High Performance Data Processing Using Rust

Faculty of EEMCS

Authors:
Alex HESSON
Anthony IROKOSU
Batu BUDIN
Julian ADAMSE
Xianzhi WU

Supervisor:
Dr. R.E. Monti

Rosen:

Eyk Haneklaus

Hendrik Niemeyer

Klaas Kole

Tina Ulbrich

Apr 14, 2023

UNIVERSITY
OF TWENTE.

Abstract

Rosen Group is a company that specializes in the inspection of pipelines. To aid in their work, Rosen created from the ground up a system for the processing of data using the c++ programming language.

With the rise in popularity of the Rust programming language, Rosen is looking into the viability of using Rust programming language as an alternative to c++.

Rust is a programming language that has gained notoriety for being excellent in aspects such as usability, cleaner and more efficient code, and more accessible library functions, as well as project and package management.

Our project involves implementing a pipes and filters design pattern, which is similar to the one used by Rosen for data processing, using the Rust programming language. Furthermore, we will share our experience of working with Rust by documenting the process.

Our approach involves conducting research on the functionality and workings of Rust, as well as exploring various Rust libraries. Additionally, we will create tests to ensure that our implementation works as intended, and document our experience throughout the process.

We found out that using Rust has some advantages including: making it impossible to have undefined behavior due to writing safe code; errors are easier to find thanks to Rust's unwrap/expect functionality; Rust's default parallelism handler passes messages instead of sharing memory leading to better memory efficiency.

Keywords: Rust, pipes and filters design pattern, data processing, parallelism.

Table of contents

Chapter 1

INTRODUCTION

In this chapter we introduce our client Rosen and the domain and goals of our project.

1.1 Rosen

Rosen is a family business founded in Lingen, Germany in 1981 by Hermann Rosen, and it is currently headquartered in Stans, Switzerland.
The company specializes in researching, developing and producing inspection tools for pipeline assessment, mostly for oil and gas pipes. The company is also active in multiple sectors such as developing and making high performance elastomers, intelligent plastic systems, pipeline cleaning services and many others.
Rosen is a leading global provider of cutting-edge solutions in pipeline inspection solutions, currently holding the world record for the longest non stop inspection run when they diagnosed the Nord Stream Pipeline for a total length of 1.224km.

1.2 Pipeline inspection process

A pipe inspection tool called a “pig” is used by being inserted into the pipe at one end, then it runs along the length of the pipe. Using a variety of on-board sensors it will detect fissures, cracks, possible stress points, or corrosion in the pipe. For pictures, see Appendix A - Pictures.
The process starts by flushing with nitrogen the launching and receiving facilities which are located at each end of the pipe, then inserting the inspection tool. The tool is then propelled by the gas pressure in the pipe along its length. Gas pressure will vary so to control the speed of the tool a valve is used to let some of the pressure pass through the inspection tool.

1.3 Sensor data and HDF5 format

The data collected from the sensors of the inspection tool is stored and processed in the HDF5 format. The HDF5 (Hierarchical Data Format version 5) file format is an open source file format made to offer support for large and complex data by making storing and organizing easier.
The HDF5 format uses only two types of of objects for the file structure:

- Datasets: typed multidimensional arrays
- Groups: container structures that can hold datasets or other groups

The biggest advantage of using HDF5 as a file format is that it allows for easy data slicing, a process that allows for a particular dataset to be split into smaller data “chunks” for faster processing times. This basically means that the entire data set does not need to be stored into memory for it to be processed, thus allowing for more efficient work with very large datasets. This is a big advantage because the pipeline inspection tool can gather as much as 40MB of data for every meter of pipe analyzed, so a pipe that is 2 km long can produce as much as 80GB of data that needs to be stored and

processed, and since the tool does not have wireless communication when it is in the pipe, all of this needs to be done locally.

1.4 Problem Description

Rosen needs to deal with huge sets of sensor data stored in hdf5 data. This data needs to be processed by several algorithms that are configured with specific parameters. The decision about which algorithms, in which order and with which parameters are taken by their experienced experts.
To achieve sufficient performance and solve these big data problems, they implement these algorithms in C++. They used a “pipes and filters” design pattern to “chain” these algorithms at runtime. Data is fed into this “chain” as a stream, to minimize file operations. The output consists of a modified data file and some result files. For this project, they want to experiment with an implementation of this framework in Rust. Rust is a modern and relatively new programming language which has gained popularity in the last couple of years. It features clean, modern syntax, a very helpful compiler and a rich ecosystem of third-party libraries. With its first-class support of parallelism, concurrency and a built-in async programming model, we believe it is a good choice for implementing a framework for Rosen’s signal and image processing and AI algorithm data pipeline.

1.5 Domain Analysis

1.5.1 Introduction

The domain of the project concerns the processing of large amounts of data. The Rosen group gets a large amount of data from their inspection tools (“pigs”), about 40 gigabytes per kilometer, and this data needs to be passed through a series of filters in order to gain useful information and insight from the raw data. The project needs to be able to handle the reading, processing and writing of large amounts of data and doing so concurrently to be efficient.

1.5.2 Target Users

The target groups for this project are the developers at Rosen. Currently, they have an existing implementation in C++ but are looking into the viability of using Rust as a modern alternative to C++.

1.6 Goal of the Project

The goal of the project is to create an efficient implementation of the pipes and filters architecture using Rust and its libraries whilst documenting our experience with the process of working with rust, both positives and negatives in the project domain.

After the data is gathered, it will be run through a series of filtering algorithms (processing chain) which run concurrently for faster processing times. The current implementation is done in C++ but Rosen wants to experiment with a possible implementation of the processing chain in Rust.

Rust is a modern, multi-paradigm and high-level programming language that is focused on safety and performance. It offers safe concurrency and provides memory and type safety without garbage collection. Rust also offers first-class support for parallelism and concurrency, which is ideal for the processing chain framework that Rosen uses. Python bindings are also required so that users who do not have experience with Rust can interact with the framework.
Our solution should be benchmarked, thoroughly tested and sent as a docker image so that it can be run anywhere.

Chapter 2

PLANING

In this chapter, we will explain how we managed our project. We will discuss the reasoning behind our approach, the planning process we followed, and how we selected our deadlines and milestones. Additionally, we will provide insights on how our team communicated throughout the project.

2.1 Project Management

We use Agile methodology for managing the project. We had daily stand-up meetings to keep track of our progress and we had weekly meetings with the client to showcase the progress made every week, the weekly meeting gave us the opportunity to get feedback from the client. We also had weekly meetings with our supervisor to discuss our progress.

We used the Trello board for the division and assignment of tasks. This allowed members to know what they had to work on and the status of each task at any given time. When a task is assigned, it is moved to the doing column; once completed, it is moved to the testing column; once it passes the test, it is moved to the completed column; and if any bugs are found, it is moved back to the working column.

2.2 Meeting planning

We had daily stand-up meetings, sometimes in person most times online, to discuss what we had done, and what we planned to do and to discuss how to tackle different problems that had surfaced. The daily meetings helped the team with being informed all the time about what tasks different members were working on and the progress of the project. Meetings with Rosen and our supervisor were planned for Fridays at 11 am and Thursdays at 1 pm respectively.

