Structured Stream Parallelism for Rust

Ricardo Pieper
School of Technology, PUCRS
Porto Alegre, Brazil
ricardo.pieper@edu.pucrs.br

Dalvan Griebler
School of Technology, PUCRS
Porto Alegre, Brazil
dalvan.griebler@edu.pucrs.br

Luiz Gustavo Fernandes
School of Technology, PUCRS
Porto Alegre, Brazil
luiz.fernandes@pucrs.br

ABSTRACT

Structured parallel programming has been studied and applied in several programming languages. This approach has proven to be suitable for abstracting low-level and architecture-dependent parallelism implementations. Our goal is to provide a structured and high-level library for the Rust language, targeting parallel stream processing applications for multi-core servers. Rust is an emerging programming language that has been developed by Mozilla Research group, focusing on performance, memory safety, and thread-safety. However, it lacks parallel programming abstractions, especially for stream processing applications. This paper contributes to a new API based on the structured parallel programming approach to simplify parallel software developing. Our experiments highlight that our solution provides higher-level parallel programming abstractions for stream processing applications in Rust. We also show that the throughput and speedup are comparable to the state-of-the-art for certain workloads.

KEYWORDS

Programming Language, Parallel Programming, Multi-core, Stream Processing, Parallel Patterns, Structured Parallelism

ACM Reference Format:
Ricardo Pieper, Dalvan Griebler, and Luiz Gustavo Fernandes. 2019. Structured Stream Parallelism for Rust. In *XXIII Brazilian Symposium on Programming Languages (SBLP 2019), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3355378.3355384

1 INTRODUCTION

In the latest years, the number of CPU cores increased significantly [10]. The need for more processing power also brought new architectures to the market (e.g. GPUs, TPUs). This wealth of options brought a challenge for application programmers. Consequently, it is not trivial to create high-level abstractions and still provide competitive performance. Many APIs for parallel programming are not following a structured approach, requiring a deep understanding of the underlying operating system and architecture. This is a challenge for separation of concerns and code maintainability. Programmers have to handle task distribution, synchronization and write optimized code for different architectures.

The structured parallel programming approach describes design patterns and algorithmic skeletons to separate the concerns between application and system programmers [2, 10]. In this approach, application programmers are those that use high-level abstractions. McCool [10] describes patterns such as Map and Reduce, which allows the application programmer to implement data parallelism.

Stream parallelism imposes additional challenges since data is often produced continuously and irregularly. Stream parallelism also comprises a substantial fraction of today's workloads [5, 6]. A common abstraction for stream parallelism is the Pipeline pattern, where the computation is structured in a sequence of stages that may run independently and in parallel. For C++, the application programmer may use libraries such as Thread Building Blocks [15] and FastFlow [1]. Using these libraries, programmers still have to significantly refactor the existing application code. This problem is addressed by SPar [4], in which C++11 annotations can be used to implement parallelism with minimal code rewriting. However, C++ is a language in which memory must be manually allocated and deallocated, and threads can mutate shared states. This error-prone approach may cause bugs such as memory leaks, use-after-free, and others.

In contrast, Rust [12] is a language that implements memory safety without garbage collection, and is data-race free [8]. This allows system programmers to explore parallelism safely by relying on the language semantics and compiler checking. APIs for task and data parallelism have been developed for Rust, such as Tokio [18] and Rayon [14]. However, abstractions for stream parallelism in Rust are still lacking.

In this paper, our goal is to provide a solution where programmers can quickly implement structured stream parallelism in Rust without the need to understand low-level details related to the operating system and computer architecture. Therefore, we present the following contributions:

SBLP 2019, September 23–27, 2019, Salvador, Brazil
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7638-9/19/09…$15.00
https://doi.org/10.1145/3355378.3355384

- A new API for Rust that supports structured stream parallelism.
- A runtime system supporting linear and non-linear pipeline pattern implementation.
- Experiments assessing performance and discussing programming abstractions with respect to the state-of-the-art libraries.

We structured this paper as follows. In Section 2 we present the background of this work, including an overview of the Rust language and the Structured Parallel Programming approach. Section 3 highlights the related works, describing available libraries for parallel programming in Rust. Section 4 introduces our approach for structured stream parallelism in Rust, including a description of the API and the underlying runtime. Finally, Section 5 discusses the results of performance and API usage experiments and Section 6 concludes this paper.

