Masterarbeit

Message passing safety and correctness checks at compile time using Rust

vorgelegt von

Michael Blesel

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6443269
Erstgutachter:	Jun.-Prof. Dr. Michael Kuhn
Zweitgutachter:	Prof. Dr. Thomas Ludwig
Betreuer:	Jun.-Prof. Dr. Michael Kuhn, Jannek Squar

Hamburg, 2021-02-15

Abstract

Message passing is the de facto standard for large scale parallelization of applications in the domain of high performance computing. With it comes a lot of complexity and possible parallelization errors. Traditional programming languages like *C* and *Fortran* and the message passing interface MPI do not provide many compile time correctness checks for such parallel applications. This thesis shows the design and implementation of a new message passing library using the Rust programming language.

Rust focuses heavily on memory safety features and rigorous compile time correctness checks. The thesis explores whether a message passing library can be designed in a way that utilizes these features to provide an easier experience for developers of HPC software by conducting better compile time correctness checks.

Many possible errors in MPI software are related to unsafe memory usage. Rust's memory *ownership* concept can be applied to message passing operations to provide better error protection. Furthermore modern language features like generic programming can be used to achieve a more convenient and safer interface for message passing functions.

The thesis demonstrates that some common erroneous code patterns of message passing with MPI can be avoided with this approach. It however also shows that particular error classes regarding the correctness of the communication schemes of message passing applications require further correctness checking tools, such as static analysis or compiler modifications.

Contents

1 Introduction

In the modern world of software development parallelization is an essential feature for getting access to all the computational power that a system can provide. Hardware has evolved in a way where multi-core CPUs are the standard and therefore computationally intensive software has to be parallelized to yield good performance. Some applications like simulations or other scientific software however need even more resources than a single machine. This is the domain of *high performance computing* (HPC), where applications are run on hundreds of computing nodes and can require terabytes of memory. Parallelization on such a scale requires different methods of parallelization than normal multi-threading.

A common method for parallelization over multiple computing nodes is *message passing*. Instead of sharing data via the local memory of a system it is exchanged with messages that are transmitted over a network between the processes of an application. Parallelization and especially message passing introduces new categories of possible errors into programs. For programmers it also increases the complexity of designing and writing software immensely. It is therefore important to provide programming interfaces and tools to developers that can decrease the likelihood of errors and help to detect them during development.

The most common technologies that are used in HPC today are the *Message Passing Interface* MPI in combination with *C*, *C++* or *Fortran* code. This thesis explores if a more modern system programming language like Rust can be used to develop a message passing library that provides better safety features and compile time correctness checks than the existing solutions do.

Chapter 2 gives a short introduction for distributed computing, message passing and the Rust programming language. These are the basic topics that this thesis builds upon and readers might not be too familiar with some of them. Chapter 3 gives a more detailed explanation of the motivations behind my work. It analyzes what the important aspects for HPC programming languages and tools are and discusses where the current solutions for message passing applications are lacking in this regard. Chapter 4 then goes on with specific examples of problems and possible errors with MPI and proposes possible solutions.

Chapter 5 is the core part of the thesis. It discusses the design ideas and implementation details for my Rust message passing library. It will highlight design decisions that lead to better compile time correctness checks for the message passing operations compared to MPI. It will also explore some more unique ideas for message passing there.

Chapter 7 performs a direct comparison between my message passing implementation and MPI by implementing a calculation heavy parallel application with both libraries and comparing the performance results. This chapter will also discuss some of the problems and peculiarities with my implementation and evaluate their significance for the practical use of the library.

Chapter 8 will give an outlook on the work that is left to be done and also some more speculative ideas for the future. Finally Chapter 9 will recapitulate which of the set goals for this thesis could be accomplished and what problems have revealed themselves.

2 Background

This chapter will go over the basics of the two main topics that are covered in this thesis, which are *distributed computing and message passing* and the *Rust programming language*. Both of these topics can of course not be covered completely but I will give a comprehensive overview of the two domains and highlight the aspects that are important for the reader to follow the rest of this thesis.

2.1 Distributed computing and message passing

The term *distributed computing* is not completely well defined and could be understood in many different ways depending on the context and the domain that it is used in. [AW04] defines a *distributed system* as: "a collection of individual computing devices that can communicate with each other". *Distributed computing* is the act of running connected software on such a system. This is still a very broad definition and could refer to many scenarios such as:

- Multiple machines communicating via the internet
- A supercomputer with many identical computing nodes all connected via a local network
- A dynamic conglomerate of machines with widely different architectures such as desktop machines, smart phones and computing clusters spread around the world and communicating via the internet, where machines may drop out or connect at any time

An important attribute that is shared by all these scenarios is the fact that they are all distributed memory systems.

