An internal domain-specific language for expressing linear pipelines: a proof-of-concept with MPI in Rust

Leonardo Gibrowski Faé, Dalvan Griebler
leonardo.fae@edu.pucrs.br,dalvan.griebler@pucrs.br
School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil

Abstract

Parallel computation is necessary in order to process massive volumes of data in a timely manner. There are many parallel programming interfaces and environments, each with their own idiosyncrasies. This, alongside non-deterministic errors, make parallel programs notoriously challenging to write. Great effort has been put forth to make parallel programming for several environments easier. In this work, we propose a DSL for Rust, using the language's source-to-source transformation facilities, that allows for automatic code generation for distributed environments that support the Message Passing Interface (MPI). Our DSL simplifies MPI's quirks, allowing the programmer to focus almost exclusively on the computation at hand. Performance experiments show nearly or no runtime difference between our abstraction and manually written MPI code while resulting in less than half the lines of code. More elaborate code complexity metrics (Halstead) estimate from 4.5 to 14.7 times lower effort for expressing parallelism.

Keywords: Programming Languages, Parallel Programming, Stream Processing, Distributed Systems

1 Introduction

As the computing requirements to solve complex problems increase, programmers begin to seek parallelism to meet computation demands [29]. Parallelism may come in the form of multiple processes in the same machine (multi-threading), multiple processes distributed over a network of machines (distributed computing or cluster computing), or through the usage of specialized hardware (Graphical Processing Units - GPUs, or other accelerators). Writing parallel programs is notoriously tricky [4]. These three forms of parallelism involve radically different programming constructs and concerns. Patterns for structured parallel programming [5, 23] ameliorate that by presenting known-to-work parallel procedures with higher-level descriptions (pipeline, map-reduce, and so on). However, the programmer still must be aware of each platform's specific characteristics and programming models to be able to use them efficiently.

We have observed a significant research effort in the past years to provide high-level parallel programming abstractions, mainly in the C++ language community. Parallel patterns are instantiated via C++-based template libraries with classes and predefined functions. Examples of abstractions that support linear pipelines are TBB [25], FastFlow [1], and GrPPI [6], where only FastFlow and GrPPI support the distributed execution of linear pipelines. More higher-level abstractions are those that design internal Domain-Specific Languages (DSLs) such as SPar [14], which offers C++ annotations to express stream parallelism, mainly in the form of linear pipelines beyond other parallelism types[22]. It also provides a source-to-source compiler for automatic parallel code generation [13]. SPar annotation methodology preserves the sequential code structure, and the programmer may choose to execute it in parallel for distributed memory [15, 28], diverse shared memory runtime systems[19, 20, 22], or GPU [31]. SPar's approach demonstrated to reduce significantly programmability metrics with respect to template-based approaches such as FastFlow and TBB [2].

Rust is a relatively new and modern low-level programming language designed to be safe and performant [21]. Recent works have shown Rust's capabilities for high-performance computing (HPC) in shared memory environments [3, 26, 27, 30, 36] analogous to FastFlow and TBB programming abstractions. Thus, a research question was raised motivated by the programmability benefits shown in C++ [2]. Could SPar's annotation-based approach be extended to the Rust language environment? An initial effort was made in [8], by identically following the SPar's annotation methodology with automatic source-to-source transformations for shared-memory architecture targeting the Rust-SSP as the runtime system [27]. That research focused on understanding the challenges and mechanisms of Rust for source-to-source code transformation. Although inspired by SPar annotations, this work proposes a new annotation methodology and a new automatic source-to-source code generation for distributed memory architectures, targeting MPI as the runtime system. To the best of our knowledge, we are the first research initiative to provide higher-level abstractions in Rust for distributed HPC clusters. Therefore, our scientific contributions are the following:

- A new high-level programming abstraction for expressing linear pipelines in Rust with automatic source-to-source code generation. Although a proof-of-concept is made with MPI, this internal DSL and code generation technique can be used to target other runtime systems and parallel architectures.
- A new annotation-based method whose main ideas and approaches could be ported into other programming

languages with metaprogramming capabilities, to ease and simplify parallel programming.

- A source-to-source transformation strategy targeting MPI with procedural macros in Rust.
- A comparative analysis of programmability and performance using several benchmark implementations for distributed HPC clusters in Rust.

Section 2 presents the essential theoretical background to understand our research implementations. Section 3 contains the full explanation of how our programming abstraction works. To showcase how general its central ideas are, and for didactic purposes, we also include a shared memory, multithreaded implementation. To switch between the two, the end user potentially has to change just a single keyword. In Section 4, we show programmability and performance metrics that indicate our programming abstraction significantly reduces code intrusion effort while incurring minimal execution time overhead. We discuss related work in Section 5 and finalize with our conclusions in Section 6.

2 Background

Parallelism exploitation is necessary for computer scientists and engineers since computing problems are becoming more complex, and computer architectures are fully parallel, ranging from cell phones to personal computers and HPC clusters. Multithreading is ultimately limited by hardware; there are only so many independent executing threads one can design a CPU chip to run. As problems and requirements grow larger still, eventually one turns away from the multithreading architecture and reaches for parallelism without shared memory: a distributed system, typically communicating through message passing. Often, the Message Passing Interface (MPI) specification is used for programming in this environment.