2.3 Milestones

We created milestones to identify important stages during the project and added expected dates of completion as deadlines and we divided the project into smaller tasks placed on a timeline. We visualize the milestones, tasks and timeline using a Gantt chart.

NB: The diamond keys represent the milestones and the blue bar represents the timeline for the tasks, each task belongs to the closest milestone below it

2.4 Team Roles

The team is composed of different members with different skill sets and interests. Each member is assigned a responsibility best suited to their interest. The role division is listed below:

Roles	**Member(s)**
Communication with Rosen	All
Communication with supervisor	All
Docker Master	Julian
Documentation Master	Xianzhi
Git Master	Anthony
Report Master	Alex
Test Master	Batu

Chapter 3

Requirement Specification

This chapter will cover the project proposal and the requirement gathering phase. We will explain how we developed the project proposal and discuss the methods used to gather and document the project requirements.

3.1 Requirements gathering

3.1.1 Functional

We gathered the requirements in multiple stages. The first stage was us reading the initial project proposal that Rosen made and discussing it. After that the second stage was discussing the assumptions that we made with Rosen. We did this in our first introductory meeting with them. Most of the requirements were derived from features that their current implementation has. This was to make the two versions comparable later on. After this the next stage was us formalizing the requirements in our project proposal. When the requirements were formalized we sent them to Rosen for feedback. In the final stage, we incorporated the feedback from Rosen into our requirements and that concluded the requirements gathering.

3.1.2 Quality

From our discussions with Rosen we were also able to infer some quality requirements. The most important quality requirement was the requirement for an extendible code base. Another requirement was that the performance was comparable to the current C++ implementation. We got these requirements from the main goal of the project. Which is: checking if Rust is a valid alternative for large scale C++ projects.

3.2 Requirements Prioritization

3.2.1 Functional

After our team got in contact with Rosen, the first couple of meetings were used to get familiar with the project and to gather the requirements. We decided to use annotations for the tasks to categorize them into three categories: (MUST): if we believed that the tasks are critical to the project success, (MAYBE): if we believed that the tasks would be helpful but they are not critical to the project success and should be implemented only if time allowed it and, finally (WON'T): those tasks have been found to not be necessary for the project to be successful. The categories were checked with Rosen.

These are the final categorized requirements related to our project.

3.2.1.1 Must

1. Docker: Create a docker environment for working with the HDF5 and the Rust programming language.
2. Read Data: Being able to read the HDF5 file format and divide them into chunks. The read_slice function is used to read and divide the data into chunks in stages. The chunk is sent to the thread after every stage. If all the data were read into a single vector, it would be necessary to wait until all the data was read before sending the first chunk.

3. Write Data: Writing the data chunks after being processed into an HDF5 file. The order of reading should be preserved whilst writing. The write_slice function is used to write incoming chunks into an HDF5 file
4. Chunk Size: Define the chunk size the dataset will be divided into with respect to optimal performance.
5. Filter : Define simple filters to manipulate the data chunks.
6. Multidimensional data: Most data is at least 1 dimensional since the pig is collecting data as it moves through the pipe. Some data has more dimensions like when there are multiple sensors of the same kind placed around the pig. For example, 12 sensors that measure the thickness of the pipe that are placed around the circumference of the pig. The data for these sensors would be 2D since we have 1 dimension for where in the pipe the measurement is taken and 1 dimension for where along the circumference of the pig the measurement is taken. To represent and work with the data from these 12 sensors, we need multidimensional data. Or we would need 12 separate data streams each with a different name. But this would add up and be not convenient if there are 120 sensors around the pig.
7. Threading: Perform experiments comparing different ways of implementing threading in rust.
    a. **Must**
        - Standard Rust Channels: Standard implementation of channels in built with Rust
        - Crossbeam Channels: Implement the concurrency with the channels provided by the crossbeam crate.
    b. **Maybe**
        - Rayon: Implement and benchmark the concurrency with the Rayon crate.
        - Intel Threading Building Block: implement and benchmark with the concurrency coming from the intel TBB.
8. Rust Documentation: Create a manual detailing how the system works. Rust has a great feature that automatically transforms code comments into HTML documentation pages.
9. Rust Testing: Use the in-built Rust unit test to perform our unit testing.
10. Reflection Report for Rosen: A report detailing our experience developing in rust, includes the timeline of when tasks were completed and libraries used, and the challenges faced during development.
11. Final report: The final report must include and explain every aspect of the project.

3.2.1.2 Maybe

1. Re-sampling: Some sensors on the pig will have a higher sensor rate than others. For example, one sensor that does 50 measurements per meter and one that does 100 measurements per meter. These values will still have to be combined sometimes (added for example). To do this, we would like to add the possibility to resample one sensor rate into the rate of another so that they match.
2. Create Python bindings: Create Python code that allows the calling of Rust functionality. This would be great as it allows personnel who are not knowledgeable in Rust to have the ability to write filters.

3. Parquet: Look into the feasibility of using the Parquet file format as opposed to the currently used HDF5.

3.2.1.3 Won’t

1. Non Sequential Filters (Won't): All filters in the project will perform their actions on chunks one after the other. Chunks do not have to be split to different filters and then recombined again.

3.2.2 Quality

We decided to prioritize the extendability of the code over the raw performance of the project. We did this because Rosen made it clear that the maintainability was key for this project. This became clear during the meeting we had with them. Performance is still important but given the language we are working with Rust it should be comparable. We will still try to make improvements but it should not interfere with the maintainability of the code base.

Chapter 4

DESIGN

In this chapter, we will provide a detailed explanation of the design used by Rosen, as well as the models and strategies we utilized to create our own design.

4.1 Architecture

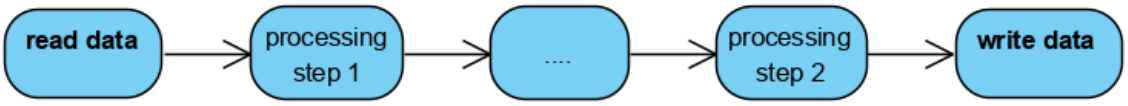
4.1.1 Processing Chain

The **Processing Chain** is the collective term given by Rosen to their system used to process the data from the sensors. This includes the implementation of the algorithms used to filter the data collected by the inspection tool, with no involvement from the user. The filters accomplish tasks such as:

- Converting the data from the measured sensor data (stored in DDF) to a more evaluation friendly format(SRH5)
- Computing additional data from the measured inputs
- Modifying existing data
- Search runs and sizing, for example, an automated search of areas of interest such as welding points or anomalies

During the evaluation process of the pipe many algorithms will be applied to the data both before and after the data can be evaluated, but also during the evaluation.
The processing chain allows for multiple algorithms to be chained together and apply them to input data as one individual operation.

The processing chain concept works by making use of an already implemented and established pattern of concurrently processing of data often referred to as Pipes and Filters.
According to Rosen, the term chain was used instead of pipe because of the already heavy use of the term pipe in their context.
The basic concept of the processing chain is linking multiple filtering algorithms in a sequence for easier concurrent processing of complex data in a chain-like manner.



