Computer Standards & Interfaces 84 (2023) 103691

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

A parallel programming assessment for stream processing applications on multi-core systems

Gabriella Andrade [a,*,1], Dalvan Griebler [a,b,**,1], Rodrigo Santos [c], Luiz Gustavo Fernandes [a]

[a] *Parallel Applications Modeling Group (GMAP), School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil*
[b] *Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (Setrem), Três de Maio, Brazil*
[c] *Department of Applied Informatics, Federal University of the State of Rio de Janeiro (UNIRIO), Rio de Janeiro, Brazil*

ARTICLE INFO

Keywords:
Parallel software
Parallel computing systems
Programming productivity
Programming effort
Stream parallelism
Programming usability

ABSTRACT

Multi-core systems are any computing device nowadays and stream processing applications are becoming recurrent workloads, demanding parallelism to achieve the desired quality of service. As soon as data, tasks, or requests arrive, they must be computed, analyzed, or processed. Since building such applications is not a trivial task, the software industry must adopt parallel APIs (Application Programming Interfaces) that simplify the exploitation of parallelism in hardware for accelerating time-to-market. In the last years, research efforts in academia and industry provided a set of parallel APIs, increasing productivity to software developers. However, a few studies are seeking to prove the usability of these interfaces. In this work, we aim to present a parallel programming assessment regarding the usability of parallel API for expressing parallelism on the stream processing application domain and multi-core systems. To this end, we conducted an empirical study with beginners in parallel application development. The study covered three parallel APIs, reporting several quantitative and qualitative indicators involving developers. Our contribution also comprises a parallel programming assessment methodology, which can be replicated in future assessments. This study revealed important insights such as recurrent compile-time and programming logic errors performed by beginners in parallel programming, as well as the programming effort, challenges, and learning curve. Moreover, we collected the participants' opinions about their experience in this study to understand deeply the results achieved.

1. Introduction

In the last decade, computers were primarily parallel architectures due to the limitations faced by the silicon industry in the design of the central processing unit (CPU) and the requirements to increase performance [1,2]. As such, multi-core CPUs emerged, coupling several cores in a single chip. Computers that have this kind of CPU, containing one or more cores, are called multi-core systems. To enable the parallelism exploitation of such architecture, the programmer must develop the software using parallel programming techniques, libraries, frameworks, mechanisms, and paradigms. In addition, the programmer has to know the computer architecture characteristics, which vary among vendors and platform types [3,4]. Therefore, this is not a simple task for application programmers, who usually focus on developing the business logic code. It is also a challenging task for system programmers who are experts in parallel programming because they need to handle all these details.

New parallel APIs (Application Programming Interfaces) have been created on top of the POSIX Threads (Pthreads) to leverage parallel programming abstractions and release programmers from dealing with lower-level implementation and architecture-specific optimizations. There are APIs based on structured and non-structured approaches for parallel programming. Structured parallel programming is a higher-level approach where the concept leverage parallel patterns that can be a receipt/guide to writing efficient parallel software. Moreover, it can be provided as ready-to-use templates (high order functions) that already implement lower-level parallelism, such as the threads' communication and synchronization, independently of the

* Corresponding author.
** Corresponding author at: Parallel Applications Modeling Group (GMAP), School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil.
E-mail addresses: gabriella.andrade@edu.pucrs.br (G. Andrade), dalvan.griebler@pucrs.br (D. Griebler), rps@uniriotec.br (R. Santos), luiz.fernandes@pucrs.br (L.G. Fernandes).
[1] Authors that majority and equivalently contributed.

https://doi.org/10.1016/j.csi.2022.103691
Received 31 January 2022; Received in revised form 7 August 2022; Accepted 4 September 2022
Available online 14 September 2022
0920-5489/© 2022 Elsevier B.V. All rights reserved.

target architecture. Conceptually, all parallel patterns can be arbitrarily nested to build new pattern compositions [5,6].

The parallel patterns can be classified mainly in data (e.g., Map, Reduce, and Stencil) and stream (e.g., Pipeline, Farm, and others) parallelism. As stream processing applications comprise collecting, processing, and analyzing high volume, heterogeneous, and continuous data streams in real-time [7], parallelism exploitation can be implemented mainly using the Pipeline and Farm patterns. Additionally, some stream processing applications present characteristics from which parallel data patterns could be applied or combined to increase the degree of parallelism [8].

Stream processing applications executing on multi-core systems (easily available on personal computers, cell phones, and servers) are recurrent in our day-to-day life [9]. Examples are video and audio processing, and data compression and analytics. This fact does not necessarily mean they are parallel software. A developer counts on a variety of languages and libraries for parallel programming based on different programming approaches to do this. Choosing one of them is a hard decision since there are few studies focused on evaluating parallel APIs.

Few existing studies were conducted to extract only specific metrics such as the development time, parallelization performance, and efficiency of a parallel application without considering the human development effort and particular challenges faced by programmers [10–14]. Moreover, most of these studies did not follow a well-controlled experimental methodology and did not provide deeper discussions [15–19]. Finally, parallel programming evaluation for stream processing applications on multi-core systems has not been studied yet (Section 3). The lack of information about usability makes it difficult for developers to make the appropriate decision and help with hints to improve the parallel API abstractions [20,21].

In this context, this work aims to assess parallel programming usability based on software engineering methodologies [22] for a quantitative and qualitative experiment. The scope of this study comprises three parallel APIs based on structured parallel programming from academia and industry for expressing parallelism in stream processing applications targeting multi-core systems. Unlike studies that perform experiments with experienced developers on parallel programming [20,23], participants of the present study were intentionally beginner developers to consider the learning impact and other qualitative information. Therefore, we aim to understand beginners' challenges.

The scientific contributions provided in this work are (1) a literature review about parallel API assessments; (2) a parallel programming assessment methodology to guide other researchers based on the literature review; (3) an analysis of three parallel APIs for expressing stream parallelism on multi-core systems; (4) mapping of quantitative and qualitative usability indicators regarding parallel programming in stream processing applications; (5) a qualitative analysis of participants' perceptions when they implement parallel programs.

This article is organized as follows. Section 2 presents the background, Section 3 presents the literature review, and Section 4 describes the applied methodology. Next, Section 5 presents the experiment plan, including goals and hypothesis, and also details how the study was conducted. In Sections 6 and 7, results are presented. Section 8 provides insights regarding the usability of parallel programming models for stream processing. Finally, Section 9 concludes this work.

2. Background

This section provides the background: Sections 2.1 and 2.2 contextualize both parallel stream processing and APIs for parallel programming on multi-cores, and Section 3 provides the related work.

Fig. 1. Parallel activity graph of the Pipeline pattern.

2.1. Parallel stream processing

Data generated from networking services, cameras, sensors, and other data sources producing streaming data can be consumed or processed by stream processing applications. They usually continually collect and process the streamed data in the form of a sequence of computing operations (filter, transform, or analyze) over data streams, which can later be stored in a permanent file system [7,24]. Parallel computing is necessary to reach the quality of service requirements, such as real-time response or high throughput [25]. Stream processing applications can be found in several domains, such as the stock market, natural systems, transportation, manufacturing, health and life sciences, law enforcement, defense, cybersecurity, and many others. These applications can process structured or unstructured data. Most stream processing applications consume structured data (e.g., relational database style records) that share a typical structure or scheme. However, commercial stream processing applications usually process unstructured data like images, audio, and video, mainly executing compression, filtering, and reproduction tasks [7,24].

Parallel stream processing applications execute as stream graphs composed of operators or stages, and FIFO (First In, First Out) communication queues [26,27]. The stream input is an infinite sequence of items (or data) stream, and the queues contain a finite number of items waiting to be consumed by each stage [28]. Each operator can process the same or a different item than the previous operator. Therefore, the number of operators usually limits the parallelism of stream processing. However, parallelism can be increased by replicating operators to process multiple items simultaneously. The replication of Stateful operators will require extra mechanisms to guarantee data consistency.

As can be noticed, developing parallel stream processing applications is not an easy activity. The structured parallel programming approach alleviates these complexities for different application domains. It supports developers with parallel patterns, which are parallelism strategies to write efficient, structured, and maintainable programs [6]. These patterns can be in the form of design patterns [5] or higher-order functions that are algorithmic skeleton libraries or high-level programming constructions equipped with well-defined functional and extra-functional semantics [26].

Figs. 1 and 2 graphically represent the two parallel patterns commonly used when implementing parallel stream processing: Pipeline and Farm. The pipeline pattern exploits parallelism in the form of a traditional manufacturing assembly line. It has well-defined tasks to be performed on data to produce modified data, which are sent to the next stage/workstation [6,24,26,29]. The Pipeline pattern applies a sequence of operations simultaneously to different data elements, so it is possible to compute each operation in a different data element at each point and at a given time. The parallel activity graph in Fig. 1 is a Pipeline with three independent stages (kernels or filters) that communicate explicitly through data channels, where the output of a stage feeds the input of the next stage, finite or infinitely.

The Farm pattern is also called split-join [24]. This pattern is similar to the Pipeline pattern and can be implemented with two or three Pipeline stages in sequence. The first stage performs as the stream item emitter or scheduler. The second stage performs as stage replicas called workers. Optionally, the last stage acts as a stream item collector [8,26]. The parallel activity graph in Fig. 2 is a Farm pattern with its three components (Emitter, Worker, and Collector).

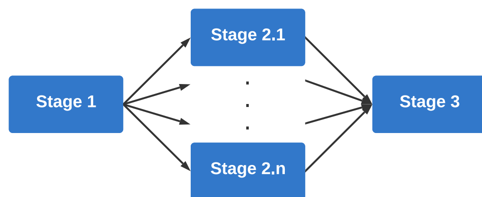


Fig. 2. Parallel activity graph of the Farm pattern.

2.2. *APIs for stream parallelism on multi-cores*

There are many parallel APIs designed for multi-core architectures. The most famous is Open Multi-Processing (OpenMP) [30], although it is only suitable for data parallelism exploitation and requires the programmer to implement extra synchronization mechanisms in the stream parallelism exploitation. For stream parallelism exploitation based on the structured programming approach, one remarkable API is StreamIt [24]. It is an external domain-specific language (new language and compiler), and its research activities ended in 2013. Maintained by Intel, Threading Building Blocks (TBB) [29,31] is an open-source and general-purpose C++ template-based parallel API from the industry. It offers a Pipeline pattern constructor that can also perform as the Farm pattern.

FastFlow [26] is a representative API from the scientific community. It has a similar C++ template-based API to TBB. However, their runtime parallelism systems are implemented differently. While the TBB API works on top of an unchangeable work-stealing task scheduler and building blocks, the FastFlow API works on top of customizable lock-free FIFO queues, building blocks, parallel patterns, and task scheduler. A more recent research initiative that promises to leverage higher-level and productive stream parallelism on multi-core systems is SPar (Stream Parallelism) [32,33]. It is an internal domain-specific language (embedded in the C++ language) in the form of C++ annotation to avoid sequential code rewriting. The parallel APIs presented in the following sections were chosen for our usability evaluation study because they are C++ APIs, have still working activities, and are suitable to express stream parallelism.

2.2.1. *FastFlow and TBB*

This section describes how to use FastFlow and TBB, explaining their main routines. The code snippets in the figures summarize the code required to develop parallel software. The first step to expressing stream parallelism in stream processing applications is to find the most computing-intensive code regions and, when possible, divide them into parallel stages to process items in sequence.

In TBB, some stages can operate in `parallel`, and others cannot (`serial`). The concept of the stage in TBB is known by the name `filter`. Fig. 3 shows how to create the first stage using TBB. Although TBB supports modeling stages using the lambda function interface, we concentrate on the default interface, in which stages are modeled as classes extending the `tbb::filter` class, where one of the following filter types should be specified as an argument to indicate the stage behavior (line 3): `tbb::filter::serial_out_of_order` is used to process the items one at a time without preserving the processing order; `tbb::filter::serial_in_order` is used to process the items one at a time in the same order. The processing order is implicitly defined by the first filter and respected by the other ones; `tbb::filter::parallel` is used to process multiple items in parallel and in no particular order [34]. Moreover, each stage class needs to implement the virtual `operator` method (lines 4–11) in which a task or stream item is processed. Every time this method returns a `void` pointer (line 8), it is implicitly sends the stream item to the next stage. When `NULL` is returned (line 10), it indicates the end of the stream to stop the stream processing [31,34].

First stage using FastFlow

```
1.  class first : public ff::ff_node_t<int>{
2.  public:
3.    int * svc ( int* ){
4.    while (1){
5.        // computation
6.        if (stop) break;
7.        ff_send_out(item);
8.      }
9.      return EOS;
10.  }};
```

First stage using TBB

```
1.  class first() : public tbb::filter {
2.  public:
3.    first(): tbb::filter(tbb::filter::serial_in_order) {}
4.    void* operator() (void* ) {
5.      while (1){
6.        // computation
7.        if (stop) break;
8.        return  item;
9.      }
10.     return NULL;
11.  }};
```

Fig. 3. First stage using FastFlow and TBB.

Middle and last stages using FastFlow

```
1.  class middle:  public ff::ff_node_t<int>{
2.  public:
3.    int * svc (int item){
4.      //computation
5.      return item;
6.  }};
7.  class last:  public ff::ff_node_t<int>{
8.  public:
9.    int * svc (int *item){
10.     //computation
11.     delete item;
12      return GO_ON;
13. }};
```

Middle and last stages using TBB

```
1.  class middle() : public tbb::filter {
2.  public:
3.    middle(): tbb::filter(tbb::filter::parallel) {}
4.      void* operator() (void *item) {
5.        //computation
6.        return item;
7.  }};
8.  class last() : public tbb::filter {
9.   public:
10.    last(): tbb::filter(tbb::filter::serial_in_order) {}
11.    void* operator() (void *item) {
12.      //computation
13.      delete item;
14.      return NULL;
15.  }};
```

Fig. 4. Middle and last stage using FastFlow and TBB.

Fig. 3 also shows how to create the first stage using FastFlow. In FastFlow, although a stage can be modeled using the lambda function interface, we focus on the default interface where a stage is modeled as a class or struct (unlike TBB), extending the `ff_node` class (line 1). Inside the stage, the virtual `svc` method has to be implemented (lines 3–10). The first stage may produce tasks (stream items) inside the `svc` method and send the produced stream items to the next stage using the `ff_send_out` method (line 7). If there are no more stream items, it is possible to return `EOS` (line 9) to propagate the end of the stream processing to the subsequent stages [26].

Fig. 4 shows how to create the middle and last stage using FastFlow and TBB. Disassociating the specific syntax and semantics, the principle for modeling the middle and last stage is similar to both APIs. The middle stage only computes the stream items and sends them to the next stage using the `return` operation (line 5 in FastFlow and line 6 in TBB). The programmer can create as many as necessary middle stages. For the last stage, the programmer has to manage the stream item data, deallocate the memory for the input item (line 11 in FastFlow and line 13 in TBB), and return a specific value to skip sending items to subsequent stages (line 12 in FastFlow and line 14 in TBB). In TBB, the programmer defines the stage behavior when writing the class and passing as an argument the filter, while in FastFlow, this is done when instantiating the parallel pattern.

Therefore, to build the parallel activity graph as a Farm pattern in TBB, the programmer specifies the `parallel` filter in the middle stage and uses the same Pipeline object to instantiate a traditional Pipeline pattern, as shown in Fig. 5. In TBB, the programmer can also specify how many concurrent threads are created in line 1. The `run` method is used to indicate the beginning of the Pipeline execution, receiving as an argument the number of concurrent tokens (it can also be understood as the number of items in the shared queue). Moreover, the class object `tbb::pipeline` is first declared (line 2) to build a parallel activity graph. Next, the objects of the three stages are declared and added to the Pipeline object using `add_filter` (lines 3–6) in the correct sequence. Lastly, by calling the `run` method in the `pipeline` object, the parallel computing will start and keep executing until a stop condition, which is a `NULL` pointer [34].

In FastFlow (Fig. 5), there is a specific object class to build the parallel activity graph for the Farm pattern. The three entities of the pattern (Emitter, Worker, and Collector) receive the class objects declared as an argument. For the Worker entity, we must create a vector

Farm using FastFlow
```
1. std::vector<std::unique_ptr<ff_node>> S2;
2. for(int i=0; i < nthreads; i++)
3.    S2.push_back(std::make_unique<middle>());
4. first S1; last S3;
5. ff::ff_OFarm<int> farm(std::move(w));
6. farm.add_emitter(S1);
7. farm.add_collector(S3);
8. farm.run_wait_end();
```

Farm or Pipeline using TBB
```
1. tbb::task_scheduler_init init(3);
2. tbb::pipeline pipeline;
3. first S1; middle S2; last S3;
4. pipeline.add_filter(S1);
5. pipeline.add_filter(S2);
6. pipeline.add_filter(S3);
7. pipeline.run(3);
```

Fig. 5. Parallel activity graph modeled according to the Farm pattern while instantiating stages using FastFlow and TBB.

Pipeline using FastFlow
```
1. first S1; middle S2; last S3;
2. ff:ff_Pipe<int> pipeline(S1, S2, S3);
3. pipeline.run_wait_end();
```

Pipeline with Farm using FastFlow
```
1. std::vector<std::unique_ptr<ff_node>> S2;
2. for(int i=0; i < nthreads; i++)
3.    S2.push_back(std::make_unique<middle>());
4. first S1; last S3;
5. ff::ff_OFarm<int> farm(std::move(w));
6. farm.add_collector(S3);
7. ff:ff_Pipe<int> pipeline(S1, farm);
8. pipeline.run_wait_end();
```

Fig. 6. Parallel activity graph modeled according to the Pipeline pattern and combined with Farm while instantiating stages using FastFlow.

that has many replicas as parallel workers are intended (`nthreads`) (lines 1–3). After, the stage object classes are declared (line 4). Next, the `ff::ff_Farm` template class is used to build the parallel activity graph, where the Worker entity is passed as an argument (a vector of workers) (line 5). The Emitter and Collector entities are added using the respective routine (lines 6–7). Then, the parallel computing will start and wait until finished at the call of the `run_wait_end` routine.