MPI. MPI is a message-passing specification used in distributed environments to facilitate their parallel programming. Programming in MPI can be challenging, as it forces the developers to think about low-level communication details and manually implement all the details of the desired parallel pattern or algorithm [23]. A possible solution for this, as this work proposes, is to create higher-level abstractions around MPI that already implement the underlying patterns, allowing the programmers to only think about the specifics of the problem at hand without worrying about the details of the parallel execution model.

Linear Pipelines. A linear pipeline is a parallel pattern described by a directed acyclic graph that represents the flow of a computation [23]. Figure 1 shows an example of a linear pipeline. The pipeline pattern explores parallelism in two ways:

1. As Figure 1 shows, within the stages of a pipeline, we can replicate the vertices so that they operate on different units of work at the same time. For example, vertex 0 could split the problem into 5 distinct units of work and send each one of those to a different vertex with label 1. Those five vertices would then work at the same time, in parallel.
2. As the pipeline fills up, the vertices working at different stages correspond to multiple steps necessary to solve the problem being worked on at the same time. For instance, after the five 1-labeled vertices finish their work, they will send their results to five 2-labeled vertices. Then, the 0-labeled vertex can send more than 5 units of work to the 1-labeled vertices. After that, we would have 10 nodes in the graph working in parallel at the same time: five at stage 1, and five at stage 2.

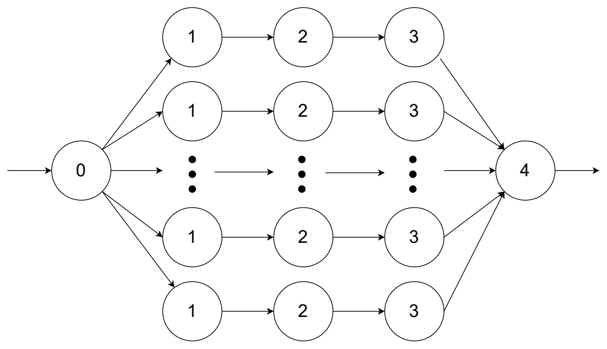


Figure 1. An example of a linear pipeline.

In a linear pipeline such as this, the nodes that begin and end the pipeline are often also called, respectively, *source* and *sink*.

Rust. Rust is a relatively new, low-level programming language with a focus on performance and safety. Since the use of parallelism in a distributed environment usually arises from a scalability issue, a performant low-level is a natural choice. Indeed, MPI applications are traditionally written in C/C++ because of this. Rust safety guarantees ensure that many categories of bugs in C/C++, particularly the ones related to undefined behavior, are impossible. Examples include use-after-free, buffer overflows, and null pointer dereferencing. Rust also offers more robust metaprogramming features than C/C++, with a macro system that gives the programmer limited access to the abstract syntax tree (AST), which they can use to perform arbitrary code transformations. This system can be used to implement custom syntax and Domain Specific Languages (DSL).

DSL. A DSL is a computer programming language of limited expressiveness, focused on a particular domain [10]. DSLs are often used in diverse and highly specialized domains because they allow for more personalized treatment of the problems common to that domain than general purposes languages [24]. In our case, we are proposing a small DSL for the domain of parallel programming with distributed systems. The DSL uses a very similar set of keywords to SPar [13, 14], a DSL originally made for C++, oriented towards stream processing. Because stream processing is usually modeled as a pipeline, adapting the DSL for the simpler case of a linear pipeline is straightforward. Section 3 explains this in further detail.

3 Proposed Abstraction

We start this section by formalizing a linear pipeline, which will then guide our implementation. We will begin by defining a series of interfaces complying with our formalization. Then, we will automatically generate parallel code that uses these interfaces. To this end, we make use of Rust's trait system and metaprogramming features, but that is ultimately just an artifact; the same underlying formalization could be used to implement the same ideas in other programming languages with metaprogramming support. The overall ideas aim to be so general that we have also created a multi-threaded implementation using the same interfaces and function transformations. Many of the code examples given are for the multi-threaded version because they are cleaner and easier to understand. They are followed by a brief explanation of what changes with regard to the one targeted for MPI. The DSL's programming interface design is internal since we aim to compile it with the host language and cause minimal interference in the source code.

3.1 Linear Pipeline Formalization

Let the graph G represent a linear pipeline with V vertices and E edges. Each vertex corresponds to a unit of computation, while each edge represents the communication between them. V contains two special vertices, V_0 and V_f, which stand for the sink and the source, respectively (the f in V_f stands for "final"). The operations performed by the sink and the source are unique, meaning they cannot be duplicated in any other vertex in V. Any other computation performed by any other vertex may be duplicated indefinitely. We represent this by annotating the vertices with the same number, as we have done in Figure 1. We may also say that vertices with the same number belong to the same *stage* of the pipeline.

Because the final goal is to implement this in a statically typed programming language, let us consider what happens in the pipeline in terms of *datatypes.* Since the edges represent simple communication between independent processes, no edge e can change the underlying type of the data they are transferring. On the other hand, because the vertices stand for arbitrary computation, they *can* change the underlying datatypes. Specifically:

Remark 1. *All vertices labeled with the same number consume the same type of input and produce the same kind of output.*

This puts constraints on how edges and vertices can be connected to build the linear pipeline. All edges that lead to a vertex must be transferring data of the same type. Similarly, for all edges leading *from* a vertex. More formally:

Remark 2. *Let (v_1, v_2) represent an edge that starts at $v1$ and ends in $v2$, and v_x is an arbitrary vertex. Given a vertex $v \in V$, all edges (v, v_x) must transfer the datatype v_x expected as input, and all edges (v_x, v) must transfer the datatype v_x generated as output.*