Figure 2.1 shows an example architecture of a distributed system with four computing nodes that are fully connected to each other via a network. Since each computing node is a separate machine they all have their own local memory that cannot be accessed from another node. If we want a parallel application to run on all nodes there needs to be a mechanism for sharing data between the nodes.

In a traditional parallel application that only runs on one machine threads would be used for parallelization and they would be able to access the same memory, which makes data sharing simpler. In the case of a distributed parallel application this is not possible

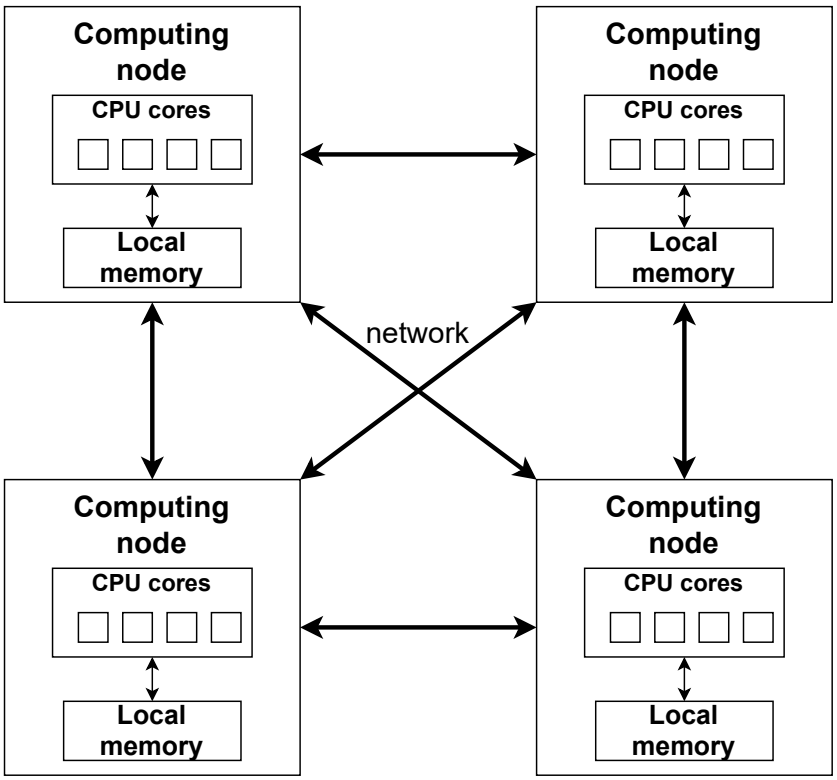

Figure 2.1: Visualization of a distributed system with 4 computing nodes that are 'all-to-all' connected via a network

and there is need for a mechanism to share the data from the local memory of each node with the other nodes. In this scenario if one node needs access to data that is stored on a different node it has to be sent over the network. In the context of distributed computing a common method for this is called *message passing* and it will be discussed in more detail in Section 2.1.1. There are other methodologies for an application to gain access to the resources of multiple computing nodes like *single system images* [BCJ01], where an abstraction layer is added between the applications and the cluster of nodes, but this thesis focuses on *message passing*.

Distributed computing appears in a lot of different domains and applications ranging from web-applications, database-systems and file systems to scientific calculations and simulations. The thesis will mostly focus on the use case of distributed parallel applications on local computing clusters in a high-performance-computing (HPC) environment. Therefore my definition of distributed computing is focused on the following attributes.

- Multiple processes belonging to the same application running on multiple computing nodes in a local computing cluster
- Using message passing to share data between the processes

2.1.1 Message passing

In the context of distributed computing the term 'message passing' refers to multiple processes of the same application communicating data over a channel. Semantically it does not matter here if the processes are running on different computing nodes or not but message passing is mostly meant to be used in that way. As discussed in the last section this method of sharing data becomes necessary when different parts of a program do not have a shared memory section and therefore do not have direct access to each other's data. The used communication channel can differ based on the context and implementation of the distributed program but for simplicity's sake we can for now imagine it as some kind of network connection.

In Figure 2.2 the basic concept behind the message passing method is shown. We see a visualization of two processes and segments of their associated local memory sections. In this example the process p1 wants to send the data stored at its local memory address d to the process p2 which wants to store the received data at address j in its local memory. In the middle of the graphic we can see the fundamental functionalities that need to be implemented to create a message passing system.

