries and data dependencies are challenging to recognize in the source code. One of the primary purposes of the SPBench framework is to extract the difficult and laborious task of dealing with the application from the users' side so they can focus on parallelism.

Figure 2 shows a simple example of how SPBench works in this sense. On the left is a short example of sequential code that computes the Mandelbrot set. This program can be implemented in parallel using a Farm pattern or three-stage pipeline. However, although it is a short piece of code, identifying the individual operators and the data dependencies among them is challenging. In the traditional form, at the top of the figure, users must write parallel code directly from sequential code. The bottom part of the figure shows the SPBench path. One of the main things we do in SPBench is rewrite sequential applications in a standardized way so that they are presented to users in a simplified interface, highlighting the main elements for exploiting stream parallelism, such as operators and data items. It eliminates much of the complexity of parallelizing these applications. For the sake of space, Fig. 2 does not present the code of the real-world applications SPBench implements, as

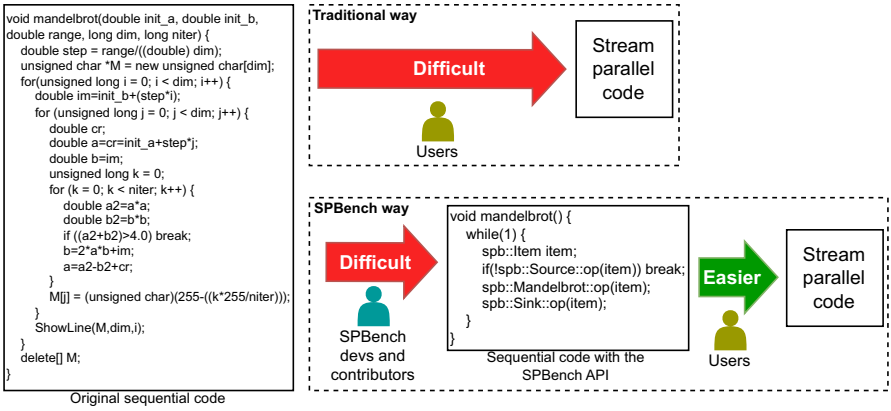


Fig. 2 Users' perspective when writing stream parallel code: traditional vs. SPBench way

they have hundreds or thousands of lines of code. Previous work has already presented and discussed the SPBench applications in more detail [9]. The workloads used in this work were characterized in [12].

SPBench also has an interface that simplifies benchmark management, such as installing application dependencies, creating specific build files, and automating the execution of benchmarks. Furthermore, its benchmarks are highly parameterizable and allow data stream frequency, batch sizing control, and instantiation of multiple data sources. Thanks to the simple and standardized design of the applications in SPBench, we implemented the GrPPI parallel code in a single benchmark. Then, we easily replicated the implementation to create benchmarks with the other applications, modifying only the specific configuration parameters required by different applications. Initially, the framework already provided benchmarks with Intel TBB and FastFLow. We have extended the SPBench framework in this work, and now it also provides GrPPI, OpenMP, and ISO C++ Threads benchmarks.

3 Related work

Our related work is papers that have evaluated the performance of applications implemented with GrPPI.

Table 1 summarizes the related work, and the last row regards this paper. In general, no work has evaluated the latency of applications implemented with GrPPI. Latency is a more sensitive metric than throughput and requires fine-tuning to keep it down for the different PPIs. Thus, evaluating this metric helps show that GrPPI also allows fine-tuning with a generic interface for distinct backends.

The main GrPPI paper [4] evaluated a video application using GrPPI-TBB, GrPPI-THR (ISO C++ threads), and GrPPI-OMP (OpenMP) against hand-written versions of these same PPIs. Although the authors explore various parallelism compositions with pipeline, farm, and stencil, only a single application was used as a benchmark in the experiments. The authors also did not investigate how performance

Table 1 Related work summary table

Related Work	Parallel patterns	Parallel programming interfaces	Performance metrics	Program. metrics	Benchmark applications
[4]	Pipeline + Farm, Pipeline + Stencil	GrPPI(TBB, THR, OMP), TBB, CUDA, ISO C++, OpenMP	Throughput	–	Gaussian blur + Sobel filter benchmark
[5]	Stream-Pool, Window-Farm, Stream-Iterator	GrPPI(TBB, THR, OMP)	Speedup	LOC, CCN	FM-Radio and 3 synthetic bench.: Traveling salesman, "Sensor" and "Image"
[6]	Pipeline-Farm	GrPPI(THR, MPI)	Speedup	LOC, CCN	Mandelbrot + Gaussian Blur
[7]	Pipeline-farm	GrPPI(THR, MPI), Boost-MPI, Spark	Speedup	–	Gaussian blur + Sobel filter, Mandelbrot + Gaussian Blur
[13]	Pipeline-farm	GrPPI(TBB, THR, FF, OMP), OpenMP	Exec. time, Mem. usage, other hardware metrics	–	pHARDI
[11]	Map, reduce, stencil, farm	GrPPI(TBB, THR, OMP, FF), FastFlow	Exec. time	LOC, CNN	Four synthetic bench. with simple math and vector operations
[14]	Pipeline-farm, map, reduce	GrPPI(THR, OMP), PThreads, OpenMP	Exec. time	LOC	From PARSEC: Swaptions, Blacksholes, Streamcluster, and Ferret
[15]	Pipeline-farm	ISO C++ thr., GrPPI(THR, TBB)	Speedup	–	Mandelbrot, Ant colony optimization, Matrix multiplication, and Image convolution
This work	**Farm, Pipeline-farm, Farm-pipeline**	**GrPPI(TBB, THR, OMP, FF), FastFlow, TBB, OpenMP, ISO C++ threads**	**Throughput, Latency, Mem. usage**	**LOC, CCN, PHalstead**	**Lane Detect., Bzip2, Face Recog., and Ferret (PARSEC)**

scales for different parallelism degrees. They did use different problem sizes, however. Also, this is the only paper we found that evaluates performance as average throughput. Although this metric is derived from execution time, we believe it is more representative for the stream processing domain, especially when extended to endless data stream scenarios.

In [5], the authors extend GrPPI to support parallel patterns for data stream applications. They evaluated their work with applications from domains that typically require low latency, such as signal and sensor data processing. However, they used only speedup as a performance metric. Muñoz et al. [6] added MPI as a GrPPI backend, allowing GrPPI applications to run on distributed platforms. They evaluated the speedup of their proposal against GrPPI-THR. In the experiments, they varied the number of distributed nodes and the degree of parallelism within each node. In [7], the authors extend the work from [6] and compare their proposal against a natively implemented version using Boost-MPI. They also compare it against an implementation in Spark, a distributed data stream processing framework. GrPPI-MPI achieved significantly higher speedups than Spark, highlighting the performance benefits of C++ over JVM-based frameworks.

The authors from [13] have implemented parallelism with GrPPI in a real-world MRI application. They exploited the pipeline-farm pattern, where GrPPI-THR and GrPPI-OMP achieved the best speedup results. They also evaluated memory usage, where TBB was the parallel backend of GrPPI that used the least memory, and FastFlow used the most. However, no backend used less memory than the handwritten OpenMP application. In [11], the authors have extended GrPPI to support FastFlow as a parallel backend framework. They tested their solution using four synthetic benchmarks for each parallel pattern ported from FastFlow. They evaluated the execution time of the GrPPI backends against implementations of the benchmarks using native FastFlow. They also evaluated programmability regarding lines of code (LOC) and McCabe's cyclomatic complexity number (CCN), where GrPPI presented similar results to handwritten FastFlow. Although FastFlow benchmarks achieved better or equivalent performance results on most of the parallel patterns evaluated, the GrPPI-TBB backend performed better with the Farm pattern.

Reference [14] evaluated the performance of GrPPI-THR and GrPPI-OMP using four benchmarks from the PARSEC suite. They compared performance against versions of the benchmarks originally implemented in Pthreads and OpenMP. In most cases, GrPPI performed equivalent to the original versions regarding execution time. In [15], the authors propose software refactoring techniques to introduce instances of GrPPI patterns into sequential C++ code semi-automatically. It supports pipeline and Farm parallelism, and they tested it with GrPPI-THR and GrPPI-OMP. Four benchmarks were used to compare the speedup of their solution against manually written versions in ISO C++ threads. They were able to achieve performance equivalent to the baseline benchmark.

Taking an overall look at the related work, we see that GrPPI has proven to be able to add generic parallelism abstractions to several PPIs with a minimal performance penalty. Compared to JVM-based PPIs such as SPark, GrPPI highlighted the performance benefit of using C++. However, as shown in Table 1, most related works evaluated performance regarding execution time/speedup only. None of them

measured latency, an increasingly important metric for real-time processing, which is one of the goals of stream processing. The evaluations considering throughput and memory were also quite limited, not considering different strategies and degrees of parallelism or different stream processing applications. This work evaluates GrPPI, considering latency, throughput, and memory usage. We used four real-world applications and varied the degree and parallel compositions. We also run experiments on a newer and more robust multi-core architecture. Regarding programmability, in addition to lines of code (LOC) and cyclomatic complexity number (CCN), we use Halstead's method for parallel applications (PHalstead) from [16].

4 Experimental methodology

We discuss the methodology used for the experiments in the next section. We evaluate performance regarding throughput, latency, and memory consumption with varying degrees of parallelism. Latency means average processing-time latency, i.e., the average elapsed time between the generation moment of an item at the source operator and its written moment in the output of the sink operator. Throughput means the number of items processed by time unit. The benchmarks we used provide these metrics and the total memory usage. We ran each benchmark three times for the performance evaluation, and the standard deviations are in all the performance charts as error bars.

4.1 Execution environment

All experiments were performed on a computer that has 144 GB of RAM and two Intel(R) Xeon(R) Silver 4210 @ 2.20GHz processors (a total of 20 cores and 40 threads). We enable the `performance` governor. The operating system was Ubuntu 20.04.4 LTS, x86-64, Linux kernel 5.4.0-105-generic, and GCC 9.4.0 using `-O3` flag. We used GrPPI v0.4.0 to implement the GrPPI benchmarks in SPBench and Intel TBB 2020 Update 2 (TBB_INTERFACE_VERSION 11102) for both GrPPI-TBB and handwritten TBB benchmarks. Hand-written code with FastFlow used version 3, while GrPPI used FastFlow 2.2.0. It is the newest supported version.