2 BACKGROUND

In this section, we present an overview of the Rust language. Additionally, we also present the Structured Parallel Programming approach.

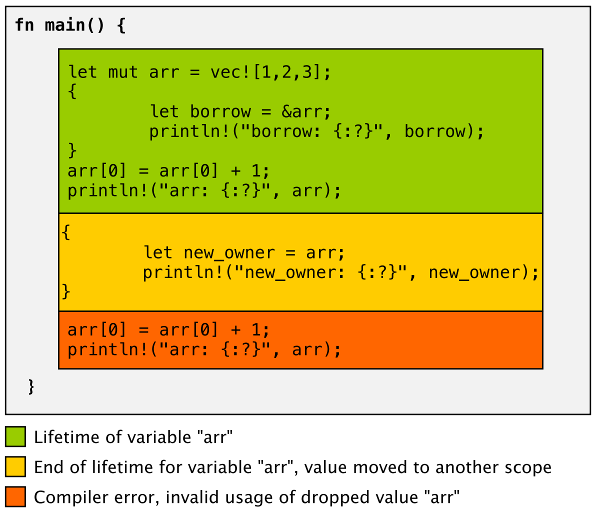
The Rust Programming Language

Rust [12] is a general-purpose, statically typed programming language designed for reliability, performance, and ergonomics. It uses LLVM for machine code generation and optimization. Rust supports a large number of platforms and can be used in browsers, servers, and embedded devices. Rust is being used in key parts of the Mozilla Firefox browser and is being maintained by Mozilla Research [9]. The language has common elements found in other languages, such as `struct`, `traits` (similar to interfaces in Java), lambda functions, and modules. Since Rust is relatively new, we provide an overview of the language and its features.

The main feature of Rust is its ownership and borrowing system [11]. This feature allows for automatic memory management without a garbage collector. C++ developers may be familiar with RAII (Resource Acquisition Is Initialization), a pattern in which resources are deallocated at the end of their scope. This helps prevent many classes of errors arising from incorrect usage of memory, (e.g. use-after-free, double-free, memory leaks). However, the programmer is responsible for implementing RAII correctly, and any potential errors are not caught by the compiler. In contrast, RAII is enforced by Rust's ownership system at compile time.

Every value in Rust has a type, a mutability modifier, an owner, and a lifetime. The owner is often the scope in which the value was declared, but ownership of the value can change by passing a value to other functions. A lifetime begins when the value is created and ends when the value is dropped. The compiler tracks lifetimes of every reference (borrow) to ensure all references are valid (*i.e.*, the value could not possibly be dropped before using the reference, preventing a use-after-free error), and lifetimes are tied to the scope in which a value exists.

In Figure 1, we provide a visualization of Rust's ownership and borrowing system in action. The code creates an array, creates a reference (borrow), and mutates the array while still being its owner. The ownership of the array is then transferred to another scope. This causes the code to fail compilation due to invalid usage of a dropped value after the block containing the `new_owner` variable.



```
fn main() {
    let mut arr = vec![1,2,3];
    {
            let borrow = &arr;
            println!("borrow: {:?}", borrow);
    }
    arr[0] = arr[0] + 1;
    println!("arr: {:?}", arr);
    {
            let new_owner = arr;
            println!("new_owner: {:?}", new_owner);
    }
    arr[0] = arr[0] + 1;
    println!("arr: {:?}", arr);
}
```

🟩 Lifetime of variable "arr"
🟨 End of lifetime for variable "arr", value moved to another scope
🟧 Compiler error, invalid usage of dropped value "arr"

Figure 1: Ownership and borrowing

In Rust, a value is borrowed when a reference to a value is created. Passing a reference to a function or variable does not transfer ownership of the original value. A value can be borrowed mutably only once at a time or immutably multiple times (with no mutable borrows). This feature avoids data races when values are passed to threads by disallowing multiple mutable borrows or simultaneous mutable and immutable borrows.

The compiler only accepts programs that follow Rust's rules for ownership and borrowing. However, due to the limitations inherent to static checking [11], some valid programs may cause the compilation to fail. To circumvent these limitations, the language provides pointer types to perform borrow checking at runtime, such as reference-counted pointers (atomic and non-atomic), and cells.