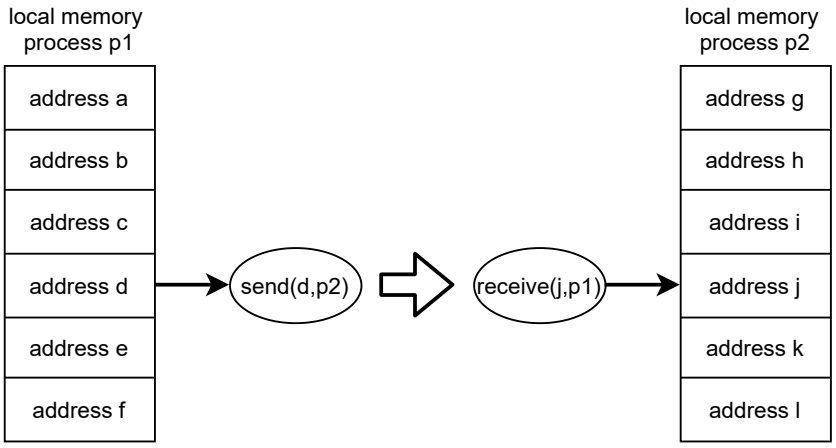

Figure 2.2: message passing example

- **Network connection**: To get any kind of communication between the two processes they have to be able to 'find' each other and create some kind of network connection. This problem can of course be solved in many different ways but the most standard approach would be to have a collective initialization phase at the beginning of each process. In this phase all started processes are registered and the necessary network information is shared between them so that they are able to establish network connections to each other.

  This can be implemented by having an externally running management process for the message passing library running, to which all processes of the application connect at first to disclose their network information. This approach is static with respect to the number of running processes for the distributed application. This

means that it has to be known at the start of the program how many processes will participate and they all need to be active before the initialization phase can complete.

For a more dynamic approach to the number of participating processes in the message passing this problem gets more complicated. If the application is for example supposed to be able to handle the emergence of additional processes during runtime the management process needs to have a more active role during the program execution to register newly connected processes after the program execution has already begun. For the purposes of this thesis we will work with the static version of choosing the number of participating processes before the start of the programs execution, but the subject of handling dynamically growing or shrinking numbers of participating processes is also noteworthy.

- **Send and receive functionality**: To get a message across between two processes the most basic functions that need to exist are `send` and `receive`. In our example we can see that the process p1 calls a send function. This function at the bare minimum needs to take two arguments. The first is the location of the data that should be sent and the second is the identification of the target of the communication, which in this case is the process p2.

  As already established the two processes at this point have the necessary information about what other processes exist and how to establish a network connection to them. Therefore the send function can now create a network packet containing the data referred to by the local memory address d and send it over the network to the process p2.

  For a successful message passing operation a send call always has to be matched by a corresponding receive call from the target process that accepts the message. In our example the process p2 calls the receive function with the first argument being the address at which the sent data is supposed to be stored and an identification of the source process of the message as the second argument. In the receive call the sent network packet from p1 is processed and the received data is stored at the local memory address j of the process p2.

  The message passing process illustrated here is still quite simplified and ignores some problems that can occur like **1.** How to assure that multiple consecutive messages from and to the same processes are handled in the right chronological order with possible network delays, **2.** How to ensure that the sent and received data types match and **3.** General error handling of the network connection. Details about how these problems can be recognized and solved will be discussed in the implementation based chapters of this thesis. This chapter's purpose is to lay down the fundamental ideas behind message passing so that the more concrete ideas and implementations in the following chapters can be better followed.

We have now seen the basic building blocks of a message passing library. Based on the send and receive functions a multitude of more complex message passing operations can be constructed that rely on those fundamentals.

While the basic concept of send and receive operations is quite intuitive they can be implemented with many different semantics which can have a big influence on the correctness and performance of distributed programs. These semantics are mainly about in which state of the send or receive operation the corresponding function call returns and the process continues its following instructions. The following three behaviours for send and receive operations are important to understand when thinking about correctness in message passing programs because using them incorrectly can easily lead to problems like deadlocks or memory corruptions in a distributed application.

- **Message buffering**: When using a send operation there are multiple ways of how and when the data should be sent to the target process. The most simple way would be to start the transmission as soon as the target process posts a receive operation. In this case the source process sends its data directly to the target where it is immediately stored at the target memory address.

  There is however an alternative to this where the message passing library contains its own internal buffer storage. In that case the data from the send operation would at first be cached by the message passing library and it would not be strictly necessary to wait for the target process to post the matching receive operation before the send operation could start transmitting its data. The buffering of messages has the advantage of making the local memory region that holds the send data be reusable more quickly.

  This is important because to ensure program correctness it must always be guaranteed that the data that is being sent will arrive in the same state that it was in when the send function was called. Therefore the local data buffer of the send data must be protected from any modifications until the send operation has transmitted the buffer. When using the buffering method this can always be done immediately and the correct state of the data will be cached by the message passing library, which makes the local data buffer free to be modified again.

  Message buffering by itself does not guarantee any correctness or performance increases but it is used as a tool by the following two semantics that might need to make use of it in their implementations.

- **Blocking and non-blocking operations**: The concept of blocking operations is about at which point a call to a send or receive function will return in the code.

  Calling a blocking operation guarantees that the local data buffer used by the operation is safe to be used again after the function call has returned. For a send this means that the correct state of the data has been transmitted. For a receive operation it means that the local data buffer now contains all of the data that it was supposed to receive. This is of course crucial for the correctness of the

program, because writing to or reading from an unsafe buffer can lead to undefined behaviour in the program. Especially in parallel applications the outcome can become completely nondeterministic.