4.2 Parallelism and tuning configurations

In addition to the parallel patterns and degree of parallelism, most PPIs require several other parameters to be tuned in stream processing applications in order to achieve the desired performance levels. For instance, by reducing the size of communication queues among the stages of a pipeline, the application can present reduced latency and lower memory usage. Although it can add some performance penalty on throughput, the impact is minimal on our test cases. We also try to use simple parallelism strategies and configurations that make a more fair performance comparison of the PPIs. By enabling a blocking behavior, threads that otherwise would be in a busy waiting state can free up computational resources. Blocking synchronizations fit well coarse grain

parallelism with milliseconds tasks or more, which is the case of the benchmarks we use in this work [2]. Blocking synchronizations allow taking advantage of work oversubscription (e.g., for load balancing). The drawback is that it may exhibit performance overheads (also due to OS involvement) [2].

We target tuning applications in our experiments to balance three performance goals: throughput, latency, and efficient resource usage. Therefore, in our FastFlow benchmarks, we enabled an `on-demand + blocking` configuration. In GrPPI, OpenMP, and ISO C++ Threads, we simulate an on-demand behavior by setting queue sizes to 1. None of this applies to the TBB benchmarks since TBB implements a completely different execution model.

In the PARSEC benchmark suite [17], the Ferret application originally implements a six-stage pipeline and each of the four middle stages is a farm. Therefore, we try to implement this same parallel pattern in the Ferret benchmarks in SPBench: $Pipeline(Source, Farm_1(n), Farm_2(n), Farm_3(n), Farm_4(n), Sink)$. As in the original version, we set the same parallelism degree n for all the farms. However, it implies that if only 10 workers are added to each of the four farms of Ferret, the application will already use 40 threads (4 farms × 10 workers) only to run worker stages. Since Ferret stages are highly unbalanced [9], most of those threads would be idle most of the time. Thus, considering the 40-thread multicore architecture we used to run the experiments, with the default configurations, there would not be available resources to run the Ferret pipe farm efficiently with more than 10 workers per farm. Not even 10, since Emitters and Collectors are also run by different threads [12].

Therefore, we chose a high over-subscription methodology for the pipeline-farm experiments with the Ferret benchmarks. We also enabled blocking behavior in the threads, so they free computing resources when they are idle. Thus, in a highly unbalanced pipeline of farms (Ferret's case), we could run each farm with many more threads in total than the number of available processor cores and still potentially achieve performance improvement. We also implemented a farm of pipelines version of Ferret: $Pipeline(Source, Farm(Pipeline(Stage_1, Stage_2, Stage_3, Stage_4)), Sink)$.

4.3 GrPPI benchmarks in SPBench

Listing 1 shows how is a SPBench benchmark with GrPPI. It represents a complete implementation of a Bzip2 benchmark using a GrPPI farm. To implement the `farm_func` function, we rely on the example code that GrPPI provides. The Bzip2 application has three stages: Source, Compress/Decompress, and Sink. Therefore, we implemented a farm where the Source operator acts as an Emitter, Compress represents the replicated farm workers, and Sink is the farm Collector. It was necessary to use the `tbb::task\_scheduler\_init()` (line 2) since we could not control the number of TBB threads directly through GrPPI.

```
1  void farm_func(grppi::dynamic_execution & ex) {
2    tbb::task_scheduler_init init(spb::nthreads);
3    grppi::pipeline(ex,
4      []() std::mutable -> optional<spb::Item> { /* first pipeline stage */
5      spb::Item item;
6      if(!spb::Source::op(item)) {
7        return {};
8      } else { return item; }},
9      grppi::farm(spb::nthreads,                  /* second pipeline stage */
10       [](spb::Item item) {
11       spb::Compress::op(item); // or spb::Decompress::op(item);
12       return item;
13     }),
14     [](spb::Item item){ spb::Sink::op(item); } /* third pipeline stage */
15   );
16 }
17 grppi::dynamic_execution execution_mode(){
18   std::string backend = spb::SPBench::getArg(0); // get the selected backend
19   bool ordering = true;  // The order of items should be guaranteed in the output
20   int queue_size = 1;
21   if (backend == "tbb"){ // configuring the TBB backend
22     int tbb_tokens = spb::nthreads*10;
23     auto tbb_exec = grppi::parallel_execution_tbb(spb::nthreads, ordering);
24     tbb_exec.set_queue_attributes(queue_size, grppi::queue_mode::blocking,
       tbb_tokens);
25     return tbb_exec;
26   } else if ... /* similar for other backends */
27 }
28 int main (int argc, char* argv[]){
29   spb::init_bench(argc, argv); // SPBench initialization routines
30   spb::Metrics::init();        // SPBench performance metrics
31   auto ex = execution_mode();  // Backend selection and configuration
32   farm_func(ex);               // Main computation (stream parallel region)
33   spb::Metrics::stop();        // SPBench performance metrics
34   spb::end_bench();
35   return 0;
36 }
```

Listing 1 Full implementation of a Bzip2 (compress mode) benchmark in SPBench using GrPPI with a single farm.

The function `execution_mode` is used to dynamically select and configure a parallel backend. We enable item sorting to ensure the correctness of the output file. We also use queues of size 1 to reduce latency and activate blocking mode to improve resource utilization. We omit the rest of this function because it is basically the same code used for `parallel_execution_tbb` (lines 21–24) but replicated to the other backends. The selection of Backends is dynamically made at execution time as an execution parameter. The command `spb::SPBench::getArg(0)` gets a custom execution argument from the user. Thus, to run this benchmark with the GrPPI-TBB backend in SPBench, varying from 1 to 40 threads, repeating the execution 3 times, and getting performance metrics, we use the following command:

```
./spbench exec -bench bzip2_grppi_farm -input huge -nthreads 1:40
 -repeat 3 -latency -throughput -memory-usage -user-arg tbb
```

Listing 2 describes the pipeline farm pattern used to implement the original topology of the Ferret application in GrPPI. We will refer to it as “pipe-farm” or “PF” in the following sections. It is similar to the single farm in Listing 1, but we add more farms in sequence.

```
grppi::pipeline(ex,
  [](std::mutable->optional<...>{/*first stage*/},
  grppi::farm(nthreads, []() {/*second stage*/}),
  ...
  grppi::farm(nthreads, []() {/*sec. last stage*/}),
  [](..){/*last stage*/}
);
```

Listing 2 Pipeline of farms implementation of Ferret with GrPPI.

To better show the ability to build composite patterns in GrPPI, we also implemented another variation called farm-pipeline (abbreviated to farm-pipe or FP). This second version has a single farm inside a pipeline, where each farm worker runs another pipeline. Listing 3 shows how we built this implementation in GrPPI.

```
grppi::pipeline(ex,
  [](std::mutable->optional<...>{/*first stage*/},
  grppi::farm(nthreads,
    grppi::pipeline(
      [](...) {/*second stage*/},
      ...
      [](...) {/*second last stage*/}
    )
  ),
  [](...) {/*last stage*/}
);
```

Listing 3 Pipeline-Farm-Pipeline (or simply "farm-pipeline") implementation of Ferret with GrPPI.

After creating the benchmarks using GrPPI and running the experiments, we integrated them into SPBench. SPBench has a self-contained policy. In other words, besides all the benchmarks presented in this work, the framework natively provides all its main library dependencies, including the GrPPI library and its backends. All benchmark source codes and tuning configurations used in this work are publicly available on the SPBench online repository.[1] More details about the applications and other features can be found in the SPBench documentation[2] and also in [9, 12].

5 Experimental results

This section presents the experimental latency and throughput performance, memory usage, and programmability results.

5.1 Performance

GrPPI is a PPI that provides structured parallel programming patterns for stream processing. It allows running an application with the backends OpenMP, TBB,

[1] https://github.com/GMAP/SPBench.
[2] https://spbench-doc.rtfd.io.

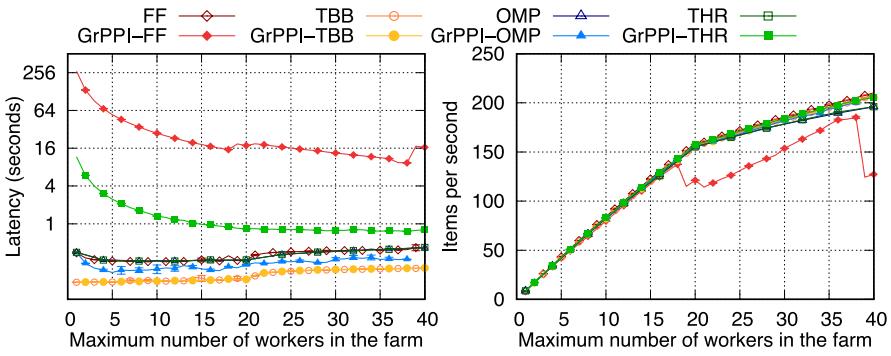


Fig. 3 Latency and throughput performance of GrPPI backends against handwritten parallel implementations with the Bzip2 (compress) benchmark

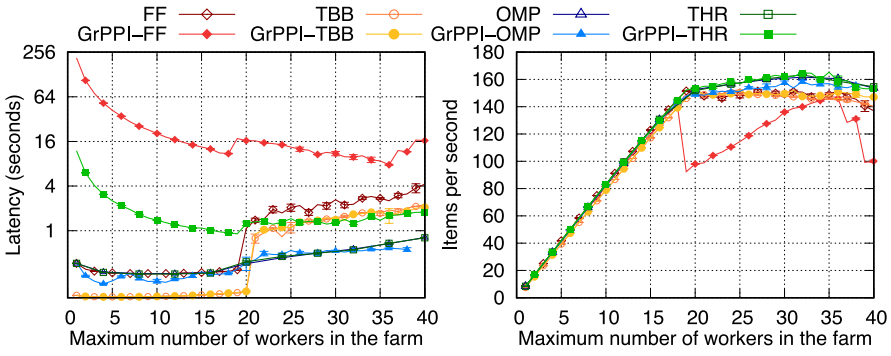


Fig. 4 Latency and throughput performance of GrPPI backends against handwritten parallel implementations with the Lane Detection benchmark

FastFlow, and ISO C++ Threads from a single parallel implementation. In this section, we evaluate GrPPI with all its backends and compare their performance against the handwritten parallel implementations.