Unlike TBB, FastFlow allows us to create other parallel patterns, combining Farm and Pipeline object classes, and reusing the same stage classes. As in the left-hand side of Fig. 6, we can start creating a simple Pipeline pattern using the `ff::ff_Pipe`, where the arguments are stage object classes. A new parallel pattern can be created by transforming one or more stages into a Farm. The example on the right-hand side of Fig. 6 shows how this can be done. The middle stage becomes the Worker entity, and the last stage becomes the Collector entity of a Farm. Then, we build the Pipeline with two stages, where the first is the serial while the second is a Farm. Other patterns and activity graphs could be created using FastFlow, which does not necessarily optimize the parallelism exploitation. It will depend on several aspects regarding the environment and application.

2.2.2. SPar

Following a domain-specific approach, SPar offers a small set of annotations to express stream parallelism, which is parsed by its own compiler to generate parallel code automatically [32]. These annotations are standard C++11 attributes with specific semantics. A C++11 annotation is declared using double brackets (`[[attr-list]]`) where there are one or more attributes. In SPar, the first attribute in the list is called an identifier (`ID`), and the rest are auxiliary (`AUX`). All attributes are part of the stream parallelism namespace (named as `spar`). The SPar `ID` attributes are `ToStream` and `Stage`, while `AUX` attributes are `Input`, `Output`, and `Replicate`.

`ToStream` indicates that a given C++ program region is going to be stream parallelism. The annotated region can be a loop or a code block. `Stage` denotes a phase within `ToStream`, where operations are computed over the stream items. At least one stage must be within a `ToStream` region. In addition, SPar supports any number of stages inside a `ToStream` region. `Input` is used to indicate the variables that will be consumed by `ID` attributes, and `Output` is used to indicate the variables that will be produced by `ID` attributes. When using these attributes, at least one argument must be present. `Replicate` is used to replicate a `Stage`. This attribute allows programmers to scale the performance on stateless stages. This attribute receives a constant value delimiting the number of workers for the stage as an argument.

Pipeline using SPar
```
1. [[spar::ToStream]]
2. while (1){
3.   // computation
4.   if (stop) break;
5.   [[spar::Stage,spar::Input(item),spar::Output(item),spar::Replicate(4)]]{
6.     // computation
7.   }
8.   [[spar::Stage,spar::Input(item)]]{
9.     // computation
10.  }
11.}
```

Fig. 7. Example of the use of Pipelines using SPar.

Moreover, this attribute can also be left empty to use the environment variable `SPAR_NUM_WORKERS`.

Fig. 7 presents a pseudocode example to indicate the SPar's ease of use. In line 1, note that `ToStream` annotation was inserted in front of a loop, indicating the beginning of the stream parallelism region. Since stream items are not consumed from and produced outside the region, auxiliary attributes are unnecessary. The codes left between `ToStream` and the first `Stage` annotation (line 5) are always an implicit serial stage that produces stream items to the following stages. The first `Stage` annotation will actually be the second `Stage` of the parallel activity graph, where if there is another `Stage` in sequence, `Input` and `Output` are required. Since this `Stage` is a stateless computation, we can add the `Replicate` attribute to increase the degree of parallelism. By the way, the last stage is stateful and does not produce anything outside. Therefore, `Output` is not needed, and `Replicate` does not apply.

Additionally, SPar offers compiler flags if programmers want to change the runtime system behavior, which may improve the performance of the application. By default, items are distributed in a round-robin way, without preserving the input order in the output. To preserve this ordering, the user can use `-spar_ordered` when compiling the program. It is important for video applications to avoid producing the wrong output video. There is also the `-spar_ondemand` flag to switch the item distribution for on-demand scheduling. Other options can be found in the online documentation.[2]

3. Literature review

This section presents the process used for the literature review and the studies returned.

3.1. Search process

Our literature review aimed to search for papers that have conducted empirical studies with students and developers of parallel applications to evaluate the usability and productivity of parallel APIs. We aimed to select journal and conference papers written in English and published from 2005 to 2022. To do so, we executed the following search string on the Scopus[3] database, which was selected because it indexes from other databases [35]. In addition, we considered the search string in the title, abstract, and keywords of the studies.

```
TITLE-ABS-KEY ((assess* OR evaluate* OR examin*)
AND (develop* OR programm* OR cod*) AND
(usability OR productivity OR effort) AND
(experiment OR empirical OR study) AND
((parallel AND (programming OR computing)) OR hpc)
AND (languages OR systems OR interfaces))
```

[2] Available at: https://gmap.pucrs.br/spar-wiki.
[3] Available at: https://www.scopus.com.

The search on the Scopus database returned 247 studies. We exported in BibTeX format to the Zotero tool[4] to help us in the review process. Using Zotero, the first filtering of the studies from reading the title, abstract, and keywords returned 26 studies. From the full-text reading, we selected 11 studies for which we applied the snowballing method. We used forward snowballing to analyze citations to the selected papers to identify additional papers to complement our literature review [36]. From this last filtering, we selected 29 final studies. Although we defined exclusion criteria studies prior to 2005, we considered a study from 1996 because it is one of the most cited in the literature.

3.2. Discussion

The popularization of parallel architecture in our daily computing systems leveraged parallel programming, which is well-known to be a complex task and most reserved for specialists. Consequently, research and industry have promoted and developed multiple parallel APIs to ease this task. Unfortunately, only a few studies aimed to evaluate the usability of such APIs. Most studies focused on evaluating their APIs regarding the performance, where only runtime, speedup, and efficiency of the parallel software are analyzed [10–14,37]. These factors could also explain why we have so many serial programs running in our parallel architectures. It is still a task for experts and it seems that only a few works really care about usability. One way to develop more parallel code developed is to make parallel programming easier for the application developers, who are not experts or specialists in system programming. Conducting empirical studies to evaluate usability, such as the one performed in this paper, is a time-consuming task that requiring considerable effort to conduct all the procedures correctly, from planning to analysis. However, this is a way to continuously improve the parallelism abstraction and create better/simpler parallel APIs.

In 1996, Szafron et al. [21] conducted a controlled experiment with graduate students in a parallel/distributed computing class to compare the Enterprise PPS interface with a PVM-like (NMP) library of message-passing routines. The main goal in [38] was to evaluate the programming effort by comparing the PRAM-like model (XMTC) and the Message Passage Interface (MPI). In [39], the main goal was to evaluate the effort of beginners in parallel programming to develop parallel applications in MPI and OpenMP. Zelkowitz et al. [40] proposed a productivity estimation model based on speedup and Lines of Code (LOC). In order to evaluate this model, Zelkowitz et al. [40] performed an experiment to evaluate the effort to develop MPI programs.

In [41] the main goal was to evaluate the effort of graduate students to develop an actual program for multi-core computers using OpenMP and Pthreads. Coblenz et al. [42] compared Cilk Plus and OpenMP to evaluate the design trade-offs in the usability and security of these approaches. In [43], Pthreads and OpenMP were compared to OpenMP _XN programming model. OpenMP _XN is an extension of OpenMP with Atomic Sections of code executed atomically and mutually exclusive from other conflicting atomic operations.

In [44], a study was performed to evaluate the effort spent by novices to develop MPI programs. Patel et al. [45] aimed to compare the performance and productivity of MPI and UPC programs. In [46], another experiment was performed to compare the productivity of C+MPI, UPC, and the x10 language of the IBM PERCS project. In [46], the productivity was evaluated through an experiment with undergraduate students with little or no parallel programming experience. In [47], the authors presented a methodology for evaluating UPC programmability against MPI through classroom studies with a group of novice programmers. Speyer et al. [48] evaluated the productivity and usability of Charm++, UPC, SCOOP, MATLAB+Star-P, and SHMEM through an experiment with computer science and engineering students.

In [49], the authors proposed a method for measuring the complexity of programming-related tasks in HPC (High-Performance Computing). The Complexity Metrics (CM) method was proposed to help determine the productivity of new parallel APIs developed by IBM. Danis et al. [49] conducted a series of real-world observations, interviews, and surveys with HPC experts to identify ecologically valid tasks for modeling with the CM method. Danis et al. [49] conducted experiments in a development environment using the instrumentation of workstations to automatically capture the programmers' behavior and to gather empirical data about productivity at a relatively fine grain. Given an activity that the programmer must perform, productivity was defined as the performance to complete that activity.

In [50], an empirical study was conducted with novice and expert Java programmers to identify multithreaded bugs in Java Threads code examples. The results were obtained through self-evaluation questionnaires, where feedback from the participants was also collected. In [51], the authors compare Java Threads and SCOOP to comprehending and debugging existing programs and writing correct new programs. The participants were undergraduate students in the software architecture course. In [52], an experiment was conducted with 13 Master's students who are, on average, in their fourth year of Computer Science studies to compare the development of parallel applications for multi-core systems using Scala and Java Threads.

Rossbach et al. [53] performed a study with 237 undergraduate students of an Operating System course. The main goal of this study was to verify and compare the different techniques used in transactional memory programming with Java Threads: using coarse- and fine-grain locks, monitors, and transactions. In [54], a study was performed to compare teams of programmers developing a parallel program from scratch using Pthreads and Intel Software Transactional Memory (STM) compiler. Similar to the previous study, the experiment performed by Castor [55] also aimed to evaluate the use of locks and transactional memory. Castor [55] evaluated the effort spent by novices to develop a simple program with mutual exclusion and synchronization requirements using Halskell's transaction memory and lock-based concurrency control mechanisms.

Nanz et al. [20] compared Chapel, Cilk, Go, and TBB in a study based on a sequential and parallel implementation of six benchmarks created by notable programmers with more than six years of experience, while Nanz et al. [23] performed an experiment that explores the claimed gap between expert and novice parallel programmers. In [23] the fraction of the original development time spent on implementing the corrections suggested by experts was also measured. Moreover, in [20] the Wilcoxon signed-rank test (two-sided variant) was used to evaluate the results.

In [56], a pilot study was conducted to assess the usability of MPI when implementing design patterns in contrast to alternative implementations of the parallel program. In [57], the authors aimed to evaluate the performance and usability of the primary/secondary pattern of the DSL-POPP (Domain-Specific Language for Pattern-Oriented Parallel Programming) in comparison to the Pthreads library. In [58], a new pattern-based process model called Patty was introduced. Patty was compared with Intel Parallel Studio in an experiment with experienced developers to evaluate its effectiveness and productivity.

Li et al. [59] conducted an empirical investigation to compare the productivity of the OpenACC and CUDA GPU programming interfaces when used by undergraduate students in a classroom environment. In 2018, the previous study was complemented by evaluating the performance and code size [61]. Another more recent study was conducted to compare the scheduling productivity of CUDA with Thrust library [63]. Miller et al. [62] conducted a study to quantify the impact of HPC training by measuring learners' productivity in heterogeneous systems.

[4] Zotero is a software used to manage bibliographic data and related research materials. Available at: https://www.zotero.org/.

Table 1
Empirical studies to evaluate the usability of parallel APIs.

Work	API	Participants	Metrics	Findings	Environment	Program
[21]-1996	Enterprise PPS and PVM-like	Graduate students	Login hours, LOC, number of edits, compiles, and execution time	Enterprise PPS is easier than PVM	Cluster of Workstations	Transitive closure
[39]-2005	MPI and OpenMP	Graduate students	Speedup, dev. time, LOC and cost per LOC	MPI presented higher effort when compared to OpenMP	Cluster of PCs	Game of life and grid of resistors
[43]-2005	OpenMP, OpenMP _XN and Pthreads	Undergraduate and graduate students	Dev. time	OpenMP _XN reduced development time compared to Pthreads	Multi-core	Benchmark applications
[40]-2005	MPI	Graduate students	Exec. time, speedup, LOC, and dev. time	It is difficult to relate productivity model to real parallel applications	Distributed	Game of life and Buffon needle
[46]-2006	MPI, UPC and IBM PERCS x10	Undergraduate students	Executing time, cleaning time, parallelization time, debugging time, authoring time, and accessing doc. time	x10 has an edge over MPI and UPC	Distributed	Smith–Waterman algorithm
[44]-2007	MPI+C	Undergraduate students	Exec. time, dev. time, number of base, non-blocking, and collective functions	Performance can be affected by the type of function used, so it is not simple to relate it to effort	Cluster	Game of life
[38]-2008	PRAM-like and MPI	Graduate students	Characterization of groups, program correctness and dev. time	XMTC programs had fewer bugs and their development time was 46% less than MPI	Linux cluster and class server	Sparse matrix and dense vector multiplication
[49]-2008	IBM parallel APIs	Novices and experts developers	The number of steps, context changes, and the number of data items operated	CM is promising, but it does not include development time, which is a key productivity measure	Workstation	Not informed
[45]-2008	MPI and UPC	Graduate students	Exec. time, speedup, and LOC	Both APIs offer performance, but UPC requires less code	Multi-core and Linux cluster	Power method algorithm
[48]-2008	UPC, SHMEM, SCOOP, Star-P and Charm++	Undergraduate students	Speedup, dev. time, correctness, and LOC	Charm++ and Star-P seem more productive for applications with modest learning curves	Linux cluster	Straightforward estimate and N-body problems
[47]-2009	UPC and MPI	Undergraduate students	Speedup, LOC, dev. time	UPC allowed an easier and faster parallelization than MPI	Cluster	Minimum distance problem
[41]-2009	Pthreads and OpenMP	Graduate students	LOC, lines with parallel constructs, dev. time, exec. time and speedup	Students preferred Pthreads because it required less code refactoring than OpenMP	Multi-core	Bzip2
[53]-2010	Java Threads	Undergraduate students	Design time, dev. time, debugging time, errors, and CCN	Over 70% of students made errors with locks, while less than 10% using transactions	Multi-core	Sync-gallery
[50]-2010	Java threads	Students and professionals	Number of multithreaded bugs	The cooperability facilitates the search for concurrent bugs	Not informed	ArchivalList, IntList and StringBuffer
[55]-2011	Haskell	Undergraduate students	Logic and Compilation errors, hanging and non-hanging errors, dev. time, and LOC	No significant difference in concurrency errors, LOC and development time	Not informed	Program with synchronization and mutex
[52]-2012	Scala and Java Threads	Master's students	Dev. time, LOC, NOC, functional styles (%), imperative style (%), errors, exec. time, and speedup	Scala code is the most compact, yet Java offers less programming and debugging effort	Multi-core	Dining philosophers, DRC project and Mergesort
[20]-2013	Chapel, Cilk, Go, and TBB	Experts in the API tested	LOC, dev. time, execution time, and speedup	TBB and Cilk have the shortest execution time and speedup, but TBB offers faster development	Multi-core	Six micro-benchmark programs
[23]-2013	Chapel, Cilk, Go and TBB	Experts and non-expert developers	LOC, dev. time, exec. time, correction time, and speedup	For all languages, an expert can only moderately improve programs written by novices	Multi-core	A suite of six benchmark programs
[51]-2013	Java Threads and SCOOP	Graduate students	Time to complete the test, Levenshtein distance, error types, LOC, number of classes, attributes and functions	The results are in favor of SCOOP, although participants have experience in Java Threads	Multi-core	Not informed
[56]-2013	MPI	Undergraduate students	LOC, CCN, dev. time, number of compilations, and exec. time	Design patterns can impact the productivity and performance	Linux cluster	Game of Life

(*continued on next page*)

Table 1 (*continued*).

Work	API	Participants	Metrics	Findings	Environment	Program
[54]-2014	Pthreads and Intel STM	Graduate students	LOC, reading time, design time, dev. time, testing time, debugging time, parallel constructs, and critical sections	Intel STM offers better debugging, but is harder to tune performance and implement queries	Multi-core	Parallel desktop search engine
[57]-2014	DSL-POPP and Pthreads	Graduate students	Dev. time, LOC, and COCOMO II	DSL-POPP requires less programming effort than Pthreads	Multi-core	Matrix multiplication
[42]-2015	Cilk Plus and OpenMP	Master's students	Number of correct programs, dev. time, and speedup	More thread-safe reductions have been developed in Cilk Plus	Multi-core	Program that finds anagrams
[58]-2015	Patty and Parallel Studio	Experienced developers	Clarity, complexity, perceivability, learnability, correctness, dev. time	Patty achieved better results because it is pattern-based	Multi-core	RayTracing benchmark
[59]-2016	OpeanACC and CUDA	Undergraduate students	Dev. time, exec. time, independence time	OpenACC requires less programming effort than CUDA	GPU	Heat transfer and message encryption
[60]-2017	Spark, Flink and Hadoop MapReduce	Master's students	Dev. time	Spark and Flink are preferred platforms over Hadoop MapReduce	Cluster	Three use cases from immunology and genomics
[61]-2018	OpeanACC and CUDA	Undergraduate students	Speedup, speedup dispersion, LOC, effort per LOC	OpenACC effort is not significantly less than CUDA	GPU	Heat transfer and message encryption
[62]-2019	MPI, OpenMP and OpenACC	Hackathons participants	Completed milestones, pre- and post-knowledge, function requirement, external library interaction, number of logical statements, efficiency, scaling, data set size, parallel fraction	The proposed productivity metrics can be applied to improve the training of HPC application programmers	Multi-core and GPU	Dot product
[63]-2020	CUDA and Thrust	Graduate students	Dev. tim, number of compiler errors, number of successful results	Thrust abstractions have decreased productivity	GPU	Six programs
Our	**FastFlow, SPar and TBB**	**Graduate students**	**Time to answer the form, accessing the material, understand the code, and read the procedure. Also, the dev. time, debugging time, number of executions, and error types**	**SPar provide better usability than FastFlow and TBB.**	**Multi-core**	**Video processing with OpenCV**

Miller et al. [62] compared applications developed by beginners students using MPI, OpenMP, OpenACC, and OpenMP+OpenACC to assess productivity. To evaluate training effectiveness, the authors proposed a new methodology with two key components: progress productivity and training productivity.