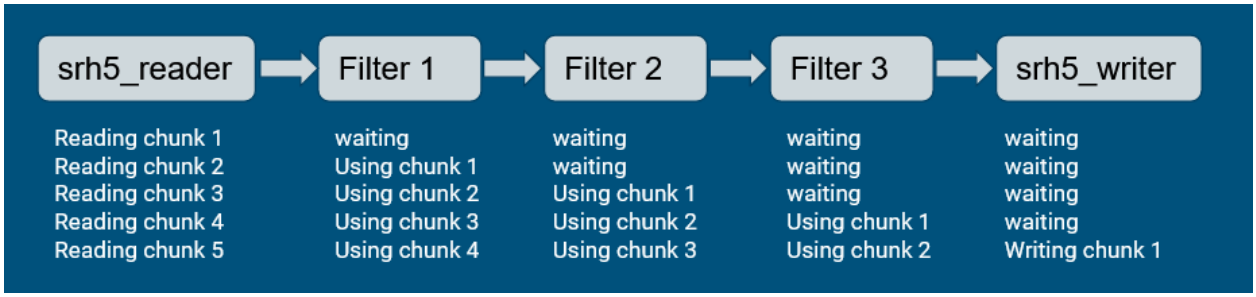
Firstly, the data is being read, then passed on to the processing chain of filters, and the final step is to write the processed data back to a hard drive.

4.1.2. Concurrency of the Processing Chain

The main advantage of using the Pipes and Filters principle in this context is that it allows the processing of very complex and large inspection data. The data is first split into smaller chunks, because as stated before, the entirety of the data file cannot be

stored into memory at once (2km of pipe can produce as much as 80GB of data). Those chunks are then individually stored into memory, processed, then stored. A signal chain with three filters used to manipulate SRH5 data will look like this:

When the processing chain is first run, the srh5_reader will read the first chunk of data, and once that is done. The filters will be idle initially. After the first read is done, the data will be passed onto the first filter, and reading the second chunk of data will start. After the first filter is done processing the data, it will pass it on to the second filter and the shr5_reader will pass the second chunk of data to filter one. Now, after the shr5_reader will read the third chunk, filter 1 and filter 2 might still be busy processing their data chunks, but after this is done filter 2 will pass it's data to filter 3, filter 1 will pas its data to filter 2 and the shr5_reader will pass the third chunk to filter 1.
Thus, by the time the shr5_reader is done reading the 5th chunk of data, the shr5_writer will be writing the results of the processed data to the hard drive. A diagram explaining this process is shown below.



The filters have to wait for one another, as one of the limitations of the data format is that it does not allow for a chunk of data to be accessed by two filters at the same time, so the system will only be as fast as the slowest filter.
If the srh5_reader cannot deliver a data set that is required for the filters, the said data set will be ignored.

4.1.3 Sensor data

In the context of the chain, the term "sensor" can refer to more than just the hardware definition. A sensor can refer to:

- The measurements are made by physical sensors, such as PHH, DSIN, DCOS, UAMPL, UTIME, ILIFT, and RLIFT.
- The results of the computation are based on other sensors, such as NIC, RAD, MINID, UWT, and UHIA.
- The distance or time axes, provide the distances and times for each sample.

The data from the sensors can be stored as a 2- or 3-dimensional array with different data types. For example, PHH is stored as a 2D array with floating point values, while UAMPL is a 3D array of 16-bit integers.

The stored data can be interpreted as follows: The first dimension refers to the sample data, i.e., the axial direction in the pipeline. The second dimension corresponds to the sensor ring on the tool, i.e., the circumferential direction. However, some sensors, like temperature sensors, have no circumferential distribution on the tool, so they may only have one element in the second dimension of the array and can be considered one-dimensional. The third dimension is reserved for certain sensors, like the UAMPL, that store multiple values per measurement. UAMPL sensors store the amplitude of a sequence of echoes for every measurement.

A defect 2D matrix of 8-bit non-zero integers is used for most sensors to indicate that the value at the corresponding position in the data matrix is defective.

4.2. Modeling

We use UML diagrams to model our designs. The UML diagrams used were: class diagrams for modeling the type system and activity diagrams for modeling the high-level acting of the system. We also used a flowchart to model the individual filter threads in greater detail. All diagrams can be found in Appendix B Diagrams.

The reason we chose to model with UML diagrams is that all of us were familiar with them. Another benefit of modeling with UML is that it can give a good impression of how things are going to work without being too much work to change later on.

The motivation for the flow chart is more specific. The filter threads are duplicated across many threads and need to work together concurrently. Since they are large pieces of code it would take a lot of work for us to rewrite them. Therefore we wanted to minimize the chance of having to do that. We created the flow charts to be able to reason about and discuss our plans for implementing the filter threads without having to rewrite code if anything changed. We think the flow charts help a lot in getting everyone on the same page concerning how the system is going to work. This allowed the whole team to reason about the system and spot potential problems before we started implementing them.

Chapter 5

PROCESS DESCRIPTION

In this chapter, we will discuss our team communication methods and the tools and technologies that we used to successfully complete the project.

5.1 Team Communication

For team communication we typically have meetings starting at 11 a.m. on Discord. In Discord we use the general channel to share quick things during meetings and the notes channel to share things with lasting importance.We also have weekly scrum masters who are in charge of summarizing the weekly jobs done to our client and supervisor. In Discord we have also added a Rosen channel to communicate with Rosen more quickly. For communication outside of meetings we had a WhatsApp group since everyone gets mobile notifications on there. For the sharing of documentation files we use a Google Drive folder.

5.2 Client and Supervisor Communication

Our first meeting with Rosen was an in-person, where we were introduced to the project, subsequent meetings were held online due to location constraints. We communicated through the use of Microsoft teams for weekly meetings, emails for formal deliverables and discord for informal asking of questions. We had our weekly meetings every Friday by 11 a.m. as this was the best time for both the team and Rosen. It also allowed us to present what we had worked on during the week and served as a feedback session for our progress. Emails were used to hand in deliverables we wanted feedback on, and discord was a convenient way to ask questions about scopes we were confused about as it allowed us to get a response in a timely manner.

We also had weekly meetings with our supervisor, we mainly met in person on university grounds on Thursday by 1pm, the time and date were chosen based on the most convenient time for both the team and our supervisor. During these meetings we reviewed: what we were working on, our progress and if we were on schedule with our planning and the general organization of the project. These sessions were also feedback sessions from our supervisor on how we can better manage our project. We communicated via emails for the handing in of deliverables for feedback and, we added our supervisor to our shared drive so he had access to our documentation.

5.3 Tools Used

5.3.1 Version Management

We used Git for version control, and GitLab for our repository. Our branching strategy involved the creation of a separate branch for each feature. After each feature has undergone thorough testing, that branch is merged into the main branch.
The reason we worked like this is so we can avoid merge conflicts, monitor the various iterations of our main branch, and stop functional code from breaking in the main branch.
Another advantage of working with GitLab is that everyone in the team is familiar with it since it is supplied by the UT.

5.3.2 Development Environment

There are some differences between the development environments of the team members. For the OSes Windows, Mac, and Linux(Pop!_OS) are all used. As an IDE VS Code is used but also Sublime Text. All of the IDE's use Rust Analyzer for Rust language support. For building, dependency management, running, and testing our code we use cargo.
As a scripting language we used to use bash/bat. It quickly became too much work to write 2 sets of scripts for every operating system. To lessen the workload we started using Python since that is portable between operating systems.

