Journal of Computer Languages 65 (2021) 101054

Contents lists available at ScienceDirect

Journal of Computer Languages

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

High-level and efficient structured stream parallelism for rust on multi-cores

Ricardo Pieper [a,1], Júnior Löff [a,1], Renato B. Hoffmann [a,1], Dalvan Griebler [a,b,*], Luiz G. Fernandes [a]

[a] School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, 90619-900, Brazil
[b] Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio, 98910-000, Brazil

ARTICLE INFO

Keywords:
Programming language
Parallel programming
Parallelism abstractions
Stream processing

ABSTRACT

This work aims at contributing with a structured parallel programming abstraction for Rust in order to provide ready-to-use parallel patterns that abstract low-level and architecture-dependent details from application programmers. We focus on stream processing applications running on shared-memory multi-core architectures (i.e, video processing, compression, and others). Therefore, we provide a new high-level and efficient parallel programming abstraction for expressing stream parallelism, named Rust-SSP. We also created a new stream benchmark suite for Rust that represents real-world scenarios and has different application characteristics and workloads. Our benchmark suite is an initiative to assess existing parallelism abstraction for this domain, as parallel implementations using these abstractions were provided. The results revealed that Rust-SSP achieved up to 41.1% better performance than other solutions. In terms of programmability, the results revealed that Rust-SSP requires the smallest number of extra lines of code to enable stream parallelism.

1. Introduction

Rust is an open-source system programming language sponsored by Mozilla. It provides memory safety, performance, reliable concurrency, and high-level features for productivity. It was built using some of the best features found in other well-established programming languages. Rust also introduces unique concepts such as ownership, borrowing, and lifetimes, that allow Rust to make memory safety guarantees without needing the extra overhead of a garbage collector. The language has been popularized in recent years and used in production code for fast, low-resource, and cross-platform system applications as well as web development [1]. Regarding parallelism, Rust has an active community and even originally written in C++ solutions completely migrated to Rust language such as Rayon [2]. However, since Rust is a relatively new language, many research opportunities remain open in the field of efficient and abstract parallelism.

Advances in technology gave place to applications whose increased computational complexity may demand parallel programming in order to achieve feasible time performance. However, parallelism is not a trivial endeavor. Programmers must deal with complex error-prone concepts such as thread creation and management, synchronization, communication mechanisms, load balancing, data dependency, critical or mutual exclusive data access, etc. This is more relevant when considering the main programming paradigm remains the sequential one [3–5]. Therefore, parallelism abstractions are crucial to increase code productivity and enable high performance. One such way to provide parallel abstractions is employing structured programming solutions [5,6]. They provide ready to use patterns or structures that represent recurrent situations commonly found in a specific study domain. Often, these structures are provided by experts in the field. This way, the general purpose developer can leverage the expertise behind the structured parallel programming provider simply by instantiating the correct API (Application Programming Interface). In fact, this approach is common in C++ language. Some examples are Intel TBB [7], Microsoft PPL [8], Parallel STL [9], among others.

Stream processing applications can be found in several fields of study. Some common examples are signal processing, audio, image, and video filtering but it can also represent applications for cryptography, data compression, etc. They are characterized by a continuous flow of data through a sequence of independent processing stages or filters. In fact, the stream processing paradigm has been studied in C++ for several years [10–12]. Some researches were conducted to create a language specific to this domain [11,13]. In Rust, parallel stream processing can be represented in the standard library using a stream of futures. Unlike the structured approach, in Rust, the programmer must compose this stream specifying communication directives, execution flow, and thread creation. To effectively execute this stream, a stream of futures execution engine such as Tokio [14] must be used.

Aiming to leverage the advantages of the structured parallel programming, we proposed Rust-SSP (Rust Structured Stream Parallelism)

* Corresponding author at: School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, 90619-900, Brazil.
E-mail addresses: ricardo.pieper@edu.pucrs.br (R. Pieper), junior.loff@edu.pucrs.br (J. Löff), renato.hoffmann@edu.pucrs.br (R.B. Hoffmann), dalvan.griebler@pucrs.br (D. Griebler), luiz.fernandes@pucrs.br (L.G. Fernandes).

[1] Authors have contributed equally to this work.

https://doi.org/10.1016/j.cola.2021.101054
Received 12 April 2021; Received in revised form 18 June 2021; Accepted 12 July 2021
Available online 31 July 2021

in [15]. It provides a pipeline skeleton that abstracts details of parallel stream processing implementation. This work extends this previous work by deeply discussing and evaluating Rust-SSP compared to other available options. Due to the absence of a benchmark suite to conduct the experiments, our goal is to provide a new benchmark suite that represent different scenarios and workload to assess parallel programming abstractions. This initiative will help other researchers to improve and evolve their works, as benchmarks become a baseline to assess overheads and optimizations. Our contributions are summarized in the following:

- A high-level, productive, and efficient parallel programming abstraction for expressing multi-core parallelism on Rust language.
- A new benchmark suite with different stream processing application workloads and parallel programming abstractions.
- A set of experiments evaluating the feasibility of Rust-SSP, Rayon, Tokio, Pipeliner, and standard Threads concerning performance and programming aspects.

The rest of this paper is organized as follows. Section 2 discusses the necessary background on structured parallel programming and previous related works. Then, Section 3 explains details of our Rust-SSP solution for structured stream parallelism. The stream processing applications benchmark suite and workloads were presented in Section 4. Subsequently, Section 5 presents a performance evaluation of the parallelized applications as well as discusses programmability aspects of each parallel programming interface. Finally, Section 6 concludes the work and discusses possible future works.

2. Background

In this section we briefly explain the structured parallel programming paradigm and its distinguished importance to the future of parallel programming. Then, we discuss the Rust programming language advances and address recent research works and community driven tools for parallel programming.

2.1. Structured parallel programming

Parallel programming is often a difficult task [16,17]. Programmers must deal with error-prone complex concepts such as task distribution, communication mechanisms, synchronization, thread management, and others. Incorrect implementations of parallelism may introduce deadlocks and data races. Also, inefficient implementations may yield limited scalability. Moreover, the non-deterministic behavior of parallel programs makes them harder to reason about when compared to sequential programs. Operations often do not execute in the same order and similar tasks may take different amounts of time to execute. Such programs may become complex and hard to debug or maintain. This problem is even more relevant considering application developers often do not specialize in parallel programming techniques. They are usually concerned with algorithmic solutions of their own specialization field rather than parallelism implementation details. On the other hand, application developers still should be able to easily achieve good performance.

Sequential programs are simpler to reason about since they often run deterministically. It always performs the same operations in the same order. Authors of [16,17] proposed pattern-based methods to bring some characteristics of serial programming into parallel programming. This way, they can help application developers understand their parallel code better since the main programming paradigm remains the sequential one [3]. McCool et al. [16] also argue that such techniques must enable programs to scale in performance as the number of cores grows. Thus, such techniques would make parallel code benefit from future hardware improvements.

Adopting a structured approach to parallel programming is a common solution to help abstract parallel programming complexity [16]. In fact, several C++ parallel programming interfaces do exactly that. Some examples are FastFlow [3], Intel TBB [7], Microsoft PPL [8], Parallel STL [9], and others. They employ parallel patterns as a way to codify commonly recurring strategies for dealing with specific parallel programming problems. The idea is for the developer to easily reuse and compose these patterns which are typically developed by experts in the field. The responsibility for the lower-level parallel details falls upon the pattern provider. For the high-level programmer, all he needs to understand is the pattern implementation semantics and its behavior and implications. There is no need to learn about internal communication protocols, synchronizations or load balancing schedulers.

Parallel patterns have two parts, high-level semantics and lower-level implementation. Semantics define how and when a parallel pattern can be used for a specific algorithm as well as data dependencies and possible limitations. For example, common parallel patterns used for data parallelism are *Map* and *Reduce*. *Map* is a parallel pattern in which a function is applied over a list of independent items, possibly in parallel. *Reduce* combines all items from a list into a single output using an associative binary function. With this semantic information, a high-level programmer is able introduce parallelism without getting involved with low-level parallel details. Besides, the responsibility for achieving efficient parallelism performance is passed to the parallel pattern and its provider. Many solutions, such as high-level parallel libraries and domain-specific languages, can provide its own parallel pattern version, which may implement different strategies and optimizations. Indeed, parallel patterns may vary drastically between different providers since they may choose different algorithms, lock mechanisms, load balancing, memory optimizations, and others.

In the field of stream processing, the most important parallel patterns are *Farm* and *Pipeline* [6]. *Farm* has a scheduler called emitter, worker replicas, and a collector that gathers results from the worker replicas. A *Pipeline* is built with a sequence of stages that perform as an assembly line, where each stage processes a different task/item in parallel. A *Pipeline* is defined as $pipeline(F_0, F_1, \ldots)$ and is composed by a set of independent sequential processing filters F_n that perform their computations on different stream data items. There is a direct producer–consumer dependency where F_n always takes input data from F_{n-1}. A pipeline can be linear or non-linear. A non-linear pipeline is a regular pipeline that has at least one of its stages replicated to increase the degree of parallelism. For the purpose of this work, the pipeline is employed as the parallel pattern to abstract stream processing applications.

2.2. Rust in prior works

This section of the paper has the goal to provide a quick overview over important advances towards Rust ecosystem. In Section 2.2.1 we present a summary of some works that aim at helping to consolidate the fundamentals for Rust programming language. In Section 2.2.2 we provide a discussion about parallelism in Rust, where we review the most used parallel programming abstractions.

2.2.1. Rust overall advances

According to the authors in [18], there is a longstanding barrier separating programming languages: safety vs. control. The former is based in rigorous type systems for statically avoiding many classes of bugs. The latter allows the access to the "bare metal" for exploiting the maximum performance of the underlying hardware. At one end of the spectrum, we have languages like Java, that enforce safety to the detriment of performance. At the other end, languages like C and C++ are standard system languages when it comes to performance, but offer little safety guarantees. Rust was proposed as an attempt to shorten this longstanding barrier. The main goal is to offer a strong type system and memory safety guarantees without giving away too much performance.