In [60], the usability of the cloud APIs Apache Hadoop MapReduce, Apache Spark, and Apache Flink was evaluated through a study with Master's students from various backgrounds, including IT and data science. This experiment was conducted as part of a cloud computing course [60].

Unlike the previous works (see Table 1), we assess the effort of beginners in parallel programming to develop stream parallel applications in FastFlow, SPar, and TBB. Since the effort to develop an application includes several activities, we evaluated the time spent by the participants when performing each of them separately: the time to answer the form, time accessing the material provided, time to understand the application code, time to read the procedure description, development time, and debugging time. In addition, we considered the number of executions and types of errors. Finally, we performed a textual analysis from the developers' perspective to deeply discuss the experimental results obtained.

4. Research method

From usability assessments, it is possible to obtain improvement indicators for designing new parallel APIs and refining the existing ones. It is possible to create better and simpler-to-use parallel APIs and manage their quality [64]. ISO 9241-11 [65] generically describes usability evaluation as "*the extent to which specified users can use a system, product, or service to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use*". ISO/IEC TR 9126-4 [66] also uses the term productivity to refer to efficiency. Translating to parallel programming, we describe the usability evaluation as **the extent to which developers can use a parallel API or tool to make source code execute computations in parallel with effectiveness, efficiency, and satisfaction in a specified context of use**. Effectiveness is "*the accuracy and completeness with which users achieve specified goals*". Efficiency or productivity evaluates the "*resources used in relation to the results achieved*", including human effort, costs, and materials. Satisfaction is defined as the "*extent to which the user's physical, cognitive and emotional responses that result from using a system, product, or service meet the user's needs and expectations*" [65].

From the literature review, we observe that most authors claim to perform usability assessments of parallel APIs without considering all the recommended practices (e.g., develop an experiment plan, perform a hypotheses test, and evaluate the validity) [20,41,42,46,62], while other works consider it [21,39,52,53,57]. Other works only consider productivity when taking experiments with people to evaluate specifically the effort when using parallel APIs [46,52,67]. Among these papers, only a few works considered programmers' satisfaction when evaluating usability [51–53]. These factors lead us to create a methodology for assessing the usability of parallel APIs. It also helps to propagate the concept without misrepresenting it. Moreover, we present a methodology to guide other researchers in this analysis and to conduct our experiment in the following sections.

To ensure that the experiment is performed correctly, a process is needed to provide steps to support the activity execution. A process

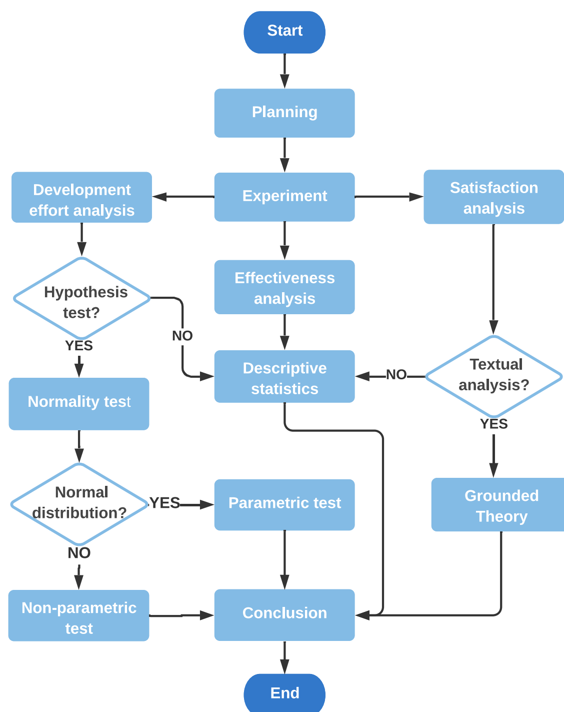


Fig. 8. Research method flowchart.

can be used as a checklist and guideline on what to do and how to do it [22]. Fig. 8 presents the methodology proposed to evaluate the usability of parallel APIs. The first step is called planning. In this phase, the problem to be solved and the experiment plan must be defined. The target problem can be defined through a gap found in a literature review. Also, the goals, context, hypothesis, procedure, study activity, and instruments must be defined.

Initially, the study goals are formulated from the problem to be solved and should reflect the purpose of the experiment. After this, the context of the experiment is defined, which can be characterized according to four dimensions: offline versus online, student versus professional, classroom versus real problems, and general versus specific problems [22]. Next, the hypotheses are formulated. The definition of the experiment is formalized into two hypotheses: The null hypothesis (H_0) and the alternative hypothesis (H_1). H_0 states that there is no significant difference between the observations, and this difference is by coincidence. H_1 is the hypothesis in favor of which H_0 is rejected [22].

After formulating the hypotheses, the study activity and the procedure are defined. The procedure presents all the rules and steps to be followed by the participants when performing the study activity. For example, whether participants will be able to access the Internet during the activity or whether there will be materials available for access, how much time they will be given to complete the activity, and other considerations to ensure there is no bias.

In the planning phase, the experimenters also choose and develop the instruments for the study, which can be objects, guidelines, or measuring instruments [22]. The experience objects can be, for example, specification documents or code. Guidelines are used to guide participants and may include, for example, procedure description and checklists. Measuring instruments are used for data collection, which can be done through forms, interviews, and instrumentation of the machines used by the participants, such as scripts to record the participants' screens and capture log files [22]. Also, before conducting the experiment, a pilot study is performed to test the experiment plan.

After the planning phase, the experiment is conducted, and the results achieved are analyzed. Effectiveness, efficiency, and users' satisfaction must be evaluated to determine the usability of a parallel API. Effectiveness should assess the accuracy and completeness with which participants achieve the goals specified in the study [65]. Descriptive statistics used to organize and summarize a data set [68] can be applied to evaluate effectiveness. For example, using the percentage, it is possible to express the proportion of participants who achieved the study goal concerning the total number of participants.

Efficiency or productivity can be evaluated through the effort spent to develop an application [65]. Descriptive statistics or hypothesis testing can be applied to evaluate the development effort results. Although descriptive statistics help to organize and summarize the evaluated data set, just the average time spent by the participants to develop a parallel application may not be enough to determine which parallel APIs provide the best productivity. Then, through a hypothesis test, it is possible to determine if there is a significant difference between the average times needed to develop the applications with each of the parallel APIs evaluated. If a hypothesis test is necessary, a normality test should first be performed to verify whether the sample has a normal distribution. A parametric test is applied if the sample has a normal distribution. Otherwise, the non-parametric test is applied [69,70].

Participants' satisfaction can be evaluated both qualitatively and quantitatively using forms [71]. For qualitative data, a textual analysis can be performed using a subset of the procedures for coding from Grounded Theory (GT), which is a specific methodology developed with the objective of building theory from data [72,73] popularly used in software engineering [74]. However, this study did not fully explore the GT methodology since we were not interested in generating theories. We aimed to use it to evaluate the qualitative perception of the participants over the activity [73]. Therefore, our methodology approached only a textual analysis based on the open and axial coding steps of GT methodology. Open coding is an interpretive process by which data is analytically divided. At this step, responses are analyzed accurately, and relevant events, actions, or interactions are compared to identify similarities and differences. They receive conceptual labels, and those conceptually similar are grouped to form categories and subcategories. In axial coding, the categories are related to their subcategories [72].

After analyzing the effectiveness, development effort, and participants' satisfaction, the conclusions over the results are presented. In addition, the study's limitations and threats to validity are discussed, and lessons learned are presented.

5. Experiment plan and execution

This section presents the study's variables, goals, hypothesis, and context. Next, the activity given to the participants and the procedure followed in this study is presented. In addition, the instruments used during the study are presented.

5.1. Independent variables

The parallel APIs evaluated in this study are independent variables. In this study, we evaluated the usability of three parallel APIs for stream processing on multi-core systems: FastFlow, SPar, and TBB. Each of them has specific characteristics that influence the development of the applications.

The experience of the programmer is one of the important independent variables [39]. In our study, the participants were intentionally beginners developers in the parallel programming domain to consider the impact of learning and help us understand the challenges faced by beginners.

The study environment is also an independent variable. Our study is conducted in a university and not in an industrial environment. Therefore, the study is not affected by factors in the industry environment. Even so, a deadline of one day was set for the participants to complete the activity.

5.2. Dependent variables

To evaluate the usability of parallel APIs, we assess the learnability, the effort required to parallelize a stream processing application, user errors, and satisfaction. The participants' learnability was assessed by the time accessing the material provided and the time to answer the form in seconds. The effort required to develop the parallel application includes the following activities: reading the procedure description, understanding the sequential application code, coding (or development), debugging, and testing. The time in seconds was considered to evaluate the first five activities. However, the time testing the application was not considered because the participants left the application running and continued to perform other activities. Therefore, the number of executions was measured. In addition, the errors made by the participants and the difficulties faced during the activity were measured.

5.3. Goals

The main goal of this study is to compare the usability of SPar, TBB, and FastFlow parallel APIs for implementing stream parallelism in C++ applications for multi-core systems. The specific goals are as follows:

- Measure the learnability;
- Measure the time spent to exploit parallelism;
- Report the implementation errors;
- Report the users' satisfaction.

5.4. Hypotheses

We consider the following seven hypotheses in our experiment based on the goals. The first two refer to learnability and the others to the development effort.

- H_{0_answer}: The time to answer the form is the same for FastFlow, SPar, and TBB;
- H_{0_study}: The time to access and study the material provided is the same for FastFlow, SPar, and TBB;
- H_{0_under}: The time to understand the application code is the same for FastFlow, SPar, and TBB;
- H_{0_read}: The time to read the procedure description is the same for FastFlow, SPar, and TBB;
- H_{0_dev}: The development time is the same for FastFlow, SPar, and TBB;
- H_{0_debug}: The debugging time is the same for FastFlow, SPar, and TBB;
- H_{0_exec}: The number of execution is the same for FastFlow, SPar, and TBB.

5.5. Context of study

The participants in this study were 15 graduate students from the graduate program in computer science (PPGCC) of the Pontifical Catholic University of Rio Grande do Sul (PUCRS), in Porto Alegre city South of Brazil. This study was part of the parallel programming course at PUCRS. Moreover, this study is conducted with participants having experience in the industry but beginners students in parallel programming.

The environment of this experiment is offline because it was conducted in an academic environment under controlled conditions and not in the industry. This study is specific because it focuses on assessing the usability of parallel APIs for stream processing in an academic environment. This study addresses a real problem common in the stream processing area: a video OpenCV processing application [22].

Video processing application

```
1. int main(){
2.   // initialization of the steps
3.   while (1){
4.     Mat src, res;
5.     total_frames++;
6.     inputVideo >> src;    // read frame
7.     if (src.empty()) break; // check if end of video
8.     vector<Mat> spl;
9.     split(src, spl);   //process - extract only the correct channel
10.    for (int i =0; i < 3; ++i){
11.      if (i != channel){
12.        spl[i] = Mat::zeros(S, spl[0].type());
13.      }
14.    }
15.    merge(spl, res);
16.    cv::GaussianBlur(res, res, cv::Size(0, 0), 3);
17.    cv::addWeighted(res, 1.5, res, -0.5, 0, res);
18.    Sobel(res,res,-1,1,0,3);
19.    outputVideo << res;   //write frame
20.  }
21.}
```

Fig. 9. Activity of study. *Source:* Adapted from [75].

5.6. Activity of study

The activity given to the participants was to implement stream parallelism in an OpenCV video processing application, which aims to extract the green channel from a video. Fig. 9 presents a piece of this application. The video processing application receives an input video and reads each video frame (line 6). A frame is processed through a series of operations to extract only the green channel (lines 9–18). Next, the frame with the green channel is written to the output video (line 19). This process is repeated until all frames have been processed (line 7) [75].

5.7. Study instruments

This section presents the instruments used in this study. A folder with materials about each evaluated parallel API was provided for the participants to access during the activity. This folder had a manual, which included an introduction to the parallel API, how to compile and run an application, essential features for implementing stream applications, and an example of use in C++. The folder also had a parallel code to be tested. In addition, a document describing the procedure was provided, which served as a guide for the participants during the execution of the activity.

The measuring instruments used in this study were questionnaires and a script to record the screens of the participants' machines. A first questionnaire was applied to collect background information on the participants (Table 2). The characterization questionnaire was used to evaluate the participants' experience with parallel programming, video processing, and others. The participants reported their level of experience among:

- **0:** None (I have never participated in such activities);
- **1:** I studied it in the classroom or in a book (I have only theoretical knowledge);
- **2:** I have practiced it in classroom projects (I have theoretical knowledge applied only in the university);
- **3:** I used it in personal projects (I have theoretical knowledge and individual practical experiences);
- **4:** I used it in some projects in industry or research (I have theoretical knowledge and little practical experience);

Table 2
Characterization questionnaire.

ID	Question
Q1.	Which is your academic background? (Technical Course/High School, Bachelor's degree, Master's degree, or Ph.D. degree)
Q2.	Which is the name of your course?
Q3.	Which is your level of experience with command line and text editor on the Linux operating system? (From 0 to 5)
Q4.	Which is your level of experience with the C++ programming language? (From 0 to 5)
Q5.	Which is your level of experience with parallel APIs (e.g., Pthreads, Cilk, TBB, FastFlow, MPI etc.)? (From 0 to 5)
Q6.	If you have any experience in Q5, please inform the parallel APIs for multi-core systems you have used and the features explored in your previous experiences:
Q7.	Which is your level of experience with developing stream processing applications (reading, writing, and processing files, network, video, audio etc.)? (From 0 to 5)
Q8.	If you have any experience in Q7, please specify which one you worked on:

Table 3
Procedure questionnaire.

ID	Question
Q1.	Which time did you start your activity?
Q2.	Which time did you finish your activity?
Q3.	Does the parallel program produce the correct result (the same as the sequential program)? (Yes/No)
Q4.	Which is the execution time of the sequential program (in seconds)?
Q5.	Which is the parallelism degree that reached the shortest execution time in the parallelized version? (From 0 to 8)
Q6.	Which is the execution time of the parallel program (in seconds) ?
Q7.	Was the manual helpful in performing the activity? (Yes/No)
Q8.	Was the activity successfully completed? (Yes/No)
Q9.	Which were your main difficulties in implementing parallelism in this activity?

Table 4
Evaluation questionnaire.

ID	Question
Q1.	With which parallel APIs did you perform your first activity?
Q2.	With which parallel APIs did you run your second activity with?
Q3.	If you had problems understanding the video application, what were they (please provide details)?
Q4.	Did you become uncomfortable with the screen-capture during the activity? Yes/No)
Q5.	Do you consider yourself able to perform activities with this parallel API after completing this activity? (From 0 to 5)
Q6.	Do you consider yourself capable of performing activities with TBB after completing this activity? (From 0 to 5)
Q7.	Do you consider yourself capable of performing activities with FastFlow after completing this activity? (From 0 to 5)
Q8.	Which is your level of experience with the C++ programming language after completing this activity? (From 0 to 5)
Q9.	Which is your level of experience with developing stream applications (reading, writing, and processing files, network, video, audio etc.) after completing this activity? (From 0 to 5)
Q10.	Which parallel API did you find the hardest?
Q11.	Justify your choice for question Q10 (please provide details):
Q12.	Which parallel API did you find the easiest?
Q13.	Justify your choice for question Q12 (please provide details):
Q14.	If you need to parallelize an application similar to the application used in the activities, which interface would you choose?
Q15.	Justify your choice for question Q14 (please provide details):

- **5:** I have used it in many projects in industry or research (I have theoretical knowledge and many real practical experiences).

One of the criteria to complete the activity was adequately filling out the Procedure questionnaire (Table 3), which was used to compare the parallel application with the sequential version. In addition, the experiment was monitored on the video to verify each participant's development process, such as the time spent reading the manual, developing and debugging. In the end, participants answered a final questionnaire (Table 4) reporting their main difficulties and facilities when performing the proposed activity, their experience level after this, and their satisfaction with the parallel APIs.