The most straightforward implementation of blocking semantics is to only return from a send call after all data has been received by the target process and for the target process to only return from its receive function after all transmitted data has been fully copied into the local data buffer. However this method can limit the performance of the program immensely. If for example a huge amount of data has to be transmitted between two processes but neither the send nor the receive data buffers have to be touched by the program for a good amount of time while other unrelated computations take place it would be wasteful to wait for the message passing operations to finish before those unrelated calculations can take place.

In cases like this a non-blocking communication mode can be used. When a non-blocking message passing operation is called it may return before the local data buffer is safe to be used again. This behaviour would let the previously described example application continue with its calculations while the big data transmission takes place in the background. However having access to unsafe data buffers can lead to a variety of errors and undefined behaviour. Therefore a method needs to be in place for the application to check when the local data buffers are safe to use again.

How to solve this problem is highly implementation specific but in the general case the non-blocking operations need to return some kind of a handle that can be queried for the current status of the buffer. This status then needs to be checked by the program before attempting to access the data buffer again and access can only be allowed as soon as the message status says that the buffer is safe again.

Implementations that have these semantics, especially the blocking case, can make use of the aforementioned buffering method to make the local data buffers safe for use again more quickly.

- **Synchronous and asynchronous operations**: While the blocking/non-blocking semantics are mainly concerned about the state of the local data buffers they do not make any assertions about the state of the transmitted message. When for example a blocking send operation returns, this does not necessarily mean that the target process has also received the message at that time. This is what the synchronous/asynchronous semantics are about.

  Using a synchronous send operation guarantees that it will not return before the matching receive operation has been posted and the message is actually being sent to its target. Whether the send will only return after the complete message was sent or as soon as the target has started receiving the message is based on the implementation but the important fact is that the send and receive operation pair has 'found' each other. This behaviour can be important for the synchronization of parallel programs but there are also cases where it is explicitly unwanted.

As an example there might be a distributed application where one process knows from the start that at it will receive a specific message at one point in the future but it is unclear when this message will occur. In this case the use of synchronous communication means that the process would have to wait in the receive operation until the awaited message arrived and would in the meanwhile not be able to do any other work.

For scenarios like this the asynchronous message passing semantic exists. An asynchronous send or receive operation will not wait for its counterpart on the target process to be posted but will return as soon as possible so that the current process can proceed with its computations while the message passing operation runs in the background. This behaviour has similarities to a non-blocking operation but as explained the return times of the two depend on different conditions.

These described semantics can be mixed and combined with each other to create different behaviours. An asynchronous, non-blocking send operation for example will return very early but neither guarantee the safety of the local data buffer nor the existence of a matching receive operation from the target process.

It is also possible to create an asynchronous, blocking send operation that after its return leaves the local send data buffer in a safe state since the underlying implementation uses message buffering and returns only after the original state of the send buffer has been cached by the message passing library.

A deeper discussion of how to implement these semantics and how they can impact the correctness of a distributed program will follow in the later chapters of this thesis.

2.1.2 High performance computing

As already mentioned parallel programs on distributed memory systems can occur in many different contexts. This thesis will mainly focus on their uses and requirements in the context of *high performance computing*. Therefore a short introduction to the field of high performance computing, which will from now on be referenced to by the abbreviation HPC, follows.

HPC can be seen as the study of *supercomputers*. The term supercomputer generally refers to a distributed memory system that consists of a high number of computing nodes, which are located together in a computing center. These massive systems contain millions of CPU cores, petabytes of memory and can achieve performance levels of multiple hundreds of PFLOP/s. Most publicly know supercomputers today are ranked by their performance in the so-called TOP500 list [TOP20] and as of writing this thesis the number one spot on that list is taken by the so-called *Supercomputer Fugaku* that is located in Kobe, Japan. This machine has over 7 million CPU cores and can achieve benchmark performances of over 400,000 TFLOP/s.

In the field of HPC supercomputers are used for a variety of huge parallelized applications like climate and weather simulations or aerodynamic simulations for cars and airplanes. These applications all have in common that they solve highly complex problems which need an extreme amount of computing power and memory. This makes it impossible to run them on shared memory systems and leads to the fact that there is a big demand for efficient software solutions to facilitate distributed computing.

Parallel applications running on these systems require the ability to communicate between the computing nodes to allow the sharing of data between their processes. This is mostly achieved by using implementations of the *message passing* mechanics explained in Section 2.1.1.

The *de facto* standard message passing implementation in HPC is the so-called *message passing interface* MPI [For21]. MPI by itself does not refer to a specific software implementation but to a standardized specification of a message passing interface that provides a lot of useful message passing functionalities. It was first conceived in 1993 by a group of domain experts in parallel computing who had the goal of creating a unified interface for message passing for distributed memory systems. Benefits of such a standard are for example better portability of software between different supercomputers and a unified infrastructure for message passing developers and hardware companies to work together at improving the HPC landscape.