Over the past years, the open-source Rust community has conducted many researches for improving core programming language features. The RustBelt project [19] provides a formal machine-checked evaluation of Rust's safety claims. In other words, the correctness of Rust's type system, borrowing mechanisms, ownership features, lifetimes, and concurrency is being investigated to build a formal foundation for Rust. The authors from [20] presented a new technique for specifying and verifying functional correctness proprieties for real-world Rust code. They provide an implementation of their technique as a plugin for the Rust compiler. The authors in [21] used their experiences for providing a discussion about memory safety and possible improvements for the safety guarantees. Also, they propose a Rust extension for memory safety in event-driven platforms.

2.2.2. Parallelism in Rust

Parallelism in Rust still is an open research topic. Rust's standard library does not provide any parallel programming abstraction. It only provides basic low-level parallelism features, such as OS threads and blocking system calls. Besides that, Rust relies on the active community for introducing new efficient parallel implementations and abstractions. Current state-of-the-art parallel programming abstractions for Rust, like Rayon and Tokio, rely in an ad-hoc parallelism approach. To the best of our knowledge, Rust does not yet have many consolidated researches on structured parallel programming techniques. Therefore, we have increased the scope of our review to also include some tools that do not implement this programming model and do not support stream parallelism.

Sydow et al. [22] implemented a graphical programming model runtime for safe stream parallelism based on data-flow model. The tool generates parallel code based on a graphical application programming interface (API). However, the tool is coupled with the IntelliJ IDEA IDE, limiting usability. An extension of this work is presented in [23]. Although the previous version of their framework could generate automatic code, the authors claim it did not achieved efficient parallelization results. In [23], the authors introduce an algorithm that combines profiling with static analysis and a performance model that estimates parallelization costs for generating more efficient parallel code.

The research presented in [24] investigates the performance when converting two parallel patterns (based in C++ FastFlow) to Rust. The results revealed that the Rust version of parallel patterns achieved similar performance with respect to the C++ version. Similarly, our goal is to investigate the benefits of the structured parallel programming on the brand-new Rust programming language. However, we work with parallel programming abstractions fully implemented in Rust instead of using Rust's FFI (Foreign Function Interface) support.

The *de-facto* library for data parallelism in Rust language is Rayon [2]. Its parallelism relies on *parallel iterators*, which extend the default sequential iterators of the standard library. Rayon has many characteristics that make it very optimized for data parallelism, but not for stream parallelism. Nonetheless, we have included Rayon to our performance evaluation due to its ubiquity on the Rust ecosystem and there are few cases where stream processing applications can be modeled using data-parallel techniques. It is also important to show the limitations of Rayon for this application domain.

As the state-of-the-art in Rust for streaming applications, Tokio [14] is a library for asynchronous programming. Tokio is entirely based on *futures*, which represent abstract units of work that might complete in the future. Tokio is composed of many modules that provide essential features for implementing asynchronous applications. For example, it offers a high-level interface for expressing asynchronous I/O (sockets, file system) and asynchronous tasks (synchronization primitives, channels, timeouts).

Pipeliner [25] is a Rust community library that offers a high-level abstraction for instantiating pipelines. It aims at avoiding unsafe Rust code and operates over a set of Iterators. Pipelines [26] is another community Rust library for implementing pipeline parallelism. This project seems to be discontinued and we high performance overheads with respect to the other parallel abstractions, therefore, we excluded this library from our study.

Data-flow parallelism is an important branch of the stream processing domain, along with stream parallelism. Although it is not the focus of our work, we consider essential to cite some important works in this sub-domain. Timely [27] is a low-latency cyclic data-flow system. It was first published in the work [28], and later was made fully available in Rust. Materialize [29] is a streaming database targeting real-time applications. This framework implements the previous Timely Dataflow runtime. Materialize allows users to interact with real-time streaming data for getting real-time information. Similarly, Noria [30] acts like a storage backend for heavy databases. While streaming data is changing, it caches precomputed but up-to-date results, so that reads to that database execute faster. Noria was first introduced in the work [31].

Table 1
A comparison of Rust parallel programming abstractions.

Implementation support	Tokio	Rayon	Pipeliner	Ours
Stream parallelism	Yes	Possibly	Possibly	Yes
Data parallelism	Possibly	Yes	Possibly	Possibly
Task parallelism	Yes	No	No	No
Data-flow parallelism	Possibly	No	No	No
Structured parallelism	No	Possibly	No	Yes
Stateful stages	Yes	No	No	Yes

Table 1 compares some main features of the parallel programming abstraction that are related to ours. The column *Ours* refers to our work, which focuses on structured stream parallelism. Tokio is a general-purpose parallel programming abstraction for asynchronous programming, therefore, it is flexible enough to support almost any parallelism domain implementation. However, implementing parallelism is not trivial, since it still exhibits too many low level parallelism details. In the future, Tokio may be used by other higher-level abstraction as an intermediate for generating efficient parallel code. Pipeliner is a parallel abstraction for modeling pipelines only. Pipelines are expressive enough to fully support stream parallelism, but also partially support data parallelism. Nonetheless, in Pipeliner the stream parallelism support is further limited since it does not allows stateful stages. Rayon targets efficient parallel execution for data parallelism. The parallel iterators that Rayon uses are also very expressive, since they are built based on Rust standard library iterators. This explains why Rayon possibly supports stream parallelism. Besides, Rayon's internal iterative functions resemble parallel patterns (maps and reduces), making this tool the closest one regarding structured parallelism. The tool proposed in this work (Rust-SSP) is the only solution that fully supports stateful stages combined with structured stream parallelism.

Another difference with respect to related works is that we contribute with a new benchmark and its parallelization using all parallel abstractions discussed in Table 1. Although the parallel programming abstractions have well-defined standards and goals, it is Rust developers' interest to know their performance behavior. Indeed, this entails in two limitations we must handle in our work. First, none of the parallel programming abstractions has yet been evaluated with real-world stream processing applications. Second, since Rust is a relatively new programming language compared to others like C and C++, there is a lack of well defined applications and benchmark suites for this programming language. For this reason, we have manually implemented a set of applications that will be described in Section 4.

3. Rust-SSP: Structured stream parallelism

This section discusses Rust-SSP, a high-level and efficient parallel programming abstraction for structured stream parallelism in Rust proposed in [15] and deeply discussed in this article. Section 3.1 reviews our design principles and discusses a possible systematic methodology

for expressing stream parallelism. Then, Section 3.2 presents the API abstractions and how programmers can use this abstractions to parallelize sequential code. Section 3.3 goes beyond by discussing low-level mechanisms and strategies used for implementing Rust-SSP runtime.

3.1. Design principles

In this section we discuss the design principles for Rust-SSP. Our main goal is to provide a high-level and efficient parallel programming abstraction. For that, we take inspiration from structured parallel streaming programming libraries for C++, namely TBB (Threading Building Blocks) [7], *FastFlow* [3], and SPar [13]. The first two are general-purpose parallel libraries, while the later is a domain-specific language (DSL) for stream parallelism. For now, our solutions remains closer to libraries like TBB and FastFlow. Nonetheless, in the future we would like to approach SPar's level of abstraction, which presents some benefits for developers. For example, in SPar developers use C++11 annotations for expressing stream parallelism within the sequential code. This bypasses the need of refactoring the sequential code. Currently, we are interested in offering a friendly and expressive interface for implementing stream parallelism in Rust, which could later be used in combination with a SPar like methodology. Rust-SSP offers a first layer of abstraction for improving programmability without giving away much performance. Our design goals are:

- Hide and abstract many parallelism details from high-level developers. Examples are synchronizations, task scheduling, low-level system mechanisms, and others.
- Support stateful stages and other efficient mechanisms that high-level programmers can use explicitly or implicitly with minimal effort. Examples are communication queues, ordering mechanism, and others.
- Avoid the use of Unsafe Rust in the API and runtime.

Rust's programming language unique features establishes new benefits and challenges for achieving this design goals. We discuss some of them as follows. Rust extensive compile-time checks may decrease concurrency issues in library code as well as programmers' specific implementations. It is necessary to clarify the extent in which Rust ensures the program is correct. Rust is designed to avoid undefined behavior such as data races such as memory and lifetime issues (i.e. double-free and dereferencing null pointers). However, Rust does not detect some concurrency problems, like deadlocks. Rust helps making the program safer by forbidding code that generates an undefined behavior. Undefined behavior in Rust is defined in the *Rustonomicon*, a guide for Unsafe Rust [1]. Even with compile-time checks, the programmer is still responsible for correctly implementing multi-threading without deadlocks or memory leaks. This can be a confusing Rust aspect: although difficult, memory leaks are possible with Safe Rust. In some scenarios Rust allows the programmer to explicitly leak heap-allocated memory with the `Box` type. This is useful when lower-level control of memory is necessary, but this is not the scope of this document.

One of the main differences between Rust-SSP and similar structured parallel abstractions from C++ concerns mutability. In FastFlow and TBB, the pipeline stages are implemented as classes or structs. Each stage is free to mutate itself. Since C++ compiler does not check data races, the programmer could accidentally implement data races. In Rust-SPP, each stage has by default local state because of Rust semantics. For instance, if a pipeline stage is running with 4 threads, each thread will see a different state. In other words, each thread has the ownership of its own local variables or state. If the programmer needs shared state, they need to use Rust's synchronization primitives, as will be better discussed in the next Section 3.2.

In Rust-SSP, each pipeline stage has input and output types, except the last one, which may only have inputs. Additionally, real streaming applications can drop some data items during computation. We support this characteristic in Rust-SSP, where stream items does not necessarily need to flow for each stage. Stages support dropping items without affecting the pipeline execution flow. Since values are moved between threads, Rust-SSP requires the data type to implement the `Send` trait as well. This trait tells the Rust compiler that the value can be safely moved between threads. Most common Rust types implement `Send` and `Sync` by default.

3.2. Rust-SSP programming interface

In this section we describe Rust-SSP (available at[2]). Our solution offers a small set of abstractions with straightforward syntax that programmers can use for expressing stream parallelism in the sequential code:

- A `pipeline!` macro to instantiate the pipeline parallel pattern.
- The `parallel!`, `sequential!` and `sequential_ordered!` macros to express stages in the pipeline.
- `post`, `collect`, `end` and `end_and_wait` methods to control the pipeline execution and data exchange.
- `In` and `InOut` trait objects to instantiate stages.