5.3.3 Programming Language

The project required the use of Rust. Rust is a systems programming language. Although Rust has some higher level features it focuses on zero cost abstraction meaning that the usage of those features does not incur (notable) performance loss. To allow for this Rust has an extensive type system including ownership and lifetimes of data. This type system also allows the compiler to give certain guarantees about concurrent programs. The ownership of the Rust type system does not allow all correct programs to be written. To still allow for this Rust has an unsafe feature that will allow for pointer dereferencing.

Rust was interesting to Rosen not necessarily because of the ownership type system but rather because of the modern features that the Rust brings. Some examples:

- Rust has a module system instead of needing "#include" statements.
- Rust has a tool called cargo that combines building, testing, package management, and documentation building.
- Rust has tests built into the language.
- Rust has documentation built into the language.

On top of that Rust also works well with the HDF5 type system by allowing things like pattern matching on data types.

A final feature of Rust that we like to bring attention to is the lack of exception handling in Rust. All Errors are handled with the type system returning a Result value that can then be matched to whether an error has occurred or not. This means that whether a function can error or not is always visible in the type signature of that function. There is however panicking that can happen anywhere and in any function, but this should only be used as a last resort for catching programming errors. The Result value vs Exception handling can be more of a subjective choice but we found that it makes learning a new library like HDF5 a lot easier as it's clear from just the signatures if something can error.

5.3.4 Libraries

The project required the use of some Rust crates in order to achieve some of our functionality. We used Rust's handy package manager named Cargo for handling our library dependencies. Cargo allowed us to easily add crates which we needed without any hassle.

We used the HDF5 Rust crate for reading and writing from and into an HDF5 file.The HDF5 rust crate provided thread-safe rust bindings for the HDF5 api written in C, this is because there is no HDF5 library written in Rust.

We also needed to handle reading and writing of multidimensional data,so we used ndarray, which is a rust crate that provides an n-dimensional container for general elements and numerics. Ndarray also allowed us to read the HDF5 file as chunk slices instead of reading the entire file into memory then chunking which improved the performance and memory management of our system.

Lastly, we used the crossbeam threading library. Crossbeam is a tool for concurrent programming. We used crossbeam as we wanted to experiment with other threading libraries to compare how they perform to the standard rust channels.

Chapter 6

IMPLEMENTATION

In this chapter, we will detail the project implementation process, including our methodologies and the reasoning behind certain design choices.

6.1 Concurrency

One of the requirements of our project is to have the reader and writer work concurrently as much as possible, since I/O operations are generally a bottleneck. It was also a requirement to have the filters on separate threads. Since data objects only flow one way, from the reader to the writer, we have identified channels as a way to implement communication between threads. The channels in the Rust standard library allow for multiple producers and a single consumer for message passing. In our implementation, all channels only use a single producer. The channel consists of two objects: a sender and a receiver. In our implementation, the reader holds a single sender. The next chain/filter holds the receiver to this sender. Similarly, filters hold both a receiver for the chain in front of them and a sender for the chain after them. A writer only holds a single receiver for the chain in front. In this way, all the filters are connected with the channels in between them. The message we pass contains either the next piece of data from the file or an end-of-file if all data was read. The reader will read and send the data down the chain until it reaches the end of the file. All threads will stop execution after an end-of-file message is received. The channels pass "owned" data from thread to thread. This means that the thread does not have to worry about other threads reading or modifying the data it is working with. This avoids many pitfalls that come with concurrent programming. Passing non-static references is also impossible in Rust. It would be possible to leak the memory before passing it through the channel. But leaking memory is not desirable given the large amount that has to be processed in the runtime of the program.

6.2 Chunking

In order to make maximal use of the concurrency of our program, we can't pass all the data at once. That would result in only one thread being able to work on it at a time. To fix this, we split the data up into chunks. Each of the chunks contains all data of a specific length of pipe. An HDF5 file can store data for multiple sensors. We store the data for each sensor in a HashMap with the name of the sensor as the key. Since the file can be chunked arbitrarily, it might occur that a filter needs access to data that is spread across multiple chunks. To allow for this access, two sets of buffers are added to each filter thread. A Pre-buffer holds chunks that come before the current chunk, and a Post-buffer holds data after the current chunk. Each time a filter finishes processing a chunk, all other chunks in the buffers get shifted in the following order: 1. A chunk is read from the receiver and placed in the pre-buffer; 2. A chunk is taken from the pre-buffer and becomes the new current chunk; 3. The old current chunk gets placed in the post-buffer; 4. A chunk is taken from the post-buffer and written to the sender. Since the current filter still needs to read chunks in the post-buffer, we have to wait with sending them to the next thread. This negatively impacts the concurrency since the next thread can't start yet. This could be solved by cloning the data and sending the copy to the next thread. But given the large amount of data that needs to be processed, making copies is not desirable. The I/O operations of reading and writing also don't utilize the buffers and are generally the bottleneck of the program. So we have decided against making copies of the data.

6.2.1 Finding Optimal Chunk Size

This feature was put in place to allow for error-free use of the dataset. We came up with the notion to determine the number that divides the entire length of the file in order to get an ideal chunk size. We accomplish this by dividing the dataset's number of rows into their prime integers, then finally multiplying 3-6 of them in reverse order to determine the ideal number of rows for a chunk. Since Rosen gave us an optimal chunk size of 4069, which we preferred to use. However with some change in the function, it may handle different sizes of datasets in a more efficient way.

6.3 Reading

Two alternative techniques were used to put the reading to use. The second arrangement was created for the purpose of increasing efficiency. We made use of the HDF Group's HDF5 rust library. First, we used the basic function to read the data that ROSEN had given us. After discovering the read:: function and becoming familiar with the library's features, we looked for a reading strategy that was more appropriate for our concurrency implementation. The issue was that the read:: methods read the file in its whole state before chunking the data into parts. After reading the entire file, this is made feasible by the process of chunking and applying filters. With our second approach, we are able to read HDF5 files in chunks of data and initiate threading after reading the first HDF5 piece. Rosen uses 1, 2, and 3 dimensional methods to acquire sensor data. This presents a challenge in our instance, so first we looked for a generic function that could handle all dimensions. Since this was not achievable, we used if statements in our read method to address the issue.

6.4 Writing

We were faced with two problems to solve when it came to writing the data. We needed to know the position the chunk belonged to in the dataset and, we needed to know the dimensions of the dataset the chunk belonged to. To address these problems we added a hashmap to our chunk Struct to hold this information. Once the chunk is ready to be written we extract the metadata from the hashmap.We create a new HDF5 file or open an existing one if available, then we create a new dataset within the HDF5 file or open an existing one if present. We then write the data into the dataset using the write_slice() function provided to use by the HDF5 crate. The write_slice() function has to be configured beforehand for the dimension of the data it has to write. To solve this problem we used if statements, so we use the write_slice() configuration for the right chunk. We tried to take a more generic approach but that proved too complicated hence we settled for if-cases.

Chapter 7

Testing strategy

In this chapter we will discuss our testing strategy.

7.1 Testing Plan

We employ distinct testing approaches for each milestone implementation and will provide a brief overview of these methods. Additionally, we will discuss the rationale behind our testing methods and explain how we utilized them.