Rust can also be used for cases where low-level pointer operations are needed. This is called Unsafe Rust [13]. In Unsafe Rust, the same ownership and borrowing rules apply, but programmers can also *dereference* raw pointers, call unsafe functions, mutate static variables, and implement unsafe traits. Programmers must explicitly mark unsafe code with the `unsafe` keyword. This feature allows using Rust's memory safety guarantees while allowing unsafe operations in a limited context.

Rust also implements a macro system. The feature is extensively used by the standard library itself. Macros are processed after the AST is constructed, making it a powerful tool to construct abstractions. Differently, in C++, macros (preprocessor directives) are processed before building the AST. In this work, we use macros to simplify the construction of a parallel pipeline, which is a structured parallel programming pattern that will be introduced in Section 2.

Structured Parallel Programming

Sequential code can grow in performance by leveraging improvements in clock rate and ILP (instruction-level parallelism). However, there are practical limitations for improvement in these areas, and these problems are collectively known as the *three walls*: the power wall, ILP (instruction-level parallelism) wall, and the memory wall. Additionally, automatic parallelization can be done only sometimes, as language semantics often introduce limitations on automatic discovery of parallelism opportunities.

Furthermore, parallel programming is often a difficult task, as explained by McCool et al. [10] and Cole [2]. Programmers must deal with task distribution and synchronization. Incorrect implementations of parallelism may also introduce deadlocks and data races. 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, leading to non-deterministic behavior. Such programs may become complex and hard to maintain.

In contrast, sequential programs are simpler to reason about since serial code is often deterministic, always performing the same operations in the same order. Authors of [2, 10] proposed pattern-based methods to bring some characteristics of serial programming into parallel programming. McCool et al. [10] 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, if the trend of core count growth continues.

Cole [2] first introduces "algorithmic skeletons". They are considered high-level abstractions for parallel programming and generalized as parallel patterns today. Additionally, McCool et al. [10] describe a set of parallel patterns with more detail along with the structured parallel programming approach, which makes the use of composable patterns to exploit parallelism in an efficient and platform-independent way. Structured parallel programming is usually discussed for *data parallelism*, which is the subdivision of a problem into multiple subproblems that can be solved in parallel. Examples are Map and Reduce patterns. Map is a pattern where a function is applied in parallel over a list of independent items. Reduce combines all items of a list into a single item using an associative function.

There are also patterns for *stream parallelism*, which are used to exploit parallelism over continuous data flow processing. Examples are Farm and Pipeline patterns. Farm has a scheduler called emitter, worker replicas, and a collector that gathers results from the worker replicas. Pipeline is built with a sequence of stages that perform as an assembly line, where each stage processes a different task/item in parallel. Stateless stages can be easily replicated to build a nonlinear pipeline composition. All different patterns can be arbitrarily nested and composed in order to extract the maximum parallelism in the underlying hardware combining either stream or data parallel patterns [3].

In the literature, many frameworks, languages, and libraries have been created for the structured parallel programming approach. For the stream parallelism domain, programmers may use libraries such as Intel TBB or FastFlow, which implement efficient parallel patterns. Higher-level approaches are DSLs like SPar [4] and StreamIt[17]. With SPar, programmers only need to annotate sequential code by using a set of C++11 attributes. The SPar compiler automatically generates parallel code with calls to FastFlow library targeting shared-memory multicore architectures. As in C++, our challenge is to allow structured parallel programming for stream parallelism on top of the Rust Standard Library, which has primitives such as threads, mutexes, and atomics. Additionally, it provides MPSC (Multiple Producer, Single Consumer) queues that can be used by multiple threads.

3 RELATED WORK

Since Rust is a relatively new language (first stable release dating back to May 2015), there are few works on parallelism abstractions. In this section, we will review some of the available tools for parallel programming in Rust, including tools that do not fit in the structured parallel programming model/approach.

Sydow et al. [16] implemented a graphical programming model and runtime for safe stream parallelism based on the *dataflow* model. The tool generates parallel skeletons of Rust code based on a graphical representation of the program. The tool is coupled with the IntelliJ IDEA IDE. The authors of [16] focused on a visual overview of the code, while our work focuses on coding for programming languages without coupling to a specific IDE.