In Listing 1 we illustrate a pipeline implementation using Rust-SPP. It resizes a stream of images, and waits until all images are processed. The pipeline parallel pattern is composed by a `Pipeline` object with three computing stages: `LoadImage`, `Resize` and `SaveToDisk`. Since stages are stateless and can be safely executed in parallel, we instantiated them using the `parallel` stage macro. By default, a programmer can specify the code that will be executed within the pipeline stages using one of the following methods:

- A function name or a closure for stateless stages. The function must return `Option`;
- A `struct` that implements `InOut` (intermediate pipeline stages) or `In` (last stage) traits for stateful and stateless stages. Rust enforces the programmer to implement the `process` method which also needs to return `Option`.

For this example, we used the last method. An illustration of such implementation is given in Listing 2. The `InOut` trait declares a `process` function that takes a value from the stream and returns a new value. The code that will be executed by the stage must be implemented inside the `process` function. The input parameter of this function are also the stage inputs. Besides, the return values will be the output data of this stage. The code for `In` has been omitted for brevity, but it is similar, except it does not have output. Both trait objects allow the user to mutate internal state.

To instantiate a stage, three macros are available:

- `parallel!`(stage, threads): defines stages that are stateless and can be safely replicated for parallel execution.
- `sequential!`(stage): defines a stage that runs sequentially in a single thread.
- `sequential_ordered!`(stage): defines a stage that runs sequentially in a single thread, receiving items in the same order they were initially produced.

Listing 1 also illustrates how data generation is performed for the pipeline. The `post` routine (line 10) is normally coupled with a data stream source argument (i.e. camera, social media logs, directories, and others). Each generated input item from the source must be posted in the pipeline so that the Rust-SSP runtime will send it in an on-demand fashion to the next stage. When required, a programmer can collect the pipeline output into a vector. In this example, we use the `collect` method. This abstraction also en-queues in the pipeline the

[2] Rust-SSP source codes: https://github.com/GMAP/rust-ssp.

"end" signal. When a thread receives the "end" signal, it propagates this signal to the subsequent stages and terminates its processing. This eventually cause all threads in the pipeline to finish. Additionally, there is a `end_and_wait` method if the programmer does not want to collect the results. If the programmer forgets to call `end_and_wait` or `collect`, the library automatically sends an end signal and waits for all threads to join. This is done by implementing the `Drop` trait, which is similar to a C++ destructor. To create an infinite stream, the programmer must simply not signal the end of stream or drop the pipeline.

```
1
2  let threads = num_cpus::get();
3  let pipe = pipeline![
4    parallel!(LoadImage::new(), threads),
5    parallel!(Resize::new(), threads),
6    parallel!(SaveToDisk::new(), threads),
7    sequential!(Collector::new())];
8  let image_paths = load_from_dir();
9  for entry in image_paths {
10   pipe.post(entry).unwrap();
11 }
12 let result = pipe.collect();
```

Listing 1: Pipeline example: resizing images.

```
1
2  struct LoadImage;
3  impl InOut<PathBuf, Image> for LoadImage {
4    fn process(&mut self, input: &str) -> Option<Image> {
5      if file_exists(input) {
6        Some(load_image(input))
7      }
8      else{ None }
9    }
10 }
```

Listing 2: Creating an `InOut` stage.

In Rust-SSP, we offer macros instead of other language features. The main reason is to simplify instantiating the stages. Macros in Rust are very common, as can be noticed in one of the most famous Rust programming instructions: `println!`. The other reason we used macros is because, in Rust, each copy of a parallel stage needs to be independent, as explained in Section 3.1. For that, an alternative would be forcing programmers to implement the `Clone` trait for each stage. Another alternative would be enforce the use of factory functions. Although simple to implement, programmers would be compelled to write extra boilerplate code for implementing new stages in these alternatives. Therefore, the best solution we found were macros. We automatized the process by transforming the first macro argument into a factory function by wrapping it into a closure. Therefore, the programmer only specifies the stage, and the macros take care of correctly and automatically instantiating it.

Rust-SSP abstracts the *Farm* concept, focusing on *Pipeline* concepts that mimic an assembly line, which is also the concept introduced on SPar for C++. However, *Farm* can be seen when there are the following structures in Rust-SSP: `let pipe = pipeline! [parallel! (...), sequential!(...)]`. Therefore, the emitter may be represented by the `post()` method like in line 10 of Listing 1. The worker is then represented by `parallel!` and the `sequential!` represents the collector.

Programmers can be guided by the following methodology for expressing stream parallelism in their sequential code:

1. *Identify code regions that could execute independently and in parallel:* A simple way to check if a code region can run in parallel is to verify whether these code regions do not mutate shared state. For instance, a loop processing items where each iteration does not depend on the previous iteration. In other words, a stateless stage. Each item processed by the loop can become a stream item, and the code inside the loop can be organized into a sequence of stages.
2. *Isolate these regions inside functions and identify their inputs and outputs:* The programmer must ensure that the input and output types of the stage are correct. The library supports returning `Option`, which allows a stage to return some item to the next stage or none (dropping an item).
3. *Wrap the functions using the API to build the pipeline:* The library will provide a mechanism to allow the user to pass a given function as a stage of the streaming application. Alternatively, the user might implement a stage using an abstraction similar to FastFlow and TBB in the form of a struct. This struct might also be necessary when a given stage must hold persistent variable states such as a file descriptor for writing output.
4. *Identify the primary input data source:* Every stream processing system will have a data producer. The source of data can be anything: files in a directory, lines in a file, data coming from the network, frames in a video, etc.
5. *Determine whether the program should wait until all items are processed (collection):* Streaming systems can be used for data parallel workloads. The abstraction will provide a way to collect the results when a stream end signal is received by the collector.
6. *Determine whether the stream items should be processed in the same order they are produced:* For some applications, it is important that the stream output is in the same order of the input. For instance: a video application. When the application programmer uses a parallel stage, they might emit stream items in an unpredictable order. Notice that ordering stream items require an extra processing step in the collector, which might cause a measurable overhead.

3.3. Rust-SSP runtime

In this section, we discuss the low-level Rust-SSP mechanisms. We describe how stages are connected to each other; how the output of a stage is sent to the next stage; the synchronization mechanism for work queues; and how threads are spawned. Our parallel programming abstraction core mechanisms are generated using the `In` and `InOut` trait objects. Internally, these trait objects become a `PipelineBlock`. A `PipelineBlock` is a trait that requires its implementations to provide a `process` method. This trait is purely internal, and it is not exposed to the user. There are two implementations: `InOutBlock` and `InBlock`. We illustrate a high-level representation of such traits and their relations in Fig. 1.

In Fig. 1, all `InOutBlock` objects have the ownership of the next block, which are implementations of `PipelineBlock`. They are stored in the `next_step` field. Whenever a parallel pipeline stage is created, Rust-SSP starts long-lived threads for each replica of the stage. Since the thread can potentially live forever, we need to have multiple ownership of the work queue and the next step. For that we use `Arc`, which is an atomic reference counter in Rust. Internally, Rust-SSP macros transform `InOutBlock` and `InBlock` objects (exposed by the public API) into pipeline blocks and spawn all the necessary threads using Rust standard threads.

To ensure threads join after computation, Rust-SSP also implements the `Drop` trait, which is similar to a destructor in C++. It is called automatically by the compiler whenever the pipeline trait goes out of scope. Joining threads works as a sort of barrier that does not allow the main program to continue executing until all threads finish. Since spawned threads can potentially live forever, this mechanism guarantees that working threads are finished and destroyed when the pipeline trait is dropped. The `Drop` implementation also sends a stream stop signal: once all threads receive the signal, they first finish their execution then stop running and join. As explained in Section 3.2, the

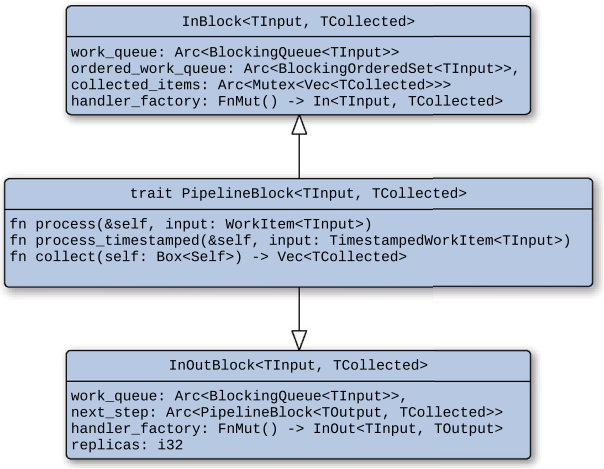


Fig. 1. Pipeline block implementations.

user can explicitly call the methods `end` or `end_and_wait`. These methods are equivalent to `Drop`: they send a stop signal and wait until all threads finish.

During runtime, Rust-SSP stores the last stage output results into the `collected_items` vector in the `InBlock` type. However, this raises the following questions: if a stream is infinite, will Rust-SSP always store processed items into this field? Will it accumulate data infinitely? The answer is yes. Nonetheless, the programmer can implement the last stage to return an empty item of type `()`. This is similar to `void` in C++. This type is a *zero-sized type*, which does not cause any allocations and can be stored in a `Vec` without ever allocating data.

Still regarding Fig. 1, the `process` routine implemented in `InOutBlock` and `InBlock` enqueues the `WorkItem` in a FIFO queue, namely `work_queue`. `InOutBlock` executes the stage code and passes the result to the next stage using the `process` method. For `InBlock`, it only executes the stage code, since this object is the last pipeline stage. By using generic types, Rust can prevent type errors at compile-time. This means the input and output types need to match for `next_step` on `InOutBlock`.