7.1.1 Unit testing

We mainly did consider our milestones in this area. We used unit testing on **read**, **chunking**, **finding optimal chunk size** (not used), **filters**, **concurrency** and **writing**.

7.1.1.1 Read

By looking at the numbers at the beginning and end of the dataset, printing them on the console, and verifying through the UI. We can see that using the visualization tool HDFView, the numbers are identical. Also feeding 4 dimensional data was handled correctly since our system works for maximum 3 dimensional datasets.

7.1.1.2 Chunking

As explained in the implementation part, we used 2 different read functions for chunking and we tested both of them by printing the values and features (length & amount of chunks) on the console.

7.1.1.3 Finding Optimal Chunk Size

This is a feature we implemented to test our chunkin process without an error/exception. We tested this feature by using it and checking if the output can divide the length of the dataset.

7.1.1.4 Concurrency

The proper closing of the threads was tested by adding an error message to the output when a thread closed before it was supposed to. We then made sure this error message was not present during all other tests.

7.1.1.5 Write

Creating our own 1, 2 & 3 dimensional hdf5 files and reading them in the visualization tool HDFView.

7.1.2 Integration testing

7.1.2.1 Read & Chunking

Two read methods were tried using the same dataset, as was mentioned in the implementation section. We assert the data to each other after reading and chunking it to ensure its accuracy. Additionally, the speed of both read techniques was compared using the standard Rust "time" library. Although they were similar in terms of speed, we discovered that the read_slice:: approach works better because the chunking starts earlier.

7.1.2.2 Finding Optimal Chunk Size

This is not yet implemented, however we implemented some for testing the read and chunking. We wanted to use a number of rows that could divide the whole file. But maybe if implemented more detailedly, it could find optimal chunk size depending on the dataset size.

7.1.2.3 Filters

Given that we only needed filters for testing purposes they were only tested as part of other tests.

7.1.2.4 Chain

All of the components that are tested by unit combined in this section. Chain process starts reading the file, it checks if the chunk amount is the same as expected. After that, applying the filter and writing in chunks to a hdf5 file. At the end values are checked again by using HDFView and see them as expected depending on the filter.

7.1.3 Benchmarking

Our plan for testing the performance is by creating a benchmark based on the sample data we got from Rosen. We don’t have access to the code of the original project. That's why Rosen will do the actual comparison. This will remove a lot of external factors and make the outcome more meaningful. Since the actual algorithm is not up to us to implement we use a dummy algorithm for the test. The dummy algorithm will be doubling all the entries in the HDF5 data. This will make it hard for the compiler to optimize away the code we are trying to test. It will also be easy to see if the test was successful as we can inspect the data and see if it has been processed successfully.

We also have a plan to test the performance impact of using different libraries for concurrency. Our comparison is going to be between the implementation of channels by the Rust standard library and the one provided by the crossbeam library. As input for our benchmark we will use a file with a large amount of random data. For the processing of the data we wrote an identity filter that moves the data without modifying it. We are planning to do multiple tests with different chunk sizes and also different crossbeam buffer sizes. We will be measuring the time it takes for the program to process the file with the identity filter and write the result to a new file.

7.2 Testing Results

7.2.3 Results benchmarking

The result of the C++ version benchmarking was 21 seconds (for graph see Appendix C - Results). The results for the Rust benchmarking was about 12-14 seconds. So from the benchmarking we can conclude that the performance is comparable within a (factor 2 of each other). This means that our goal for performance is reached. It can also indicate that our solution is faster in this case. Due to slight differences in input we can't conclude a performance increase for certain. A possible increase in performance can be explained by our solution being able to concurrently read and write a single file. This advantage would become negligible with multiple files as input. So in conclusion we have achieved similar performance as was the goal of the project.

We also compared using Crossbeam instead of the Rust std for the channel implementation (for graph see Appendix C - Results). Out of our comparison we concluded that small chunk sizes lead to a lot of overhead caused by moving the chunks. We found the optimal chunk sizes to be within 100,000 and 1,000,000. Chunk sizes larger than 1 million will not be beneficial as the concurrency will suffer. Another result is that Crossbeam implementation can be faster than the Rust std implementation. For this to happen Crossbeam needs to be properly configured. Crossbeam can be configured with different types of buffers (store chunks in between threads). A buffer can be bounded or unbounded. A bounded buffer will force a thread to wait until there is space in the buffer. An unbounded buffer will grow to allow for insertion. We found bounded threads with a size of 10 or 100 gave a performance improvement.

Chapter 8

Future work

In this chapter, we will focus on potential improvements that can be implemented in our project in the future.

8.1 Support for More Data Types

At current time only the double data type is supported. This data type is not suitable for all use cases. For example, it will not store data efficiently when working with whole numbers. Future modifications to the program might include support for more of the HDF5 types like booleans and integer types. Implementing these types should not face challenges as it will only involve copying the implementation of the double data types for other data types. The reason it was not done during the project was because of time constraints and since the scope of the project was only to show what was possible in Rust.

8.2 Resampling

One of the “maybe” requirements of the project was to allow for the resampling of sensor data. At current time this feature is not implemented. While implementing this feature we realized that we needed access to the sample rate of the original data. Obtaining this sample rate was less straightforward than we thought and we ran into time constraints while deciding on an approach. Future work could provide a solution to this problem. When the sample rate can be determined it can be used to chunk data with different sample rates and then they can be resampled inside of the filters to match.

8.3 Compression

The file size resulted after running the data through our Rust implementation was noticeably larger, about 7 times larger than the original file.

The reason for the resulting large file size is due to the way the HDF5 data format attributes overhead. Because we write in slices, the HDF5 file allocates for an incoming chunk size extra blocks and it does this for every chunk, which leads to the total file size growing very large very fast.

A fix to this problem is to implement file compression. File compression will allow us to have smaller output file sizes. HDF5 has a library called “zlib” that would allow us to compress the file. We researched how to implement compression, but due to time constraints we were not able to provide a fully tested version of the project with compression, but in future implementations, having file compression would greatly improve the project.

8.4 Python Binding

PyO3 was used to do Python binding for Rust. When doing Python binding, an error was found: the trait ‘PyClass’ implemented for ‘hdf5::Dataset’. This means PyO3 doesn’t support hdf5 Dataset type. The trait ‘PyClass’ is needed to be implemented for hdf5 Dataset type. Since we can’t implement any traits for any types outside of the rust hdf5 crate, PyO3 can’t be used to do Python binding for Rust.
We found a solution to this problem with the new-type pattern of Rust but we ran into time constraints trying to implement it.

Chapter 9

Conclusion

In this chapter, we will review the results of our project and reflect on the work that we have accomplished.

9.1 Evaluation

The core of the functional requirements that were categorized as MUST were implemented, but we did not have enough time/ran into some complexity issues to be able to implement some MAYBE requirements.
Some time was lost to re-do the planning of the project in the early stages, but the project development became more organized thanks to this. We have learned a lot from this project about professional workflow and will plan a lot more and explicitly in the future.