5.1.1 Varying the parallelism degree

We measure latency and throughput by varying the degree of parallelism in each farm stage from 1 to 40, the number of cores (with hyper-threading) in the architecture we use. We enabled blocking mode on the PPIs as this allows more efficient use of resources and can improve performance when using hyper-threading, especially in applications that implement a pipeline farm [9].

Figures 3, 4, 5, 6, and 7 present each application's latency (left) and throughput/items per second (right) results. First, all four applications implement a single farm. In the case of Ferret-farm, we unified the internal operators of the application into a single pipeline stage. However, one of the goals of this work is to check whether GrPPI is flexible enough to build different compositions for all

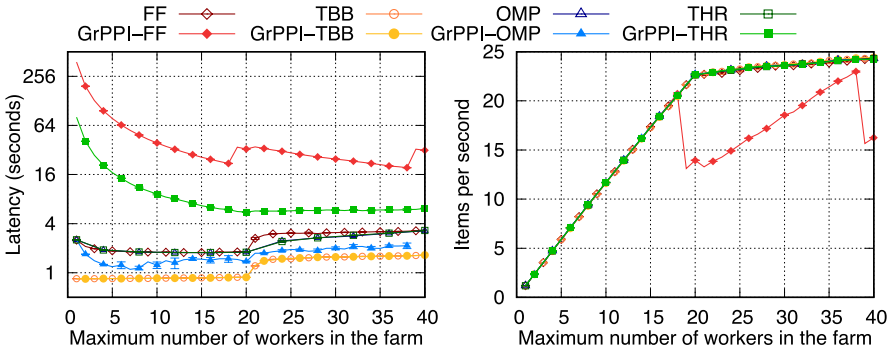


Fig. 5 Latency and throughput performance of GrPPI backends against handwritten parallel implementations with the face recognizer benchmark

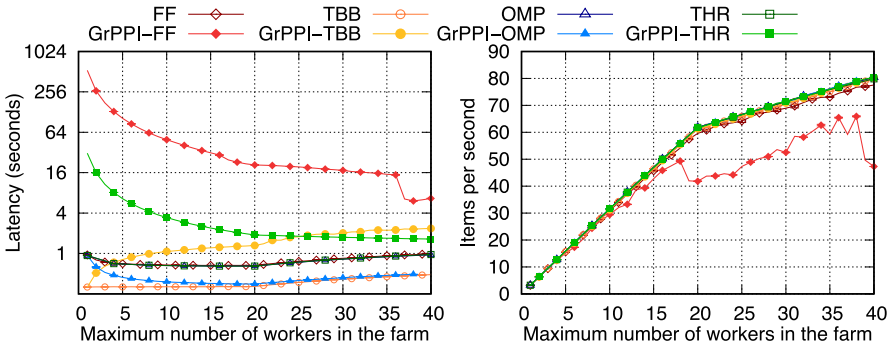


Fig. 6 Latency and throughput performance of GrPPI backends against handwritten parallel implementations with the Ferret (single farm) benchmark

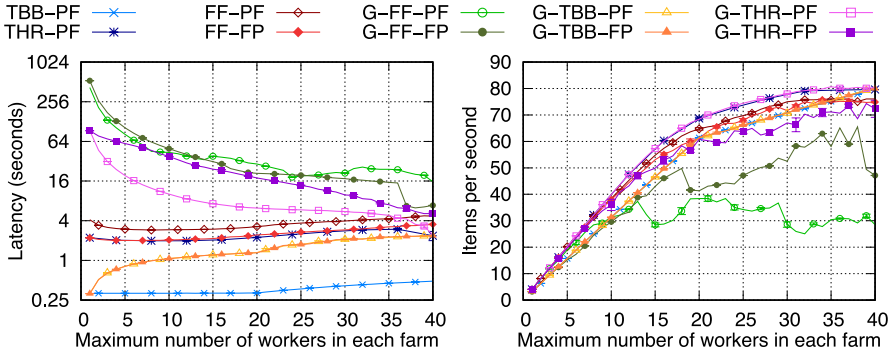


Fig. 7 Latency and throughput performance of GrPPI backends against handwritten parallel implementations with the Ferret benchmark using different compositions of pipelines and farms

the backends from a single generic code. This way, we also implemented Ferret benchmarks using a pipeline of farms (PF) and a farm of pipelines (FP) compositions, as demonstrated in Listings 2 and 3. The pipeline of farms is the original parallel structure of Ferret in the PARSEC suite [17].

The x-axis of the graphs represents the maximum number of workers on each farm. It is fundamental to point out that the actual degree of parallelism of the farms and the number of threads varies according to how each PPI implements it. In fact, one may notice that the GrPPI-OpenMP backend is not run up to 40 parallel workers but only up to 38 instead. It is a matter of internal implementation logic within GrPPI. In FastFlow, for instance, if the user builds a single farm with two workers, it will run four threads: two threads for the workers plus two threads to run the Emitter and Collector stages. It works the same way with GrPPI-FastFlow and GrPPI-Threads. However, GrPPI does not behave the same way for the OpenMP backend. For a two-worker farm in GrPPI-OpenMP, users must explicitly set the farm parallelism degree attribute as four. Therefore, in a single farm, for the same farm parallelism degree, GrPPI-OpenMP runs with two fewer workers than the FastFlow and C++ threads backends. This way, to make the results more comparable, we have shifted GrPPI-OpenMP results by two. It is not a concern with TBB, however, because of its work-stealing task scheduler that behaves entirely differently.

The charts in Figs. 3, 4, 5, and 6 show that GrPPI's throughput with the TBB, OpenMP, and ISO C++ threads backends is equivalent to that of the benchmarks with handwritten code in the single farm benchmarks. In the Lane Detection application (Fig. 4), GrPPI-THR and GrPPI-OMP even achieve better performance using hyper-threading (around 20 workers).

A similar behavior occurs in Fig. 3 with the Bzip2 benchmark, where GrPPI backends (not including GrPPI-FF) perform better than handwritten OpenMP and C++ Threads. With Face Recognizer and Ferret (farm) benchmarks, in Figs. 5 and 6, the throughput performance of GrPPI is equivalent to or better than the handwritten benchmarks. In the case of GrPPI-FF, it achieves throughput comparable to the other PPI with lower degrees of parallelism. Still, the inability to enable blocking mode in GrPPI's FastFlow knocks down performance when using hyper-threading above 20 workers.

Regarding latency, the results of PPIs vary widely. In addition to the inability to enable blocking mode in GrPPI-FF, it is also not possible to enable on-demand mode or reduce the queue size, which could bring a similar effect. Therefore, GrPPI-FF has an unlimited buffer between stages that stores many items simultaneously. These items wait a long in the buffers/queues until the other stages can process them. It incurs a significant increase in latency for all the evaluated applications.

Although GrPPI-THR has the best throughput performance, it has the second-worst latency performance overall. However, it manages to have lower latency than the TBB benchmarks in Lane Detection and Ferret when using hyper-threading. GrPPI-TBB has comparable latency to the handwritten TBB in all applications except in the Ferret (Figs. 6 and 7). In this case, it presents an increasing latency from the beginning, an unexpected behavior. Ferret differs from the other applications because it does not require item order, and how GrPPI implements it in TBB

may explain this difference. GrPPI-OMP presented a similar and even better latency than handwritten OpenMP and C++ Threads in Lane Detection over 20 workers.

Figure 7 presents the performance results for the pipeline with multiple farms (PF) and farm of pipelines (FP) compositions. We only present the result for the Ferret application, which is the application that originally implements a farm pipeline [17]. The GrPPI-OMP PF and FP parallel implementations did not work. Although they start running without errors, threads do not work and enter an infinite waiting. Thus, we do not present the results for these implementations. The highest throughput in the results in Fig. 7 was achieved by ISO C++ Threads (THR-PF) and GrPPI-THR-PF at all degrees of parallelism, followed closely by the handwritten version of FastFlow. A pipeline of farms in TBB behaves very similarly to a single farm in an application with no ordering requirement (this case) since a TBB thread can avoid buffering and process an item from the beginning to the end of the pipeline if resources are available.

TBB is the PPI that presents the best latency results in most test cases. In most benchmarks, handwritten TBB's latency is about half of the second-best result when running less than 20 parallel workers. TBB uses a dynamic task scheduler with a work-stealing policy [3] that benefits from the type of applications we use in these experiments. In the other PPIs, each thread runs the same computation over different data items, i.e., a thread statically runs a single pipeline stage all the time. In this case, data items move from one stage to the next through queues/buffers, which can add extra waiting time, increasing latency. In TBB, threads can run different tasks. These tasks are placed in each thread's double-ended queue (deque). Also, tasks can be chained to subsequent tasks. For instance, if a specific task involves applying the computation of a pipeline stage to a single data item, the return of this task is a function call for the next task, which can be running the code of the next pipeline stage over the data item, and so on. Therefore, a single TBB worker thread can process a data item through the entire pipeline, which means that items stay shorter in the inter-stage buffers, reducing average latency. However, some factors can prevent a task from being executed, such as a data ordering constraint or if no computing resource is available. In this case, the thread may try to steal and execute a task from the back of another thread's deck. After processing the stolen task, the thread returns it to its respective place. This work-stealing can also happen if the thread empties all its deque tasks.

Bzip2 and Ferret are applications that pose no significant challenges for the TBB. In the case of Bzip2, its workload is relatively stable during computation, meaning that data items have similar computational costs. This generates less clutter in the pipeline. All the workloads used here have been properly characterized in previous work [12]. Despite having a very unstable workload, Ferret has no restrictions on the order of items. In other words, in both benchmarks, the TBB worker threads benefit from these aspects of the workload. When Ferret-TBB starts using hyper-threading (more than 20 parallel workers), latency does not increase as much because there is only competition for computational resources. In the case of Bzip2, this competition for resources causes extra clutter of items, and latency jumps slightly with more than 20 parallel workers. In Face Recognizer, when hyper-threading is used, TBB latency practically doubles. This is because this application has a very unstable workload,

as it depends on detecting and recognizing faces in different video frames. It is also necessary to guarantee the order of the frames in the output video. In Lane detection, the scenario is similar, but the latency increases by almost 10x when using more than 20 parallel workers with hyper-threading. This big difference occurs because Lane Detection processes many more video frames and processes them faster than Face Recognizer [12]. In addition, the computational cost of each frame fluctuates even more. The result of these factors is a large item clutter before the last stage of the pipeline. This implies that TBB threads cannot process end-to-end data items in the pipeline at once and need to switch contexts to perform other tasks. This combination of factors causes this large step in TBB's Lane Detection latency when using more threads than physical processor cores. The overhead caused by increased context switching potentially negatively affects the overall performance as well. This item clutter problem also similarly impacts FastFlow, but not as much in the OpenMP and C++ Threads implementations. Since our handwritten C++ Threads and OpenMP benchmarks use the same data structures for buffers and the same ordering algorithm, how FastFlow implements it may explain this difference in the latency performance.