Remark 3. *Because edges do not transform data, Remark 2 implies that, for every edge (v_1, v_2), v_1's output must be of the same type as v_2's input.*

Given the internal computations of V, Remarks 1-3 are sufficient to fully specify the types of every element of G. To see why, consider a more straightforward pipeline, without any replication. It would be equivalent to just using the top nodes in Figure 1. Once the programmer has specified which computations will occur in each stage, with their inputs and outputs, the type of every edge is forced by Remark 2. If one stage's input does not correspond to its previous stage's output (in other words, if Remark 3 is false), that is a detectable type error made by the developer. To transform the simplified pipeline into the one in Figure 1, we can simply create more vertices at any stage between the source and the sink that will execute the exact same code as the other vertices with the same label, and fill in the edges as necessary. Remark 1 guarantees that if we have a valid implementation for one vertex, we can simply replicate it for all vertices in the same stage, and the pipeline will still be valid.

3.2 Trait based implementation

Using what we have presented in Section 3.1, we can create Rust traits representing all mentioned constraints. A *trait* in Rust is similar to an interface in other languages: it defines a series of functions that all data types that implement the trait must have. We can then use the trait as parameters to generic functions, which the compiler will monomorphize during compilation — essentially, creates a different version of the function for every necessary type. We will present an overview of the several traits implemented that conform to our formal specifications. For the sake of clarity, they will be slightly simplified from our actual implementation, though they will retain the general idea. Furthermore, because the MPI implementation is too verbose to transcribe in this paper, we give examples of the multi-threaded implementation and then explain the differences between the two.

3.2.1 Edges. To represent an edges' ability to send and receive data, we have the `Sender` and `Receiver` traits (Listing 1). Note that the type `T` that `Sender` and `Receiver` communicate, as well as `Sender` and `Receiver` themselves, must all implement the `Send` trait. `Send` is a marker trait that indicates these types can be safely sent across thread boundaries, which will be necessary for the multi-threaded implementation. Now, one needs only implement these traits for a given data type. As an example, the multi-threaded implementation can be found in Listing 2.

```
pub trait Sender<T: Send>: Send + Clone {
    type Error: std::fmt::Debug;
    fn send(&mut self, elem: T) -> SendResult<Self::Error>;
}
pub trait Receiver<T: Send>: Send {
    type Error: std::fmt::Debug;
    fn recv(&mut self) -> ReceiveResult<T, Self::Error>;
}
```

Listing 1. `Sender` and `Receiver` traits

```
pub struct MtSender<T: Send> { sender: mpsc::Sender<T> }
impl<T: Send> Sender<T> for MtSender<T> {
    type Error = ();
    fn send(&mut self, elem: T) -> SendResult<Self::Error> {
        match self.sender.send(elem) {
            Ok(()) => SendResult::Ok,
            Err(_) => SendResult::End,
        }
    }
}
pub struct MtReceiver<T: Send> { receiver: mpsc::Receiver<T> }
impl<T: Send> Receiver<T> for MtReceiver<T> {
    type Error = ();
    fn recv(&mut self) -> ReceiveResult<T, Self::Error> {
        match self.receiver.recv() {
            Ok(elem) => ReceiveResult::Ok(elem),
            Err(_) => ReceiveResult::End,
        }
    }
}
```

Listing 2. Multi-threaded `Sender` and `Receiver`

The MPI implementation is more involved. Since we need to send values through the network, we must serialize them somehow. To do this, we have made use of the `serde` and `bytecode` libraries. This means that, to use the MPI versions of these functions, type `T` must also implement the `Serialize` and `Deserialize` traits from `serde`. Then, our implementation sends an unsigned 64-bit integer indicating the length of the serialized data, followed by the data itself. A length of 0 means the channel has closed.

3.2.2 Workers.

Now that we can communicate, we must perform the actual computations. We will define, in Listing 3, a `Worker` as a data type that has a `Receiver` and a `Sender` of independent types (since, as discussed, nodes can perform arbitrary code transformations).

It is important to note that this trait specification does not include the actual computation performed by the worker as part of its definition. Instead, `run` accepts a function (or lambda) declared elsewhere, which will take the `Worker` as an argument. This function then would have to use the worker's `sender` and `receiver` methods to communicate with the rest of the pipeline.

```
pub trait Worker<In: Send, Out: Send>: Sized {
    type R: Receiver<In>;
    type S: Sender<Out> + 'static;
    /// returns this worker's sender
    fn sender(&self) -> &Self::S;
    /// returns this worker's receiver
    fn receiver(&mut self) -> &mut Self::R;
    /// execute the worker's code. Note we accept a function that
    /// is defined elsewhere as an argument.
    fn run<F: FnOnce(Self) + Send + 'static>(self, f: F);
}
```

Listing 3. `Worker` trait

Furthermore, the specification does not include a way of replicating workers. This is because replication of independent execution units is highly dependent on the environment we are developing for. For instance, multi-threaded applications can simply fire new threads anywhere within their code, while all units of execution for an MPI environment are created all at once, globally, when the environment is initialized.