Stream processing systems frequently use queues to communicate data between stages. In Rust-SSP, this is implemented using the `BlockingQueue` type, which is based in a `VecDeque` type wrapped by a `Mutex`. When ordered items are needed, Rust-SSP uses a different communication queue, namely `ordered_work_queue`. This queue is a `BTreeMap<u64, WorkItem>` type wrapped in a Mutex. It is a map in which the key is the number of the work item, which increases by 1 for every new work item posted in the pipeline. This map keeps the items in the order they were produced, and the worker threads always search for the next item, advancing by 1 at every iteration and removing from the map. However, stages are free to drop items from the stream by returning `None`. To avoid losing track of the counting, even when an item is dropped from the stream, it still produces a stream signal. The `WorkItem` represents all possible signals:

- `Value(T)`: A value to be processed by a stream stage.
- `Dropped`: Represents a stream item drop signal. It gets processed by all stages in order to keep tracking of ordering.
- `Stop`: A stream stop signal. Once it reaches a stage, all threads of that stage finish execution. The first thread that receives a Stop signal in a parallel stage broadcasts the signal to all other threads.

Finally, macros are also used to link together all blocks of the pipeline. If we had opted for a manual linking approach, where the programmer must construct the stages and connect them using a method call, the ownership system would require the linking of the blocks to be done in a counter-intuitive *reversed* order. If a pipeline sequence has stages in the order `s1`, `s2` and `s3`, the programmer would have to create links in the order `s2.link_to(s3);s1.link_to(s2);` because the subsequent stage is owned by the previous stage. This effectively ends the scope of the object, preventing it to be used again. The `pipeline!` macro allows the stages to be defined in the intuitive order without worrying about ownership. Macros are also the only way to make variadic functions in Rust. If a standard function was used, the programmer would have to pass a vector of stages. Therefore, the macro provides a syntactical advantage by offering a clean way to pass a varying number of stages without boilerplate code.

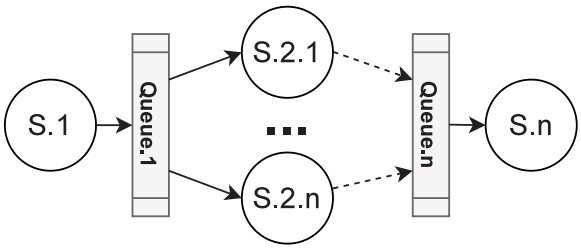


Fig. 2. Data flow in the runtime.

A general visualization of the runtime is shown in Fig. 2, representing an abstract stream processing application. In it, the first Stage ($S.1$) is a sequential stream generator that insert stream items in the first queue (Queue 1). Then, the parallel Second stage ($S.2.1, \ldots, S.2.n$) consume from Queue 1 in an on-demand fashion. The interface supports as many stages, with one queue for each producer–consumer relation, as necessary. In this example, we end the pipeline with a sequential nth stage $S.n$. The queues connecting stages are protected by a single `Mutex` and developed with a FIFO access pattern. When an item is added, a condition variable is used to notify that an item was enqueued. In the case of a sequential ordered stage, we re-order items using a tagging and reordering algorithm based on [32]. The tagging process is performed by the `post` method. It assigns to the stream item a tag value based on an internal counter that increments each time the `post` method is called. The reordering algorithm applied to the sequential stages dequeues work items in order based on this tag, blocking if the next item is not in the queue yet.

4. Rust stream benchmark suite

Rust is a relatively new language. As a result, it does not have as many applications, benchmark suites, and scientific studies as an older language such as C++. In C++, there is a plethora of well established benchmark suites available in the literature to choose from. These benchmark suites are composed of a set of applications with very well defined characteristics and workloads. This way, when a new parallel processing solution in C++ is developed, researchers can perform a solid comparison with regards to the state-of-the-art. Some examples of great popularity are Parsec [33], Splash [34], and Rodinia [35].

On the other hand, evaluating new Rust solutions require the development of ad-hoc benchmark suites. Often, these are particular to each research or to a specific scenario. This means that different solutions to a similar problem will likely be compared with totally different methods. Another important aspect is that in this benchmark suite we do not consider distributed data stream applications. Instead, we only focus on multi-core environment. In the future, we can consider this domain of applications, such as the ones used in DSPBench suite [36]. They recently worked with streaming applications in distributed computing environments like clusters and cloud. With that in mind, our goal is to provide a Rust benchmark suite suitable for parallel stream processing applications targeting multi-cores.

In this work, the benchmark suite will be used for evaluating Rust-SSP with respect to state-of-the-art parallel programming abstractions

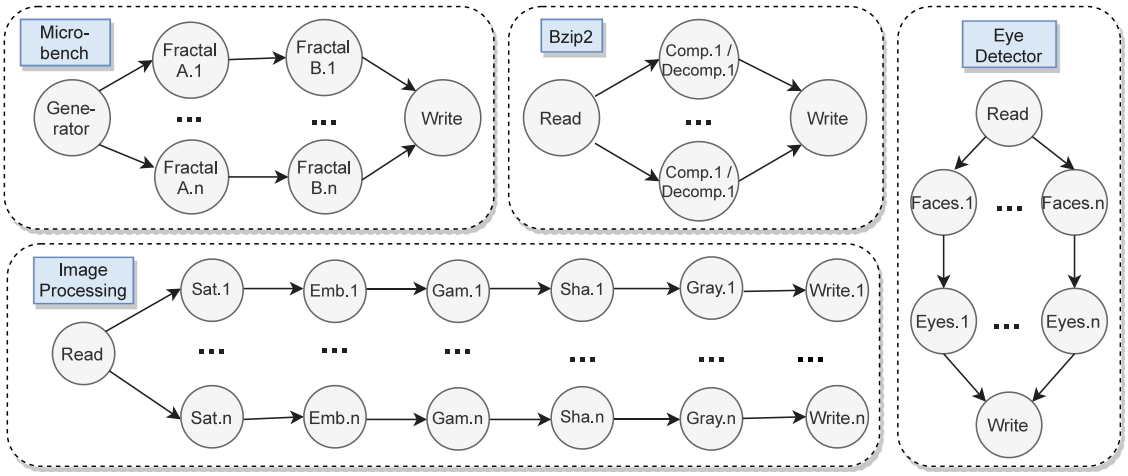


Fig. 3. The flow-graph of the benchmark applications.

in Rust targeting stream parallelism. The benchmark suite[3] is available to the community so that other researches focusing on stream processing applications can use it. Consequently, other researchers can use the benchmark suite to evaluate qualitatively or quantitatively their own parallel programming abstractions, programming strategies, computer architectures, compilers, and other computing systems. In the following, Section 4.1 characterizes and explains each one of the 4 selected applications. Then, in Section 4.2 we characterize 3 different workloads for each one of these applications.

4.1. Applications

In this Section we characterize and explain each one of the four applications selected for our benchmark suite. Two of them are simpler test cases while the other two are more robust and closer to real-world cases. The test applications are Micro-bench and Image Processing. The real-world cases are Bzip2 compression and Eye Detector. They were chosen to represent different computational characteristics that can be commonly found in the stream processing scenario. These involve different memory access patterns, disk access, throughput variation, number of processing stages, and computational intensity.

The Fig. 3 depicts the streaming data flow for each application within our benchmark suite. Each sphere represents a sequential block of code, or, in other words, a stage of computation. The arrows represent the direction data is being moved. The ellipsis symbol indicates that the current stage is stateless, meaning the stage can be replicated.

The Micro-bench is a mathematical synthetic application we designed using a fractal in the complex plane, namely the Mandelbrot set. The main reason we choose this application is because of its unbalanced computation. For example, points from the complex plane that are within the Mandelbrot set may compute to infinite, while others iterate much less. Micro-bench data flow is depicted in the superior part of Fig. 3. We have modified the Micro-bench (Mandelbrot set fractal) to have two sequential stages of intensive computation, namely Fractal A and Fractal B. For each one of these stages, the computational intensity can be customized by setting a maximum number of iterations. Usually, the Mandelbrot set is parallelized using a single stage, instead of two stages design. This design tends to stress the abstractions' schedulers by increasing the level of complexity required to balance the use of computer resources.

The next test application chosen for the benchmark suite is called Image Processing. It is characterized by a stream of images flowing through 5 different filters, as seen in Fig. 3. They are Saturation, Emboss, GammaCorrection, Sharpen, and Grayscale. All filters were provided by *Raster* library from Rust. Some filters may take longer than others to complete and all of them are stateless, which means they can be parallelized. It is important to highlight that these filters, together, do not represent any specific computation. It was created to represent a deeper pipeline and stress different computational workload characteristics for our benchmark suite.

The Bzip2 application is an important open-source tool for lossless data compression/decompression. Bzip2 uses the Burrows–Wheeler transform to optimize frequently-recurring characters. This effectively makes the compression computationally costly whereas the decompression is shipper. The algorithm divides the input file in blocks of bytes with sizes ranging from 100 kb up to 900 kb, where the block size must be multiple of 100 kb. In our version of the algorithm, we used 900 kb blocks as default. One of the main specific characteristics of Bzip2 is that the performance heavily depends on the characteristics of the input file. Bzip2 is built using three stages as depicted in Fig. 3. The first one (Read) reads data from the input, the middle one is a computational intensive stage compress or decompress, and the last stage collects and writes the output items, which items need to be re-ordered.

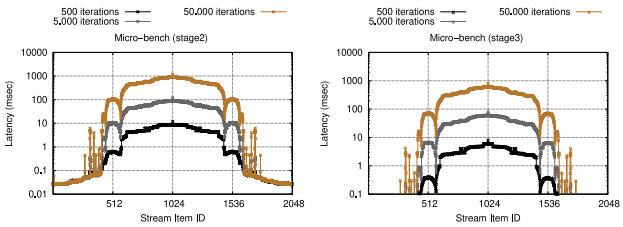
The last application chosen to compose part of the benchmark suite is Eye Detector. It is a stream processing application that detects the eyes in the faces within an input video. One possible use of this type of application would be tracking the eye movement of a group of people to automatically identify their interest in a certain product inside a store. The image processing is performed using Rust's OpenCV library support. Eye Detector can be separated in four processing stages. They are illustrated in the right-hand part of Fig. 3. The first stage constantly reads frames from an input video. Then, the second stage is responsible for detecting faces in each individual frame. These detected faces are forwarded to the third stage, which detects a set of eyes in each face. Finally, the fourth stage writes the resulting detection to an output video.

4.2. Workloads

In this Section, we describe three different workloads for each application. They were selected to represent different computational characteristics for each application such as computational intensity, contrasting memory or disk read/write overheads, and throughput variation. The main goal is to later use these workloads to assess performance and to help identifying potential issues or advantages of each parallel programming abstraction. For this workload characterization, we ran the experiments in the same machine described in Section 5.1.