The *de-facto* library for data parallelism in Rust is Rayon [14]. Rayon includes common data parallelism patterns such as Map and Reduce, and the runtime implements a work-stealing thread pool. The level of parallelism can be controlled by configuring the global thread pool size, or by instantiating a new thread pool. Rayon implements *parallel iterators* that extend the sequential iterators on the standard library. This approach allows programmers to easily parallelize existing sequential code written using Rust iterators. However, Rayon does not support parallelism for iterators with unknown size, and pipelines for iterators with a known size can be imitated by implementing chained *map* functions. In our work, we focus on stream parallelism, but we include Rayon in our comparison tests due to its ubiquity on the Rust ecosystem.

Tokio [18] is a Rust library for asynchronous programming. Tokio is entirely based on *futures*, which represent abstract units of work that might complete in the future, and they execute in a work-stealing thread pool. Tokio provides a different approach for stream parallelism, called *asynchronous streams*. The API provides a `Stream` trait, which is analogous to a stage of a pipeline in a structured parallel programming approach. `Stream` implementations can have mutable state and the pipeline can be implemented by linearly connecting streams together. For stream parallelism, the programmer needs to manually spawn futures in each stage and create a `channel` to return a result. Tokio can not be considered as a structured parallel programming approach due to the low-level parallelism control (futures and channels). We built a parallel pipeline in Tokio using asynchronous streams in Section 5 for performance evaluation.

In Table 1, we compare the main feature set of each library. The column *Ours* refers to our work, which focuses on structured stream parallelism. For Tokio, we mark Stream Parallelism and Data Parallelism with "Yes" because the user can build it from scratch by using futures and channels. Although Tokio provides fewer abstractions, it is flexible enough to support all types of parallelism. Moreover, Tokio and Rayon implement a work-stealing scheduler. Work-stealing is very good for task and data parallelism. However, it may present performance drawbacks for stream processing since threads are not dedicated to each stage of the pipeline, which may cause latency problems [3]. Work-stealing implementation can help to improve resource utilization but not necessarily improve performance. When analyzing the available parallel programming options for Rust, we observed that none of them offer structured stream parallelism for ready-to-use in parallel programming.

4 STREAM PARALLELISM FOR RUST

Rust offers several features that put it as a candidate to be used for HPC applications in the next years. Some of them are memory safety, thread-safety, and LLVM optimizations. Therefore, we designed a high-level programming library based on structured stream parallelism for Rust, which allows users to quickly express parallel stream operators through the pipeline pattern using few building blocks. This section will discuss the design principles, API of the library, and runtime system.

Table 1: A comparison of Rust parallel programming APIs.

Characteristics	Tokio	Rayon	Ours
Stream Parallelism	Yes*	No	Yes
Data Parallelism	Yes*	Yes	Yes
Task Parallelism	Yes	No	No
Work-stealing	Yes	Yes	No

Design Principles

Our programming interface was designed based on TBB (Thread Building Blocks) [15], *FastFlow* [1], and SPar [4] because they follow the structured parallel programming approach as well. In TBB, each stage of the pipeline is called a *filter*, which are classes that extend the `tbb::filter` class. Differently, FastFlow provides the `ff_node_t` generic class that can be used as building blocks to instantiate Farm or Pipeline patterns. SPar provides code annotations to mark stream parallelism regions, stage, input/output dependencies, and stage replication. We want to be as close as possible to the abstraction level provided by SPar with the implementation principles (library) of TBB and FastFlow. Therefore, our design principles are summarized as follows:

- Avoid the use of Unsafe Rust in the API and the runtime.
- Do not expose synchronization, task scheduling or any system related programming to application programmers.
- Application programmers must be able to specify the end of the stream for bounded streams and choose whether a stage receives items in the order that they were initially produced.

Our design focuses on type-safety and thread-safety. Each stage in the pipeline indicates the data type of their inputs and outputs. At compile-time, during macro expansion, the links between stages are established, and any type errors are caught by the compiler during type-checking. Additionally, we use Rust's thread-safety guarantees to prevent data races by having each replica of a parallel stage contain its own isolated state. A replica cannot access the state of another replica. If this becomes necessary, the programmer will be forced by the compiler to use synchronization primitives. This approach allows the programmer to have a limited form of mutable state. The following steps might guide application programmers to introduce stream parallelism on existing sequential code using our library:

(1) Identify code regions that could execute in parallel.
(2) Isolate these regions inside functions and identify their inputs and outputs.
(3) Wrap the functions using the API to build the pipeline
(4) Identify the primary input of data.
(5) Determine whether the program should wait until all items are processed.