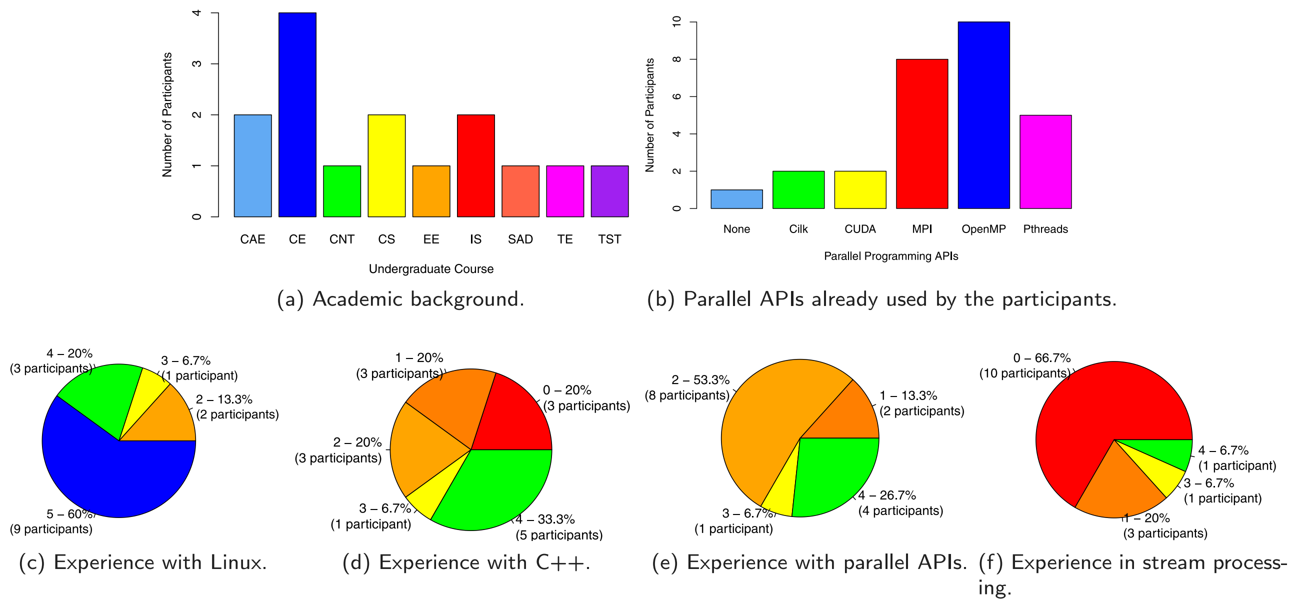
5.8. Procedure

This section presents the procedure of this study. Before a participant began performing the activity, a member of the experiment committee ran the screen-capture script to record the participants' screens during the activity. The participants could only access the folder with the provided material. It was forbidden to consult any material on the Internet for any purpose. Using the cell phone for consultation or distraction during the experiment was prohibited. The participants could only ask technical questions about the experiment. In addition, a deadline of one day was set for the participants to complete each activity.

In this study, we aimed to assess the usability of parallel APIs to express parallelism in the stream processing application domain rather than evaluate optimizations to achieve more efficient parallelism. Therefore, we defined some criteria for the activity to be considered complete: The participants could only complete the activity if the parallel application achieved a speedup greater than or equal to 3; if the parallel application produced the same result as the sequential version; and if the form was correctly filled out. Additionally, a member of the experimentation committee should verify that the activity has indeed been completed. If the activity has been completed, the member of the experiment committee stops the screen-capture script, collects everything inside the experiment folder, and deletes all records from the computer used.

5.9. Experimental setup

The study participants used multi-core workstations with an Intel® Core™ i7-4790 processor with eight cores (four physical), 3.6 GHz, and 15.6 GB of RAM. The operating system was Linux Mint 17.3 and G++ compiler version 5. The OpenCV version used was 3.1.0. The parallel APIs used were FastFlow 2.1.3, TBB 4.4.6, and SPar 1.



(a) Academic background. (b) Parallel APIs already used by the participants.

(c) Experience with Linux. (d) Experience with C++. (e) Experience with parallel APIs. (f) Experience in stream processing.

Fig. 10. Participants' background.

5.10. Pilot

A pilot is a test of the experiment plan before performing the execution itself. Generally, in a pilot, the methods and procedures are similar to the target study to produce data that help to evaluate the procedures [76]. In this work, a pilot was conducted with a few participants to guide the future study. These participants were not present in the final experiment. It was essential to adjust environmental problems and fix software issues. With this, we were sure that the procedure would work and that the participants would feel comfortable when performing the activity.

5.11. Experiment execution

This section presents how the experiment was performed. Experiments to evaluate usability may follow different approaches depending on the factors being evaluated. In this experiment, the participants were divided into three groups (each group with five students), varying the sequence of using the parallel APIs to parallelize the same OpenCV video processing application. The first group used SPar, TBB, and FastFlow; the second group used TBB, SPar, and FastFlow; and the third group used SPar, FastFlow, and TBB. After the experiment, the forms and screen-capture videos were analyzed to obtain data that can be used to evaluate the usability of the parallel APIs.

5.11.1. Participants' profile

This section presents the profile of the participants in this study. Fig. 10 details the participants' profiles, bringing participants' responses to the characterization questionnaire (see Table 2). Fig. 10(a) shows that all participants have graduated in one of the following undergraduate courses (Q2): Control and Automation Engineering (CAE), Computer Engineering (CE), Computer Network Technologist (CNT), Computer Science (CS), Electrical Engineering (EE), Information Systems (IS), System Analysis and Development (SAD), Teleinformatics Engineering (TE), and Telecommunications Systems Technologist (TST).

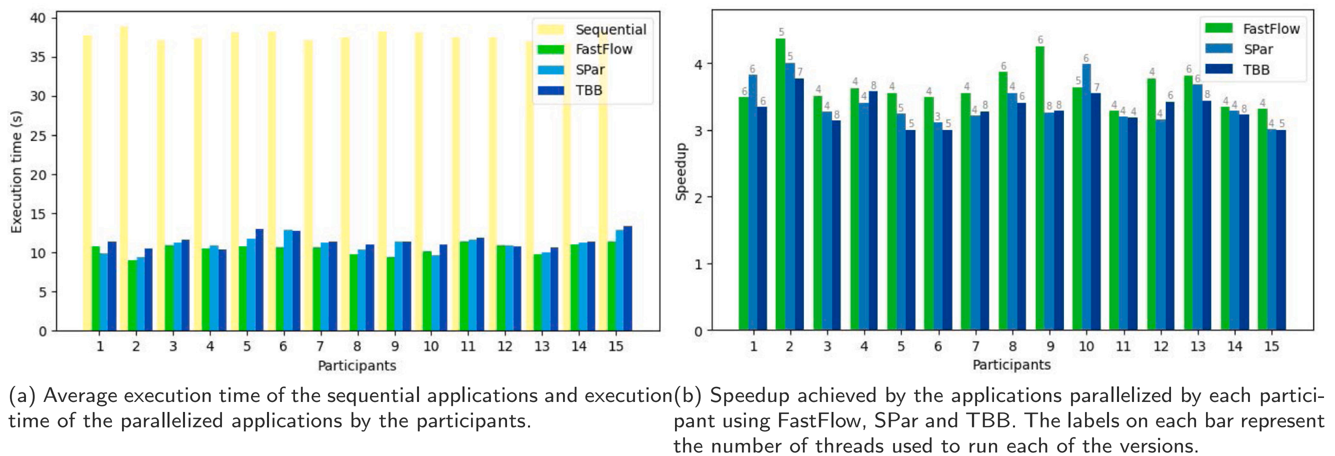
Fig. 10 also presents the participants' experience levels from 0 to 5 with command line and text editor in the Linux operating system (Q3), with the C++ programming language (Q4), with parallel APIs (Q5), and with the development of stream processing applications (Q7). Fig. 10(c) shows that all participants are familiar with the Linux operating system, using it in the classroom (2 participants), in personal projects (1 participant), and some (1 participant) or many (1 participant) projects in industry or research.

Fig. 10(d) shows that only 3 participants have no experience developing C++ applications, and 3 have only theoretical knowledge. The other participants have already developed some applications using C++ in classroom activities (3 participants), in personal projects (1 participant), and industry or research (5 participants).

Fig. 10(e) shows that most participants have already used parallel APIs, either to parallelize applications in the classroom (8 participants), on personal projects (1 participant), or in industry or research (4 participants). However, these participants are not experts in parallel programming because they have had a short time in contact with the parallel APIs. In addition, only 2 participants have never used parallel APIs, having only theoretical knowledge about them. Fig. 10(b) shows that most participants have used parallel APIs for multi-core (OpenMP, Pthreads, and Cilk) and distributed (MPI) systems. Only 2 participants have already developed GPU applications using CUDA. Moreover, only 1 participant has no experience with any parallel APIs.

Fig. 10(f) shows that most participants (10 participants) have no experience developing stream processing applications or only theoretical knowledge (3 participants). Although 2 participants had practical experience in developing stream processing applications, they are not considered experienced developers in this domain. The main reason is the little contact these participants have with this subject because they usually focus on developing the business logic code and not on developing these types of applications. Therefore, the participants in this group are considered beginners in the stream processing domain and satisfy the target sample of this experiment.

The answers to Q3, Q4, Q5, and Q7 were used to divide the groups of this experiment. An analysis of variance (ANOVA) was performed to verify if the division of the groups is close to equal. Two hypotheses were considered: the null hypothesis (H_0) stating that there are no differences between the experimental condition means, and the alternative hypothesis (H_1) stating that some means of the experimental condition are different. Given the conventional significance level (α) of 0.05, the ANOVA was applied for each question [38,77]. Table 5 shows that all P-values are greater than α, so all H_0 can be accepted. Therefore, there is no significant difference among the three groups, and the division of participants was done in a balanced way.



(a) Average execution time of the sequential applications and execution time of the parallelized applications by the participants.

(b) Speedup achieved by the applications parallelized by each participant using FastFlow, SPar and TBB. The labels on each bar represent the number of threads used to run each of the versions.

Fig. 11. Execution time and performance achieved by each participant using FastFlow, SPar, and TBB.

Table 5
ANOVA results.

Questions	Q3	Q4	Q5	Q7
F-value	0.86	0.46	0.38	0.27
P-value	**0.45**	**0.64**	**0.69**	**0.76**

5.11.2. Effectiveness

This section presents the effectiveness evaluation, which refers to the accuracy and completeness in which the participants achieve the specified objectives [65]. In this study, the participants' objective was to parallelize a video processing application using the parallel APIs FastFlow, SPar, and TBB. Furthermore, this goal would only be achieved if the participant met the three criteria presented in Section 5.8: performance, program correctness, and the correct completion of the Procedure questionnaire (Table 3).

The speedup was calculated to evaluate performance by dividing the sequential execution time (Q4) by the parallel execution time (Q6) [5]. Fig. 11(a) shows that all participants could reduce the execution time of the parallel applications. In addition, Fig. 11(b) shows that all applications developed by the participants achieved the minimum required performance (speedup $\geq$ 3) using the three parallel APIs. Each participant used a certain number of threads shown on each bar's label in the graph. However, this speedup is not necessary the optimal performance. To achieve more performance would require the participants to have more knowledge about the architecture, which is beyond the scope of this study. Therefore, we did not ask for the maximum speedup but only the minimum speedup as an indicator of parallelization success and the conclusion of the activity.

In order to evaluate the correctness of a program, it is necessary to verify if its execution against a known input generates the expected output [38]. All participants were able to parallelize the application in order to produce the expected output using the three parallel APIs. In addition, the Procedure questionnaire was adequately filled out by all participants. Therefore, FastFlow, SPar, and TBB showed effectiveness in this study. Thanks to our pilot experiment.

6. Development effort analysis

This section presents the evaluation of the effort required to develop a video processing application using FastFlow, SPar, and TBB. Development effort is usually evaluated considering the time required to develop an application. In Q1 and Q2 of the Procedure questionnaire (Table 3), a participant must inform the start and end time of the activity, which can be used to calculate the total development time. However, the effort to develop an application includes several activities, such as planning, coding, and debugging. Therefore, in this study, we analyzed the time taken for each of the activities performed by the participants during the development of the video processing application rather than evaluating the total development time.

Eight factors were collected from the screen-captured videos to assess the development effort: time to answer the form, time accessing the material available, time to understand the application code, time to read the procedure description, development time, debugging time, number of executions, and error types. To analyze the first seven metrics, we performed a paired hypothesis test. To analyze the error types, we used descriptive statistics. In addition, the programming errors analyzed were divided between compile-time errors and programming logic errors.

6.1. Hypothesis test

The development effort is a factor that must be evaluated to determine the usability of a parallel API [65]. To assess this factor, we considered the time taken by participants to complete the following activities in seconds: time to answer the form, time accessing the material provided, time to understand the application code, time to read the procedure description, development time, and debugging time. The time for testing and running the application was not measured, as the participants would leave the application running and continue to perform other activities, e.g., accessing the material provided. Instead of these metrics, the number of executions of the application was measured. This metric represents how many times a participant has tested the parallel application until they can reduce the execution time by up to three times the execution time of the sequential version.

Fig. 12 shows the box plots for each of the metrics evaluated. Based on the box plots, it can be seen that the lowest medians for the time to answer the form, time to access the provided material, development time, debugging time, and the number of executions were obtained when participants used SPar. On the other hand, when they use FastFlow, the lowest medians were obtained for the time spent understanding the application code and reading the procedure description. Since the value of the medians alone cannot determine which parallel API offers the best usability, it is necessary to perform a hypothesis test.

To verify whether the data collected has a normal distribution or not, we first performed the Shapiro–Wilk test at the conventional significance level (α) of 0.05 [70]. This test was chosen because it is one of the most powerful tests for all distribution types and is independent of sample sizes [78]. Two hypotheses were considered: H_0 stating that the samples show a normal distribution, and H_1 stating that the samples do not have a normal distribution. Table 6 shows the results of the Shapiro–Wilk normality test. The P-values greater than 0.05 are

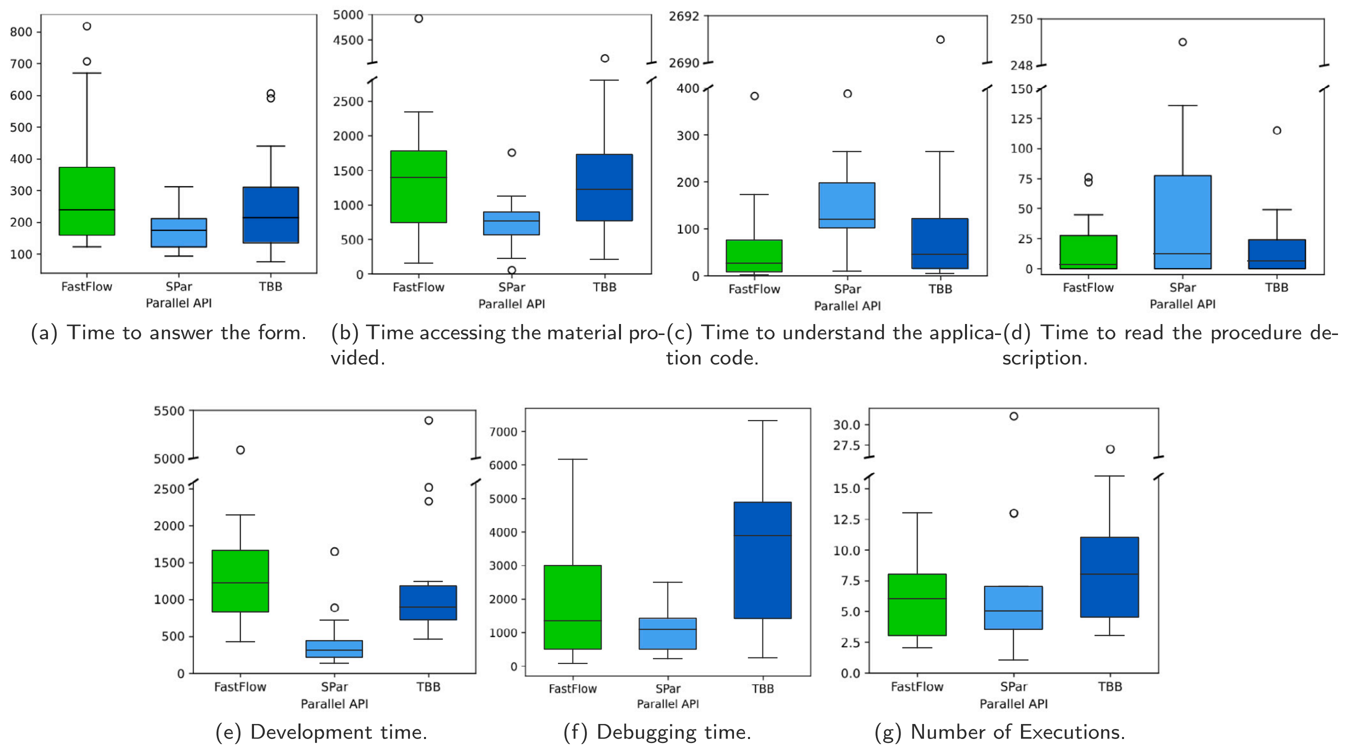


Fig. 12. Box Plot for the times collected in the experiment.

Table 6
Shapiro–Wilk and Hypothesis Test.

Metrics	Shapiro–Wilk			Hypothesis Test		
	SPar	FastFlow	TBB	SPar x FastFlow	SPar x TBB	FastFlow x TBB
	p-value	*p*-value	*p*-value	*p*-value	*p*-value	*p*-value
Time to answer the form (s)	**0.5166**	0.0033	0.0286	0.0353	**0.1914**	**0.6788**
Time accessing the material provided (s)	**0.4828**	0.0085	**0.0811**	0.0026	0.0147	**1.0000**
Time to understand the application code (s)	**0.3586**	0.0001	4.06E−07	0.0170	**0.2524**	**0.2524**
Time to read the procedure description (s)	0.0010	0.0002	0.0001	**0.0618**	**0.1959**	**0.8753**
Development time (s)	0.0003	0.0004	4.82E−05	0.0006	0.0067	**0.6788**
Debugging Time (s)	**0.5247**	0.0333	**0.2026**	**0.0730**	0.0012	**0.1688**
Number of Executions	0.0003	**0.1289**	0.0068	**0.7257**	**0.2778**	**0.1015**

presented in bold in Table 6. H_0 can be accepted for these cases because the samples have a normal distribution. For all other cases, the P-values are less than α, so H_0 can be rejected because these samples do not have a normal distribution.