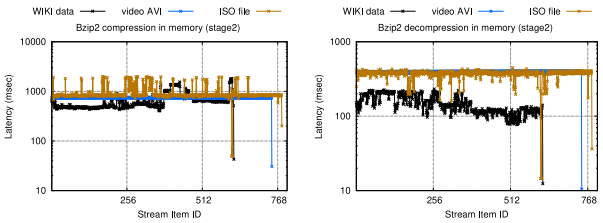
We have measured latency to characterize each workload. Latency represents the time it takes from one stage to compute a single stream item from the beginning to the end of that stage. For example, if a

[3] Benchmark source codes: https://github.com/GMAP/RustStreamBench.



(a) Fractal A. (b) Fractal B.

Fig. 4. Computational intensive stages of Micro-bench.



(a) Compress. (b) Decompress.

Fig. 5. Characterization of Bzip2 Compress/Decompress stages.

Table 2
Workload sizes before and after Bzip2 compression.

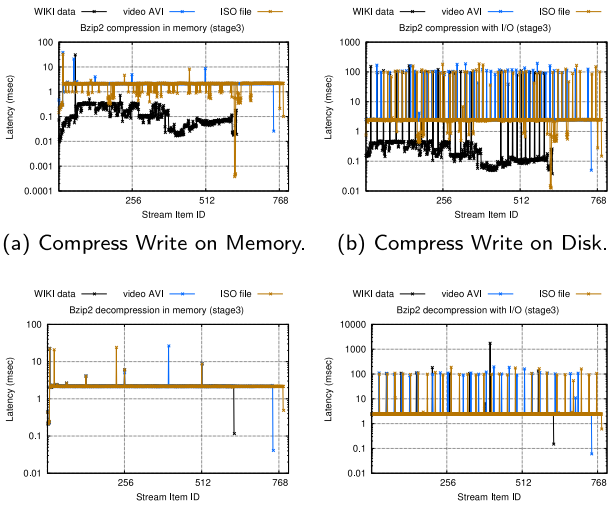
Bzip2	WIKI data	AVI video	ISO file
Original file	557 Mb	673 Mb	704 Mb
Compressed file	78 Mb	665 Mb	658 Mb

certain workload has 100 items in the stream, there will be 100 different latency measurements for each processing stage. The latency was individually measured in milliseconds for each stage of the applications (stages defined in Section 4.1). All latency data was collected from the sequential implementation. This way, there is no interference from any parallel programming abstraction. Next, we demonstrate all latency measurements.

In Fig. 4 we present the two computational intensive stages from the Micro-bench application, which are the Fractal A and B stages. Since the Mandelbrot set fractal is a known and well-defined set of complex numbers, the three workloads have the same characteristic with different intensities. We have made the number of stream items constant, using 2048 total items. However, we have modified the maximum number of iterations for each stage using 500, 5.000, and 50.000 iterations. In our experiments, we also used asymmetric values for splitting the iterations between the two stages. For example, one stage computes 60% of the iterations while the other the remaining 40%. This explains why Fractal A (Fig. 4a) has maximum latency around 35% higher than the maximum latency achieved on Fractal B (Fig. 4b). Concerning the Read and Write stages, we do not illustrate the results because the latencies are very low and similar for different input configurations.

Fig. 5 presents the results for the computational intensive stages of Bzip2, which are the Compress and Decompress stages. We have chosen three different workloads from distinct domains: (1) WIKI data — a binary file containing text information about the page view statistics from Wikipedia. It represents a workload with significant variation. (2) AVI video — a video with 300 fps and dimensions 880*x*540. It represents a constant input, with few variation. (3) ISO file — an ISO file that represents a combination of a constant input with an unbalanced workload.

Table 2 presents the original sizes of the three workloads and the corresponding sizes after executing Bzip2 compression. We were careful to select workloads with similar amount of computational work.



(a) Compress Write on Memory. (b) Compress Write on Disk.
(c) Decompress Write on Memory. (d) Decompress Write on Disk.

Fig. 6. Characterization of Bzip2 Write stages.

In Fig. 6 we present the characterization of the writing stages. We used two versions of the Bzip2 application: One writes the results on memory, while the other version writes the results on disk. The versions were designed having in mind the parallel implementation. For that, the memory version exhibits only overheads arising from the parallel programming abstractions while the I/O version exhibits other sources of overhead commonly found on the stream processing domain. As can be seen in Figs. 6a and 6c, that operate on memory, the latencies are similar for all stream items, except the WIKI data on Compression. The reason is that compressing the WIKI data may compress the file with a factor of 7.14 times, while others have a factor of 1.07 times at best. This is shown in Table 2. On the other hand, Figs. 6b and 6d show that writing on disk exhibits high latency variations. The former graphic shows more variations than the later since compressing blocks results in unknown sizes, which depends on how much data was compressed.

Regarding Image Processing application, Fig. 7 showcases latency results for each one of three different workloads: *small*, *mixed*, and *big*. The idea is that the higher the resolution, the longer each filter may take to process the image. With that in mind, *small* workload is a set of 100 images of the same size (640 × 427 pixels), *big* is composed of 100 higher resolution images (1920 × 1280 pixels), and *mixed* is composed of half of each workload to create a more unbalanced scenario. All images are encoded using the JPEG format. The latency results in Fig. 7 showcase that *small* and *big* are relatively stable across all filters while *mixed* oscillate between the two. Finally, read and write image operations perform in a similar pattern, as depicted in Fig. 8.

Fig. 9 represents the latency measurements for Eye Detector's workloads. We have named the first workload as *one*, the second as *mixed* and the third as *several*. All workloads are MPEG-4 videos with 450 frames, 15 s duration, and 640 × 360 resolution. The difference between workloads was in the video characteristics. The *one* video was characterized by a single face in the whole video. On the other hand, *several* had several faces in the whole video and *mixed* had moments with only one face and moments with several faces. This way, the variations in the workloads are actually how many elements are there to process in the image and not the image itself. The first and last stage in Fig. 10 are similar between different workloads since they perform I/O operations. The second stage in Fig. 9a detects faces, therefore the more faces in it, the longer it takes to process. The third stage, which detects eyes, in Fig. 9b has even more variation. That is because if there are more faces, there is more eyes to process. This workload has moments where the latency is close to zero since no faces were detected in that particular frame.

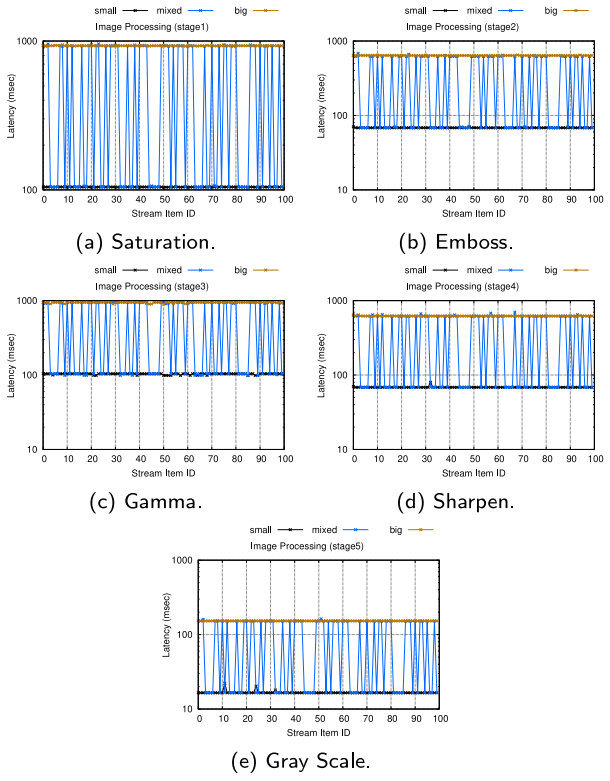


Fig. 7. Characterization of computational intensive stages for Image Processing.

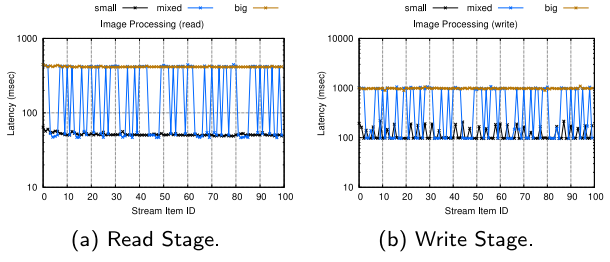


Fig. 8. Characterization of Read and Write stages for Image Processing.

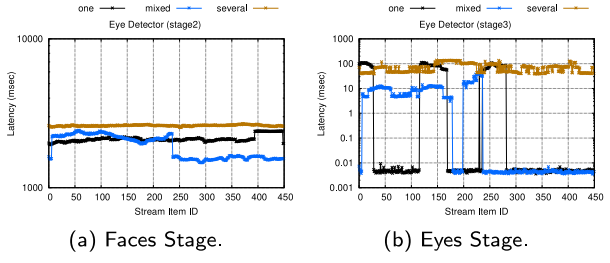


Fig. 9. Characterization of Eye Detector Read and Write stages.

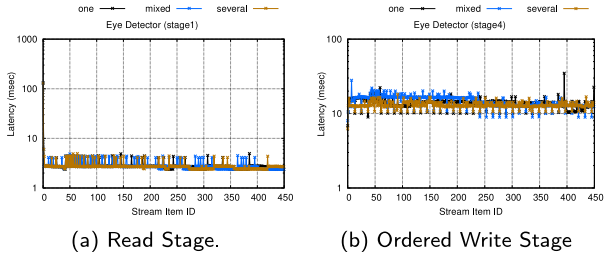


Fig. 10. Characterization of Eye Detector Read and Write stages.

5. Evaluation

This Section presents the experimental analysis of this paper. First, Section 5.1 showcases the performance results using three different workloads for each application, by first explaining the parallel implementation for each parallelism abstraction used. Second, we deliberate on Rust parallelism development and programmability for stream processing applications in Section 5.2.