Rust-SSP Programming Interface

In this section we describe our API (public available [1]), which has the following elements:

- A macro called `pipeline!` to setup a pipeline.
- `parallel!`, `sequential!` and `sequential_ordered!` macros to define stages in the pipeline.
- `post`, `collect` and `end_and_wait` methods to control pipeline execution.
- `In` and `InOut` trait objects to define the code executed by the stage.

In Listing 1 we implemented a pipeline that resizes a stream of images and waits until all images are processed. A pipeline is composed of a `Pipeline` object and one or more stages (`LoadImage`, `Resize` and `SaveToDisk`). Each stage is defined using a macro. The code of a stage can be defined using one of the following methods:

- A function name or a lambda function for stateless stages.
- A `struct` that implements `InOut` or `In` traits for stateful and stateless stages.

The example shown in Listing 1 uses `struct` that implement `InOut` and `In` traits. These traits are the building blocks of the API, which are wrappers for sequential code. The `InOut` trait declares a `process` function that takes a value from the stream and returns a new value, as shown in Listing 2. The `In` trait declares a `process` function that takes a value from the stream, but does not return a result. The code for `In` has been omitted for brevity, but it is very similar to `InOut`. Both trait objects allow the user to mutate internal state.

To define a stage, three macros are available:

- `parallel!(stage, threads)`: defines a stage that can be replicated for parallelism.
- `sequential!(stage)`: defines a stage that runs in a single thread.
- `sequential_ordered!(stage)`: defines a stage that runs in a single thread, receiving items in the order they were initially produced.

The last stage in the pipeline must be a `sequential!` or `sequential_ordered!` stage that receives an `In` trait object. Listing 1 shows the use of the `post` method, which is used to send items to the pipeline. In this example, we are interested in collecting the results after processing all images in a directory (bounded stream computation). The results can be collected by calling the `collect` method. Also, it encompasses the "end" signal to notify the alive threads waiting for items in the shared queues. When a thread process the "end" signal, it stops producing items to the next steps. This signal is propagated to the other threads and other steps, which eventually causes all threads to finish. 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. In the case of an infinite stream where the programmer uses an infinite loop, the threads will never finish, and there is no need to signal the end of the stream.

Listing 1: Pipeline example: resizing images.

```rust
let threads = num_cpus::get();
let pipeline = pipeline![
  parallel!(LoadImage::new(), threads),
  parallel!(Resize::new(), threads),
  parallel!(SaveToDisk::new(), threads),
  sequential!(Collector::new())];

let image_paths = load_from_dir();
for entry in image_paths {
  pipeline.post(entry).unwrap();
}

let result = pipeline.collect();
```

Listing 2: Creating an `InOut` stage.

```rust
struct LoadImage;
impl InOut<PathBuf, Image> for LoadImage {
  fn process(&mut self, input: PathBuf) -> Image {
      //code for loading image omitted
    }
}
```

Rust-SSP Runtime

In this section, 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. As described in Section 4, the building blocks of our API include `In` and `InOut` trait objects. Internally, these trait objects become a `PipelineBlock`. A `PipelineBlock` is a Rust trait that requires its implementations to provide a `process` method. There are 2 implementations: `InOutBlock` and `InBlock`. These traits and implementations are described in Figure 2. Some details of these structures have been omitted for simplicity.

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

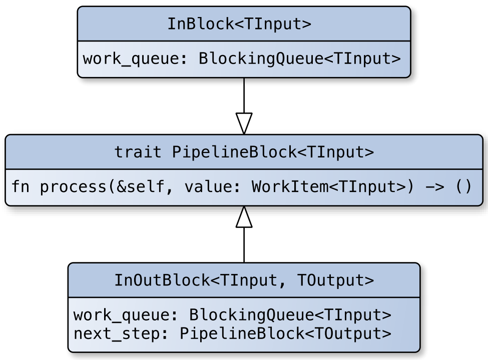

InBlock<TInput>
work_queue: BlockingQueue<TInput>
trait PipelineBlock<TInput>
fn process(&self, value: WorkItem<TInput>) -> ()
InOutBlock<TInput, TOutput>
work_queue: BlockingQueue<TInput>
next_step: PipelineBlock<TOutput>

Figure 2: Pipeline block implementations.