- The core of the functional requirements that were MUST were implemented.
- We didn't have enough time/ran into some complexity issues to implement some MAYBE requirements.
- We lost some time by re-doing the planning but the project became more organized
- We have learned a lot about a more professional workflow and will plan a lot more explicitly in the future

9.2 Results

9.2.1 Speed

The speed of the Rust implementation was a bit faster than the C++ implementation, but not because of the language. In our implementation when the first chunk is split, it immediately starts to get processed, while in the C++ implementation we think it waits for the entire file to be split into chunks before being processed.
In terms of raw speed, Rust and C++ are similar in speed.

9.2.2 Extendability

- Hard to determine given that we don't have access to the C++ project and extendability can be highly project specific
- The lack of inheritance did not significantly affect our progress
- The trait system allowed us to write easily extensible code that was unaware of the actual implementation of the trait

9.2.3 Testing

- From our experience with C++ we would say that rust testing is easier since it is built into the language and into cargo making setting up tests not necessary.
- Also the integration with the documentation is really beneficial

9.2.4 Experience with Rust and final conclusion

Working with Rust was a nice experience. It is a modern programming language that offers some advantages over C++ mainly enhanced thread safety, and better concurrency support. The documentation, although not as good as the C++ one, is very thorough on the standard libraries, but the documentation for the HDF5 library was lackluster.

The syntax is a bit verbose but it offers strong and flexible error handling. In Rust, a function can not throw an exception that gets handled later. Instead a function has to define its output type to be a Result. A Result has 2 variants: an “Ok” variant that contains the successful output or an “Err” variant that contains the error that occurred. This makes it very clear to the caller of a function that the function can error. It also makes the code more verbose. However, a combination of pattern matching and the question mark operator help to lessen this. For errors that cannot be handled, Rust provides panicking. When a function panics, it stops execution. Panicking is usually reserved for programming errors.
After working on this project, we believe that Rust is a viable alternative to C++ as it offers better concurrency support and thread safety, the documentation is solid, but keep in mind that the HDF5 support in Rust is lacking.