In a pipeline of farms, the inability to optimize FastFlow code in GrPPI is a critical factor for both throughput and latency. If on-demand mode is not enabled, it adds multiple unlimited queues/buffers in the pipeline, increasing latency. Without enabling blocking mode, idle workers are in a busy wait state and do not free up resources. This combination causes a further load imbalance in this application. Since Ferret-PF has a pipeline with four farms and the architecture has 40 threads, the expectation is that the pipeline of farms in non-blocking mode will have a significant drop in performance above ten workers per farm. On the other hand, the farm-pipeline pattern avoids the additional collectors/emitters between stages that there would be in a pipeline farm. This performance difference is because a lower number of queues is required for a pipe of farms, where all works in a farm share a single queue. Consequently, the number of queues is independent of the farm multiplicity. Thus, it requires fewer threads and has fewer shared queues, leading to better load balancing and mitigating the performance impact.

Except for the TBB benchmarks, all the PPIs considerably increased latency with pipeline-farm implementations. The difference between TBB and FastFlow in these situations has been extensively discussed in detail in previous work [9]. Despite the difference in throughput between GrPPI-FF-FP and GrPPI-FF-PF, the two strategies behaved similarly, showing the highest latencies. However, with GrPPI-THR, the pipeline of farms (PF) composition presented far better results than the FP composition. After all, it achieved the best throughputs and reduced latency to the same level as GrPPI-TBB with 40 workers.

5.1.2 Varying the computational cost of the workload

The Lane Detection benchmark is the SPBench workload that poses the most unique challenges. The computational cost of each video frame varies according to the number of lanes that can be detected in the input video. Therefore, intersections, road bends/curves, nearby vehicles, and other elements greatly influence the

(a) 360p resolution (640 × 360) (b) 720p resolution (1280 × 720)

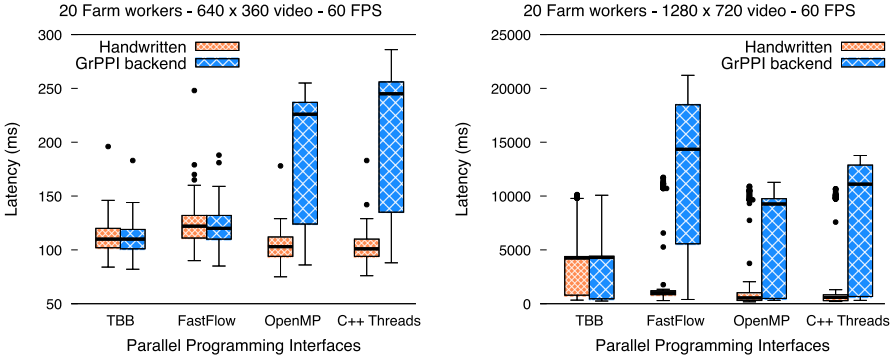
Fig. 8 Output of Lane Detection benchmark with two different input video resolutions

computational cost of each frame. In addition, this application needs to guarantee the correct order of the frames processed in the output. Frames with different computational costs mean that parallel workers process these frames at different speeds, which creates a major frame clutter problem. This ordering cost increases the processing time for each frame, increasing latency and reducing throughput. However, Lane Detection is a latency-critical application in real-world applications. For instance, if it is running in a self-driven vehicle, milliseconds can make the difference in avoiding an accident on the road.

Vehicles can include camera sensors of different resolutions. Naturally, higher resolutions can improve the quality of the result, but they require more processing power from the system. In addition to the cost of processing more pixels, a higher resolution also makes it possible to identify elements in the image more precisely, adding to the overall cost of computing each frame. Figure 8 shows a frame from Lane Detection's output video with two different resolutions. With 360p resolution (Fig. 8a), the application can only identify the lane where the vehicle is, while with 720p resolution (Fig. 8b), the application can identify all the lanes on the road. Therefore, the processing cost is increased by the need to process more pixels and by processing these additional lanes.

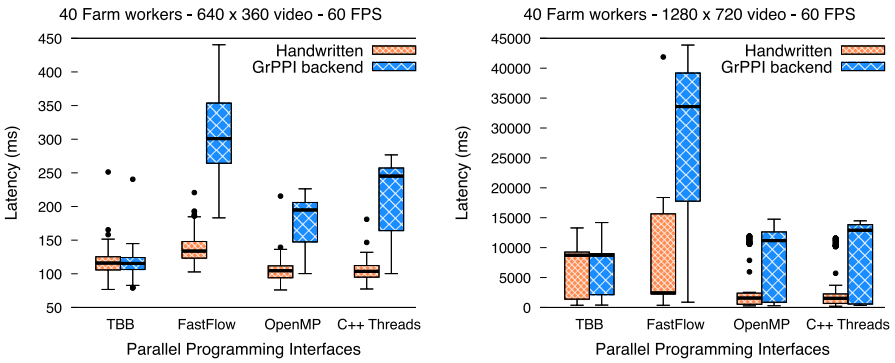
In [4], the authors evaluated GrPPI with one video application and different frame resolutions, but they only evaluated the average throughput and used a synthetic benchmark. Therefore, we evaluate GrPPI's latency when running the Lane Detection benchmark at two different input video resolutions (360p and 720p). Typically, video sensors in autonomous vehicles capture 60 frames per second (FPS) [18]. Therefore, to simulate a more realistic scenario, we also added the `--frequency 60` parameter when running the benchmark in SPBench, which limits the input stream to 60 FPS in this case.

Figures 9 and 10 show the latency results in more detail using dispersion diagrams. The diagrams use the box-and-whisker representation, where the 1st, 2nd, and 3rd quartiles are the dataset's 25th, 50th, and 75th percentiles. The whiskers are based on the 1.5 interquartile range (IQR) value [19], and the diagrams include outliers. The samples in the dataset are the latency of the frames measured every 250 ms during the execution of the benchmarks. We avoided increasing the number of samples to reduce overheads over the application's performance. Therefore, as each



(a) 360p input frames. (b) 720p input frames.

Fig. 9 Latency (250ms sample) of Lane Detection benchmarks running with 60 FPS in the input stream and a maximum of 20 parallel workers on a single-farm configuration



(a) 360p input frames. (b) 720p input frames.

Fig. 10 Latency (250 ms sample) of Lane Detection benchmarks running with 60 FPS in the input stream and a maximum of 40 parallel workers on a single-farm configuration

sample can represent different frames when comparing two results, single outliers are not significant for analysis.

Figure 9 shows the results of running the Lane Detection benchmark with 20 parallel workers. The architecture of the execution environment has 20 physical cores and 40 threads, so this scenario has little competition for computing resources. With 360p input video, the benchmarks with handwritten parallel code showed similar latency results (Fig. 9a). It contrasts with the difference shown in Fig. 4 in Sect. 5.1.1. There, the input FPS was defined by the hardware speed and the application's limitations. Here, we can see that in a more realistic scenario, with 60 FPS, FastFlow can sustain a low latency both in handwritten code and with GrPPI-FastFlow. So, the workers in this application are processing items fast enough that they do not queue for long. This is not the case of GrPPI with OpenMP and C++ Threads backends, which more than double the latency of handwritten parallel code. If we compare it with the results of Fig. 4 again, we can see that with 20 parallel workers,

the latency of both backends should be equal to or less than that of handwritten FastFlow when decreasing the FPS. Therefore, this indicates that GrPPI is either generating poorly optimized code for OpenMP and ISO C++ Threads or is adding extra structures that cause a large overhead compared to handwritten parallel code.

Many differences occur when the video resolution is increased. As mentioned earlier, in addition to the application processing more pixels, a sharper image also makes it possible to identify additional elements that add to the workload. Although FastFlow, OpenMP, and C++ Threads present low latency in Fig. 9b, this is due to the limited size of the queues. In other words, the pipeline cannot ingest more frames, and these frames would have to be buffered elsewhere, or they would be dropped. If they were buffered elsewhere, the latency of these three PPIs should be similar to that of the TBB in this case. The main difference here is GrPPI-FF, which does not limit the size of the queues and a higher load on each worker on the farm. In this case, the source can still load these larger frames into the queues faster than the workers can process them, even though the input is limited to 60 FPS. Furthermore, even when there is no prior buffering or frame loss, which is the case with the TBB benchmark, the latency would still be very high, as the median is almost 5 s. This is unacceptable for latency-critical applications. Therefore, it would be necessary to either reduce the input FPS or the video resolution or improve the hardware performance for this application to run in realistic scenarios.

Figure 10 shows the same experiment but increases the number of parallel workers to 40, which is the number of threads in the architecture. The main difference compared to 20 parallel workers is the latency of GrPPI-FastFlow, which more than doubles with both resolutions. So, instead of reducing latency as the number of workers increases, as seen in experiments without FPS limits (Fig. 4), the latency increases. It happens again due to the impossibility of reducing queue sizes in GrPPI-FF and activating blocking mode, which would cause idle threads to release resources. Therefore, 40 workers hold more frames for longer but have to compete with other threads for computational resources, even when those threads are not processing other frames. Despite this, even when blocking mode is activated when the frame resolution is increased, threads spend more time working on a single frame, intensifying competition for resources. It can be seen in the larger distribution of high latency items for the handwritten FastFlow in Fig. 10a compared to the same result in Fig. 9b.