All `InOutBlock` objects own the object of the next block, which are implementations of `PipelineBlock`. The implementation of `process` in `InOutBlock` and `InBlock` simply enqueue the `WorkItem` in the `work_queue`, which is a FIFO queue. `InOutBlock` executes the stage code and passes the result to the next stage (`next_step`) by calling the `process` method. `InBlock` only executes the stage code, as it represents the end of the pipeline. By using generic types, Rust can prevent type errors at compile-time, since the input type of `next_step` of `InOutBlock` needs to match the output type.

The threads that read items from the `work_queue` are spawned by the stage definition macros. The macros generate the necessary code to transform the external API stage objects into internal pipeline blocks, spawns all the necessary threads and register them for joining if the programmer needs to wait for the end of stream processing.

The stage code is defined using Rust traits. In Rust, `struct` fields and function parameters must have a known size at compile time. Traits have unknown size, and cannot be stored in structures or declared as function parameters. Rust requires the programmer to "box" the value to store it in the heap, which can be done with the Box API. The programmer would have to manually create the box, and transfer ownership of the data to the box. Instead, we use macros to generate the necessary boxing code. Therefore, application programmers can simply pass the trait object without manual boxing. Another way to solve this problem would be to store a reference to a trait object. However, the programmer needs to specify lifetimes when storing a reference in a Rust `struct`, which makes the library code more complex. In languages like C++, these limitations do not exist since abstract class objects and references can be freely passed to methods and stored in class fields. However, the lifetime and memory allocation of these objects must be carefully thought by the programmer. In contrast, Rust is more strict regarding passing abstract objects to methods and *structs*. Overcoming these restrictions can be challenging, and Rust-SSP overcomes this by using macros.

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 macros allow the stages to be defined in the intuitive order. Although the reverse linking could be performed by a function instead of a macro, we would face the aforementioned problem with unknown-size arguments, requiring the user to create a Box object before passing it into the function.

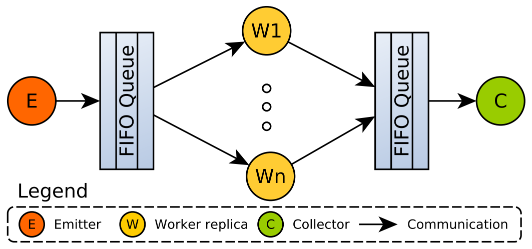


Figure 3: Flow of data in the runtime.

A general visualization of the runtime is shown in Figure 3, representing an abstract application that follows the well-known Farm (with emitter, worker, and collector) pattern. In order to provide parallelism, we store all work items in a double-ended queue protected by a Mutex. 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 store items in an ordered set, where the key is the tag of the item. The tagging process is done by the `post` method, and the tag value is assigned by an internal counter that increments each time the `post` method is called. Work items are then dequeued in order, blocking if the next item is not in the queue yet. This ordering algorithm was based on [7]. Section 5 evaluates the performance of our implementations.

5 RESULTS

In this section, we present a performance evaluation and discuss programming abstractions. Our library was named as SSP-Rust. All tests were run in a machine with 2 Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz (12 cores, 24 threads) with 32GB of RAM memory. The operating system used was Ubuntu 16.04 64 bits with kernel 4.4.0-146-generic. The code was compiled using Rust 1.34.1 using the `--release` flag.

The tests involved 2 applications:

(1) Calculate a 1000x1000 Mandelbrot Set.
(2) Fetch a stream of images and apply a sequence of 6 filters.

For the Mandelbrot Set, we calculate multiple lines of the image in parallel, resulting in 1000 work items. This test shows that SSP-Rust can also be used for simple data parallelism workloads. We also test Tokio and Rayon on the same workload. For the Image Filter test, we applied 6 filters in each image following this order: Saturation, Emboss, Gamma Correction, Sharpen, Grayscale, and Resize to 500px width. Some filters may take longer than others to complete, and the images have different sizes. This workload will test how the system behaves when the load is not balanced. For this test, we compared SSP-Rust and Tokio. We do not compare with Rayon because it does not support stream parallelism.

We ran each benchmark 5 times to take the average. The standard deviation was less than one percent. Take into account that some comparisons are unfair: Rayon and Tokio have different runtimes and implement work-stealing thread pools which are expected to use resources more efficiently. For this version of Rust-SSP, we implemented shared queues controlled with *mutexes* to communicate between stages. We do not implement any optimization for lock contention and cache. Therefore, we expect that Rayon and Tokio perform better than Rust-SSP.