3.2.3 Replicating Workers.

To replicate workers for the multi-threaded case, as mentioned, we can simply fire a new thread. The multi-threaded `Worker` implementation can be found in Listing 4.

```
pub struct MtWorker<In: Send, Out: Send> {
    thread_pool: ThreadPool, // any implementation will do
    receiver: MtReceiver<In>,
    sender: MtSender<Out>,
}
impl<In: Send + 'static, Out: Send + 'static> Worker<In, Out>
    for MtWorker<In, Out> {
    type R = MtReceiver<In>;
    type S = MtSender<Out>;
    /// runs the code in a separate thread
    fn run<F: FnOnce(Self) + Send + 'static>(self, f: F) {
        std::thread::spawn(move || f(self));
    }
    /// spawns a new unit of execution
    fn spawn<F: FnOnce() + Send + 'static>(&self, f: F) {
        self.thread_pool.spawn(f);
    }
    fn sender(&self) -> &Self::S { &self.sender }
    fn receiver(&mut self) -> &mut Self::R { &mut self.receiver }
}
```

Listing 4. Multi-threaded `Worker`

There are only two traits missing: `Source` and `Sink`. `Source` is the same as `Worker`, but does not have a `Receiver`. `Sink`, on the other hand, has just a method to turn itself into a `Receiver`, so we can call `recv` on it to retrieve the data that is exiting the pipeline.

With this, the multi-threaded implementation is complete. To create a linear pipeline, one must simply call and concatenate the implementations together. In order to facilitate this, we created a declarative macro called `to_stream`. A declarative macro in Rust is a macro declared inline (later, we will see that procedural macros are defined in a separate program instead) that performs text substitution. Let F be a set of functions, $f_0, f_1, f_2, ..., f_i$, where f_0 and f_i are the source and sink logic, respectively. f_0 and f_i must accept as their first arguments a type that implements `Source` and `Sink`, while all other functions must accept a multi-threaded `Worker` as their first argument. Any function except the `Sink` may have any number of extra arguments (this is just an implementation artifact; `Sink` could theoretically be made to accept extra arguments). Then, `to_stream` is called as depicted in Listing 5, generating roughly the code in Listing 6.

```
to_stream!(multithread: [
    f_0 (/* extra args for f_0 */),
    (f_1(/* extra args for f_1 */), /* number of workers */),
    (f_2(/* extra args for f_2 */), /* number of workers */),
    ...
    f_i,
]);
```

Listing 5. `to_stream` multi-thread example

```
let (snd_0, rcv_1) = create_sender_receiver_pair();
let f_0 = SequentialSource::new(snd_0);
f_0.run(f_0, (/* args for f_0 */));

let (snd_1, rcv_2) = create_sender_receiver_pair();
let worker =
    MultiThreadedWorker::new(/* number of workers */, rcv_1, snd_1);
worker.run(move |w| f_1(w, /* extra args for f_1 */));

let (snd_2, rcv_3) = create_sender_receiver_pair();
let worker =
    MultiThreadedWorker::new(/* number of workers */, rcv_2, snd_2);
worker.run(move |w| f_2(w, /* extra args for f_2 */));
...
let sink = SequentialSink::new(rcv_i);
sink // sink is returned at end
```

Listing 6. `to_stream` multi-thread generated code

To generate MPI code instead, one would simply change in Listing 5 the token `multithread` to `mpi`. Now, because MPI can not simply fire a new process from an arbitrary code location, `to_stream`'s MPI implementation must do more work to ensure the right amount of replication. It creates one worker per replication amount and uses the processes' rank to determine what code they should run. For example, if Listing 5 had $i = 3$, a level of replication of 2 for both its workers, and was generating MPI code, it would assign `f_0` and `f_3` to process of rank 0, `f_1` to processes $[1 - 2]$, and `f_2` to processes $[3 - 4]$. This implies an MPI world size of at least 5, which we validate during startup, exiting with an error if it is not the case. Communication between the processes is done in a round-robin fashion, starting at an offset according to the process's rank. In the example given, process 1 would send to processes $[3, 4]$ while the process 2 would send to $[4, 3]$. This is done to minimize contention over the communication channels.

The final generated code in Listing 6 will automatically validate our pipeline according to Remarks 1-3 by virtue of simple type checking. Now, at this point, we are still demanding the programmer write the set of functions F taking into consideration this implementation since their first arguments must accept types that implement the relevant traits we have presented. If we can eliminate this final hurdle, we will have a nearly completely transparent abstraction that lets programmers build pipelines while specifying just "normal" functions.

3.3 Procedural macro function transformation

Procedural macros in Rust are separate programs written by the developer that perform code transformations. During compilation, when one of these macros is called, the compiler will execute the respective program and feed the parsed Rust AST as input, expecting a stream of valid Rust tokens as output. Within the procedural macro, the developer can do any arbitrary computation. It is possible to, for example, generate code conforming to a specification contained in a separate file, detect special hardware features, adjust the generated code accordingly, and so on. There are 3 types of procedural macros according to the Rust Reference [35]. We will be using attribute macros to perform function transformations. When used in this way, attribute macros look very similar to annotations in other programming languages such as C++ and Java. Our ultimate goal is to allow the developer to write sequential function implementations and then transform those into functions that execute in parallel using the traits we implemented in the previous section. Listing 7 shows what that will look like.