Today many different implementations of the MPI standard exist, ranging from open-source projects to proprietary solutions. But as long as the MPI implementations and the MPI applications are standard compliant they can be combined at will and they should behave the same semantically. MPI officially offers bindings for the two programming languages *Fortran* and *C* which dominate the current HPC landscape. It can additionally be used in languages like *C++*, which can be compatible with *C*, and other languages that offer MPI wrapper libraries around the original *C* bindings.

To the current day MPI is still the predominantly used library for parallelization in HPC but since the conception of MPI the world of HPC has changed a lot, with on the one hand large developments and changes in the existing hardware and on the other hand the emergence of new kinds of HPC applications that bring with them different requirements and priorities.

On the hardware side it has for a long time been the case that the increase in performance of single-core CPUs is growing ever slower because of physical limitations like the speed of light and problems with heat dissipation. This has lead to the modern multi-core CPU architectures we are familiar with today and their numbers of cores are still steadily growing from generation to generation. This development has brought new demands for parallel programs in a HPC setting. When working with single-core architectures an application purely using message passing for its parallelization is ideal but when this concept is transferred to multi-core systems a lot of potential computing power is either wasted because there is still only one process run per CPU or the message passing method has to also be used for running multiple processes on the same computing node.

This is not ideal because intra-node parallelization is working on a shared memory system and therefore does not necessarily require methods like message passing. In this scenario it is often preferable to use multi-threading inside the computing nodes instead of message passing because it can generally net a better performance. Since nearly all modern supercomputers today use multi-core CPU architectures this has lead to the fact that modern HPC applications have to mix the two parallelization methods of message passing and multi-threading in what is often called a hybrid parallelization style.

The other big change to the HPC landscape over the last decades is the increasing adoption of GPU computing for tasks like machine learning and other big-data tasks. This brings its own set of challenges for HPC applications on how to scale these kinds of applications over big GPU computing clusters but it is not a topic that will be discussed in this thesis.

Overall HPC today is still mostly dominated by the languages *C* and *Fortran* in applications that predominantly use MPI for parallelization and also, in the case of more modern applications, multi-threading parallelization. This is mainly because these languages have stood the test of time and also because many HPC applications contain a lot of legacy code that has been in use for many years.

However we can also see the emergence of more modern technologies and programming languages being introduced. This thesis explores the topic of whether a modern systems programming language like *Rust* can bring any benefits for message passing applications compared to the traditional languages.

2.2 The Rust programming language

Rust is a strongly typed, compiled, systems programming language initially developed by Graydon Hoare. It is a modern take on low-level programming languages like *C* or *C++* that promises to deliver good performance but also comes with an emphasis on *safety*. Foremost the topic of memory safety is tackled by *Rust*, which can be a great concern in *C/C++*. How exactly this is handled will be discussed later in this chapter.

While this thesis is not the place to give an extensive explanation of the *Rust* programming language the key concepts that make it suitable for implementing a message passing library are highlighted in this section.

On the official Rust website [Rus21] it is stated that the three main principles guiding the development of Rust are **performance**, **reliability** and **productivity**. The following will shortly go over how each of them manifests itself in Rust and point out how they can be beneficial in a HPC environment.

- **Performance**: In regards to performance Rust promises to be comparable to languages like *C* that are generally considered to deliver good performance results out of the box. Performance is of course a very complex topic and especially a

general performance comparison between two languages can prove to be difficult in practice because the results are highly dependent on the specific programs that are being compared and other outside influences.

However when talking about the performance of a whole programming language it is mostly about analyzing how efficiently the abstract concepts from the source code are translated into machine code. In programming languages like *C++* the term 'zero-cost abstractions' is often used in this context. This term was described by the original inventor of *C++* Bjarne Stroustrup with the quote: "What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better."[Str95]

This means that when using the more complex abstraction features provided by the language they still get compiled into the same concise machine code that would have been generated if the developer would have implemented the feature themselves using the low-level primitives of the language. Broadly speaking the language promises the developers that using its 'nice' features that offer better usability and readability will also yield the best possible performance. Similar to *C++* this is also one of Rust's performance features.

In the context of this thesis the performance of Rust code is quite important since we are looking at its application in a HPC environment where good performance results are required of the parallel programs run on supercomputers. This is firstly because the problems that HPC applications try to solve are often very complex and already time intensive even with the most optimal implementations and secondly because lowering the runtime of large scale parallel applications on a cluster can save substantial amounts of money that would otherwise be spent on powering the system. If Rust wants to be a suitable choice for HPC it needs to fulfill comparable performance requirements to *C* and *Fortran*.