Appendices

Appendix A - Pictures

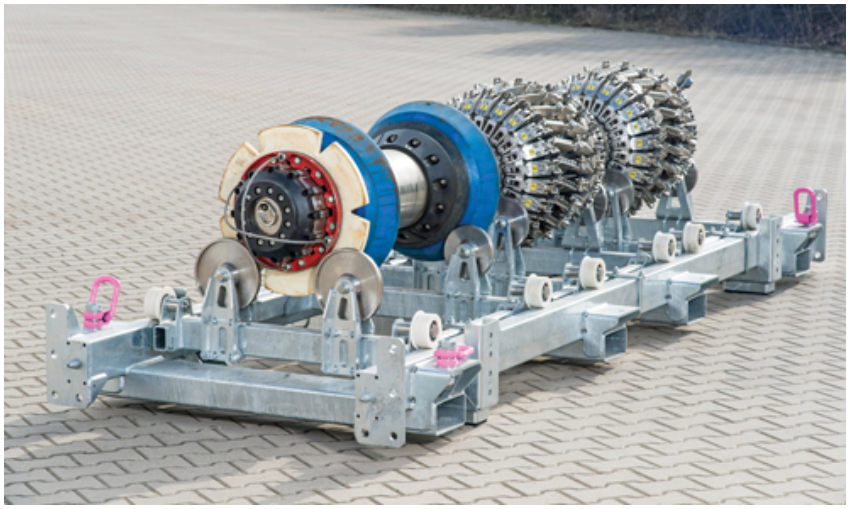


Figure 1. A ‘PIG’ above ground

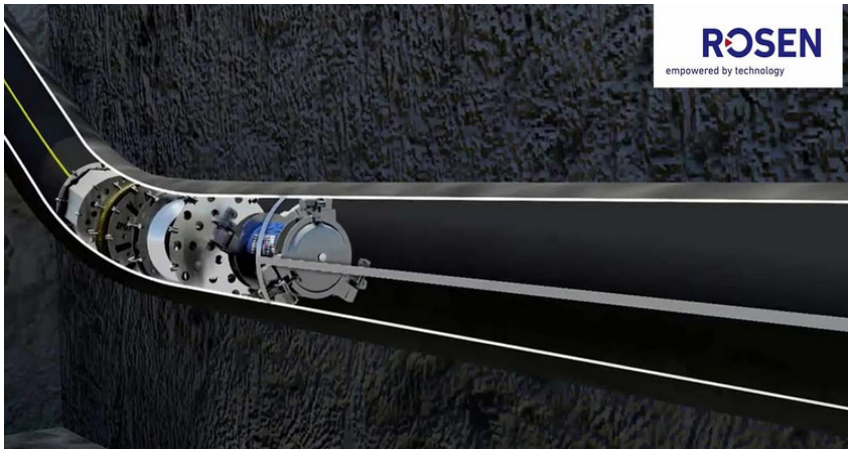


Figure 2. A ‘PIG’ in a pipe

Appendix B - Diagrams

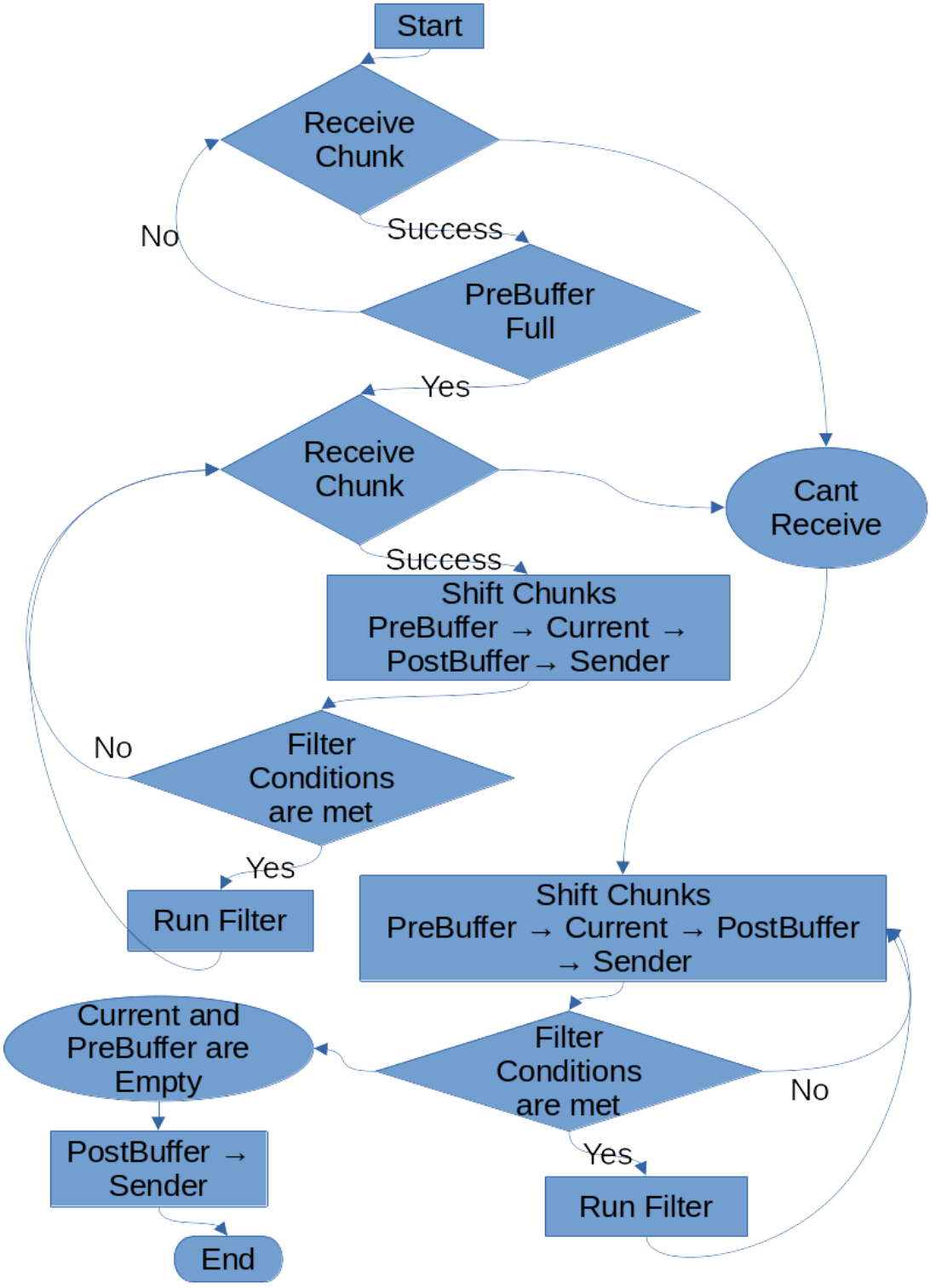


Diagram 1. flow chart for the functioning threads

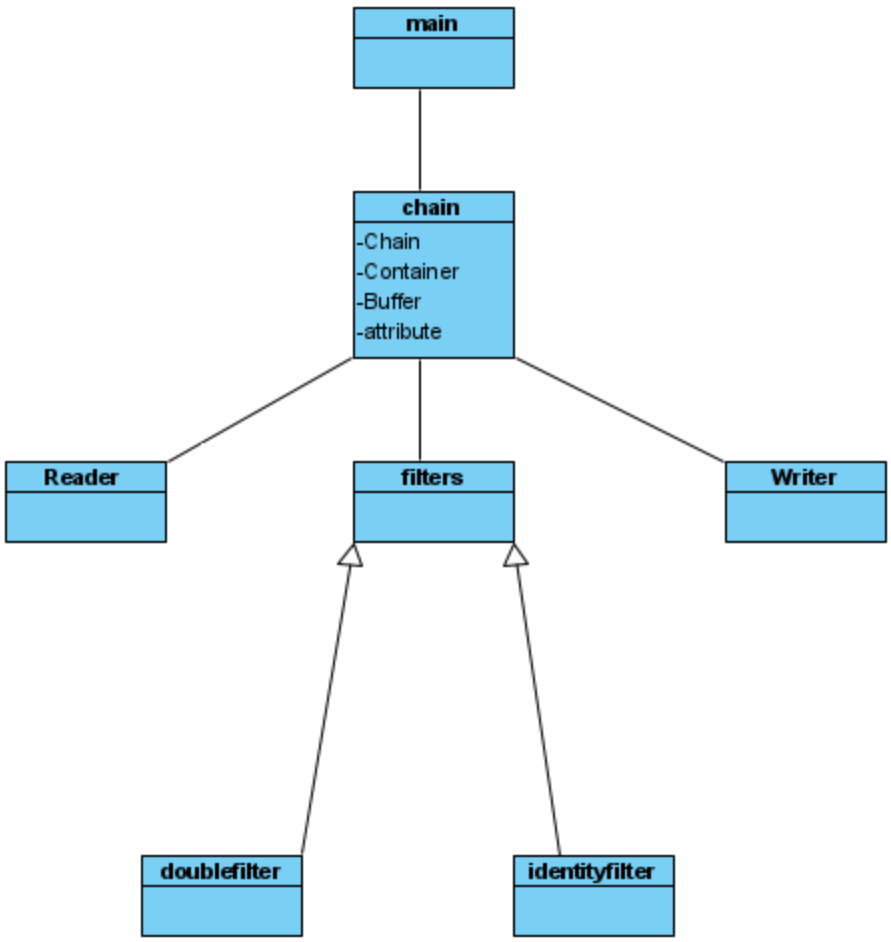
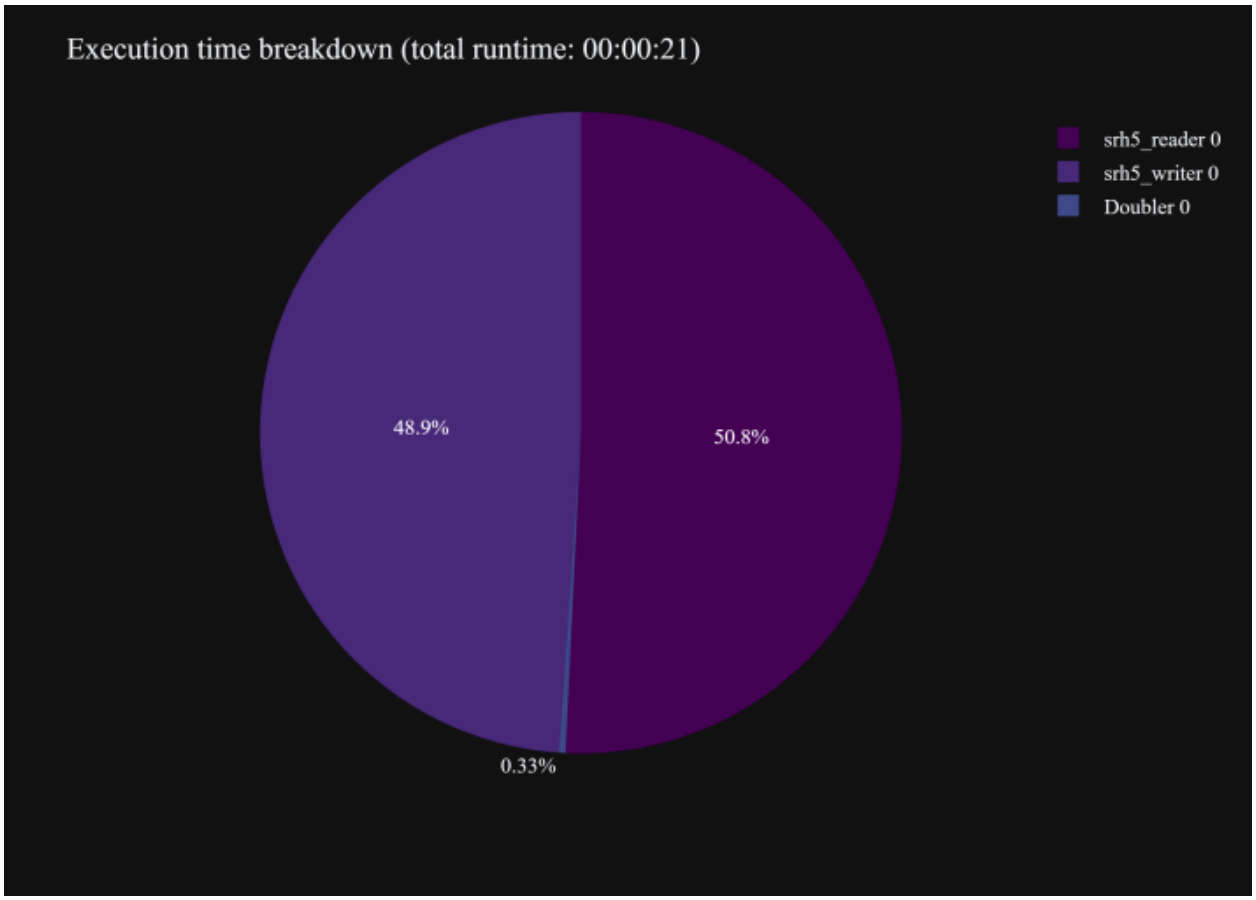
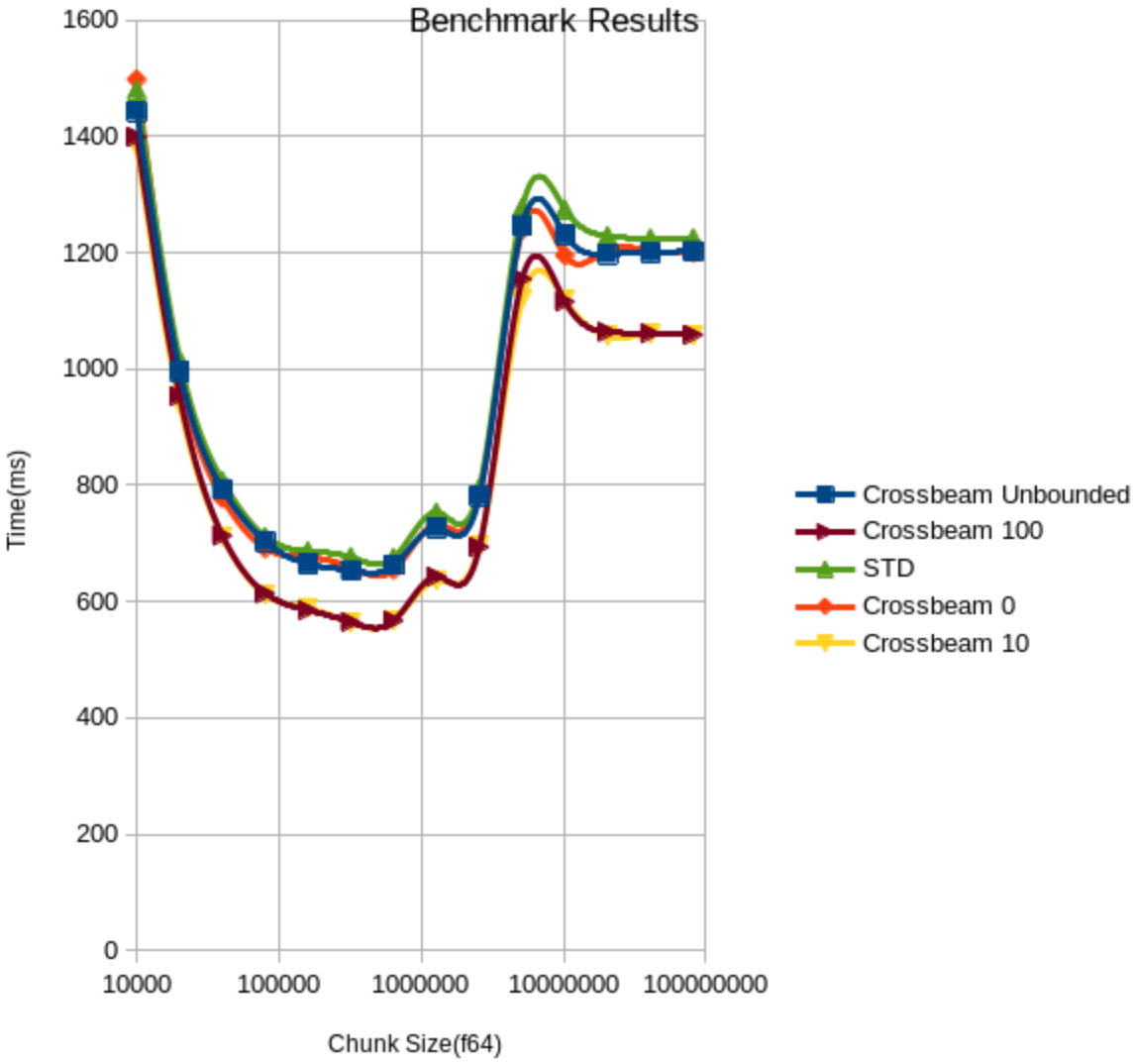