Considering the hypotheses presented in Section 5.4, we performed a paired hypothesis test for two samples [79]: SPar versus FastFlow, SPar versus TBB, and FastFlow versus TBB. The hypothesis test between SPar and TBB was performed using the parametric Student's t-test for the time to consult material and debugging time because the samples have a normal distribution for such metrics. For all other cases, the non-parametric Wilcoxon test was performed. The t-test evaluates the hypotheses considering the populations' mean, while the Wilcoxon test considers the medians [70,79].

Table 6 also shows the results of the Wilcoxon and t-test considering a conventional α of 0.05. For the metric time to answer the form, the P-value is greater than α for SPar versus TBB, and FastFlow versus TBB (in bold). So, there is no significant difference with respect to the time spent by the participants to answer the form in the activities using SPar and FastFlow compared to the TBB activity. Between SPar and FastFlow, the P-value is smaller than α, so there is a significant difference between these results. The time spent by participants to answer the questionnaire during the FastFlow activity may have been longer due to the volume of information reported by the participants.

For the time accessing the material provided, the P-value is 1 (greater than α) for FastFlow versus TBB. Therefore, there is no difference between the time spent by the participants studying the FastFlow and TBB APIs. On the other hand, the P-value is less than α for SPar versus FastFlow and TBB, confirming the results in Fig. 12(b). Therefore, H_{0_study} can be rejected because the effort in accessing the material for learning FastFlow, TBB, and SPar were not equal. It was lower for SPar.

As can be seen in Table 6, the P-value is greater than α for SPar versus TBB, and FastFlow versus TBB (in bold). Therefore, there is no significant difference in the time spent by the participants to understand the sequential application in the activity using TBB compared to SPar and FastFlow. However, when comparing SPar with FastFlow, the P-value is lower than the α value, confirming there is a significant difference between the time spent by participants to understand the sequential application code in each of the activities. These results highlight that the effort to understand the sequential application was the lowest in the activity using FastFlow because two groups finished the activity using it.

As seen in Table 6, the P-value is greater than α for the time to read the procedure description in all cases. Therefore, H_{0_read} cannot be accepted, although the time spent by the participants reading the procedure description is different for each of the parallel APIs used (Fig. 12(d)). These results show that the effort to understand the study

Table 7
The most frequently occurring compile-time errors in the activity using SPar.

ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Type	Description	No.
Group	2	1	3	3	3	2	1	3	2	2	3	1	2	1	1			
	7	9	9	10	8	6	12	3	4	4	7	34	2	26	15	Variable was not declared	A variable was called inside `ToStream/Stage` but is not in the required `Input/Output`	156
	–	–	43	–	–	–	–	–	–	–	–	39	33	–	–	A compilation directive is missing	The OpenCV compile directive is missing from the command line	115
	12	–	–	–	–	–	–	6	–	–	6	24	–	–	12	Invalid character	A character was copied from the PDF file	60
	–	1	–	–	1	6	3	–	2	–	2	2	–	–	2	Expected]	There is no closing bracket (]) at the end of annotation or a syntax error has been made	19
	–	2	–	–	3	–	2	–	4	–	4	2	–	–	–	Expected }	A closing curly bracket (}) is missing at the end of annotation	17
	–	–	1	–	–	–	7	–	–	–	2	1	1	–	2	Annotation with syntax error	There is a syntax error in the annotation, e.g., a letter or a parenthesis is missing	14
	–	–	1	–	–	–	7	–	–	–	2	–	1	–	2	Expected {	An opening curly bracket ({) is missing after an annotation	13
	–	–	–	–	–	–	10	–	–	–	–	1	–	–	–	Variable was not received	The required variables are not in the `Input` to the compute OpenCV functions	11
	–	–	–	–	–	–	–	–	1	–	2	4	3	–	1	Unrecognized command line option	4.86cmA compile flag is in the wrong place or its syntax is wrong	11
	–	–	–	–	–	–	2	–	–	–	–	2	3	–	2	Semantic error in `ToStream`	At least one `Stage` attribute must be declared inside `ToStream`	9

procedure is greater for the activity using SPar because two groups started the activity using it. However, it was possible to demonstrate through a hypothesis test that there is no significant difference between the efforts spent by the participants to understand the procedure in each of the activities.

For the development time, the P-value is less than α for SPar versus FastFlow, and SPar versus TBB. Therefore, H_{0_dev} should be rejected because there is a significant difference in the time required to develop an application with SPar regarding FastFlow and TBB. The P-value is greater than α for FastFlow versus TBB, so there is no significant difference between the time spent by the participants to develop the applications using both APIs. These results highlight that the effort to parallelize the application was lower using SPar than FastFlow and TBB.

For debugging time, the P-value is greater than α for SPar versus FastFlow, and FastFlow versus TBB (in bold). Therefore, there is no significant difference between the medians. Although the debugging time for the applications developed with SPar is the shortest (see Fig. 12(f)), from the hypothesis test, it is possible to show that there is a significant difference only between the debugging times of SPar and TBB (P-value $= 0.0012 < 0.05$). There is no significant difference between the debugging times of SPar and FastFlow. Then these results highlight that the effort to correct programming errors is lower for SPar and FastFlow.

In all comparisons, Table 6 shows that the P-value is greater than α for the number of executions. Although the participants have executed a smaller number of times the application parallelized with SPar concerning the other parallel APIs (Fig. 12(g)), from the hypothesis test, it was possible to show that there is no significant difference in the number of times the applications were executed. These results highlight that the effort to test the applications parallelized with FastFlow, SPar, and TBB was equal.

6.2. Compile-time errors

This section shows the results of compile-time errors collected from the screen-captured videos.

6.2.1. Common compile-time errors in SPar

Table 7 shows the most common compile-time errors made by the participants in the activity using SPar. A compile-time error occurred when participants tried to call a variable within the scope of an `ID` attribute (`ToStream` or `Stage`), but this variable was not declared within its scope or as a global variable (156 times). Some participants did not inform the correct variables to the `Input` and `Output`, resulting in a compilation error (11 times).

When the participant did not use the OpenCV flag (`-cflags opencv'`) and library (`-libs opencv'`) on the command line to compile the code, a linker error occurred (115 times). Since the linker errors do not allow the execution of the code as well as the compile-time errors, they were included in Table 7. Some participants also used the SPar compile flag (`-spar_blocking`) in the wrong place on the command line, or there was a syntax error in it, resulting in a compile-time error (11 times). In addition, invalid characters were also found in the code, resulting in compile-time errors 60 times.

In SPar, the annotated region must be between curly brackets ({...}), and the annotations must be declared using double brackets (`[[attr-list]]`) [32]. Compile-time errors occurred when some participants did not use a curly bracket at the beginning of the annotated region (13 times) or at the end of the annotated region (17 times). Compile-time errors also occurred when some participants did not put a closing bracket (]) at the end of annotation, or put it in an incorrect place (19 times), or there was a syntax error in the SPar annotation (14 times). Some participants annotated the code using only the `ToStream` annotation, which also resulted in an error. SPar semantic requires that at least one `Stage` is within an annotated region (`ToStream`).

6.2.2. Common compile-time errors in FastFlow

Table 8 shows the most common compile-time errors made by participants in the activity using FastFlow. A compile-time error occurred when the participants called a local variable (usually declared in the main function of the program) within a stage (160 times). As with the SPar, some participants copied the example code from the manual (in PDF format), resulting in compile-time errors (46 times). A participant

Table 8
The most frequently occurring compile-time errors in the activity using FastFlow.

ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Type	Description	No.
Group	2	1	3	3	3	2	1	3	2	2	3	1	2	1	1			
	2	8	51	43	9	2	4	12	5	2	6	2	2	2	10	Variable was not declared	A variable was called inside a stage, but it was not declared in this scope	160
	1	–	14	28	10	–	2	–	–	2	–	9	–	–	1	No matching function for call	The function received an argument of the wrong type	67
	–	–	2	44	6	–	3	–	–	7	–	–	–	–	–	Invalid initialization of the variable	A variable was declared using a pointer but was initialized without using a pointer	62
	–	–	6	17	4	–	–	1	–	–	–	6	–	–	12	Parameter incorrectly passed to stage	Variables were passed as parameters to the `svc` method	46
	–	–	–	36	–	–	–	3	–	7	–	–	–	–	–	Invalid character	A character was copied from the PDF file	46
	5	–	1	–	8	–	1	5	–	15	–	2	–	1	6	Cannot convert types in `return`	A variable of a different type than the return type of the `svc` method was returned	44
	–	–	–	22	9	–	–	3	–	1	–	–	–	–	3	Cannot declare an object of abstract type	The `svc` method was declared of one type and its parameter of another type	38
	–	2	4	1	3	–	7	–	3	4	–	2	2	–	4	Expected template name before < token	FastFlow header or namespace is missing, or there is a syntax error in ff_node	32
	–	–	6	6	–	–	2	6	2	–	4	1	–	–	–	Template argument 1 or 2 is invalid	The `ff_node_t` was declared of type `std::Mat`, but the `Mat` type belongs to `cv namespace`	27
	–	–	2	8	5	–	–	1	3	3	–	–	–	–	–	Expected ;	There is no semicolon after a member declaration, `struct` definition, or variable name	22

forgot the semicolon at the end of a variable, `class`, or `struct` declaration (22 times).

In FastFlow, each stage has an `svc` method, which has a pointer as a parameter. We observe that some participants tried to pass variables as parameters to the `svc` method, resulting in compile-time errors (46 times). Some participants tried to return a variable of a different type than the `svc` method returns (44 times). Some participants passed an argument of the wrong type to a function or method (67 times). Other participants declared a variable using a pointer, but they called it without using the pointer (62 times).

Moreover, other participants forgot to include the header files of the FastFlow libraries (#include <ff/pipeline.hpp> and #include <ff/farm.hpp>), or they did not include the `using namespace ff;` directive, resulting in a compile-time error (32 times). When the participant declared the `ff_node_t` of type `std::Mat` (ff_node_t<std::Mat>), a compile-time error occurred (27 times) because the `Mat` type belongs to the `cv namespace` and not the `std namespace`.

6.2.3. Common compile-time errors in TBB

Table 9 shows the most frequently compile-time errors made by participants in the activity using TBB. In TBB, each stage can be modeled as classes extending the `tbb::filter`. The most common error made by the participants was calling a variable declared in the main function within a stage (243 times). In TBB, each stage class needs to implement the virtual `operator` method, which has a pointer as a parameter. A compile-time error occurred when some participants declared the `operator` method of a given data type and its parameter of another type, e.g., `void *operator(Mat* item)` (44 times). An error also occurred when a variable was declared using a pointer, but it was called in the code without a pointer (33 times).

Other participants passed an argument of the wrong type to a function or method (89 times). In addition, a linker error occurred when a participant did not use the OpenCV flag (`-cflags opencv‘`), OpenCV and TBB libraries (`-libs opencv‘ -ltbb`) on the command line when compiling the code (73 times).

In the middle and last stages of TBB, the `static_cast` operation must be used to convert the `void` pointer to the `Mat` data type, and the `new` command must be used to allocate memory for the `void` pointer. We observed that some participants did not use the `static_cast` (54 times) command, or they did not use `static_cast` or `new` commands correctly (68 times), or there was a syntax error in the `static_cast` command (41 times), resulting in compile-time errors. Some participants did not use the `static_cast` command to convert the `void` pointer to the <vector>Mat data type before calling `spl[0].type`, resulting in another compile-time error (45 times). Finally, some participants tried to return a different variable type from the return type of the `operator` method.

6.3. Programming logic errors

Types of programming errors are divided between runtime errors and compile-time errors [80]. The compile-time errors collected from the screen-captured videos were discussed in the previous section (Section 6.2). On the other hand, the runtime errors occur while a program is running and, in most cases, are caused by programming logic errors. This section shows the results of programming logic errors, which occur because of some flaw in the program logic and causes incorrect, unexpected, or unintended output [80].

6.3.1. Common logic errors in SPar

Table 10 shows the programming logic errors made by participants in the activity using SPar. It also presents some programming logic errors in C++ language. The most recurrent error made by participants was to pass the global variables `inputVideo` and `outputVideo` as parameters of the `Input` and/or `Output` attributes of `ToStream` or a `Stage` (20 times). Thus, the output video could not be generated.

The stream processing logic foresees the creation of at least three stages for the video processing application in this study: a first stage for reading an input video frame, a middle stage for extracting the green channel from the video, and a last stage for writing the frame to the output video. If less than three stages are created, the read and write

Table 9
The most frequently occurring compile-time errors in the activity using TBB.

ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Type	Description	No.
Group	2	1	3	3	3	2	1	3	2	2	3	1	2	1	1			
	53	10	1	2	3	2	2	4	20	61	9	7	56	4	9	Variable was not declared	A variable was called inside a stage, but it was not declared in this scope	243
	3	6	1	–	–	–	1	–	4	8	–	32	15	4	15	No matching function for call	The function received an argument of the wrong data type	89
	–	–	–	–	–	–	–	–	–	–	–	–	42	31	–	A compilation directive is missing	The OpenCV and TBB compile directive are missing	73
	4	13	–	–	–	–	–	–	2	20	–	17	3	–	9	Invalid conversion from types	There is an assignment between variables of different data types or a conversion error	68
	6	2	3	4	–	8	–	–	5	9	1	2	3	16	9	Cannot convert types in return	A variable of a different type than the return type of the operator method was returned	68
	–	11	33	–	–	–	–	–	1	–	–	7	–	2	–	`void*` is not a pointer to object type	The `static_cast` method was not applied before calling the operator parameter	54
	–	8	–	–	–	–	–	–	–	–	–	25	–	–	12	Error calling the variable `spl`	A pointer is missing or there is a syntax error in `static_cast` when converting `spl`	45
	–	1	12	3	–	–	–	–	–	3	–	19	–	6	–	Cannot declare an object of abstract type	The operator method was declared of one type and its parameter of another type	44
	–	2	–	–	–	–	–	–	3	2	1	15	–	13	5	Data type conversion error	There is a syntax error when converting the variable using `static_cast`	41
	2	3	3	–	–	1	2	3	7	–	–	–	6	3	3	Invalid initialization of the variable	A variable was declared using a pointer but was initialized without using a pointer	33

Table 10
The most frequently occurring programming logic errors in the activity using SPar.

ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Type	Description	No.
Group	2	1	3	3	3	2	1	3	2	2	3	1	2	1	1			
	–	5	–	2	2	–	2	–	1	3	–	5	–	–	–	Global variable in `Input/Output`	Global variables in the `Input/Output` prevent the generation of the output video	20
	–	1	–	–	–	–	5	–	–	–	–	5	3	–	5	Less than three stages	Reading, extracting and writing operations are performed in one or two stages	19
	–	4	–	–	–	–	–	–	–	–	–	6	–	–	5	`Replicate` attribute at the wrong stage	`Replicate` attribute is used in the frame reading stage or the frame writing stage	15
	–	3	–	–	–	–	–	–	1	1	2	2	–	3	1	Variable in `Output` and not in `Input`	A variable was placed in the `Output` of the `Stage/ToStream` instead of the `Input`	13
	9	–	–	–	–	–	–	–	–	–	1	–	–	–	–	Wrong variable in `Input`	A variable must be consumed in a `Stage`, but another variable was placed in `Stage Input`	10
	–	–	1	5	–	–	–	–	–	–	–	–	–	–	–	SPar flag on execution command line	The Spar `-spar_ordered` flag was used in the command line for the execution	6
	–	–	–	–	–	–	–	–	–	–	–	1	–	3	–	The incorrect region was parallelized	A region of the code that does not perform stream processing has been parallelized	4
	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	Wrong variable in `Output`	A variable must be produced for the next `Stage` but it is not in the current `Stage Output`	4

operations must be performed together with the processing operation. Therefore, the `Replicate` attribute cannot be used, and no parallel processing can be applied. The participants implemented less than three stages 19 times. Some participants also tried to create more than three stages. Although it was not a programming logic error, it did affect the performance of the application. Moreover, when the `Replicate` attribute was used in the reading and writing stages (first and last stages), the application execution was suddenly interrupted.

We observed that some participants produced logic errors because they could not understand the producer/consumer relation and data dependency with the `Input` and `Output` attributes. For example, they provided wrong/unused variables as arguments for these attributes.

Table 11
The most frequently occurring programming logic errors in the activity using FastFlow.

ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Type	Description	No.
Group	2	1	3	3	3	2	1	3	2	2	3	1	2	1	1			
	3	1	12	13	3	–	4	3	3	3	6	5	3	–	11	`ff_node_t` declared with the wrong type	The `ff_node_t` was not declared of type `Mat`	70
	3	–	–	–	1	1	–	–	–	5	–	8	–	1	–	`return *variable` or `return &variable`	The pointer or the address of a local variable has been returned	19
	–	–	–	15	-	–	–	–	–	–	–	3	–	–	–	Parameters of Stage in `svc` method	Variables were passed as parameters of a stage inside the `svc` virtual method	18
	3	–	3	–	–	–	–	–	–	–	–	6	–	–	3	One of the stages is missing	In the main function, one of the stages was not called within the template class	15
	–	2	2	3	–	2	2	1	1	–	–	–	1	–	–	`ff_Pipe/Farm` declared with the wrong type	The `ff_node_t` class was declared of `Mat` type but the `ff_Pipe/Farm` template class was not	14
	3	–	4	3	–	–	–	–	–	2	–	–	–	–	–	The `return` command is missing	The first stage returns `EOS`, the middle returns the stream item, and the last returns `GO_ON`	12
	–	–	2	1	2	–	2	1	–	2	–	–	–	1	–	Wrong return	A variable must be received by the next stage but it is not returned by the current stage	11
	–	–	–	4	1	–	–	–	–	3	1	–	–	1	–	The `delete` command is missing	In FastFlow, the last stage must contain the `delete item` command before returning `GO_ON`	10
	1	–	1	–	–	–	–	–	–	–	–	2	–	–	1	Pipeline was not run	The method `run_and_wait_end()` was not called to execute the Pipeline in the main function	5

Another logic error committed by participants was to parallelize the incorrect region of the application code. Finally, there was a command-line logic error because they were specified `-spar_ordered` in the correct order.

6.3.2. Common logic errors in FastFlow

Table 11 shows the most frequently programming logic errors made by participants during the activity using FastFlow. A programming logic error occurred when the participants declared the `ff_node_t` class with a data type other than `Mat*` type (70 times). In addition, some participants tried to provide a variable within a stage as a parameter to the `svc` method (18 times) and not to the `stage()` constructor method.

Some participants had difficulty using the `return` command at the end of each FastFlow stage, producing logic errors. Some participants forgot to return `EOS` in the first stage, the variable produced by the middle stage for the last stage, or `GO_ON` in the last stage (12 times). Some participants did not return the expected variable (11 times), returned a pointer to a variable (`return *variable`) or the address of a local variable (`return &variable`) (19 times). Also, some participants forgot to use the `delete` command in the last stage (10 times) to deallocate memory before call `return GO_ON`.

In FastFlow, each stage that will be executed must be specified in the `ff_Pipe<>` or `ff_Farm<>` methods. Some participants did not include one of the stages in Pipeline or Farm, producing a logic error (15 times). Other participants declared in the main function a Pipeline or Farm pattern of a different data type from the `ff_node_t` class data type (14 times). Finally, some participants also produced logic errors when they did not execute the Pipeline or Farm using the `run_and_wait_end()` method in the main function (5 times).

6.3.3. Common logic errors in TBB

Table 12 shows the programming logic errors most frequently made by participants in the activity using TBB. A logic error occurred when the participants parallelized code regions that should remain sequential in the main function (97 times). As mentioned above, the stream processing logic provides for the creation of at least three stages for the video processing application in this study, and only the intermediate stage can be processed in parallel. When the parallel filter was used in the first and last stages (reading and writing), the application execution was suddenly interrupted (19 times). On the other hand, when less than three stages were created, the reading or writing operations were performed together with the processing operation, producing the error when participants used the parallel filter (16 times). Some participants also created more than three stages, affecting the performance of the application.

In TBB, the parameters of a filter class had to be provided within the `stage()` construct method (e.g., `stage1(int channel) :c(channel)`). Some participants incorrectly provided a variable as a parameter to a stage incorrectly (97 times) or a variable as a parameter to the `operator` method (31 times). We also observed logic errors due to incorrect use of the `return` command by participants (29 times), resulting in the suddenly interrupted execution of the application. In addition, there were errors related to the `delete` command usage. Some participants did not use the delete command for deallocating memory in the last stage (7 times) or used it in the wrong stage (5 times).

After creating the classes for each stage, the objects of the stages had to be declared and added to the Pipeline (in order) through the `add_filter` command. Some participants did not add one of the stages to the Pipeline (12 times), resulting in the application execution being interrupted. At last, segmentation faults occurred when several instances of the same stage were added to the Pipeline. Moreover, if a stage was not instantiated in the main function, it could not be added to the Pipeline using the `add_filter` command (see Fig. 5) (6 times).

Table 12
The most frequently occurring programming logic errors in the activity using TBB.

ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Type	Description	No.
Group	2	1	3	3	3	2	1	3	2	2	3	1	2	1	1			
	45	–	–	–	–	–	–	–	5	37	–	–	10	–	–	Incorrect computations inside the stage	Computations that should continue in the main function were placed within the stages	97
	4	–	–	–	–	2	1	–	–	18	–	22	–	–	–	The stage is receiving parameters incorrectly	No attribution of the declared variables as parameters of a stage was performed	47
	4	–	–	–	–	–	1	–	–	14	–	12	–	–	–	Parameters of stage in `operator` method	The variables were passed as parameters of the `operator` instead of the stage constructor	31
	8	1	–	–	1	–	1	–	3	8	–	–	–	6	1	The `return` command is missing	First and middle stages return the stream item, and first and last stages return `NULL`	29
	4	–	1	–	–	–	–	–	1	9	–	1	3	–	–	Parallel at the wrong stage	Parallel filter is used in the frame reading stage or the frame writing stage	19
	–	–	–	–	–	–	–	2	2	3	–	–	5	2	2	Less than three stages	Reading, extracting and writing operations are performed in one or two stages	16
	2	1	–	–	–	–	–	–	3	1	4	–	–	1	–	One of the stages is missing	In the main function the `add_filter` method was not used to add a stages to the Pipeline	12
	–	–	–	–	1	–	–	–	1	2	–	–	–	3	–	The delete command is missing	In TBB, the last stage must contain the `delete item` command before returning `NULL`	7
	–	1	–	–	–	–	–	–	1	–	4	–	–	–	–	A stage was not instantiated	One of the classes of filters (stage) was not instantiated in the main function	6
	–	–	–	–	–	–	1	–	–	–	2	–	–	2	–	The `delete` command at the wrong stage	In the TBB, only the last stage should contain the `delete item` command	5

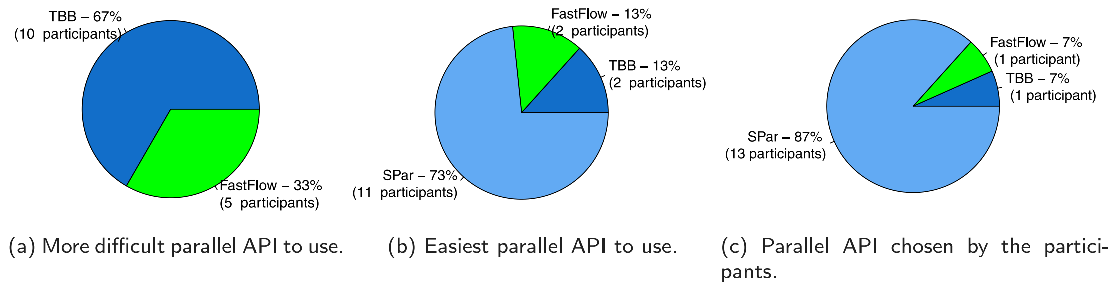


Fig. 13. Participants' opinion regarding the parallel APIs used.

7. Satisfaction analysis

After evaluating the development effort, the satisfaction of the participants was surveyed. For this purpose, quantitative and qualitative data collected through the Procedure (Table 3) and Evaluation (Table 4) questionnaires were analyzed. These questionnaires are self-assessing, so they reflect the participants' opinions.

7.1. Quantitative analysis

This section presents the quantitative results of participant satisfaction. In Q7 of the Procedure questionnaire, the participants reported whether the manual was useful to perform the activity or not. All participants agreed that the SPar and TBB manuals were useful. However, two participants diverged from the others, saying that the FastFlow manual was not useful. Moreover, all participants agreed that the activity was completed successfully (Q8).

In Q5, Q6, and Q7 of the Evaluation questionnaire, participants were required to inform their ability to perform activities with SPar, TBB, and FastFlow on a scale from 0 to 5. To compare these responses, the average of the 15 participants was taken. The average of SPar equals to 2. On the other hand, the averages of TBB and FastFlow are equal to 1.8. From the average analysis, the participants considered themselves more capable of developing applications using SPar.

In Q8 and Q9, the participants were required to inform their experience developing C++ applications and stream applications on a scale from 0 to 5. Before conducting the experiment, participants answered the same questions on the characterization questionnaire. In order to compare these responses, the average of the 15 participants was taken. The average of participants' experience in developing C++ applications before the experiment is equal to 2.13 and after the experiment is equal to 2.53. The average of participants' experience in developing stream processing applications before the experiment is equal to 0.66 and after the experiment is equal to 2.33. This analysis is just an overall idea and comparison. The ideal approach to evaluate the participants' knowledge is through a proficiency test before and after the experiment [51]. This would demand much more effort and time, therefore, we opted for not following the ideal approach as it is not the central point of this work. Despite that, these results were expected and reflect positively on our experiments.

(a) Difficulties with FastFlow. (b) Difficulties with SPar. (c) Difficulties with TBB.

Fig. 14. Difficulties in the activity.

In Q10 of the Evaluation questionnaire, participants are asked which parallel API they had more difficulty performing the activity. Fig. 13(a) shows a graph with the parallel APIs chosen by the participants. None of the participants considered that it was more difficult to program with SPar. From the 15 participants, 10 considered it was harder to program with TBB, and 5 considered it was harder to program with FastFlow. On the other hand, in Q12, participants are asked which parallel API they considered easiest to perform the activity. Fig. 13(b) shows that 11 participants found it easiest to program with SPar. In addition, only 2 participants thought it was easier to program using FastFlow or TBB.

In Q14, participants should inform which parallel API they would choose if they needed to parallelize an application in the future. Fig. 13(c) shows that most participants would have chosen SPar (13 participants). FastFlow and TBB were chosen by only 1 participant each.

7.2. Qualitative analysis

The qualitative analysis was conducted using the open and axial coding of Grounded Theory (GT) (Section 4). In the open coding, participants' answers were analyzed in detail to create codes related to specific passages of those responses. Next, the axial coding was performed, where categories were identified and related to their subcategories, creating relationships. Graphical representations were created using ATLAS.ti tool.[5] The relationships between codes and categories are represented by a red line, and the relationships between the codes are represented by black lines with a label describing the type of relationship.

In the Procedure questionnaire (Table 3), participants should report their main difficulties in implementing parallelism with FastFlow, SPar, and TBB for Q9. Fig. 14 shows a graphic representation for the participants' difficulties. Regarding to FastFlow (Fig. 14(a)), the participants reported difficulties in using the **OpenCV** functions, **modified the code**, **division of the stages**, **compiled the code**, **found programming errors**, among others. Due to the participants' difficulties in **understanding FastFlow skeletons** and **the difference between Farm and Ordered Farm**, there were difficulties in **using Ordered Farm**. Consequently, there was difficulty in **generating the output video correctly**.

The main difficulty reported by the participants was understanding the **communication among the stages**. The quotes below highlight that this difficulty can be caused by the difficulty in **selecting the type of pointer for communication among the stages**:

> "In general, the biggest difficulty was managing C++ pointers. I was confused about how to turn variables into pointers and how to pass them on to the other stages (e.g., from Emitter to Worker)." [Participant 10]

> "I had a small problem understanding how to correctly define the types of data that are transmitted among the stages." [Participant 11]

This may be related to the participants' difficulties in using the C++ programming language. Participants reported difficulties in **using namespaces**, **C ++ data structures**, and **C++ pointers**. These difficulties occurred due to the participants' **lack of practice and experience with the C++ language**. Due to the **lack of C++ documentation**, participants also reported **difficulties in using C++ pointers**. In addition, the main problem reported by the participants was the **lack of practice/experience with the C++ language**, which together with a **little experience with stream processing**, is also related to the participants' difficulty in **understanding the sequential code**.

In the activity using SPar, participants did not report specific difficulties with C++, but only with the parallel API. Fig. 14(b) shows that the participants had difficulties in **finding the programming errors** in the application because they could not **distinguish the code errors from the warnings**. Participants had difficulty **understanding the sequential code** and **defining the number of stages required**, resulting in a **failure to increase the speedup**. In addition, participants reported difficulties in **compiling the code** and **identifying the middle stage**.

The main difficulty was to correctly **fill in the attributes of Input and Output**, which is caused by the participants' difficulty in **understanding SPar logic**. In this regard, the main error of the participants was in **using a global variable in the Input and Output attributes**. Due to this error, participants had problems with the **scope of variables**, where they could not define which variables should be global and local. This type of error occurred due to the lack of more detailed explanations in the manual provided. They could not **understand the meaning of the term variable consumed**. The quotes below exemplify this aspect:

> "I forgot that a global variable cannot be used within the process of parallelism, because all workers see and write on them." [Participant 4]

> "Due to the global variables that are not in `spar::Input()`, it was not clear when reading the manual." [Participant 5]

[5] ATLAS.ti is a software used for qualitative data analysis and mixed methods research in academic, market, and user experience research. Available at: https://atlasti.com/.

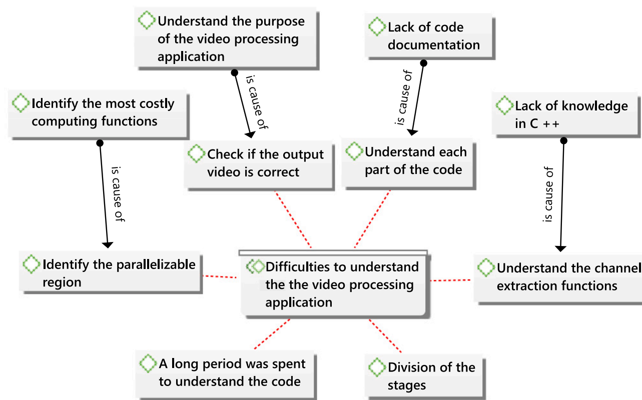


Fig. 15. Difficulties to understand the video application.

In the activity using TBB, participants also report specific difficulties with C++, as in the activity with FastFlow. Fig. 14(c) shows that the participants reported difficulties using specific commands (`new Mat(res)` and `static_cast`), **data structures**, **classes**, **object-oriented programming**, and pointers. The main difficulties reported by the participants in relation to C++ were the use of **pointers** (8 quotes) and **`static_cast` command** (5 quotes), which are caused by the lack of practice with the C++ programming language.

Regarding TBB, participants reported difficulties in **understanding the logic of the TBB**, **understanding how filters work**, **passing parameters in the constructor method** of a stage, **identifying the parallel region of the program**, and defining the **scope of the variables**. Due to the **lack of illustrations and more detailed explanations in the manual**, the participants have difficulty **understanding the meaning of tokens** and the **difference between the number of tokens and the number of threads**. The main difficulty reported by the participants was **understanding how the communication among the stages worked**. The quote below indicates that this difficulty is related to the difficulty to **find the programming errors**:

> "My biggest problem with TBB was not noticing that it was necessary to create a new copy of the data before they could be sent to the next stage. Therefore, as can be seen in the video, it took me 10 to 20 min to try to find the error. As soon as I discovered that it was necessary to use the `new Mat(res)` to send the data to another stage, the program was almost complete." [Participant 6]

In Q3, the participants should report their main difficulties in understanding the video application. Fig. 15 shows that the participants reported difficulties in **identifying the most costly computational functions**; in **identifying the parallelizable region of the program**; and in performing the **division of the stages**, determining which stages would be the first, the second and the third ones. Due to the difficulty in **understanding the purpose of the video application**, there were difficulties in **verifying if the video output was correct**. Due to the **lack of code documentation**, there were difficulties in **understanding each part of the code**. In addition, the main difficulty reported by the participants was **understanding the functions for extracting the green channel** from the video. The quote below indicates that this difficulty may be related to a **lack of knowledge of the C++ programming language**:

> "Although I do not know much about C++, consequently I do not know what color channel extraction function in the video is doing. It was easy to understand the context and what each part of the algorithm was doing." [Participant 10]

In Q10, the participants must inform which parallel API is the most difficult to perform the activity. Participants found it more difficult to program using TBB (10 participants) and FastFlow (5 participants) (see Fig. 13(a)). In turn, in Q11, participants must justify their choice. Fig. 16(a) shows a graphical representation of the reasons presented by the participants. Participants reported some difficulties with TBB, such as **difficulties in using the C++ `static_cast` command**, **difficulties in defining the number of tokens and threads**, and **difficulties with C++ object-oriented programming**. Due to the difficulties in **identifying the program's parallelizable region** and **difficulties in understanding the logic of TBB**, **a long period was spent to complete the activity** with TBB. In addition, since in TBB was **the first parallel API used by some of the participants**, **a long period was spent to complete the activity**. This indicates that starting with TBB, is much more complicated.

The main difficulty reported by the participants was to understand the **communication among the stages**. The quotes below highlight that it may be related to the difficulty in **using C++ pointers** and **using the new command**:

> "I took time to understand how to correctly pass the values through pointers among the TBB stages." [Participant 1]

> "My biggest difficulty was to notice the details in the definition of the variables shared among the stages. For example, it was necessary to instantiate the shared variable with `new` in the first stage. In the following stages, the `static_cast` was used. I was having problems because I was not doing `Mat res = new Mat` in the first stage." [Participant 2]

Fig. 16(b) shows the reasons for FastFlow being the most difficult parallel API. Participants reported difficulties in **implementing the Farm pattern**; **declaring global variables correctly**; **understanding communication among the stages**; and **dividing operations among the stages**. Due to the lack of practice with C++ programming, there were difficulties in using C++ pointers. In addition, due to the difficulty in using C++ pointers and the lack of more code examples, FastFlow was more difficult than TBB.