- **Reliability**: The Rust website defines Rust's most important reliability features as *memory-safety* and *thread-safety*.

  Memory safety is a big concern for programming languages. It is one of the most frequently occurring errors in *C* programs [BSD05] and can lead to program crashes or undefined program behaviour. While of course being avoidable it is easy to occur in *C* since there are not many protections in place that either prohibit unsafe memory accesses or catch them at compile time. Rust has very extensive measures in place to prohibit unsafe memory accesses and to detect them at compile time. This feature of Rust will be key in the later described message passing implementations and a more detailed explanation is given in Section 2.2.1.

  Thread-safety is concerned with the topic of possible errors in concurrent, multi-threaded code. Out of the box Rust comes with multiple concurrency functionalities for writing safe multi-threaded code and the compiler is able to detect a multitude of common concurrency errors like data-races. Since this thesis is concentrating on the message passing aspect of parallel applications the details about this are

not covered here. Nonetheless it is a beneficial feature for HPC applications since, as already discussed, a hybrid parallelization of mixing message passing with multi-threading is common in today's HPC applications.

- **Productivity**: Productivity is a more abstract concept than performance and reliability. It is concerned with how the language Rust and its associated tools can make the task of software development easier for the programmers and save development time.

  Rust tries to achieve this by providing a comprehensive standard library that provides solutions for many frequently occurring tasks in programming. This is however not unique to Rust and most other languages feature comparable standard libraries. Where Rust sets itself more apart from languages like *C* and *C++* is in the tools that accompany it.

  Firstly, the Rust compiler highly emphasizes the output of clear and concise error and warning messages. This can be a problem in other languages where it can often be very time intensive to find the actual origin of a compilation error.

  Another productivity feature of Rust is *Cargo*. Cargo is a full build system and package manager for Rust that bundles all the infrastructure that is needed to build and distribute software packages. It is for one a wrapper around the Rust compiler that manages external dependencies of programs and automatically provides different build templates like *Debug* and *Release* builds as well as the infrastructure for unit testing. Additionally it makes it easy to share Rust software in the form of *Crates*. A crate is a a specific form of a Rust project that is automatically created by Cargo when creating a new project. Rust crates can then be uploaded to the accompanying website "crates.io" which is the central hub for all published Rust software. Using this infrastructure it is quite easy to include external dependencies in a Rust project by simply adding their names and wanted version to the local Cargo configuration file, which will then make Cargo automatically download and include them in the build process without any more manual work for the developers.

  Especially the easy management of dependencies could be a great feature for HPC. Many complex HPC applications bring in a lot of dependencies and it can be rather difficult to get the programs to build correctly on different clusters. The integrated build and dependency system of Cargo could save a lot of time here and make it easier to share HPC applications between different data centers. This would however require the whole software stack of the application to be written in Rust, or at least the existence of Rust wrappers for the most commonly used libraries in HPC.

```
1 {
2     // Memory allocation
3     let s = String::from("I am a String on the heap.");
4
5     // Valid access of s
6     println!("{}", s);
7 } // Scope of s ends here and the memory is freed
```

Listing 2.1: Example of Rust's ownership model

2.2.1 Rust's memory-safety mechanisms

Rust's memory management is from the ground up designed with safety in mind and it is built around a key concept of the language that is called *Ownership*. This feature is quite unique to Rust and it is therefore explained here what implications the ownership model has for Rust code.

One of the main problems with memory management in other languages like *C* is that once memory has been allocated on the heap it also has to be manually freed again later in the program to avoid memory leaks that will over time fill the memory of the process with old data that never gets removed until the program is terminated. This problem can get quite complicated as the program complexity increases and its control flow features a high number of possible branches. It has to be ensured that an allocated heap variable is always freed exactly once after the program's last access on it. If it is not freed a memory leak arises but if it is freed too early and the program tries to access an already freed memory region it can lead to crashes like *segmentation faults* or undefined behaviour.

One way to solve this specific memory problem is to implement *garbage collection* for your language. With this concept manually freeing no longer used memory is not necessary anymore because the garbage collector monitors all allocated memory during the program's runtime and in regular intervals frees all memory that has no more active references to it. This solves the problems but it also has the drawback of runtime overhead that can impact the program's performance negatively. Rust does not feature a garbage collector but approaches this problem differently.

In Rust every heap variable has one dedicated *owner*. The allocated memory lives as long as the owner variable is in scope and gets automatically freed after that. Listing 2.1 shows the basic idea of the ownership model. First a `String` variable is created and thereby allocates memory on the heap. The variable `s` is now the explicit owner of that memory and if `s` goes out of scope the memory will be freed.