5.1.3 Varying the input data frequency

Although cameras in autonomous vehicles typically capture 60 frames per second [18], there are several variations and types of camera sensors for vehicles of this type that work with lower capture rates, reaching up to just 10 FPS [20]. Additionally, this frame rate may fluctuate. For example, if the vehicle is stationary or traveling very slowly in a straight line, many redundant frames may be discarded before the main processing. This type of technique can be used to improve system efficiency [21]. In this section, we evaluate the latency of the benchmarks using a frames-per-second rate that fluctuates between 10 and 60 FPS in a sinusoidal fashion. To do this, we added the following execution parameter to SPBench:

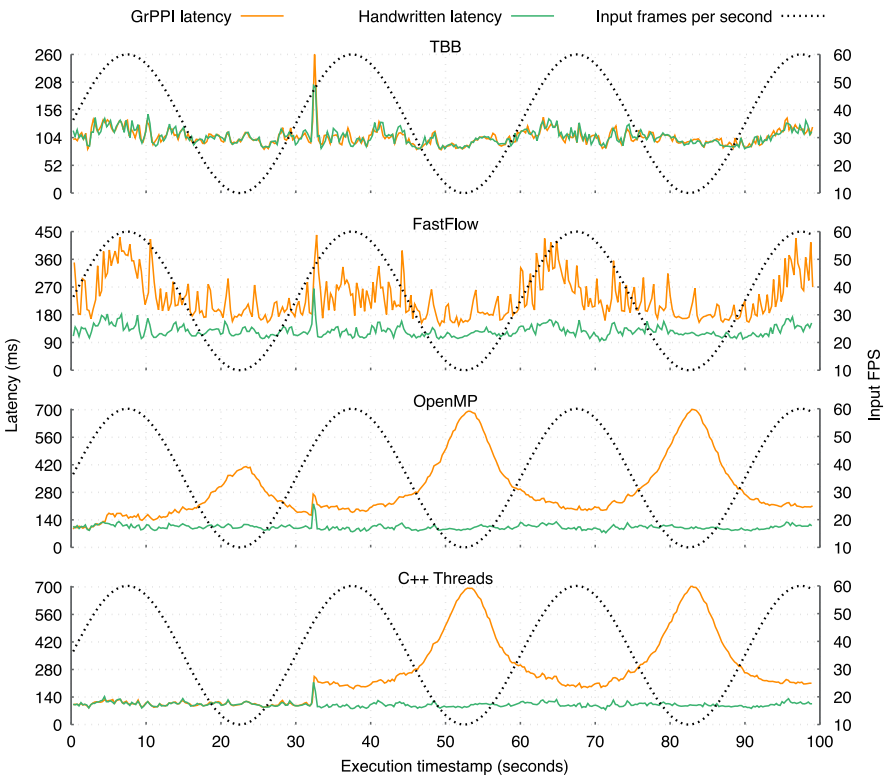


Fig. 11 Latency of handwritten vs. GrPPI parallel code for the Lane Detection benchmark running with a maximum of 40 parallel workers in a single farm configuration and varying the number of video frames available per second at source from 10 to 60

`--freq-pattern wave,10,10,60` . This command automatically creates a frequency pattern in the input stream in a sine wave format, which has a wavelength of 10 s, with the trough at 10 and the crest at 60 items (frames) per second [12].

Figure 11 compares the latency results of each GrPPI backend against handwritten parallel code. The benchmarks were run with 40 parallel workers on the farm and 360p resolution in the input video. Other configurations do not add significant new conclusions. We used SPBench's monitoring functionality to monitor latency and throughput during the execution of the benchmarks. Every 250 ms, a latency/throughput sample was taken, and each sample represents an average of the frames processed during the last 250 ms. It is all done automatically by SPBench and can be configured using a single execution parameter. This type of metric is usually called instantaneous latency/throughput. It attenuates outliers and helps to visualize the experimental results better.

Regarding the handwritten parallel code, all the PPIs showed similar results and a latency of around 120 ms throughout execution. A noticeable latency spike occurs around 32 s into the execution (x-axis). It is because there is a short sequence of

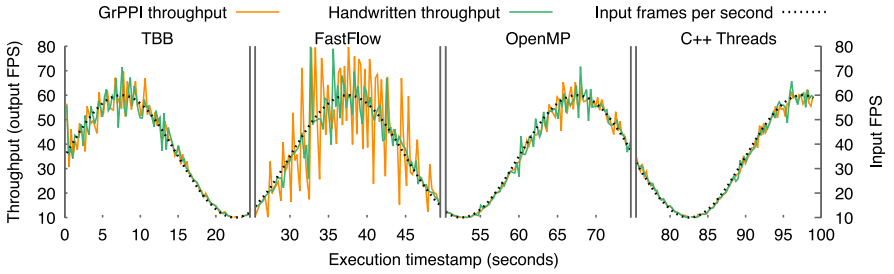


Fig. 12 Throughput of handwritten vs. GrPPI parallel code for the Lane Detection benchmark running with a maximum of 40 parallel workers in a single farm configuration and varying the number of video frames available per second at source from 10 to 60

frames in the input video where the road changes and several lanes are detected. These frames impose a greater workload on the workers. In this experiment, we configured the sine wave so that this spike would occur when the FPS is in the rising phase. It generates an additional stress point and allows us to test how the PPIs deal with critical load spikes.

All the handwritten benchmarks and GrPPI-TBB and GrPPI-FF quickly reduced the latency after the load spike. It is not valid for GrPPI with the OpenMP and ISO C++ Threads backends. In the case of GrPPI-OMP, it did not even manage to keep latency low after the crest of the first wave, even before the spike. Since we believe that GrPPI implements the same inter-stage buffer/queue mechanism for both OpenMP and C++ Threads, this higher initial latency of GrPPI-OMP is probably linked to differences in how it allocates work to threads. With C++ Threads, the thread executing the source operator has to compete with the farm workers for computing resources. Nevertheless, if GrPPI is run with the OpenMP backend using the same parallelism settings, it will create two fewer workers on the farm than ISO C++ Threads. So, the Source on GrPPI-OMP has more computational resources and unnecessarily fills the worker queues. When running GrPPI-OMP with no FPS limits (Fig. 4, the on-demand mechanism seems to work, since it presents the best latency results when running with the maximum number of workers in parallel. Therefore, there is some bad optimization causing this unexpected behavior with floating input FPS, even with the peak FPS being about three times lower in this case.

Regarding the ISO C++ Threads GrPPI backend, the increase in latency only occurs after the heavy workload spike. The application fills the buffers on high-FPS phases, and this causes an increase in latency in the future, reducing after low-FPS phases. Based solely on the results of Sect. 5.1.1, we hypothesize that the high latency could result from a flaw in the on-demand mechanism, similar to the problem with GrPPI-FF. However, the latency behavior in the experiments with reduced FPS in Sect. 5.1.2, and with fluctuating FPS in this section, shows that GrPPI-FF has very similar behavior to GrPPI-OMP. It leads to the conclusion that the problem involves other aspects, such as data contention caused by poorly optimized lock mechanisms. In the case of GrPPI-FF, the average latency is about double compared

to the handwritten FastFlow code, but both show similar behavior throughout execution. Therefore, the only issue in this case is the lack of a functional on-demand mechanism for GrPPI-FF.

Figure 12 compares the output FPS with the input FPS of the benchmarks. Despite large variations in latency and possibly poorly optimized parallel code, all the PPIs managed to sustain the same level of throughput. The greater variation in throughput shown by GrPPI-FF is again a consequence of the impossibility of activating blocking and on-demand mode. The presence of output FPS spikes higher than the input FPS means that at some point in the pipeline, items are buffered and then processed in a shorter interval in the future. For instance, this can result from out-of-order items waiting for the correct item.

5.2 Memory usage

The evaluation of memory usage is a crucial aspect of developing and optimizing stream processing applications. To meet real-time processing demands, SP applications typically require handling large volumes of data and running tasks concurrently to achieve high throughput and lower latency. By evaluating memory consumption, developers can identify potential bottlenecks and optimize their code for better performance. It is also a relevant factor in the scalability of stream processing applications. As the amount of data being processed increases, so does the memory required to handle that data. By evaluating memory consumption, developers can check whether their application can scale effectively as the volume of input data increases.

In this section, we use the SPBench benchmarks to evaluate the memory usage of TBB, FastFlow, OpenMP, ISO C++ Threads, and GrPPI in stream parallelism. We get the memory usage from the SPBench `memory-usage` metric. This metric returns the total memory used by a benchmark during its execution (peak resident set size). We ran the benchmarks with a parallelism degree of 10, 20, 30, and 40. We ran the benchmarks without a limit on the frequency of data item ingestion, the same parameters used in the experiments in Sect. 5.1.1. Figure 13 shows the results of the benchmarks with a single farm. Note that the *y*-axis is in a logarithmic scale for better data visualization. In most cases, PPIs demanded similar amounts of memory in each application. The apparent exception was GrPPI-FF, where unlimited queues/buffers loaded, at once, all data into memory. It occurs due to the inability of GrPPI to adjust the size of FastFlow queues and the FastFlow developers' decision not to set a lower default boundary. Another contribution to this effect derives from the GrPPI's strategy of using value-oriented bounded queues instead of pointer-oriented ones. That avoids excessive allocation/deallocation at the price of preallocating more memory.

GrPPI-THR (ISO C++ Threads backend) was the second case that used memory most in all applications except Lane Detection and Bzip2 when using more than 20 parallel workers. The difference from the other PPIs is more prominent when using fewer parallel workers on the farm. However, we can see that this result is directly linked to the high latencies that this backend presented in the performance