Regarding the Q12, participants informed which parallel API is the easiest to perform the activity. Most participants pointed out SPar (see Fig. 13(b)). In turn, in Q13, participants must justify their choice and Fig. 17(a) shows the reasons presented by participants. In SPar, **there is no need for major code changes**, which is one of the main reasons reported by participants. The quote below indicates that **there is no need for major code changes** due to **SPar annotations**:

> "With SPar, there were no major changes in the code, only new annotations for the parts that can be parallelized." [Participant 6]

Moreover, in SPar, **there is no need to create a class or structure for each stage**, due to its **simple syntax**. The quote below reinforces that **SPar code is easier to understand and program**:

> "Its syntax is extremely simple, it does not require the use of structure or classes as in FastFlow and TBB, making the code easier to understand and apply parallelism." [Participant 4]

Only 2 participants think it is easiest to program with TBB. Fig. 17(b) shows that TBB **enabled faster implementation** for the participants, **the manual is enough to understand how TBB works**, and **it was simpler to use**. In addition, **participants had no difficulty in using it** because of its **similarity to FastFlow**.

Only 2 participants think it is easier to program with FastFlow. Fig. 17(c) shows that the activity using FastFlow was the easiest, and **there were few difficulties** due to the **similarity of the video processing application to the code example**. **One participant had already presented work in class** about FastFlow, so this participant

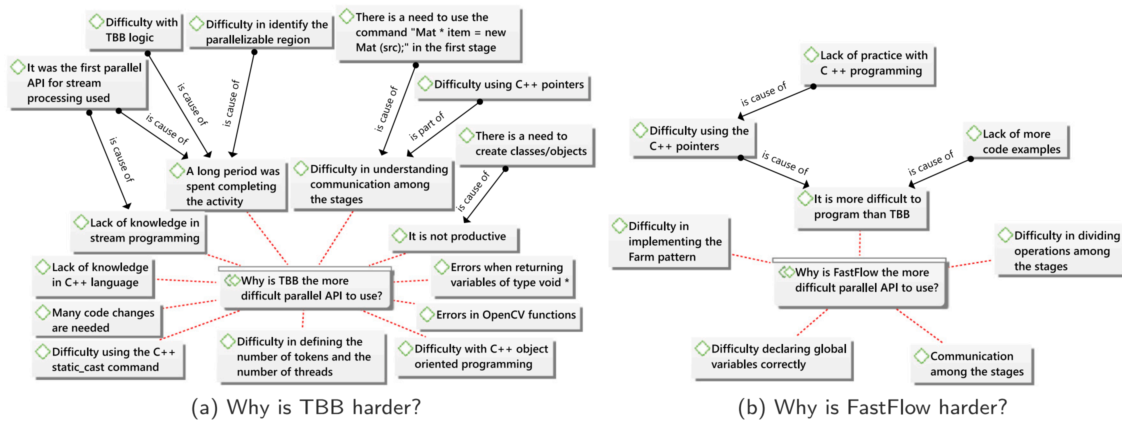


Fig. 16. More difficult parallel API to use.

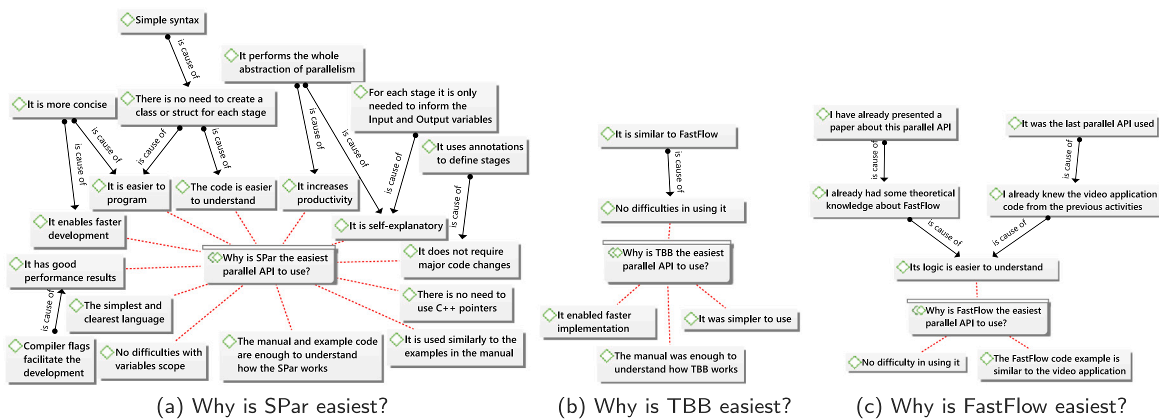


Fig. 17. Easiest parallel API to develop stream processing applications.

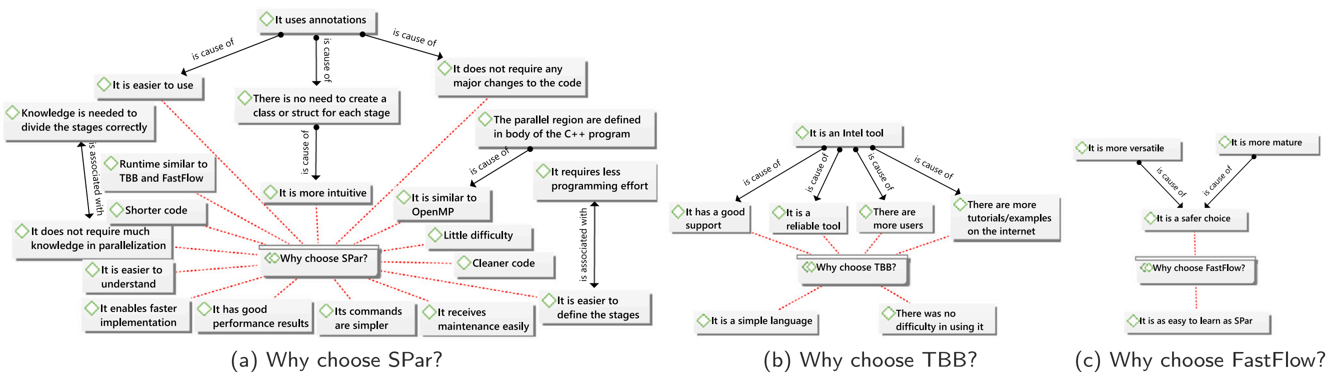


Fig. 18. Parallel API chosen by the participants.

already had **some theoretical knowledge about it**. In addition, as **FastFlow was the last parallel API used** by the participant, they already **knew the video application**.

In Q14, participants should inform which parallel API they would choose to parallelize a stream processing application in the future. Most participants chose SPar (see Fig. 13(c)). In Q15, participants must justify their choice. Fig. 18(a) shows the reasons presented by the participants. The participants would choose SPar because it allows them to **implement parallelism faster**. In SPar, the programmer only inserts **annotations** in the code to enable parallelism. Due to its **annotation-based model**, **SPar is easier to use** than FastFlow and TBB, **no need to create a class or structure for each stage**, and **no need for major code changes**. In addition, since **there is no need to create a class/structure**, the language **is more intuitive**. The following quotes indicate this:

> "Undoubtedly, the ease and agility that SPar provided at the time of coding and parallelization of the application would make it the one chosen to parallel a world-real application." [Participant 10]

> "I find the SPar interface more intuitive, without the need to previously define the parallel stages as classes or structures. With SPar, it is possible to define the region and the parallel stages in the program body (similar to OpenMP), which is very good." [Participant 14]

Only 1 participant chose TBB to parallelize stream processing applications. Fig. 18(b) shows that this participant pointed out that **TBB is a simple language** and **there was no difficulty in using it**. According to the participant, as **TBB is a tool from Intel**, it offers some advantages. The quote below indicates the advantages reported by the participant:

> "Being from Intel, it is reliable and has good support. An important factor is that there should be more users, tutorials, and examples on the Internet." [Participant 7]

Only 1 participant chose FastFlow to parallelize stream processing applications (Fig. 18(c)). This participant reported that **FastFlow is as easy to learn as SPar**. However, due to its **versatility**, and **maturity**, he thinks that **it is a safer choice**.

8. Discussion and threats to validity

The results show that SPar, FastFlow, and TBB offer effectiveness in solving the activity given to the participants because all applications produced the output video correctly with a minimum speedup of 3. However, SPar is the easiest language to use for parallel stream processing because it presented a lower learning curve in relation to the other parallel APIs, and provided more productivity to the developers. With a short time of access to the material provided, the participants were already able to parallelize the application using this language. In addition, the learning curve for SPar is shorter, since the simplicity of this parallel API results in less time accessing the material provided.

When using SPar, the participants did not have to make major modifications to the code in order to provide parallelism. With FastFlow and TBB, it was necessary to create structures (`class` and `struct`) for each stage, resulting in errors related to the C++ programming language. In addition, the need for the `return` command on a stage implemented through C++ structures is related to one of the main programming logic errors made by the participants using FF and TBB. From the satisfaction analysis, we observed that the use of C++ structures and pointers is one of the main difficulties reported by the participants when performing the activity using FastFlow and TBB. In addition, participants found it easier to use SPar (11 participants) and would choose it if they needed to solve a real problem in future activities (13 participants) since it is not necessary to use pointers and create structures to implement parallelism with this parallel API.

From the participants' point of view, SPar is easier to use parallel API, because of its annotation-based programming model. This feature allowed the participants to be able to develop parallel stream processing applications with less effort. Therefore, the evaluation of the participants' satisfaction confirmed the results concerning the development effort and productivity. The participant's opinion is still little considered by the literature since few studies have evaluated it [51–53]. However, this analysis is essential since it can help to complement the evaluation of the results.

From the results, we conclude that SPar offers the best usability in developing stream processing applications for multi-core systems. It was shown that SPar provides the best indicators of productivity and users' satisfaction and consequently better usability. This is the first study aiming to evaluate the usability of structured programming-based parallel APIs in the stream processing domain. In addition, through this study, it was possible to test and validate the proposed methodology in order to facilitate the evaluation of the usability of parallel APIs.

This study showed promising results. However, some threats to validity remain. In this experiment, the sequence of the parallel APIs may impact the results, since no group started the activity with FastFlow, and it was the last parallel API used by two groups. Therefore, participants may have learned faster how to solve the problem with FastFlow, which is a threat to internal validity. To reduce this risk, two groups could have started their activities with, in principle, the simpler parallel API (SPar). Thus, the participant has the chance to understand the problem by performing a simpler activity first [39].

The way in which the study is conducted is another threat to internal validity, which is directly influenced by how the study instruments have been formulated. For example, if there is a poorly formulated question on a questionnaire, the study could be adversely affected. To face this threat, a pilot study was previously conducted to test the instruments used in this study and capture any confusing/error factors. Another threat to internal validity is related to instrumentation [81]. Data are collected using self-assessment questionnaires completed by the participants. Then a participant may enter incorrect data into the questionnaire, such as the activity start and end time. To face this threat, in addition to the questionnaires, the machine screens used by the participants during the study were automatically recorded using a monitoring script and later checked.

As a threat to external validity, it is not possible to represent all the situations of stream processing in multi-core environments. As such, other experiments on different tasks should be performed towards the generalization. However, a strength of our experiment is that we explored beginner programmers and applied quantitative and qualitative analysis to understand the results as precisely as possible in a non-explored scenario, to the best of our knowledge.

The design of the experiment may be considered a threat to construct validity. SPar programming model is different from the TBB and FastFlow programming models. It is difficult to justify that this difference does not favor one parallel API over the others. Although TBB and FastFlow have similar programming models, each one presents its own characteristics. However, all of them are based on structured parallel programming and have support for stream parallelism. Recurrent parallelism patterns are easily identified/captured by developers, such as Farm, and Pipeline patterns.

Finally, a threat to conclusion validity is the sample size that may not be so representative from the statistical perspective. To reduce this threat, our analysis included all data collected from the participants. Unfortunately, this is a recurrent difficulty for empirical studies with developers, especially for approaches that require specific profiles, as in our case. Thus, our study presents a limitation on the size of the samples, which are considered indications (and not evidence). This does not discredit our research and the conclusions, where several lessons-learn were taken from qualitative and quantitative results.

9. Conclusion

In this article, we conducted a literature review for assessing parallel APIs and a usability evaluation of three parallel APIs (FastFlow, SPar, and TBB) for expressing parallelism in stream processing applications that execute on multi-core environments. The study was conducted with beginners in parallel programming to understand their main challenges when developing parallel applications for multi-core machines. We assigned the activity of parallelizing an OpenCV video application. Moreover, based on the literature review, we presented a methodology to serve as a guide for researchers to assess parallel programming and follow it in other application domains and parallel APIs.

Our results may also help in teaching parallel programming because the participants were beginners in this domain, and we identified the main challenges they faced in the study. We also presented the specific difficulties with each parallel API. The main difficulties reported by the participants referred to communication among the stages and the challenges of the correct division of tasks in order to achieve performance. We also observed difficulties with the C++ language, especially with pointers and object-oriented programming.

Concerning the data extraction, recording the screens of the machines used by the participants during the study allowed us to observe

in detail their behavior while performing the proposed activities. However, using a human observer has some challenges because it is time-consuming and requires double-checking. One approach that could reduce this effort is automatically generating logs from the experiment execution. However, none of the tools available met our requirements.

Evaluating the parallel programming as we performed in this study was a time-consuming task due to the planning and execution of the experiment. Moreover, the detailed analysis of the data provided in this study took several months. Some studies in the parallel programming domain use classical coding metrics to evaluate programming usability and productivity [16–18,33,82–88], such as LOC, McCabe's CCN [89], Halstead's measures [90], and COCOMO II [91]. These classical metrics can rapidly provide some programming indicators based on code size, complexity assessment, and development effort estimation. However, they are designed to evaluate general-purpose software and do not consider factors that impact the parallel programming effort, such as the programming model (structured and non-structured), recurring programming errors, user satisfaction, and human difficulties [92]. In future work, we intend to conduct further investigation of such metrics applied to the parallel programming domain.

Parallel programming assessment presents several challenges to be addressed in the future. We observe a gap in the usability analysis of parallel APIs targeting GPUs such as CUDA, OpenACC, and others. Moreover, there is a gap in evaluating emerging parallel APIs for HPC clusters, such as HPX and Apache APIs for big data processing (e.g., Spark and Flink). Since few studies explore such architectures [59–63], there is also room for creating methodologies to assist parallel programming usability experimentation, such as the one proposed in this study.

CRediT authorship contribution statement

Gabriella Andrade: Conceptualization, Methodology, Formal analysis, Writing – original draft, Visualization, Data curation, Software. **Dalvan Griebler:** Conceptualization, Methodology, Investigation, Writing – review & editing, Data curation, Software, Funding acquisition, Resources. **Rodrigo Santos:** Methodology, Writing – review & editing. **Luiz Gustavo Fernandes:** Supervision, Funding acquisition.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.csi.2022.103691.

Acknowledgments

This research is partially funded by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, FAPERGS 05/2019-PQG project ParAS (Nº 19/2551-0001895-9), FAPERGS, Brazil 10/2020-ARD project SPar4.0 (Nº 21/2551-0000725-7), Universal MCTIC/CNPq, Brazil Nº 28/2018 project SParCloud (Nº 437693/2018-0), and UNIRIO, Brazil and FAPERJ, Brazil (Grant: 211.583/2019).