We will call the original function f_o and the generated function f_g. We must not change f_o's internal semantics, lest the results differ from their sequential implementations. This is trivially handled by simply copying f_o's body into f_g. Furthermore, because f_g will use types that implement the traits we discussed in the previous section, the exact same function transformations work for any implementation, both multi-threaded and distributed. We will now consider what transformations we must do to each function in Listing 7, starting with the `stage`.

```
#[source]
fn source(/*inputs*/) -> /*outputs*/ {/* sequential source logic */}

#[stage]
fn stage(/*inputs*/) -> /*outputs*/ {/* sequential stage logic */}

#[sink]
fn sink(/*inputs*/) -> /*outputs*/ {/* sequential sink logic */}

// later, the developer calls to_stream!(), using these functions
```

Listing 7. Desired application level code

3.3.1 Stage.

For this function, f_g will accept a $Worker$ implementation with a `Receiver` that will receive a tuple consisting of f_o's input and a `Sender` that will send the same type as f_o's output. In between them, we execute f_o to perform the desired computation. Listing 8 shows the result of that.

```
fn stage(worker: impl Worker<(/*inputs*/), /*outputs*/>) {
    let mut r = worker.receiver().recv();
    while r != ReceiveResult::End {
        let output = { /* Run f_o with what we received in 'r' */ };
        worker.sender().send(output);
        r = worker.receiver().recv();
    }
}
```

Listing 8. `stage`'s transformed function

3.3.2 Source.

The `source` function has an extra requirement to let us transform it: it **must return an iterator as its output**. If it does not, the procedural macro returns an error (this is trivially detectable since we only need to see if the function's signature output begins with `Iterator`). This corresponds to the notion that the source function must create the units of work that will be sent through the pipeline. A function that simply returns an integer does not make sense for the source of a pipeline. Since `source` must return an iterator, f_g just executes f_o and iterates over its results, sending each of those to a worker in the next stage in the pipeline.

3.3.3 Sink.

While the `source`'s f_o function must return an iterator, the `sink`, being the opposite of `source`, generates an f_g that returns an iterator. This allows the developer to iterate over the results outputted by the pipeline. The iterator's implementation will call $receiver.recv()$ to get the next item, executing f_o before returning it to the caller.

3.4 Ordering and stateful computations

There are two last considerations to finalize our abstraction: ordering constraints and stateful computations.

3.4.1 Ordering.

Some data streams demand the processed items be outputted in the same order as they were inputted. A typical example is video frames. This means our `sink`'s f_g must allow for that. We implemented the algorithm described in [16] to order the output. It works by tagging each item in the `source` with a monotonically increasing integer. Then, in the `sink`, we use a priority queue to determine which item should be outputted next. The current implementation is not robust against data losses throughout the pipeline (it would wait for the missing data forever), though that could be improved in future works. To specify that we want the output to be ordered, we use `#[sink(Ordered)]`.

3.4.2 State.

Some workers may benefit from having some sort of immutable state for performing their computations. For example, they could read a specification from a file. If we do not allow f_g to receive extra variables representing that state, f_o would have to read the specification on every execution, which would clearly be sub-optimal. To accommodate stateful computations, `sink` and `stage` accept an optional `State(args)` argument (`#[stage(State(var1, var2, ...))]`). These correspond exactly to the "extra arguments" in the `to_stream!` implementation we explained in Section 3.2.3.

4 Results

This Section discusses the results of the MPI implementation explained in Section 3. We created distributed implementations for the 4 programs in *RustStremBench*[1] provided by [27]: `bzip2` – performs compression with the bzip2 algorithm; `micro-bench` – calculates a mandelbrot set; `eye-detector` – uses OpenCV to detect people's eyes from a video; and `image-processing` – applies a series of filters to a list of images. Their execution graph can be found in Figure 2a. We slightly changed `eye-detector` from the original work, introducing an extra computation stage.

There are two distributed implementations: one using raw MPI calls and one using our abstraction. We implemented both versions, and the raw MPI one was written to mimic the generated code as closely as possible. To make the necessary calls to MPI, we used the `rsmpi` library [12, 32], with openMPI as its underlying implementation [11]. When it was necessary to serialize certain data types, we copy-pasted the code into both implementations to make sure this would not affect our results.

We begin with an analysis of code intrusion effort since our primary goal with our implementation was to make distributed parallel programming easier. We then proceed to evaluate its impact on the final program's performance. We show that our abstraction leads to considerable programmability benefits with negligible performance overhead.

4.1 Programmability

The primary reason for creating a higher-level abstraction is to simplify the programmer's activity. Therefore, it is important to measure programming complexity in some form, so that we may confidently state we have achieved our goal. To measure programmability, we compare significant lines of code, as reported by the `scc` tool[2], and Halstead complexity metrics [17] between the sequential and distributed implementations. Table 1 shows the results.

Discussion. Table 1 indicates our abstraction has greatly simplified the act of implementing a parallel pipeline compared to using raw MPI calls. The extra lines of code and complexity we observe in Eye Detector and Image Processing are mostly due to the extra code necessary to implement

[1]source code available at https://github.com/GMAP/RustStreamBench
[2]available at https://github.com/boyter/scc

the serialization of the datatypes we need to communicate between the workers. We do not need to do that for Bzip2 and Micro-Bench because Bzip2 only needs to communicate raw data, and Micro-Bench's structures are simple enough that we can use `serde`'s default serialization methods. Comparing the multithreaded version of the programs to their MPI counterparts shows how programability between the applications that do need complex serialization is almost identical . Halstead estimations differences between the raw MPI and our abstraction's implementation are explained by the fact that raw MPI leads itself to very branch-happy code. To give each process its role, we must branch on its MPI-given rank. This inflates Halstead's estimations since it considers branches to increase code complexity. By contrast, the branching part of MPI is completely hidden away when using our abstraction, which greatly reduces Halstead's estimated complexity.