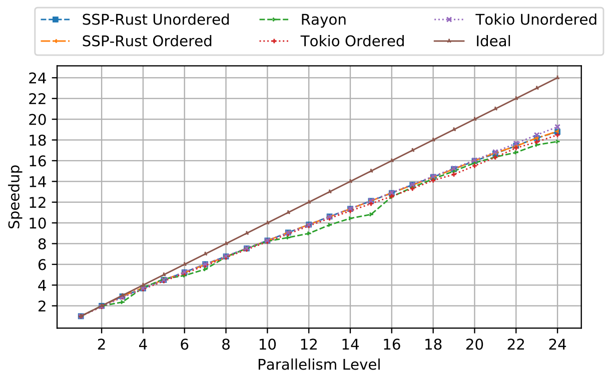


Figure 4: Mandelbrot with Speedup Comparison.

Figure 4 shows a speedup graph for the Mandelbrot Set benchmark. For Tokio and SSP-Rust, we also test the ordered and unordered options. We observed similar performance across all parallelism levels. Rayon varies more across all tests. This is a behavior that we attribute to the implementation of its work-stealing scheduler. Tokio and SSP-Rust show less variation and similar performance.

In Figure 5, we show two graphs displaying the throughput of Tokio and SSP-Rust. Both libraries show approximately the same peak throughput and a similar curve. For Tokio, we observed that by configuring only 1 pending future on each stage, there are no performance improvements. A speedup was only achieved when the stage is configured to work with 2 or more pending futures. Tokio did not exhibit the same behavior in the Mandelbrot Set benchmark because there is only 1 stage. It is difficult to analyze the behavior since the buffer size in Tokio does not necessarily mean the number of threads that are running.

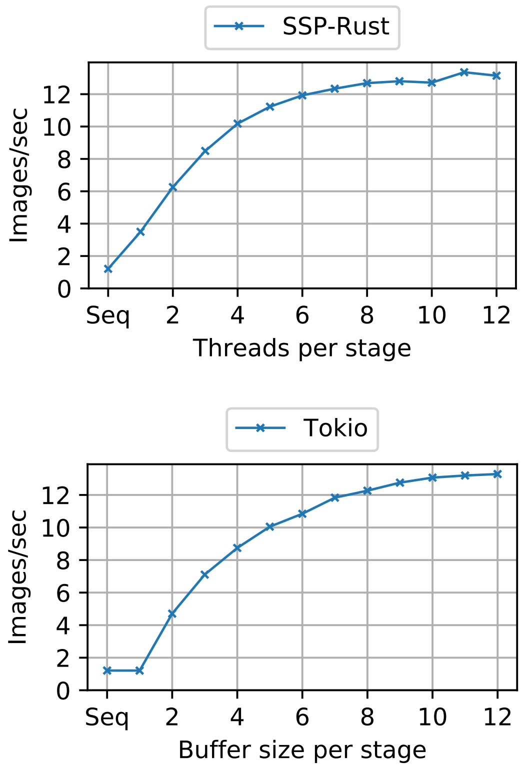


Figure 5: Image Processing Throughput Comparison.

It is worth to discuss also the abstractions when implementing the parallelism for the Mandelbrot Set benchmark in terms of source lines of code added beyond the sequential source code. Therefore, we measured the code intrusion. For Rayon, we measured 4 lines of code. Tokio requires 14 lines of code, and SSP-Rust takes 7 lines of code. The effort of parallelization in Rayon is very low since it follows Rust's Iterator API design in order to provide easy ways to work with collections of data. For existing single-threaded code that already uses the Iterator API, migrating to Rayon is trivial. For Tokio, most of the effort is in spawning a future and creating a `oneshot` channel to retrieve the results. This process takes approximately 7 lines of code.

The programming effort in SSP-Rust is larger than Rayon, and the code is shown in Listing 3. However, the code itself is simple because it only defines a pipeline, adds items to the stream, and waits for the stream processing termination.

Listing 3: SSP-Rust Mandelbrot Set.

```
let pipeline = pipeline![
  parallel!(calculate_line, threads),
  sequential!(RenderLine::new())];
for i in 0..lines {
  pipeline.post(i).unwrap();
}
let rendered_lines = pipeline.collect();
```

For the Image Processing benchmark, we used an external library called `raster` to apply a filter to each image. Each filter takes 2 lines of code. The Tokio implementation takes 68 lines of code, and SSP-Rust takes 30 lines. To use Tokio, it is necessary to use almost twice the number of lines of code to provide stream parallelism when compared to SSP-Rust. To alleviate this, we created a simple macro to perform the setup, reducing the line count to 46 lines, which is still 55% more lines of code than SSP-Rust.