5.1. Performance evaluation

The experiments were executed on a machine with two Intel(R) Xeon(R) Silver 4210 CPU @ 2.20 GHz, featuring 20 cores and 40 threads. Each hyper-threaded core has 64 kB private L1, 1 MB private L2 and 13.75 MB of L3 shared. The machine has 64 GB of RAM @ 2400 MHz. The machine is also equipped with four HDD 3.5 @ 7200rpm using SATA 3.1 with 6.0 Gb/s. The kernel was Linux 5.4.0-59-generic and the OS Ubuntu 20.04.1 LTS. The code was compiled using Rust 1.47.0 and the optimization flag `--release`. Other software details are, Tokio version 0.1.17, Rayon version 1.5, Pipeliner version 1.0.1, Crossbeam utils version 0.8, and Crossbeam channels version 0.5.

The execution time metric was chosen to evaluate performance. With the execution time, workload information (Section 4.2), and degree of parallelism, most other performance metrics can be derived. Regarding the graphs, the x axis is always the degree of parallelism, which goes from 1 up to 40 (number of cores with hyper-threading in the system). The position at $x = 0$ in the graphs represents the sequential version of each application.

Another important aspect to consider is that the number of replicas in the x axis may not represent the actual active thread count in the system. Rather, each runtime may spawn one dedicated thread for each stage, or in the case of parallel stages, a thread pool determined in size by the value of the degree of parallelism. Ultimately, each parallel runtime and its abstractions define the active thread count. Finally, the y axis in the graphs represents the execution time in seconds, which are represented in logarithmic scale 2.

Each plotted value on the graphs was obtained from the arithmetic mean of 5 executions performed for each parallelism degree value ranging from 1 up to 40. Moreover, the standard deviation was plotted in the form of error bars, which may not be visible in some cases as it is mostly negligible. To guarantee the correctness of the parallel versions, we compared the hash value of the output with the sequential version.

For the parallel versions using `std-threads`, we have not used async-std mechanisms, which employs lightweight tasks handled as futures and streams. Instead, we have opted for spawning persistent threads. The downside of this approach is to handle communication and synchronization, in which we used efficient lock-free crossbeam bounded channels.

5.1.1. Micro-bench

Previous Fig. 3 has shown the Micro-bench data flow contains four stages, in which two are computational intensive and stateless. An efficient parallel version was obtained by implementing a non-linear pipeline with two sequential stages (Generator and Write) and two parallel stages (Fractal A and B). Because of parallelism, it is mandatory to re-order stream items after the parallel stages for obtaining the correct output. Rust-SSP has a sequential ordered stage for that. For Tokio, Rayon, and Pipeliner we have collected all stream items and executed an efficient ordering from Rust's standard library. In real work stream processing, this is not a viable option since the stream will never end. The reason to do in this way is that Rayon and Pipeliner do not support stateful sequential stages for collecting stream items. Although Tokio has an ordered buffer, it achieved poor performance with respect to the other parallel abstractions and we avoid using it. *std-threads* ordering was manually implemented using the strategy presented in the following work [32].

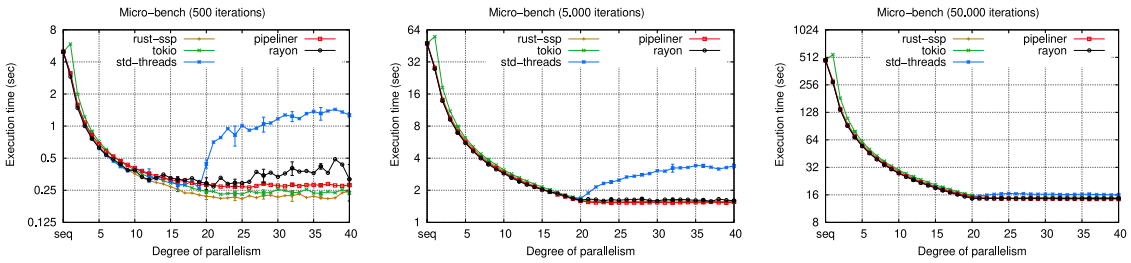


Fig. 11. Execution time in seconds of Micro-bench.

Fig. 11 shows the results. As can be seen, with the smallest workload, Rust-SSP presents on average lower execution times than others, followed by Tokio with a difference close to 10%. In Tokio, executing with a single thread is 18% slower than executing Micro-bench sequentially. However, it scales well later. Both Rayon and *std-threads* results are unstable for the smallest workload. Rayon uses work-stealing communication queues, which exhibits to much overhead for this small workload (250 ms). The mechanism overhead only pays off for bigger workloads. *std-threads* was implemented manually and we did not optimize everything possible. The reason is that the current *std-threads* version uses busy-wait channels, which introduces starvation when the execution has more threads than available cores. In degree of parallelism 20, there are 41 threads running (20 threads for each parallel stage + 1 thread for the generator). More details about the problem can be found in Section 5.1.4, where we introduce an optimized queue for *std-threads*. With the largest workload, the differences between the parallel programming abstractions are similar, except *std-threads*. At maximum degree of parallelism, Rust-SSP and Rayon present the same execution time, with difference less than 0.1%, while Tokio is 2.3% slower.

5.1.2. Image processing

Image Processing results were depicted in Fig. 12. For this application, it was developed a version using Rust-SSP, Tokio, *std-threads*, Pipeliner, and Rayon. In this case, we did not consider the stateful read and write operations in the implementation. Instead, they were performed before and after processing respectively. Therefore, Image Processing implementation comprises solely the 5 image filtering stages previously described.

Among all results in Fig. 12, Rayon demonstrated a relatively high standard deviation, which was 5.05 s at its highest in *big* workload. Another trend was the load balancing issue with *std-threads* version. This was due to the busy-wait scenario introduced by the crossbeam queue implementation. In fact, we observed that Rust-SSP performs up to 2.47 times faster than *std-threads*. Overall, Rust-SSP, Tokio, and Pipeliner have mostly comparable results, but there were cases where the difference is 5% between them.

5.1.3. Bzip2

Bzip2 is a straightforward application from the parallel stream processing viewpoint. Bzip2 embraces two complementary applications, a compress and a decompress. As shown in Fig. 3, it has the same data flow for both versions, which is composed by three stages. The Read and Write are sequential stateful stages, since they use a single file descriptor for reading and writing data. Reading from and writing into the original file uses by default blocks of 900 kb. However, operating over the compressed file shows unbalancing, since the block size can vary depending on how much data Bzip2 algorithm is able to compress. To measure the performance we designed two versions, one operates entirely on memory, while the other includes disk operations. The difference is the memory version is an ideal scenario for parallelism in which other sources of overhead are hidden. For instance, local disk read and write operations, network communication latencies, and others.

Fig. 13 reports the results for Bzip2 application. The first two Figs. 13a and 13b present the results when executing on memory. As can be seen, the behavior of different parallel programming abstractions is very similar, even for different workloads. In Fig. 13a, the parallel abstractions exhibit maximum variation of 3.8%, but on average are close to 1%. This maximum variations are from Tokio and can be seen in the decompress version (Fig. 13b) using the AVI video and ISO file workloads. Tokio's performance scaling decreases around degree of parallelism 30. Also, we observed that on maximum degree of parallelism the *std-threads* version again exhibits starvation. On the other hand, implementing I/O streaming applications indeed reveals some parallel programming abstractions' weaknesses, meaning it was worth introducing such additional Bzip2 version. Results are depicted in Figs. 13c and 13d. In this case, Bzip2's last stage needs to maintain the state of a file descriptor for saving compressed/decompressed data. Rayon and Pipeliner cannot implement those applications because they do not support stateful stages. A possible workaround would be using external mechanisms. That is, manually spawning a new thread that receives data from a channel connected to all of the third stage parallel threads. In this work, we focus on supported default library mechanisms and did not implement such version.

Rust-SSP and *std-threads* present similar performance behavior (difference close to 1% on average). Tokio results show variation depending on the input. The worst results (up to 41.1% slower than the other two) were observed in unbalanced workloads, as discussed in Section 4.2. The reason Tokio achieved this performance is mainly because of their ordering mechanism. In other applications (Micro-bench, Bzip2 on memory) we avoided using it on Tokio, which resulted in similar performance with respect to the others. However, since Bzip2 on disk requires on-the-fly output writing, there is no option for not using it besides manually implementing our own ordering algorithm. The only experiment Tokio obtained the highest speedup was in Fig. 13c, using the AVI video workload (middle one). Yet, the results for Tokio are unstable and oscillated between Tokio being 11.3% slower and 6.0% faster than Rust-SSP.

5.1.4. Eye detector

Fig. 14 showcases the results of the Eye Detector application. As explained previously, it is a pure stream processing application composed of 4 computational stages. Since the last stage writes the resulting output to file during execution time, Pipeliner and Rayon could not be used. Instead, this application could be developed using Rust-SSP, Tokio, and *std-threads*. The **std-threads* version is a test case we will explain below. Regarding the parallel implementation, both second and third stages can be safely parallelized as they are stateless stages. That means they do not share or hold previous states of data. On the other hand, the first and last stage respectively perform sequential input and output operations. They only have one file descriptor for reading and another for writing data. Therefore, these are stateful stages and cannot be developed with Rayon and Pipeliner as previously explained.