	Bzip2		**Micro Bench**		**Eye Detector**		**Image Processing**	
	SLOC	Hours	SLOC	Hours	SLOC	Hours	SLOC	Hours
Sequential	190	39.50	37	1.37	50	3.06	26	0.76
This Work (MT)	226	46.65	100	10.50	108	8.66	64	2.14
Pure MPI	504	178.77	213	27.93	370	60.10	287	47.29
This Work (MPI)	226	46.65	103	10.49	199	19.87	112	5.64

Table 1. Programmability metrics in terms of significant lines of code (SLOC), and Halstead estimated development time in hours. "MT" stands for "multi-threaded".

4.2 Performance

Distributed computing is often used not only for increased reliability but also for increased scalability. Thus, we should make sure our abstraction does not negatively affect execution time. This section measures the throughput in items per second of the 4 programs presented at the beginning of this Section. The "item" in questions changes for each application: for `bzip2`, it is a chunk of 900kB, from a Fedora Core 6 ISO file; for `micro-bench`, it is each row of the final 4096x4096 image; for `eye-detector`, it is the frames of the video from a 450 frames video with many faces; and for `image-processing` it is each image in a set of 1000 small (640x427) images. The results are presented in Figure 2, where "0" replicated stages represent the original sequential program.

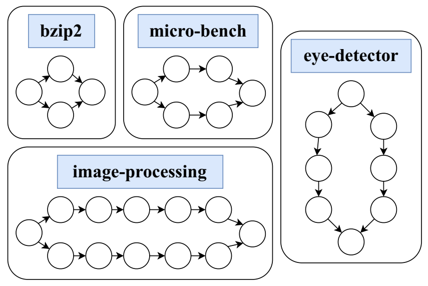
We conducted our experiments in a cluster with 4 machines, connected through 1Gb ethernet, each equipped with two Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40GHz, and 32GB of RAM, for a total of 48 physical cores. We applied a constant factor of parallelization for every stage of the pipeline, which explains why the x-axis values change for each graph. For example, `micro-bench` has 2 stages, which means we can only execute with $2n$ replicated stages, plus 1 extra process for each sequential sink and source. This was done purely to avoid bottlenecks between stages; as Section 3 demonstrated, changing the amount of replication for each stage in our abstraction is trivial.

Discussion. `eye-detector` and `image-processing` are the applications with the highest number of stages (3 and 5, respectively). The proposed abstraction and MPI performed roughly the same in these circumstances. The other applications show a visible difference between the implementations, with `bzip2` favoring raw MPI, and `micro-bench` favoring us. `bzip2`'s raw MPI has the advantage that, since in a compression algorithm, we are working with raw bytes, no serialization needs to be done. But our abstraction always serializes the data. Even if, in this case, the serialization consists of a simple copy, when done over large amounts of data, the costs add up. This also explains why there is a much larger difference in decompression: there is less work for the CPU, and so the extra copying of data makes a more significant impact. `micro-bench`'s difference is so tiny it could be explained by variations in the network. Overall, we see our abstraction significantly increased execution times only for the most straightforward application, which had the least number of stages (only 1) and needed not serialize its data. All other applications that involved more complicated pipelines and data serialization show either no or a very small difference.

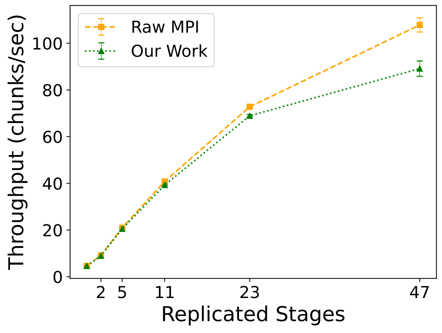
5 Related Work

Rust has had many abstractions for executing parallel code in shared memory environments. From academic efforts [3, 27, 34], to non-academic, though widely used, ones [30, 36]. These developments will likely continue since Rust offers distinctive benefits to multi-threading programming (no data races and a standard library with many efficient synchronization primitives). Although we implemented a version of our library for shared memory, it was only done as a way of validating that our abstraction was generic enough to support multiple parallel environments. Our main contribution lies in the MPI, the distributed implementation. Our work also distinguishes itself from other academic works that leverage Rust's metaprogramming capacities [7, 18, 33] because they do not focus on generating code to run in distributed environments.

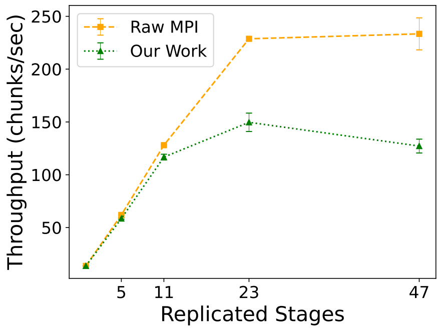
Specifically, regarding MPI, there are the works of Tronge, Pritchard, and Brown [37, 38]. In [37], the authors reimplemented core components of Open MPI used for intra-node communication in Rust, showing its performance is close to the existing C code. In [38], the authors examined ways to implement or use point-to-point communication within memory-safe programming languages. To this end, they've datatype implemented type matching on top of Open MPI, and a UCX-based library written in Rust. Both of these works are lower-level than ours, focusing on programming feasibility and low-level performance. In theory, our abstraction could have been built on top of their work, rather than `rsmpi`. This could be done in possible future works.