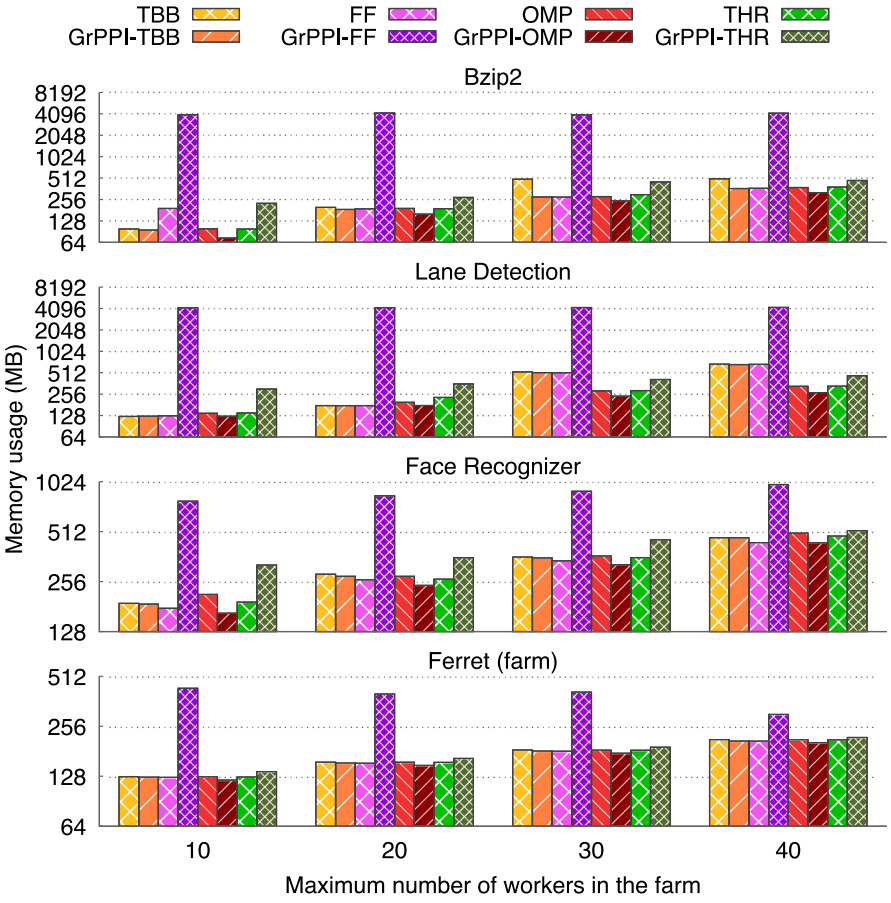


Fig. 13 Total memory consumption of benchmarks with a single farm

evaluation. It may indicate the presence of bad optimizations in GrPPI or the inability to enable the on-demand mode for this backend, as occurs with GrPPI-FastFlow. However, when running the Lane Detection benchmark with 20 and 40 parallel workers, GrPPI-THR used less memory than FastFlow and TBB, both handwritten and GrPPI versions. These results are tightly linked to the latency results presented in Fig. 4. The PPI that used the least memory in the big picture was GrPPI-OpenMP, followed very closely by handwritten FastFlow in most test cases. It may not seem like it because of the logarithmic scale of the chart's y-axis, but GrPPI-OMP uses about 18% less memory than the handwritten version in most cases. We can correlate this difference to the good latency results of GrPPI-OMP from Sect. 5.1.1, which were run under the same parameters. We expected similar results between the OpenMP and ISO C++ Threads backends of GrPPI, as it happens with the handwritten benchmarks. However, GrPPI does not seem to share the structures common between these two PPIs.

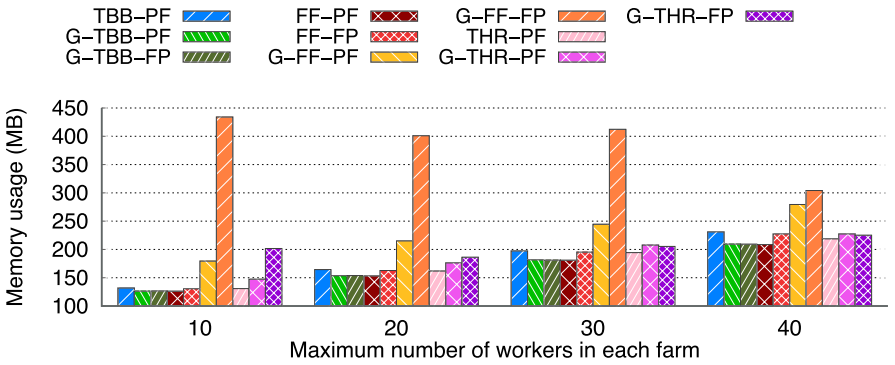


Fig. 14 Total memory consumption of Ferret benchmarks using pipeline-farm (PF) and farm-pipeline (FP) compositions

Figure 14 shows the memory usage of the pipe-farm (PF) and farm-pipe (FP) compositions. GrPPI-FF again shows high memory usage. However, this is only in the farm-pipe composition and not so much in the pipe-farm. However, the memory usage of GrPPI-FF-FP with 40 workers is reduced. An explanation for this is that resources are very tightly contested with blocking mode disabled, and more intensive stages (the last ones) get more processing priority. Thus, such a lack of resources can lead to the inability of the first stage (emitter) to send enough items to fill the queues. That is because there is a bottleneck reduction in the more costly task. Therefore, it increases the bottleneck at the emitter. Although GrPPI-FF-PF has more intermediate queues because of its multiple farms, each farm runs an emitter and a sink stage. The lack of computing resources makes it worse in this case since it has to run these extra stages. It may slow down the fulfillment of the queues, keeping the memory usage lower than in the farm-pipe case.

GrPPI-THR-FP also uses more memory with ten workers, but we can see that this is directly linked to the latencies shown in Fig. 7. The same is true for GrPPI-THR with a single farm (Fig. 13). Explaining this would require further investigation into how GrPPI implements the communication mechanisms between the stages using ISO C++ threads.

The handwritten FastFlow and GrPPI-TBB achieved the lowest memory usage with a pipeline of farms. In the single farm benchmarks, handwritten TBB and GrPPI-TBB got similar results. That is true for memory usage but also for latency and throughput performance. However, GrPPI-TBB presented such an unexpected behavior regarding latency when using a pipeline of farms composition. We are not sure what may cause that behavior, but we believe it also impacts the memory usage of this benchmark. On the other hand, our highly optimized handwritten FastFlow managed to use over 10% less memory than the handwritten TBB benchmark. The benefits of TBB's work-stealing execution model 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 [3]. We believe this contention may explain the extra memory TBB demanded compared to FastFlow.

5.3 Programmability evaluation

Parallel programming is usually evaluated in terms of execution time and speedup. In the stream processing domain, latency and throughput are also important performance metrics. [1] says that a PPI should balance three properties: performance, portability, and programmability/productivity. However, programmer productivity is a critical factor in parallel programming that is usually not addressed. It is directly related to the lack of methods and tools that support parallel programming and the difficulty of performing experiments on humans [16, 22]. Thus, code metrics such as lines of code (LOC) or cyclomatic complexity number (CCN) are the most productive evaluations of parallel programming. Most related works use such metrics, as discussed in Sect. 3. However, these metrics may be inaccurate and lead users to wrong assumptions [23].

If we disregard parallel programming, there are well-known code-based methods in the literature for evaluating the programmability cost of applications. One of the most known is Halstead's method, a code-based metric used to measure the complexity of a program [24]. The basis of the method is the observation that the complexity of a program is related to the number of unique operators and operands used in it. The application of the Halstead method is predicting the effort required to develop or maintain a software program and to estimate the number of bugs that may be present in the program [23]. We also can use the method to compare the complexity of different programs and to identify sections of code that may be particularly difficult to understand or maintain.

Halstead's method, however, does not correctly address parallel programming since it cannot recognize specific code tokens from most of the PPIs. Andrade et al. [23] overcame this problem and adapted Halstead's method to support some PPIs, including the ones used in this work. This method is based on tokens of code, classified as operators or operands, and proposes a series of measures, including estimated development time. [23] added specific tokens from different PPIs and developed a tool called PHalstead[3] from this.

Here, we evaluate the programmability/productivity of the PPIs using the most common metrics found in the literature and Halstead's method leveraged by the PHaltead tool. We measured LOC and CCN using the Lizard 1.17.10 tool.[4] Figure 15 shows the results of the single farm plus pipeline of farms (PF) implementations. For the LOC and cyclomatic complexity, we also added the results of the sequential applications for baseline comparison.

Regarding GrPPI, we consider two versions: "GrPPI-static" is a more straightforward implementation that invokes the executor of a specific backend statically within the code; "GrPPI-dynamic" is an implementation that allows switching between the four backends dynamically at execution time. This second version requires the addition of the backend selection mechanism represented in line 17 of listing 1. We consider these two versions because, although dynamic

[3] https://github.com/GMAP/phalstead.
[4] https://github.com/terryyin/lizard.

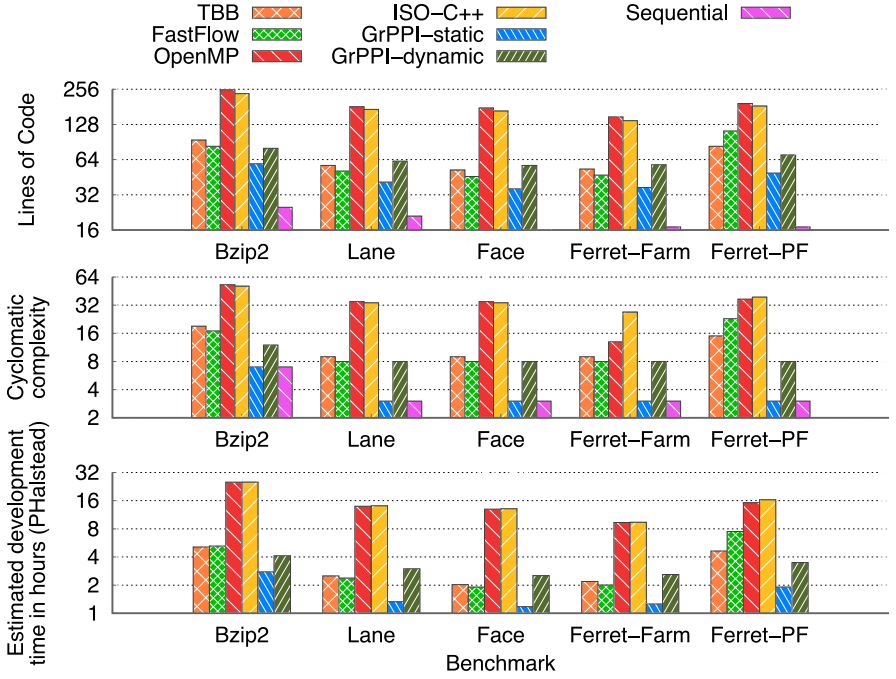


Fig. 15 Number of lines of code, cyclomatic complexity, and estimated development time (PHalstead [23]) of the PPIs with each benchmark

backend selection is a valuable feature of GrPPI, its use is a user option and not a requirement.

GrPPI-dynamic is similar to TBB and FastFlow in parallel implementations with a single farm. However, we can see that GrPPI shows better results in pipe-farm (PF) implementations, where the complexity of programming with FastFlow increases considerably. Concerning cyclomatic complexity, GrPPI-static presented equivalent results to the sequential application. On the other hand, PHalstead estimated a longer development time for GrPPI-dynamic with Lane Detection, Face Recognizer, and Ferret-Farm. The addition of the backend switching mechanism gives this extra cost. In Bzip2, which has two execution modes (compress and decompress), this cost is diluted, and GrPPI-dynamic can maintain a lower development time than TBB and FastFlow. In Ferret-PF, this is due to the significant increase in implementation complexity with TBB and FastFlow.