Evaluating the graphs in Fig. 14, we observed a similar trend among all three workloads. Tokio and Rust-SSP perform similarly (maximum difference of 6.03% in favor of Rust-SSP) while *std-threads* suffers from a load balancing issue. This issue is less prevalent with the *several*

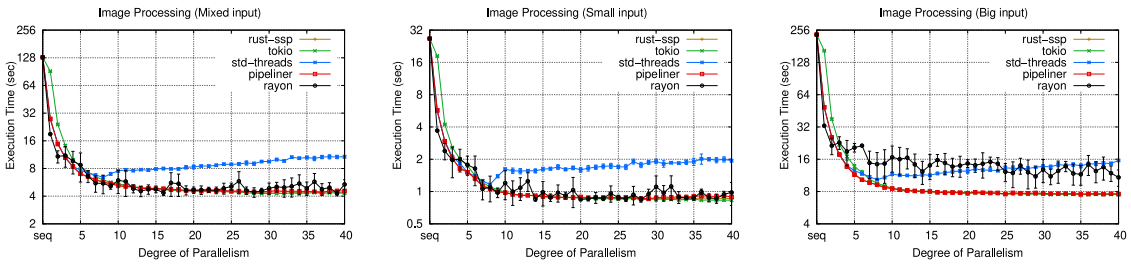
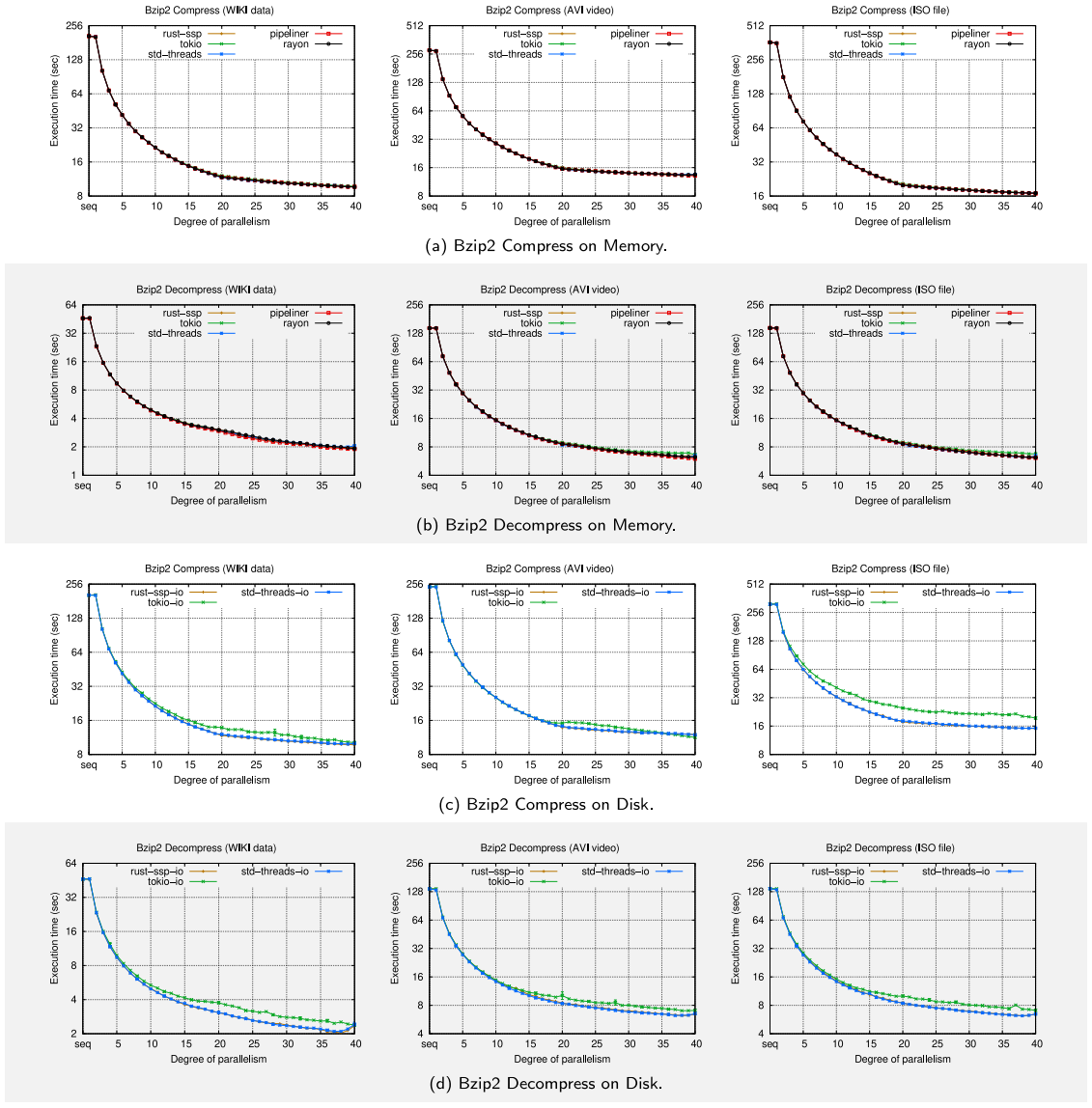


Fig. 12. Execution time in seconds of Image Processing.



(a) Bzip2 Compress on Memory.

(b) Bzip2 Decompress on Memory.

(c) Bzip2 Compress on Disk.

(d) Bzip2 Decompress on Disk.

Fig. 13. Execution time in seconds on Bzip2.

workload in Fig. 14 (right-hand side) since it is a more naturally balanced workload (refer to Section 4.2). The reason for these problem is because the default crossbeam communication channel deployed is affected work starvation and does not relinquish the thread access until a new work arrives. This effectively means that when there is no work available, it infinitely loop checks the queue for new work or the end of processing. To test prove that this is the problem, we have developed the **std-threads* version, which uses the default Rust *RwLock* for fine grain queue access control combined with standard thread parking and unparking. The resulting tests from **std-threads* demonstrated that the performance was improved by up to 52.56%.

5.2. Programmability

In this Section we discuss programmability aspects regarding the parallel implementation techniques for Rust-SSP, Tokio, Rayon, Rust's standard threads, and Pipeliner. For that, we discuss coding characteristics from each interface and measured SLOC (Source Lines of Code) for the applications described in Section 4.1. SLOC does not consider blank

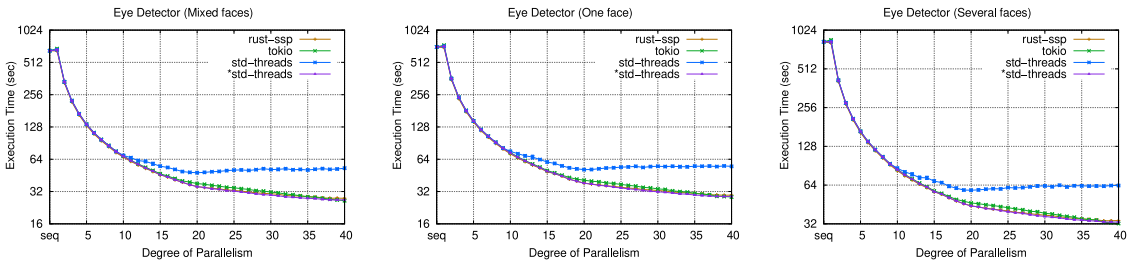


Fig. 14. Execution time in seconds on Eye Detector.

lines and comments, only practical code lines and language syntax. We measured SLOC with the cloc[4] tool. SLOC gives a general idea of the level of code intrusion that each interface requires to enable parallelism. It does so by comparing the number of extra lines of code each interface introduces over the original sequential version. It has however its limitations. SLOC alone is not sufficient to claim that one parallel programming interface is easier to code with. Other aspects need to be considered such as time to develop, programmer experience and background, etc. [37,38].

Table 3 demonstrates the measured SLOC for each benchmark application and parallel programming interface. The label *seq* represents the sequential code. Every line of code comparison is made considering the *seq* as base. Save for Eye Detection application, it can be observed that Rust-SSP is consistently the parallel interface that requires the smallest amount of extra code. This is due to the structured approach Rust-SSP adopts. It is modeled specifically for the stream processing paradigm, completely abstracting from the developer communication and thread management details. On the other hand, in the Eye Detection application, Rust-SSP introduces more lines of code than Tokio. The cause for that is the use of persistent unique local variables in the parallel stages, namely eye and face detector modules. Therefore, it requires a more verbose struct implementation, which ends up increasing SLOC. None of the other benchmark applications require persistent and unique local variables. Regarding the **std-threads* version for Eye Detector presented in the previous Section 5.1, the measured SLOC was 290. This showcases that the extra performance benefits come at the cost of several extra lines of code.

The version that introduces the highest overall number of extra lines of code in Table 3 is *std-threads*. This is the lowest-level programming interface. It requires the developer to directly handle most implementations aspects including thread management, execution flow, data reordering, and communications. Concerning Tokio, the Image Processing application is a case where Tokio can leverage Rust's default iterators to create the stream generation phase. In every other application, Tokio requires the developer to create a special polling function for the stream generation stage, which increases the total SLOC. Tokio also requires the programmer to manually handle thread spawning and stage communications. However, Tokio does not require the programmer to introduce manual reordering algorithms in the applications where its necessary (Micro Bench, Eye Detector, Bzip2).

The Rayon and Pipeliner interfaces in Table 3 were not developed in Bzip2 Disk and Eye Detector applications due to lack of support for sequential stateful stages as previously stated. Both interfaces have similar SLOC results since they are implemented with similar level of abstractions. Additionally, despite the name, Pipeliner does not implement a structured pipeline such as Rust-SSP. It does however, implement parallelism that can be represented with iterators. Conceptually, Rayon supports Pipeliner features and further makes available a plethora of other data parallelism features such as reductions, fork-joins, and others.

[4] https://github.com/AlDanial/cloc.

Table 3
Benchmark applications source lines of code (SLOC).

Interfaces	Benchmark applications				
	Micro-bench	Image Proces.	Bzip2 memory	Bzip2 disk	Eye Detec.
seq.	35	26	103	98	57
rust-ssp	84	47	150	137	120
Tokio	119	74	185	170	107
std-threads	119	121	204	253	187
Rayon	102	44	190	–	–
Pipeliner	103	52	182	–	–

6. Conclusion

In this work, we argued that Rust parallelism abstractions are not yet fully explored due to the relatively small language lifespan. We specifically focused and explained structured parallel programming techniques which are common in C++ but mostly untouched in Rust. Aiming to leverage the structured parallel programming paradigm, we proposed to use Rust-SSP. Therefore, we detailed its design principles, programming interface, and runtime. We have also observed that Rust still lacks well defined stream processing applications benchmark that could be used to compare performance between different parallel implementations. For that, we have composed such benchmark by selecting and describing a set of Rust applications with different computational characteristics. Furthermore, we implemented parallel versions of these applications using different Rust parallel programming abstractions: Tokio, Rayon, Pipeliner, standard Rust threads, and our Rust-SSP. From this point, we also have selected, and explained three different workloads for each application. Combining the applications and workloads, we have created a benchmark for parallel stream processing in Rust that will be made publicly available for the community. With this benchmark suite, we have performed a set of experiments for assessing performance and programmability aspects of the parallelizations.