References

[1] G.E. Moore, Cramming more components onto integrated circuits, Proc. IEEE 38 (8) (1965).
[2] P.S. Pacheco, Introduction to Parallel Programming, Morgan Kaufmann, Burlington, MA, USA, 2011.
[3] K. Gregory, A. Miller, C++ AMP, O'Reilly Media, Sebastopol, CA, USA, 2012.
[4] D.B. Kirk, W. mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann, Cambridge, MA, USA, 2016.
[5] T.G. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, in: Software Patterns Series, Pearson Education, Boston, USA, 2004.
[6] M. McCool, J. Reinders, A. Robison, Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann Publishers, Waltham, USA, 2012.
[7] H.C. Andrade, B. Gedik, D.S. Turaga, Fundamentals of Stream Processing: Application Design, Systems, and Analytics, Cambridge University Press, Cambridge, United Kingdom, 2014.
[8] D. Griebler, Domain-Specific Language & Support Tool for High-Level Stream Parallelism (Ph.D. thesis), Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, 2016.
[9] D. Corral-Plaza, I. Medina-Bulo, G. Ortiz, J. Boubeta-Puig, A stream processing architecture for heterogeneous data sources in the internet of things, Comput. Stand. Interfaces 70 (2020) 103426.
[10] A. Atashpendar, B. Dorronsoro, G. Danoy, P. Bouvry, A scalable parallel cooperative coevolutionary PSO algorithm for multi-objective optimization, J. Parallel Distrib. Comput. 112 (2018) 111–125.
[11] J. Lòff, D. Griebler, G. Mencagli, G. Araujo, M. Torquati, M. Danelutto, L.G. Fernandes, The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures, Future Gener. Comput. Syst. 125 (2021) 743–757.
[12] L. Riha, R. Smid, Acceleration of acoustic emission signal processing algorithms using CUDA standard, Comput. Stand. Interfaces 33 (4) (2011) 389–400.
[13] V. Soni, A. Hadjadj, O. Roussel, G. Moebs, Parallel multi-core and multi-processor methods on point-value multiresolution algorithms for hyperbolic conservation laws, J. Parallel Distrib. Comput. 123 (2019) 192–203.
[14] I.M. Spiliotis, M.P. Bekakos, Y.S. Boutalis, Parallel implementation of the image block representation using OpenMP, J. Parallel Distrib. Comput. 137 (2020) 134–147.
[15] F. Cantonnet, Y. Yao, M. Zahran, T. El-Ghazawi, Productivity analysis of the UPC language, in: 18th International Parallel and Distributed Processing Symposium, IPDPS'04, IEEE, Santa Fe, New Mexico, USA, 2004, pp. 254–260.
[16] S. Nanz, C.A. Furia, A comparative study of programming languages in rosetta code, in: 37th IEEE International Conference on Software Engineering, Vol. 1, ICSE, IEEE, Florence, Italy, 2015, pp. 778–788.
[17] V. Narayanan, R. Kavitha, R. Srikanth, Performance evaluation of Brahmagupta-Bhaskara equation based algorithm using OpenMP, in: Proceedings of Data Analytics and Management, Springer, 2022, pp. 21–28.
[18] B. Peccerillo, S. Bartolini, Flexible task-DAG management in PHAST library: Data-parallel tasks and orchestration support for heterogeneous systems, Concurr. Comput.: Pract. Exper. 34 (2) (2022) e5842.
[19] S. Wienke, J. Miller, M. Schulz, M.S. Müller, Development effort estimation in HPC, in: SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2016, pp. 107–118.
[20] S. Nanz, S. West, K.S. Da Silveira, B. Meyer, Benchmarking usability and performance of multicore languages, in: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, IEEE, 2013, pp. 183–192.
[21] D. Szafron, J. Schaeffer, An experiment to measure the usability of parallel programming systems, Concurrency, Pract. Exp. 8 (2) (1996) 147–166.
[22] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science & Business Media, Heindelberg, Germany, 2012.
[23] S. Nanz, S. West, K.S. Da Silveira, Examining the expert gap in parallel programming, in: European Conference on Parallel Processing, Springer, Aache, Germany, 2013, pp. 434–445.
[24] W. Thies, S. Amarasinghe, An empirical characterization of stream programs and its implications for language and compiler design, in: 2010 19th International Conference on Parallel Architectures and Compilation Techniques, PACT, IEEE, 2010, pp. 365–376.
[25] D. Griebler, A. Vogel, D.D. Sensi, M. Danelutto, L.G. Fernandes, Simplifying and implementing service level objectives for stream parallelism, J. Supercomput. 76 (2019) 4603–4628.
[26] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, Fastflow: high-level and efficient streaming on multi-core, in: Programming Multi-Core and Many-Core Computing Systems, Parallel and Distributed Computing, Wiley-Blackwell, 2017, pp. 261–280.
[27] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, R. Grimm, A catalog of stream processing optimizations, ACM Comput. Surv. 46 (4) (2014) 1–34.
[28] S. Schneider, M. Hirzel, B. Gedik, K.-L. Wu, Safe data parallelism for general streaming, IEEE Trans. Comput. 64 (2) (2013) 504–517.
[29] J. Reinders, Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism, O'Reilly Media, Inc. Sebastopol, USA, 2007.
[30] T.G. Mattson, Y. He, A.E. Koniges, The OpenMP Common Core: Making OpenMP Simple Again, in: Scientific and Engineering Computation, MIT Press, Cambridge, MA, USA, 2019.
[31] M. Voss, R. Asenjo, J. Reinders, Pro TBB: C++ Parallel Programming with Threading Building Blocks, A Press, New york, USA, 2019.
[32] D. Griebler, M. Danelutto, M. Torquati, L.G. Fernandes, SPar: A DSL for high-level and productive stream parallelism, Parallel Process. Lett. 27 (01) (2017) 1740005.

[33] D. Griebler, R.B. Hoffmann, M. Danelutto, L.G. Fernandes, High-level and productive stream parallelism for dedup, ferret, and Bzip2, Int. J. Parallel Program. 47 (1) (2018) 253–271.

[34] Intel, Intel threading building blocks documentation, 2020, URL https://www.threadingbuildingblocks.org/docs/help/index.html.

[35] J.L. Barros-Justo, F.B. Benitti, S. Tiwari, The impact of use cases in real-world software development projects: A systematic mapping study, Comput. Stand. Interfaces 66 (2019) 103362.

[36] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, 2014, pp. 1–10.

[37] H. Park, P. Kim, H. Kim, K.-W. Park, Y. Lee, Efficient machine learning over encrypted data with non-interactive communication, Comput. Stand. Interfaces 58 (2018) 87–108.

[38] L. Hochstein, V.R. Basili, U. Vishkin, J. Gilbert, A pilot study to compare programming effort for two parallel programming models, J. Syst. Softw. 81 (11) (2008) 1920–1930.

[39] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J.K. Hollingsworth, M.V. Zelkowitz, Parallel programmer productivity: A case study of novice parallel programmers, in: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, IEEE, Seatle, USA, 2005, p. 35.

[40] M. Zelkowitz, V. Basili, S. Asgari, L. Hochstein, J. Hollingsworth, T. Nakamura, Measuring productivity on high performance computers, in: Software Metrics, IEEE International Symposium on, Citeseer, 2005, p. 6.

[41] V. Pankratius, A. Jannesari, W.F. Tichy, Parallelizing bzip2: A case study in multicore software engineering, IEEE Softw. 26 (6) (2009) 70–77.

[42] M. Coblenz, R. Seacord, B. Myers, J. Sunshine, J. Aldrich, A course-based usability analysis of cilk plus and OpenMP, in: 2015 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC, IEEE, 2015, pp. 245–249.

[43] J.B. Manzano, Y. Zhang, G.R. Gao, P3i: The delaware programmability, productivity and proficiency inquiry, in: Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications, 2005, pp. 32–36.

[44] R. Alameh, N. Zazworka, J.K. Hollingsworth, Performance measurement of novice HPC programmers code, in: Third International Workshop on Software Engineering for High Performance Computing Applications, SE-HPC'07, IEEE, 2007, p. 3.

[45] I. Patel, J.R. Gilbert, An empirical study of the performance and productivity of two parallel programming models, in: 2008 IEEE International Symposium on Parallel and Distributed Processing, IEEE, 2008, pp. 1–7.

[46] K. Ebcioğlu, V. Sarkar, T. El-Ghazawi, J. Urbanic, An experiment in measuring the productivity of three parallel programming languages, in: Proceedings of the Third Workshop on Productivity and Performance in High-End Computing, Austin, USA, 2006, pp. 30–36.

[47] C. Teijeiro, G.L. Taboada, J. Tourino, B.B. Fraguela, R. Doallo, D.A. Mallón, A. Gómez, J.C. Mourino, B. Wibecan, Evaluation of UPC programmability using classroom studies, in: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models, 2009, pp. 1–7.

[48] G. Speyer, N. Freed, R. Akis, D. Stanzione, E. Mack, Paradigms for parallel computation, in: 2008 DoD HPCMP Users Group Conference, IEEE, 2008, pp. 486–494.

[49] C. Danis, J. Thomas, J. Richards, J. Brezin, C. Swart, C. Halverson, R. Bellamy, P. Malkin, Towards applying complexity metrics to measure programmer productivity in high performance computing, in: Proceedings of the 2008 International Conference on Software Engineering, ICSE, in: First International Workshop on Software Engineering for Computational Science and Engineering, ACM, Gothenburg, Sweden, 2008, pp. 1–8.

[50] C. Sadowski, J. Yi, User evaluation of correctness conditions: A case study of cooperability, in: Evaluation and Usability of Programming Languages and Tools, PLATEAU '10, Association for Computing Machinery, New York, USA, 2010, pp. 1–6.

[51] S. Nanz, F. Torshizi, M. Pedroni, B. Meyer, Design of an empirical study for comparing the usability of concurrent programming languages, Inf. Softw. Technol. 55 (7) (2013) 1304–1315.

[52] V. Pankratius, F. Schmidt, G. Garreton, Combining functional and imperative programming for multicore software: An empirical study evaluating scala and java, in: 2012 34th International Conference on Software Engineering, ICSE, IEEE, Zurich, Switzerland, 2012, pp. 123–133.

[53] C.J. Rossbach, O.S. Hofmann, E. Witchel, Is transactional programming actually easier? ACM Sigplan Not. 45 (5) (2010) 47–56.

[54] V. Pankratius, A.-R. Adl-Tabatabai, Software engineering with transactional memory versus locks in practice, Theory Comput. Syst. 55 (3) (2014) 555–590.

[55] F. Castor, J.P. Oliveira, A.L. Santos, Software transactional memory vs. Locking in a functional language: A controlled experiment, in: Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11, in: SPLASH '11 Workshops, Association for Computing Machinery, New York, NY, USA, 2011, pp. 117–122.

[56] A. Nanthaamornphong, A pilot study: design patterns in parallel program development, in: Proceedings of the 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering, 2013, pp. 17–20.

[57] D. Griebler, D. Adornes, L.G. Fernandes, Performance and usability evaluation of a pattern-oriented parallel programming interface for multi-core architectures, in: The 26th International Conference on Software Engineering & Knowledge Engineering, Knowledge Systems Institute Graduate School, Vancouver, Canada, 2014, pp. 25–30.

[58] K. Molitorisz, T. Müller, W.F. Tichy, Patty: A pattern-based parallelization tool for the multicore age, in: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, 2015, pp. 153–163.

[59] X. Li, P.-C. Shih, J. Overbey, C. Seals, A. Lim, Comparing programmer productivity in OpenACC and CUDA: An empirical investigation, Int. J. Comput. Sci. Eng. Appl. (IJCSEA) 6 (5) (2016) 1–15.

[60] B. Akil, Y. Zhou, U. Röhm, On the usability of hadoop MapReduce, apache spark & apache flink for data science, in: 2017 IEEE International Conference on Big Data, Big Data, IEEE, 2017, pp. 303–310.

[61] X. Li, P.-C. Shih, X. Li, C. Seals, A case study of novice programmers on parallel programming models, J. Comput. 13 (5) (2018) 490–502.

[62] J. Miller, M. Arenaz, Measuring the impact of HPC training, in: 2019 IEEE/ACM Workshop on Education for High-Performance Computing, EduHPC, IEEE, 2019, pp. 58–67.

[63] P. Daleiden, A. Stefik, P.M. Uesbeck, GPU programming productivity in different abstraction paradigms: a randomized controlled trial comparing CUDA and thrust, ACM Trans. Comput. Educ. (TOCE) 20 (4) (2020) 1–27.

[64] F. Domínguez-Mayo, M. Escalona, M. Mejías, M. Ross, G. Staples, A quality management based on the quality model life cycle, Comput. Stand. Interfaces 34 (4) (2012) 396–412.

[65] ISO 9241-11:2018, Ergonomics of Human-System Interaction – Part 11: Usability: Definitions and Concepts, International Organization for Standardization, Geneva, Switzerland, 2018, URL https://www.iso.org/standard/63500.html.

[66] ISO/IEC TR 9126-4:2004, Software Engineering – Product Quality – Part 4: Quality in Use Metrics, International Organization for Standardization, Geneva, Switzerland, 2004, URL https://www.iso.org/standard/39752.html.

[67] J. Miller, S. Wienke, M. Schlottke-Lakemper, M. Meinke, M.S. Müller, Applicability of the software cost model COCOMO II to HPC projects, Int. J. Comput. Sci. Eng. 17 (3) (2018) 283–296.

[68] Z.C. Holcomb, Fundamentals of Descriptive Statistics, Routledge, New York, USA, 2016.

[69] Y. Chan, Biostatistics 102: quantitative data–parametric & non-parametric tests, Singapore Med. J. 44 (8) (2003) 391–396.

[70] D.J. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures, third ed., Chapman and Hall/CRC, New York, USA, 2004.

[71] C.M. Barnum, Usability Testing Essentials: Ready, Set... Test!, Morgan Kaufmann, Burlington, USA, 2010, p. 408.

[72] J.M. Corbin, A. Strauss, Grounded theory research: Procedures, canons, and evaluative criteria, Qual. Sociol. 13 (1) (1990) 3–21.

[73] B.B.N. de França, H. Jeronimo, G.H. Travassos, Characterizing DevOps by hearing multiple voices, in: Proceedings of the 30th Brazilian Symposium on Software Engineering, SBES '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 53–62.

[74] P. Sharma, A.L. Sangal, Building a hierarchical structure model of enablers that affect the software process improvement in software SMEs—A mixed method approach, Comput. Stand. Interfaces 66 (2019) 103350.

[75] OpenCV, Creating a video with openCV, 2019, URL https://docs.opencv.org/2.4/doc/tutorials/highgui/video-write/video-write.html.

[76] L.M. Connelly, Pilot studies, Medsurg Nurs. 17 (6) (2008) 411–412.

[77] A. Rutherford, ANOVA and ANCOVA: A GLM Approach, second ed., John Wiley & Sons, Hoboken, USA, 2011.

[78] N.M. Razali, Y.B. Wah, et al., Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests, J. Stat. Model. Anal. 2 (1) (2011) 21–33.

[79] M. Rubert, K. Farias, On the effects of continuous delivery on code quality: A case study in industry, Comput. Stand. Interfaces 81 (2022) 103588.

[80] G.J. Bronson, A First Book of C++, fourth ed., Cengage Learning, Boston, USA, 2011.

[81] E.-M. Ihantola, L.-A. Kihn, Threats to validity and reliability in mixed methods accounting research, Qual. Res. Account. Manage. 8 (1) (2011) 39–58.

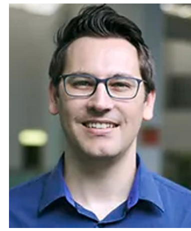
[82] D. Adornes, D. Griebler, C. Ledur, L.G. Fernandes, A unified MapReduce domain-specific language for distributed and shared memory architectures, in: The 27th International Conference on Software Engineering & Knowledge Engineering, Knowledge Systems Institute Graduate School, Pittsburgh, USA, 2015, p. 6.

[83] D. Adornes, D. Griebler, C. Ledur, L.G. Fernandes, Coding productivity in MapReduce applications for distributed and shared memory architectures, Int. J. Softw. Eng. Knowl. Eng. 25 (10) (2015) 1739–1741.

[84] J.Á. Cid-Fuentes, P. Alvarez, R. Amela, K. Ishii, R.K. Morizawa, R.M. Badia, Efficient development of high performance data analytics in python, Future Gener. Comput. Syst. 111 (2020) 570–581.

[85] J. Fernàndez-Fabeiro, A. Gonzalez-Escribano, D.R. Llanos, Simplifying the multi-GPU programming of a hyperspectral image registration algorithm, in: 2019 International Conference on High Performance Computing & Simulation, IEEE, Dublin, Ireland, 2019, pp. 11–18.
[86] M.A. Martínez, B.B. Fraguela, J.C. Cabaleiro, A highly optimized skeleton for unbalanced and deep divide-and-conquer algorithms on multi-core clusters, J. Supercomput. (2022) 1–21.
[87] S. Okur, D. Dig, How do developers use parallel libraries? in: 20th International Symposium on the Foundations of Software Engineering, SIGSOFT, ACM, New York, USA, 2012, p. 54.
[88] G. Rodriguez-Canal, Y. Torres, F.J. Andújar, A. Gonzalez-Escribano, Efficient heterogeneous programming with FPGAs using the controller model, J. Supercomput. 77 (12) (2021) 13995–14010.
[89] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 (4) (1976) 308–320.
[90] M.H. Halstead, Elements of Software Science, Vol. 7, North-Holland, New York, NY, USA, 1977.
[91] B.W. Boehm, C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D.J. Reifer, B. Steece, Software Cost Estimation with COCOMO II, Prentice Hall PTR, 2000.
[92] G. Andrade, D. Griebler, R. Santos, M. Danelutto, L.G. Fernandes, Assessing coding metrics for parallel programming of stream processing programs on multi-cores, in: 47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA, IEEE, 2021, pp. 291–295.

Gabriella Andrade is a Ph.D. student of the graduate program in computer science (PPGCC) at the Pontifical Catholic University of Rio Grande do Sul (PUCRS), where she is a member of the Parallel Applications Modeling Group (GMAP). She received her master's degree in Electrical Engineering from the Federal University of Pampa (UNIPAMPA), where she also received her bachelor's degree in Computer Science. Her main research interests include parallel computing, methodologies, and metrics for evaluating parallel application development.



Dalvan Griebler is an associate professor of the graduate program in computer science (PPGCC) at the Pontifical Catholic University of Rio Grande do Sul (PUCRS), where he is research coordinator at the Parallel Applications Modeling Group (GMAP). Also, he is an associate professor at Sociedade Educacional Três de Maio (Setrem) where he leads the Laboratory of Advanced Research on Cloud Computing (LARCC). He received the Ph.D. in computer science from both PUCRS and University of Pisa in 2016. His main research interests are: parallel and distributed computing, methodologies, languages and libraries for high-level parallel programming; benchmarks; big data; and cloud computing.

Rodrigo Santos received the Ph.D. degree in computer science from the Alberto Luiz Coimbra Institute for Graduate Studies and Research in Engineering, Federal University of Rio de Janeiro (UFRJ). He is currently an Associate Professor with the Department of Applied Informatics, Federal University of the State of Rio de Janeiro (UNIRIO), where he leads the Complex Systems Engineering Lab (LabESC). His research interests include complex systems engineering (specially software ecosystems and systems-of-systems) and computer science education and training.

Gustavo Fernandes is an associate professor of the graduate program in computer science (PPGCC) at the Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre (Brazil). His primary research interests are Parallel and Distributed Computing, High Performance Applications Modeling, Green Computing and Parallel Programming Interfaces. Dr. Fernandes received his Ph.D. in Computer Science from the Institut National Polytechnique de Grenoble (France) in 2002. He currently leads the Parallel Applications Modeling Group (GMAP) at PUCRS.