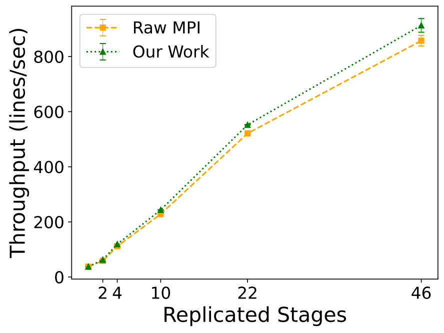
(a) Applications execution graph



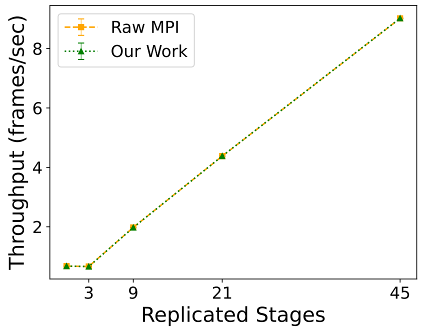
(b) `bzip2` compression



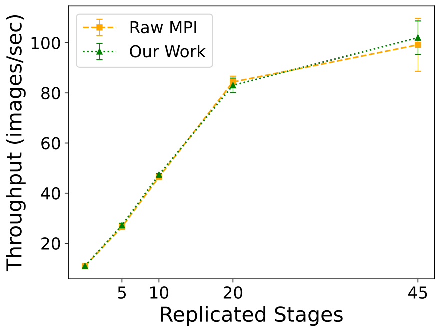
(c) `bzip2` decompression



(d) `micro-bench`



(e) `eye-detector`



(f) `image-processing`

Figure 2. *RustStreamBench* with handwritten MPI implementations vs. automatic parallelization using this work.

RStream [9] is a data-processing platform written in Rust whose results are competitive with custom MPI implementations. The authors created a platform for stream processing, directly comparing their work's performance with Flink's. Our work proposes just a library (not a platform), based on code transformations to deal with the specific case of a linear pipeline, and with a greater focus on ease of programming. Moreover, our ideas presented in Section 3 are generic and abstract, such that they may be implemented in nearly any programming language with metaprogramming support and for multiple parallel environments.

6 Conclusion

In this work, we presented a new internal DSL for expressing linear pipelines with minimal code intrusion in Rust. We also performed source-to-source transformations targeting HPC clusters using MPI as a runtime system. We demonstrated that our proposed abstraction is general enough to target multi-threaded or message-passing implementation with the same annotation interface. We collected programmability metrics in the form of SLOC and Halstead estimations to demonstrate how our DSL simplifies the implementation of parallel code. Furthermore, we have presented performance measurements showing that the generated code is competitive with the raw MPI equivalent, except for the case where the data does not need to be serialized since the raw MPI code can then omit an extra copy of the data.

Although the results are promising, this work's validation is limited to the application set and computing environment. More experiments are necessary in the future to generalize the achievements. We plan to evaluate our DSL in different cluster environments using the Infiniband network and newer servers, as well as test other workloads. Also, as mentioned in Section 5, we aim to use another underlying communication protocol or an alternative MPI implementation, making use of prior work like [37] or [38]. We want to provide another work distribution with an on-demand scheme, as opposed to its current round-robin, measuring performance differences. We could also investigate the possibility of detecting when we can bypass serialization, thus improving Bzip2's performance. Finally, we plan to use the same programming abstraction approach to implement easy-to-use parallelization for GPUs or other similar accelerators.

Acknowledgments

This work is funded by PUCRS University, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, CNPq Research Program (Nº 306511/2021-5), and FAPERGS call 09/2023 - PqG.

References

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. *Fastflow: High-Level and Efficient Streaming on Multi-core.* John Wiley & Sons, Ltd, Chapter 13, 261–280. https://doi.org/10.1002/9781119332015.ch13

[2] Gabriella Andrade, Dalvan Griebler, Rodrigo Santos, and Luiz Gustavo Fernandes. 2023. A parallel programming assessment for stream processing applications on multi-core systems. *Computer Standards & Interfaces* 84 (March 2023), 103691. https://doi.org/10.1016/j.csi.2022.103691

[3] Valerio Besozzi. 2024. PPL: Structured Parallel Programming Meets Rust. In *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 78–87. https://doi.org/10.1109/PDP62718.2024.00019

[4] Paweł Czarnul, Jerzy Proficz, Krzysztof Drypczewski, and Pedro Valero-Lara. 2020. Survey of Methodologies, Approaches, and Challenges in Parallel Programming Using High-Performance Computing Systems. *Sci. Program.* 2020 (jan 2020), 19 pages. https://doi.org/10.1155/2020/4176794

[5] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. 1993. Parallel programming using skeleton functions. In *PARLE '93 Parallel Architectures and Languages Europe*, Arndt Bode, Mike Reeve, and Gottfried Wolf (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 146–160.

[6] David del Rio Astorga, Manuel F Dolz, Javier Fernández, and J Daniel García. 2017. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience* 29, 24 (2017). https://doi.org/10.1002/cpe.4175

[7] José Duarte and António Ravara. 2022. Taming stateful computations in Rust with typestates. *Journal of Computer Languages* 72 (2022), 101154. https://doi.org/10.1016/j.cola.2022.101154

[8] Leonardo Faé, Renato Barreto Hoffmann, and Dalvan Griebler. 2023. Source-to-Source Code Transformation on Rust for High-Level Stream Parallelism. In *XXVII Brazilian Symposium on Programming Languages (SBLP) (SBLP'23)*. ACM, Campo Grande, Brazil, 41–49. https://doi.org/10.1145/3624309.3624320

[9] Alessio Fino, Alessandro Margara, Gianpaolo Cugola, Marco Donadoni, and Edoardo Morassutto. 2021. RStream: Simple and Efficient Batch and Stream Processing at Scale. In *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 2764–2774. https://doi.org/10.1109/BigData52589.2021.9671932

[10] Martin Fowler. 2010. *Domain-Specific Languages (Addison-Wesley Signature Series (Fowler))* (1 ed.). Addison-Wesley Professional. https://martinfowler.com/books/dsl.html

[11] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 97–104.