Diagram 2. General class Hierarchy Diagram

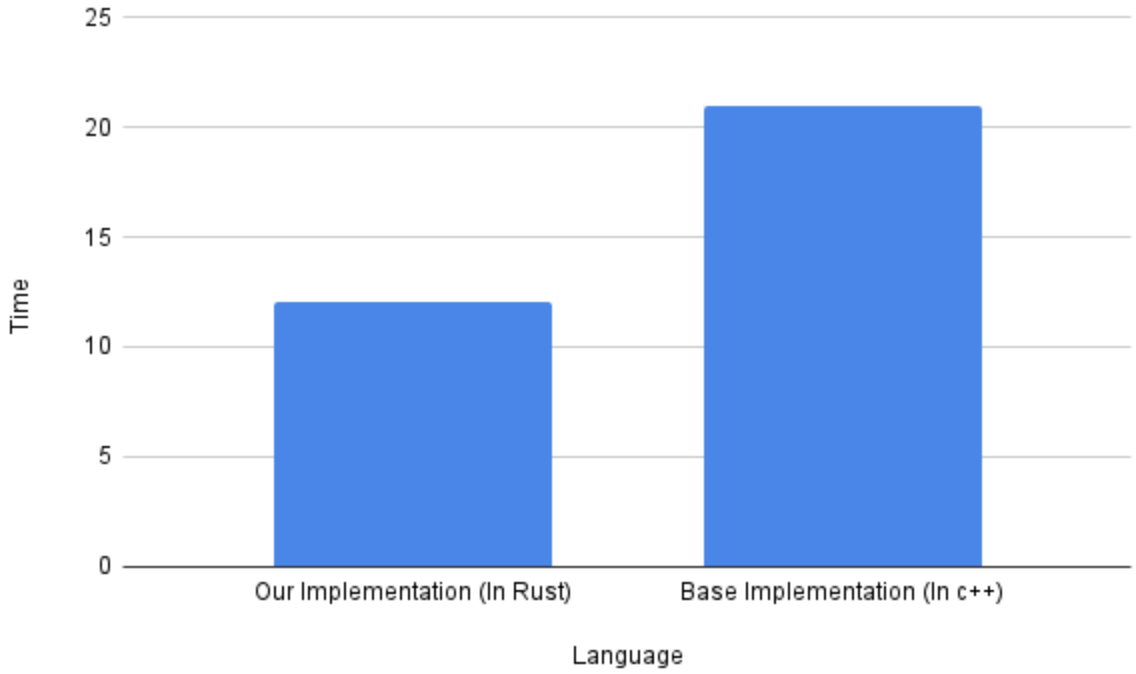
Appendix C - Results



Result 1. Benchmarking of the C++ code (done by Rosen)



Result 2. Result of the CrossBeam vs Rust std comparison



Result 3. Result of comparison of speed between our implementation and the control implementation.