As we can see no manual call to a `free` function is necessary but the memory deterministically gets freed at the end of the scope of `s` without the need of any runtime garbage collection. The most important fact here is that all ownership rules can be enforced at

```
1  fn main()
2  {
3      let outer_scope: String;
4      {
5          let inner_scope = String::from("I am a String");
6          println!("{}", inner_scope);
7          // Ownership of the memory gets moved to outer_scope
8          outer_scope = inner_scope;
9          // Compilation error!
10         println!("{}", inner_scope);
11     }
12     println!("{}", outer_scope);
13 }
```

Listing 2.2: Example of moving ownership

compile time, meaning that possible errors are caught by the compiler and do not appear only at runtime.

Of course the lifetime of heap variables in Rust is not limited to only one scope as we have seen in the last example. To allow heap variables to outlive the scope they were allocated in, it is possible to transfer the ownership to other variables. In Listing 2.2 we can see an example of this. The data of the String is first owned by the variable `inner_scope` and the same ownership rules as in the last example apply.

This time however the ownership of the string is moved to the variable `outer_scope` before the end of the inner scope. From an intuitive perspective coming from other languages it might seem that now both of the variables `inner_scope` and `outer_scope` reference the string's data on the heap but in Rust this is not the case. It is essential that there can always be just one owner of the string's data in Rust and therefore the variable `inner_scope` becomes invalid immediately after it is assigned to `outer_scope`. The ownership is explicitly moved to `outer_scope` and `inner_scope` can no longer be accessed. If something, for example the second print statement, tries to access the old owner of the string it causes a compilation error.

We have now seen how the ownership model works. By enforcing that there can at all times just be one owner of allocated data the Rust compiler is empowered to detect all kinds errors stemming from memory management.

One last Rust specific feature related to memory management that needs to be shown is how the ownership model can coexist with passing references to heap variables around in the program. Rust has some specific restrictions on the type and number of references that can exist for a variable to still be able to guarantee memory-safety. The language specifically distinguishes between *mutable* and *immutable* references. In Listing 2.3 both kinds are used.

```
1  fn modify_string(s: &mut String)
2  {
3      // Appending to s
4      s.push_str(" and I was modified");
5  }
6
7  fn main()
8  {
9      let mut s = String::from("I am a String");
10     // Passing a mutable reference to s to the function
11     modify_string(&mut s);
12     // Creating a immutable reference to s
13     let s_ref = &s;
14     println!("{}", s_ref);
15 }
```

Listing 2.3: Example of references

In this example the main function creates a string and then passes a mutable reference to it to another function where the contents of the string are modified. As we can see the `modify_string` function has to explicitly state that it expects a mutable string reference as argument. When it is called in `main` it is again necessary to explicitly pass a mutable reference in the form of `&mut s`. This syntax comes from the fact that in Rust all variables are immutable by default and it always has to be stated by the programmer when a variable is supposed to be mutable. Enforcing this gives the compiler the necessary information to detect possible errors and also has beneficial influence on its ability to create optimized machine code because it is always known whether a modification of a variable is possible at some point.

After the string has been modified in the example code an immutable reference to `s` is created with the `&s` syntax and its content gets printed.

Rust has special a restriction for the coexistence of multiple references to the same variable. There can always **either** be one mutable reference to it or an arbitrary number of immutable ones. This restriction allows the detection of possible data races at compile time because it is not possible for there to be a scenario where in one part of the code an immutable reference to a variable exists whose contents might be changed at any time from another part of the code where a mutable reference to the same data is active.

The rest of this thesis will explore how these memory-safety mechanisms of Rust can be used in the context of message passing and if it is possible to use them to mitigate possible error sources in the development of distributed parallel programs.

3 Motivation

This chapter will lay out the reasons that led me to think that the Rust programming language would be a good fit for the field of HPC. It will mainly focus on its potential benefits for a message passing library implementation but it will also briefly explore how the design principles of Rust and especially its accompanying tools make it suitable for the requirements of a HPC programming language.

3.1 What makes a programming language suitable for HPC?

Before we can discuss why Rust might be a good choice of language for HPC applications we first need to define more closely what the desirable features of a such a language are.

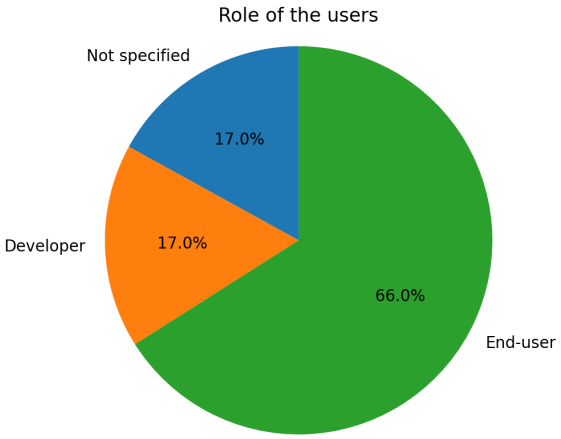

Figure 3.1: What is the role of a typical HPC user? source: [ANG+20]

It is especially important here to look into who the people are that work on HPC software and what their specific needs and demands for programming languages and libraries are. The March 2020 edition of the 'Parallel Computing Journal' featured a paper [ANG+20] that included a study about programming languages for data-intensive HPC applications. In this publication the researchers analyzed hundreds of articles relevant to HPC that

were published in the period of 2006-2018 and extracted information about the user base of HPC languages and what they considered their most important aspects. This section will review some of the results from this study and reason about how well Rust fits the desired features for HPC programming languages that were defined in the study.