[12] Andrew James Gaspar. 2018. Rust in HPC. (12 2018). https://doi.org/10.2172/1485376

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

[14] 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 (March 2017), 1740005. https://doi.org/10.1142/S0129626417400059

[15] Dalvan Griebler and Luiz Gustavo Fernandes. 2017. Towards Distributed Parallel Programming Support for the SPar DSL. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing (ParCo'17)*. IOS Press, Bologna, Italy, 563–572. https://doi.org/10.3233/978-1-61499-843-3-563

[16] 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, 8 (July 2018), 4042–4061. https://doi.org/10.1007/s11227-018-2482-7

[17] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc., USA.

[18] Kyle Headley. 2018. A DSL embedded in Rust. *IFL 2018: Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*, 119–126. https://doi.org/10.1145/3310232.3310241

[19] Renato B. Hoffmann, Dalvan Griebler, Marco Danelutto, and Luiz G. Fernandes. 2020. Stream Parallelism Annotations for Multi-Core Frameworks. In *XXIV Brazilian Symposium on Programming Languages (SBLP) (SBLP'20)*. ACM, Natal, Brazil, 48–55. https://doi.org/10.1145/3427081.3427088

[20] Renato Barreto Hoffmann, Júnior Löff, Dalvan Griebler, and Luiz Gustavo Fernandes. 2022. OpenMP as runtime for providing high-level stream parallelism on multi-cores. *The Journal of Supercomputing* 78, 1 (January 2022), 7655–7676. https://doi.org/10.1007/s11227-021-04182-9

[21] S. Klabnik and C. Nichols. 2023. *The Rust Programming Language, 2nd Edition.* No Starch Press. https://books.google.com.br/books?id=SE2GEAAAQBAJ

[22] Júnior Löff, Renato Barreto Hoffmann, Dalvan Griebler, and Luiz Gustavo Fernandes. 2022. Combining stream with data parallelism abstractions for multi-cores. *Journal of Computer Languages* 73 (December 2022), 101160. https://doi.org/10.1016/j.cola.2022.101160

[23] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured parallel programming: patterns for efficient computation.* Elsevier.

[24] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.* 37, 4 (Dec. 2005), 316–344. https://doi.org/10.1145/1118890.1118892

[25] Chuck Pheatt. 2008. Intel® threading building blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.

[26] Ricardo Pieper, Dalvan Griebler, and Luiz G. Fernandes. 2019. Structured Stream Parallelism for Rust. In *XXIII Brazilian Symposium on Programming Languages (SBLP) (SBLP'19)*. ACM, Salvador, Brazil, 54–61. https://doi.org/10.1145/3355378.3355384

[27] Ricardo Pieper, Júnior Löff, Renato Berreto Hoffmann, Dalvan Griebler, and Luiz Gustavo Fernandes. 2021. High-level and Efficient Structured Stream Parallelism for Rust on Multi-cores. *Journal of Computer Languages* 65 (July 2021), 101054. https://doi.org/10.1016/j.cola.2021.101054

[28] Ricardo Luis Pieper. 2020. *High-level Programming Abstractions for Distributed Stream Processing.* Master's Thesis. School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil.

[29] Thomas Rauber and Gudula Rünger. 2013. *Parallel programming.* Springer.

[30] Rayon. 2019. Rayon. https://github.com/rayon-rs/rayon

[31] Dinei A. Rockenbach, Júnior Löff, Gabriell Araujo, Dalvan Griebler, and Luiz G. Fernandes. 2022. High-Level Stream and Data Parallelism in C++ for GPUs. In *XXVI Brazilian Symposium on Programming Languages (SBLP) (SBLP'22)*. ACM, Uberlândia, Brazil, 41–49. https://doi.org/10.1145/3561320.3561327

[32] Rsmpi. 2023. Rsmpi. https://github.com/rsmpi/rsmpi

[33] Arash Sahebolamri, Thomas Gilray, and Kristopher Micinski. 2022. Seamless Deductive Inference via Macros. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/3497776.3517779

[34] Stefan Sydow, Mohannad Nabelsee, Sabine Glesner, and Paula Herber. 2020. 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). 289–296. https://doi.org/10.1109/SBAC-PAD49847.2020.00047

[35] The Rust Project. 2024. The Rust Reference. https://doc.rust-lang.org/reference/

[36] Tokio. 2019. Tokio - The asynchronous runtime for the Rust programming language. https://tokio.rs

[37] Jake Tronge and Howard Pritchard. 2023. Embedding Rust within Open MPI. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (<conf-loc>, <city>Denver</city>, <state>CO</state>, <country>USA</country>, </conf-loc>) *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 438–447. https://doi.org/10.1145/3624062.3624112

[38] Jake Tronge, Howard Pritchard, and Jed Brown. 2023. Improving MPI Safety for Modern Languages. In *Proceedings of the 30th European MPI Users' Group Meeting* (<conf-loc>, <city>Bristol</city>, <country>United Kingdom</country>, </conf-loc>) *(EuroMPI '23)*. Association for Computing Machinery, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3615318.3615328