6 CONCLUSIONS

This paper introduced a new parallel programming library for structured and safe stream parallelism. The API exposes only a few methods that provide important features in stream parallelism without calling Unsafe Rust methods. Our abstraction provides high-level and structured mechanisms, simplifying the parallel software development for stream processing applications. Although a fair comparison is hard to achieve due to the different design goals of the available tools, we provided similar performance and demonstrated simpler parallel programming abstractions for stream processing applications.

Besides concluding the important steps, we visualize future works. First, more benchmarks with real-world applications could be performed to obtain more data regarding performance and programming effort. Second, performance improvements could still be applied in the work queues by using non-blocking mechanisms. Third, support arbitrary nesting of data and stream parallel patterns. Fourth, support more advanced stream processing techniques, such as sliding window. Lastly, support for other architectures such as GPU could be implemented in the runtime.

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, the FAPERGS 01/2017-ARD project called ParaElastic (No. 17/2551-0000871-5), and the Universal MCTIC/CNPq No. 28/2018 project called SParCloud (No. 437693/2018-0). We also thank Laboratório de Alto Desempenho (LAD) from PUCRS for the computing resources.

REFERENCES

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2014. FastFlow: High-Level and Efficient Streaming on Multi-core. In *Programming Multi-core and Many-core Computing Systems (PDC)*, Vol. 1. John Wiley & Sons, 14.

[2] Murray I. Cole. 1989. *Algorithmic Skeletons: Structured Management of Parallel Computation.* University of Glasgow, Glasgow, United Kingdom.

[3] Dalvan Griebler. 2016. *Domain-Specific Language & Support Tool for High-Level Stream Parallelism.* Ph.D. Dissertation. Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil.

[4] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. 2017. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters* 27, 01 (2017), 1740005.

[5] Dalvan Griebler, Renato B H Filho, Marco Danelutto, and Luiz Gustavo Fernandes. 2017. Higher-Level Parallelism Abstractions for Video Applications with SPar. In *3rd International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms* (2017-09-01) *(RePara'17)*. IOS Press, Bologna, Italy.

[6] Dalvan Griebler, Renato B H Filho, Marco Danelutto, and Luiz Gustavo Fernandes. 2018. High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. *International Journal of Parallel Programming* (2018), 1–19. https://doi.org/10.1007/s10766-018-0558-x

[7] Dalvan Griebler, Renato B. Hoffmann, Marco Danelutto, and Luiz Gustavo Fernandes. 2018. Stream Parallelism with Ordered Data Constraints on Multi-Core Systems. *Journal of Supercomputing* 75 (July 2018), 1–20. https://doi.org/10.1007/s11227-018-2482-7

[8] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

[9] Mozilla Research Link Clark. 2019. *Inside a super fast CSS engine: Quantum CSS (aka Stylo).* Retrieved May 2, 2019 from https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo

[10] Michael McCool, Arch D. Robison, and James Reinders. 2012. *Structured Parallel Programming: Patterns for Efficient Computation.* Elsevier, Waltham, MA.

[11] Mozilla Research. 2019. *Rust Book.* Retrieved May 2, 2019 from https://doc.rust-lang.org/book/

[12] Mozilla Research. 2019. *Rust Programming Language.* Retrieved May 2, 2019 from https://www.rust-lang.org

[13] Mozilla Research. 2019. *The Rustonomicon.* Retrieved May 2, 2019 from https://doc.rust-lang.org/nomicon/

[14] Rayon. 2019. *Rayon.* Retrieved May 12, 2019 from https://github.com/rayon-rs/rayon

[15] James Reinders. 2007. *Intel Threading Building Blocks* (first ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[16] Stefan Sydow, Mohannad Nabelsee, Helge Parzyjegla, and Paula Herbe. 2018. A Safe and User-Friendly Graphical Programming Model for Parallel Stream Processing. *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2018), 239–243.

[17] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction.* Springer, Grenoble, France, 179–196.

[18] Tokio. 2019. *Tokio - The asynchronous runtime for the Rust programming language.* Retrieved May 12, 2019 from https://tokio.rs