The two PPIs that showed the worst programmability results were OpenMP and ISO C++ Threads (THR). These PPIs do not provide structured parallel patterns, nor do they abstract away the concurrency control mechanisms. Therefore, they require much more programming effort. Since both PPIs share some structures in the SPBench benchmarks, such as communication queues, they have similar results in all three programmability metrics.

6 Critical analysis of experimental results

The performance results of GrPPI varied significantly among the four backends. Since it provides no mechanism to configure FastFlow queues and scheduling policies, this greatly limited the performance of this backend. In addition, it only supports an older version of the FastFlow library (2.2.0), which also limits the users' option to use newer features of FastFlow, such as the `BOUNDED_BUFFER` compiling macros. Therefore, enabling support for on-demand and blocking modes in GrPPI-FF or making it support more recent versions of FastFlow could be an important improvement for GrPPI in the context of stream processing.

GrPPI-TBB performed as well as the handwritten TBB in most cases. However, it presented unexpected increased latency when running the Ferret benchmarks. Due to GrPPI's internal implementations and lack of documentation, we could not find why it shows this behavior. We tested different configurations and all the combinations of mechanisms available with this application. Although the main difference between Ferret and other applications from a stream processing perspective is that it does not need to sort the items, enabling or not enabling sorting mechanisms in GrPPI doesn't significantly affect latency. Further investigation is needed to understand this issue.

GrPPI-THR (ISO C++ Threads) did not present the throughput loss faced by the GrPPI-FF backend when using hyper-threading. However, it presented the second-worst latency, probably due to issues with the GrPPI's fine-tuning mechanisms. On the other hand, GrPPI-OMP presented a surprisingly good performance. In most cases, it achieved lower latency than handwritten OpenMP, FastFLow, and ISO C++ threads. In the Lane Detection benchmark, GrPPI-OMP was the PPI that presented the lowest latency when using hyper-threading. With the Ferret (farm) benchmark, it presented equivalent latency to handwritten TBB at high parallelism degrees. Although GrPPI-OMP seems a good option, we could not run it with the pipeline of farms composition. Our preliminary experiments observed that it creates more and an arbitrary number of threads than expected. For instance, it creates around 17 threads to run the Ferret pipe-farm with two workers per farm. With little parallelism, it already used all the available threads of the architecture it was running. Therefore, GrPPI-OMP needs to be improved in this direction so that more complex stream parallelism patterns can be effectively used.

We could find most of the aforementioned GrPPI limitations because this work is the first to evaluate it under a latency perspective, as we showed in the related work Sect. 3. All its backends presented performance or behavior issues in some of our experiments. The main issues are higher latency for GrPPI-FF and GrPPI-THR, throughput drop for GrPPI-FF, unexpected high latency for GrPPI-TBB in Ferret benchmarks, limited parallelism scaling with GrPPI-OMP in more complex compositions, and poor throughput performance and high memory usage of GrPPI-FF with a pipeline of farms implementation.

Therefore, while **GrPPI-THR** can deliver high throughput and does not require installing additional libraries, it delivers poor latency performance. While

GrPPI-OMP delivers decent latency and good throughput performance, it presents unpredictable behavior in complex parallel compositions. While **GrPPI-TBB** presents good latency and throughput performance, it presents unexpected performance behavior in some cases and may demand more effort to set it up. In summary, **GrPPI-FF** did not show any advantage over the other backends at any point in our evaluation and presents many limitations that incur high memory usage.

Regarding programmability, OpenMP and ISO C++ threads demand the most programming effort, as expected, since they do not provide structured parallel patterns or abstract concurrency control mechanisms. In single-farm benchmarks, TBB and FastFlow require similar programming effort in our results. GrPPI, as expected, shows the best results in this regard. One of the main features of GrPPI is that it allows the switching of backends from a single generic implementation. Meanwhile, our analysis shows that implementing these switching mechanisms can raise the programmability cost of an application with GrPPI to the same levels as a hand-written implementation with FastFLow and TBB in some cases. For the pipeline of farms composition, the three programmability metrics we used pointed out that handwritten FastFlow required less than twice the programming effort of TBB. However, while the productivity/programmability evaluation can somewhat address the effort of writing parallel code, it cannot address the difficulty of finding parallelism configurations that perform decently. In the pipe-farm case, we argue that the effort required to implement a FastFlow version that achieves performance competitive with TBB demands much more effort than the metrics point out. This goes beyond the complexity of the code itself but also encompasses the parallelism settings required to achieve the desired performance levels. GrPPI eliminates this difference in programmability between PPIs while reducing the total programming effort at the cost of limitations that can lead to poor performance and misuse of computing resources.

Code-based metrics also fail to address other aspects, such as the lack of documentation of PPIs, which would increase development time. In any case, we believe that the code-based metrics could at least point in the right direction when evaluating PPIs in our test cases. However, they may fail with respect to proportionality in more specific cases.

Andrade et al. [22] also evaluated the LOC and CCN of three applications we use in this paper: Bzip2, Lane Detection, and Face Recognizer (Person Recognition). They evaluated parallel implementations of these applications using PThreads, Intel TBB, FastFlow, and others. Although they also measured estimated development time, they used the pure Halstead's method, considering only standard C++ keywords/tokens and not addressing specific keywords/tokens used by the PPIs. If compared the LOC and CCN numbers in Fig. 15 against the data in Table III of [22], it further highlights how SPBench can reduce the complexity of dealing with the sequential code. For example, in the original sequential Bzip2 presented in [22], users face 1327 lines of code and a cyclomatic complexity number equal to 288. In SPBench, we rebuild this sequential code in a way that users have to face only 28 LOCs and a CCN of 8. In SPBench, we do not eliminate the original source code, but we bring the main parts required for typical stream parallelism implementation

to the surface. It works the same way as shown in the example in Fig. 2. The simplified sequential version allows users to focus on implementing and tuning parallelism. Also, all SPBench sequential benchmarks follow the same implementation standards, which can make parallel code highly portable, except for specific tuning configurations.

Although the last release of the GrPPI library allows users to run the benchmarks with four backends, it does not follow an "all-inclusive" style and does not provide TBB and FastFlow backends within it. It means it is up to the users to build and configure the backend PPIs before installing GrPPI. Installing them and setting up the system environment so GrPPI can find TBB and FastFlow can be tricky for less experienced users who do not have admin access to the system. Therefore, users could also benefit from some proper guidelines in this regard.

7 Suggestions for improving GrPPI

In this section, we suggest improvements to future versions and extensions of GrPPI. Most of the suggestions are based on an analysis of the experimental results on performance and usability and also on the critical analysis of the authors regarding the use of GrPPI in general, which goes beyond programming.

Suggestions for functional and technical improvements:

- *Add more use cases to the samples available with GrPPI* The examples available in the latest version of GrPPI only cover a couple of the simplest use cases and configurations regarding stream parallel patterns. No pipeline or farm examples make use of fine-tuning mechanisms, for example. Such mechanisms are also not presented and explained in the user documentation.
- *Update support for FastFlow 3* or add support for fine-tuning mechanisms in the currently supported version. The ideal scenario is to implement support for a newer FastFlow version and fine-tune mechanisms in a generic way in GrPPI. However, more current versions of FastFlow also allow various mechanisms to be tuned through compilation directives. It would eliminate the need to translate specific mechanisms, although it would not improve programmability.
- *Improve fine-tuning mechanisms* In addition to the lack of mechanisms for adjusting queue size and activating blocking mode for the threads in FastFlow, some of these mechanisms do not seem to work properly in other backends either. However, this could result from poor optimization of these mechanisms, their lack, or other GrPPI optimization problems. It is also not possible to set the maximum number of threads in the TBB backend via GrPPI directives. Therefore, there is plenty of room for improvement in this direction, and a comprehensive evaluation of each mechanism should be carried out before future updates.
- *Improve user documentation* GrPPI is not all-inclusive, i.e., it relies on users installing and configuring backends beforehand. Although GrPPI does provide some documentation on activating or deactivating backends during installation, it would be useful to have documentation to help the user prepare the environment so that GrPPI properly sees the backends. Also, it should provide compre-

hensive documentation on tuning mechanisms, showing what options are supported or not and providing more details on each.
- *Standardize and be more transparent about system behavior* While running experiments, we noticed that applications implemented with GrPPI did not behave as expected in some situations. For example, instead of setting the degree of parallelism with the TBB backend, this parameter sets the number of tokens in practice. Additionally, our experiments with the OpenMP backend were limited as it ran many more threads than expected. These limitations and unpredictability may prevent a fair comparison and analysis of the PPIs.

In addition to the practical suggestions for technical improvements, we also have some substantive suggestions that we believe would be important for improving GrPPI in the future.

- *Include latency as a performance metric when evaluating GrPPI* in the context of parallel stream processing. The requirement for low-latency stream processing applications is set to grow, and GrPPI needs to demonstrate its ability to operate under these requirements. Furthermore, this work demonstrates that latency is a metric that allows us to highlight optimization problems and tool limitations.
- *Evaluating usability and programmability using experiments with people* In our analysis of the usability/programmability of the PPIs used in this work, we concluded that evaluating PPIs solely based on written code can give a distorted view of reality. As GrPPI is a library with a strong focus on making parallel programming easier, we believe that a practical experiment using real people is very relevant to demonstrate how much GrPPI helps in this regard. This type of experiment also helps highlight the points that demand the most time from the user and need improvement.
- *Evaluating energy consumption* Our evaluation of memory usage complemented our performance analysis and helped us better understand some of GrPPI's optimization problems and limitations. Energy consumption is a growing concern in HPC, and we believe it can also complement future analyses and indicate the presence of optimization problems in GrPPI.
- *Test tuning mechanisms in future versions of GrPPI* Stream processing applications usually have different performance goals, such as throughput, latency, or resource usage. Achieving these performance goals may require specific mechanisms, which, although partially implemented by GrPPI, are not usually analyzed.