The experiments have shown that Rust-SSP achieved similar, or in some cases, better performance with respect to other parallel abstractions. Rust-SSP, Standard Threads, and Tokio showed to be the most expressive options to develop stream processing applications. In fact, Tokio is a state-of-the-art solution for stream processing in Rust. However, Tokio has shown performance issues, specially when data reordering is needed (up to 41.1% worse than Rust-SSP). In some cases, Rayon performed up to 30.3% and Pipeliner up to 27.7% worse when compared to Rust-SSP. In terms of programmability, among all parallel abstractions, Rust-SSP was the one that required the lowest average amount of extra lines of code to enable parallelism.

This study has revealed some opportunities for future works. First, our benchmark suite could include streaming applications from the distributed processing domain such as DSPBench [36]. For instance, they could represent heavily distributed IoT applications. Second, there is a possibility that Rust-SSP's communication queue might be a performance bottleneck due to lock contention. Other queue implementation

with fine grained locks could be studied to mitigate this potential problem. Third, we could support more structured programming patterns besides pipeline and allow arbitrary pattern combination. Especially aiming to provide support for data parallelism such as map and reduce patterns and combine/nest it with pipeline pattern.

CRediT authorship contribution statement

Ricardo Pieper: Software, Investigation, Validation, Visualization, Writing – original draft. **Júnior Löff:** Software, Investigation, Validation, Visualization, Writing – original draft. **Renato B. Hoffmann:** Software, Investigation, Validation, Visualization, Writing – original draft. **Dalvan Griebler:** Conceptualization, Project administration, Validation, Writing – review & editing. **Luiz G. Fernandes:** Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to acknowledge the support of GMAP research group and PUCRS university. 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 19/2551-880001895-9), FAPERGS - 10/2020-ARD project SPar4.0 (Nº 21/2551-890000725-7), MCTIC/CNPq - Nº 28/2018 project SParCloud (Nº 437693/2018-0), and MCTIC/CNPq - call 25/2020 (Nº 130484/912021-0).

References

[1] Mozilla Research, The rustonomicon, 2019, URL: https://doc.rust-lang.org/nomicon/.

[2] Rayon, Rayon, 2019, URL: https://github.com/rayon-rs/rayon (Accessed on 27.03.2021).

[3] 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, in: PDC, vol. 1, Wiley, 2014, p. 14.

[4] D. Griebler, Domain-Specific Language & Support Tool for High-Level Stream Parallelism (Ph.D. thesis), Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, 2016, URL: http://tede2.pucrs.br/tede2/handle/tede/6776.

[5] M. McCool, A. Robison, J. Reinders, Structured Parallel Programming: Patterns for Efficient Computation, Elsevier Science, 2012.

[6] T. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, first ed., Addison-Wesley Professional, 2004.

[7] J. Reinders, Intel Threading Building Blocks, O'Reilly, Sebastopol, CA, USA, 2007.

[8] Microsoft, Parallel patterns library (PPL), 2016, URL: https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl (Accessed on 27.03.2021).

[9] A. Moskalev, A. Fedorov, Get started with parallel STL, 2018, URL: https://software.intel.com/content/www/us/en/develop/articles/get-started-with-parallel-stl.html (Accessed on 27.03.2021).

[10] H.C.M. Andrade, B. Gedik, D.S. Turaga, Fundamentals of Stream Processing, Cambridge University Press, New York, USA, 2014.

[11] W. Thies, M. Karczmarek, S.P. Amarasinghe, Streamit: A language for streaming applications, in: Proceedings of the 11th International Conference on Compiler Construction, Springer, Grenoble, France, 2002, pp. 179–196.

[12] W. Thies, S. Amarasinghe, An empirical characterization of stream programs and its implications for language and compiler design, in: Inter. Conf. on Par. Arch. and Compil. Tech., in: PACT '10, ACM, Austria, 2010, pp. 365–376.

[13] 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, http://dx.doi.org/10.1142/S0129626417400059.

[14] Tokio, Tokio - the asynchronous runtime for the rust programming language, 2019, URL: https://tokio.rs (Accessed on 27.03.2021).

[15] R. Pieper, D. Griebler, L.G. Fernandes, Structured stream parallelism for rust, in: XXIII Brazilian Symposium on Programming Languages (SBLP), in: SBLP'19, ACM, Salvador, Brazil, 2019, pp. 54–61, http://dx.doi.org/10.1145/3355378.3355384.

[16] M. McCool, A.D. Robison, J. Reinders, Structured Parallel Programming: Patterns for Efficient Computation, Elsevier, Waltham, MA, 2012.

[17] M.I. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, University of Glasgow, Glasgow, United Kingdom, 1989.

[18] R. Jung, J.-H. Jourdan, R. Krebbers, D. Dreyer, Safe systems programming in rust, Commun. ACM 64 (4) (2021) 144–152, http://dx.doi.org/10.1145/3418295.

[19] R. Jung, J.-H. Jourdan, R. Krebbers, D. Dreyer, Rustbelt: Securing the foundations of the rust programming language, Proc. ACM Program. Lang. 2 (2017) http://dx.doi.org/10.1145/3158154.

[20] V. Astrauskas, P. Müller, F. Poli, A.J. Summers, Leveraging rust types for modular specification and verification, Proc. ACM Program. Lang. 3 (2019) http://dx.doi.org/10.1145/3360573.

[21] A. Levy, M.P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, P. Pannuto, Ownership is theft: Experiences building an embedded OS in rust, in: Proceedings of the 8th Workshop on Programming Languages and Operating Systems, in: PLOS '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 21–26, http://dx.doi.org/10.1145/2818302.2818306.

[22] S. Sydow, M. Nabelsee, H. Parzyjegla, P. Herbe, A safe and user-friendly graphical programming model for parallel stream processing, in: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, IEEE, 2018, pp. 239–243.

[23] S. Sydow, M. Nabelsee, S. Glesner, P. Herber, Towards profile-guided optimization for safe and efficient parallel stream processing in rust, in: 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2020, pp. 289–296, http://dx.doi.org/10.1109/SBAC-PAD49847.2020.00047.

[24] L. Rinaldi, M. Torquati, D.D. Sensi, G. Mencagli, M. Danelutto, Improving the performance of actors on multi-cores with parallel patterns, Int. J. Parallel Program. 48 (4) (2020) 692–712, http://dx.doi.org/10.1007/s10766-020-00663-1.

[25] Cody Casterline, Pipeliner: a rust library for pipelines, 2021, URL: https://docs.rs/pipeliner (Accessed on 27.03.2021).

[26] David King, Pipelines: A tool for pipelines, 2021, URL: https://docs.rs/pipelines (Accessed on 27.03.2021).

[27] Frank McSherry, Timely dataflow: A framework for managing and executing data-parallel dataflow computations, 2021, URL: https://docs.rs/timely (Accessed on 27.03.2021).

[28] D.G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, M. Abadi, Naiad: A timely dataflow system, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, in: SOSP '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 439–455, http://dx.doi.org/10.1145/2517349.2522738.

[29] I. Materialize, Materialize: a streaming database for real-time applications, 2021, URL: https://github.com/MaterializeInc/materialize (Accessed on 27.03.2021).

[30] J. Gjengset, M. Schwarzkopf, Noria: a streaming data-flow system, 2021, URL: https://docs.rs/noria (Accessed on 27.03.2021).

[31] J. Gjengset, M. Schwarzkopf, J. Behrens, L.T. Araújo, M. Ek, E. Kohler, M.F. Kaashoek, R. Morris, Noria: Dynamic, partially-stateful data-flow for high-performance web applications, in: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, in: OSDI'18, USENIX Association, USA, 2018, pp. 213–231.

[32] D. Griebler, R.B. Hoffmann, M. Danelutto, L.G. Fernandes, Stream parallelism with ordered data constraints on multi-core systems, J. Supercomput. 75 (8) (2018) 4042–4061, http://dx.doi.org/10.1007/s11227-018-2482-7.

[33] C. Bienia, S. Kumar, J.P. Singh, K. Li, The PARSEC benchmark suite: Characterization and architectural implications, in: 17th International Conference on Parallel Architectures and Compilation Techniques, in: PACT '08, ACM, Toronto, Ontario, Canada, 2008, pp. 72–81.

[34] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, SIGARCH Comput. Archit. News 23 (2) (1995) 24–36.

[35] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), in: IISWC '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 44–54.

[36] M.V. Bordin, D. Griebler, G. Mencagli, C.F.R. Geyer, L.G. Fernandes, Dspbench: a suite of benchmark applications for distributed data stream processing systems, IEEE Access 8 (na) (2020) 222900–222917, http://dx.doi.org/10.1109/ACCESS.2020.3043948.

[37] 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, URL: https://gmap.pucrs.br/dalvan/papers/2014/CR_SEKE_2014.pdf.

[38] 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, http://dx.doi.org/10.1142/S0218194015710096.

Ricardo Pieper is a software engineer at Ubots Brasil. He received his M.Sc. Degree in Computer Science from Pontifical Catholic University of Rio Grande do Sul (PUCRS) in 2020. His thesis included stream parallelism using parallel patterns in cluster environments. His interests are: high-performance computing, parallelism, computer graphics and programming languages.

Júnior Löff is a M.Sc student in Computer Science at the Pontifical Catholic University of Rio Grande do Sul (PUCRS), and research member of the Parallel Applications Modeling Group (GMAP) at PUCRS. He received his B.Sc Degree in Computer Engineering from PUCRS in 2020. His research interests include: Parallel and distributed systems, high-performance applications modeling and hardware/software co-design.

Renato B. Hoffmann is a M.Sc student in Computer Science at the Pontifical Catholic University of Rio Grande do Sul (PUCRS), and research member of the Parallel Applications Modeling Group (GMAP) at PUCRS. He received his B.Sc Degree in Computer Engineering from PUCRS in 2020. His research interests include: High performance computing, parallel programming, parallel architectures, and high-performance algorithms.

Dalvan Griebler is an Associate Professor at the Pontifical Catholic University of Rio Grande do Sul (PUCRS). Also Associate Professor at Três de Maio Faculty (Setrem) and head of the Laboratory of Advanced Research on Cloud Computing (LARCC) at Setrem. 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; benchmarking; and cloud computing.

Luiz 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). 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.