8 Conclusion

In this work, we evaluated the GrPPI library targeting stream processing scenarios. Unlike related work, we use latency and throughput as more representative metrics to measure stream processing performance. We also used the SPBench benchmarking framework to simulate more realistic workloads for evaluating GrPPI. We also evaluate memory usage using different parallel pattern compositions and multiple

degrees of parallelism. The SPBench allowed us to build, configure, and run the benchmarks easily and quickly, providing a highly parameterized environment. Finally, we evaluated GrPPI regarding programmability/productivity, including an adapted version of Halstead's method for parallel applications.

GrPPI with the OpenMP backends, ISO C++ threads, and TBB showed throughput equivalent to or better than the handwritten benchmarks with TBB and Fastflow in most cases. On the other hand, all GrPPI backends underperformed at some point in the test scenarios. Considering latency, the GrPPI FastFlow and ISO C++ threads backends performed much worse than the others. GrPPI-TBB also showed high latency in the Ferret application, which is an unexpected result. The poor performance of GrPPI-FastFlow is because it does not allow configuring FastFlow buffers/queues or enabling blocking mode on the threads. The memory usage results showed the impact of the lack of fine-tuning on system resource utilization and that high memory usage is commonly associated with high latency. GrPPI authors argue that FastFlow is targeted to expert parallel programmers, as GrPPI provides a more simplified and user-friendly interface aimed at a broader range of application developers while providing fewer fine-tuning options [11]. However, some of the performance issues we've observed result from GrPPI's internal implementation rather than a lack of fine-tuning mechanisms.

We can see that such a simplified interface fulfills its role because GrPPI showed the best programmability results. By applying Halstead's method, we estimated that implementing a pipeline-farm composition for FastFlow takes about three times less development time through GrPPI than directly writing the parallelism using the structured parallel patterns provided by FastFlow. The source code-based metrics we use can give an indication of how easy it is to use/program in a given PPI. However, they fail to represent the real differences between PPIs more accurately.

We can say that GrPPI does what it promises as an overall conclusion. It can deliver competitive performance while foregoing fine-tuning. However, such fine-tuning can be a crucial requirement for current stream processing applications, which demand real-time processing. For example, a Lane Detection application, which would be running in a self-driving vehicle, cannot have more than a few milliseconds of latency in each frame. Such factors may limit the applicability of GrPPI in more realistic scenarios. We have integrated GrPPI into SPBench as one of the contributions of this work. This means that SPBench now natively provides benchmarks with GrPPI and support for creating new benchmarks using this library. We hope that this will also help evaluate possible new versions of GrPPI in the future.

One of the limitations of this work is that it does not evaluate the parallel patterns for data stream processing that GrPPI implements, which are more complex patterns [5]. This is because, although SPBench allows the creation of data stream processing benchmarks, applications in this subdomain frequently require keyed data partitioning (grouping stream data by key) [25], and GrPPI does not support this functionality yet. In future work, GrPPI could be tested on different architectures, preferably a non-Intel one, as has been done so far in previous work. It would also be interesting to evaluate the latency of the GrPPI parallel patterns for data streams using appropriate benchmarks. In addition, a comparison of GrPPI with similar solutions such as SPar [26] could be done, which generates code for the same backends

as GrPPI but has a code annotation-based abstraction approach. We hope the results, analysis, and suggestions can help guide improvements and future development of GrPPI and also other PPIs in this context.

Acknowledgements The authors acknowledge the High-Performance Computing Laboratory of the Pontifical Catholic University of Rio Grande do Sul (LAD-IDEIA/PUCRS, Brazil) for providing support and technological resources, which have contributed to the development of this project and the results reported within this research.

Funding Open access funding provided by Università degli Studi di Torino within the CRUI-CARE Agreement. This work was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brazil (CAPES) - Finance Code 001, European Union's Horizon 2020 JTI-EuroHPC research and innovation program under Grant Agreement No. 956748, project "Adaptive multi-tier intelligent data manager for Exascale" (ADMIRE), by the Spanish Ministry of Science and Innovation, and FAPERGS 10/2020-ARD ProjectSPAR4.0(No. 21/2551-0000725-7).

Availability of data and materials Not applicable.

Declarations

Conflict of interest The authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Consent for publication All authors have read and approved the final manuscript and agree with its submission to The Journal of Supercomputing.

Ethics approval and consent to participate Not applicable.

References

1. McCool M, Reinders J, Robison A (2012) Structured parallel programming: patterns for efficient computation. Elsevier, Amsterdam
2. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M (2017) Fastflow: high-level and efficient streaming on multicore, Chap. 13. In: Pllana S, Xhafa F (eds) Programming multi-core and many-core computing systems. Wiley, Hoboken, pp 261–280. https://doi.org/10.1002/9781119332015.ch13
3. Voss M, Asenjo R, Reinders J (2019) Pro TBB: C++ parallel programming with threading building blocks, vol 295. Springer, Berkeley
4. Rio Astorga D, Dolz MF, Fernández J, García JD (2017) A generic parallel pattern interface for stream and data processing. Concurrency Comput Pract Exp. https://doi.org/10.1002/cpe.4175
5. del Rio Astorga D, Dolz MF, Fernández J, García JD (2018) Paving the way towards high-level parallel pattern interfaces for data stream processing. Future Gen Comput Syst 87:228–241. https://doi.org/10.1016/j.future.2018.05.011

6. Muñoz JF, Dolz MF, Rio Astorga D, Cepeda JP, García JD (2018) Supporting MPI-distributed stream parallel patterns in GrPPI. In: Proceedings of the 25th European MPI Users' Group Meeting, EuroMPI'18. ACM, New York, NY, USA. https://doi.org/10.1145/3236367.3236380
7. López-Gómez J, Fernández Muñoz J, del Rio Astorga D, Dolz MF, Garcia JD (2019) Exploring stream parallel patterns in distributed MPI environments. Parallel Comput 84:24–36. https://doi.org/10.1016/j.parco.2019.03.004
8. Garcia AM, Griebler D, Schepke C, García JD, Muñoz JF, Fernandes LG (2023) A latency, throughput, and programmability perspective of GrPPI for streaming on multi-cores. In: 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), PDP'23. IEEE, Naples, Italy, pp 164–168. https://doi.org/10.1109/PDP59025.2023.00033
9. 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
10. Vogel A, Griebler D, Danelutto M, Fernandes LG (2022) Self-adaptation on parallel stream processing: a systematic review. Concurrency Comput Pract Exp 34(6):6759. https://doi.org/10.1002/cpe.6759
11. Garcia JD, Rio D, Aldinucci M, Tordini F, Danelutto M, Mencagli G, Torquati M (2020) Challenging the abstraction penalty in parallel patterns libraries. J Supercomput 76(7):5139–5159. https://doi.org/10.1007/s11227-019-02826-5
12. Garcia AM, Griebler D, Schepke C, Fernandes LG (2023) Micro-batch and data frequency for stream processing on multi-cores. J Supercomput. https://doi.org/10.1007/s11227-022-05024-y
13. Garcia-Blas J, Rio Astorga D, García JD, Carretero J (2019) Exploiting stream parallelism of MRI reconstruction using GrPPI over multiple back-ends. In: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp 631–637. https://doi.org/10.1109/CCGRID.2019.00081
14. Vílchez Moya C (2020) Application parallelization and debugging using pattern-based programming. Technical report, Undergraduate Thesis of Double Degree in Computer Engineering and Mathematics, Faculty of Informatics UCM, Department of Computer Architecture and Automation. https://eprints.ucm.es/id/eprint/62014/
15. Brown C, Janjic V, Barwell AD, Garcia JD, MacKenzie K (2020) Refactoring GrPPI: generic refactoring for generic parallelism in C++. Int J Parallel Prog 48(4):603–625. https://doi.org/10.1007/s10766-020-00667-x
16. Andrade G, Griebler D, Santos R, Danelutto M, Fernandes LG (2021) Assessing coding metrics for parallel programming of stream processing programs on multi-cores. In: 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), SEAA'21. IEEE, Pavia, Italy, pp 291–295
17. 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
18. Liu S, Gaudiot J-L (2020) Autonomous vehicles lite self-driving technologies should start small, go slow. IEEE Spectrum 57(3):36–49. https://doi.org/10.1109/MSPEC.2020.9014458
19. Dekking FM, Kraaikamp C, Lopuhaä HP, Meester LE (2005) A modern introduction to probability and statistics: understanding why and how, vol 488. Springer, Berkeley
20. Ignatious HA, Sayed H-E, Khan M (2022) An overview of sensors in autonomous vehicles. Procedia Comput Sci 198:736–741. https://doi.org/10.1016/j.procs.2021.12.315
21. Bagwe GR (2018) Video frame reduction in autonomous vehicles. Master's Thesis, Michigan Technological University, Michigan, USA. https://doi.org/10.37099/mtu.dc.etdr/645
22. Andrade G, Griebler D, Santos R, Fernandes LG (2023) A parallel programming assessment for stream processing applications on multi-core systems. Comput Stand Interfaces 84:1–25. https://doi.org/10.1016/j.csi.2022.103691
23. Andrade G, Griebler D, Santos R, Kessler C, Ernstsson A, Fernandes LG (2022) Analyzing programming effort model accuracy of high-level parallel programs for stream processing. In: Proceedings of the International Conference on Software Engineering and Advanced Applications, pp 229–232. https://doi.org/10.1109/SEAA56994.2022.00043
24. Halstead MH (1977) Elements of software science, vol 36. Elsevier, New York, pp 4–41
25. 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(na):222900–222917. https://doi.org/10.1109/ACCESS.2020.3043948

26. 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

Authors and Affiliations

Adriano Marques Garcia[1] · Dalvan Griebler[2] · Claudio Schepke[3] · José Daniel García[4] · Javier Fernández Muñoz[4] · Luiz Gustavo Fernandes[2]

✉ Adriano Marques Garcia
adriano.marquesgarcia@unito.it

Dalvan Griebler
dalvan.griebler@pucrs.br

Claudio Schepke
claudioschepke@unipampa.edu.br

José Daniel García
jdgarcia@inf.uc3m.es

Javier Fernández Muñoz
jfmunoz@inf.uc3m.es

Luiz Gustavo Fernandes
luiz.fernandes@pucrs.br

[1] Department of Computer Science, University of Turin, Turin, Italy

[2] School of Technology, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, RS, Brazil

[3] Laboratory of Advances Studies in Computation (LEA), Federal University of Pampa, Alegrete, RS, Brazil

[4] Department of Computer Science, University Carlos III of Madrid, Madrid, Spain