Figure 3.1 shows the study's results for the question of how HPC programmers usually interact with programming languages. The graph distinguished between the two categories '*Developer*' and '*End-user*'. A developer is defined as a person that is mainly concerned with using the programming language to develop tools and libraries for other users. These kinds of programmers are mostly experts in the language with further knowledge in the more technical aspects of the systems that they develop for.

The majority of HPC programmers however can be classified as end-users who use the language to solve specific problems. In HPC this group mostly consists of scientists from a wide spectrum of domains. They are the actual userbase of big computing clusters and use the programming languages as a tool to solve their domain specific problems. The end-users are not necessarily experts in the used programming language and parallel computing but they need them as a tool to implement a solution to their problems.

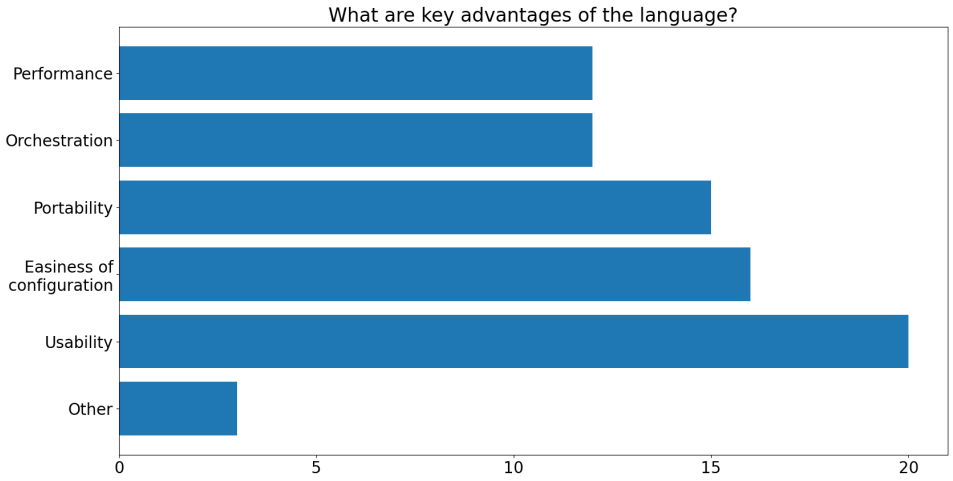

Figure 3.2: What are the key advantages of the language? source: [ANG+20]

Scientific applications often start out as sequential programs that only get parallelized later when it turns out that the sequential approach can no longer solve the underlying problem with the resources of only one computing node. This parallelization process can be very complex since it is often not trivial to parallelize the underlying sequential algorithms. It is therefore important that the used programming language and especially parallelization tools and libraries are designed to be as easy to use as possible to not make the parallelization efforts even harder on the programmer.

In Figure 3.2 we see another result from the mentioned study that shows what the users see as key advantages of the used programming languages in HPC. As we can

see concepts like ‘usability’ and ‘easiness of configuration’ are perceived to be the most important features here even before raw performance. Based on this the authors of the study conclude that the key features of a good HPC programming language are *performance*, *programmability* and *availability of tools and libraries*.

As already discussed in Section 2.2 Rust promises to deliver comparable performance to *C* and would therefore fulfill the first criterion. In the availability of tools and libraries for HPC aspect Rust naturally lacks behind languages like *C* and *Fortran* since they have a rich history in HPC and core tools like *MPI* and *OpenMP* are designed for them.

This thesis focuses mainly on the programmability aspect. Programmability firstly encompasses aspects of a language like a clear and intuitive syntax. A good debugging experience and the existence of strict safety and correctness mechanisms that will not let erroneous code compile at all are however also important. Rust’s previously discussed memory-safety mechanisms and clear compiler error messages are very fitting for this. The following chapters will explore how to make use of them to provide a more pleasant experience for programming message passing applications.

3.2 Problems with C and MPI

As already stated MPI has been the *de facto* standard for parallelization in HPC for multiple decades now but it is not perfect and brings multiple drawbacks and challenges with it.

Some of MPI’s problems stem from its age. The HPC environment has changed a lot since the 1990s as new hardware technologies and also new use cases have emerged. Even though MPI still gets regular updates to the standard and its implementations, some of the core features are basically unchanged since its beginning. Further it is natively bound to even older programming languages that are lacking the usability and comfort features that many modern languages provide to developers.

MPI’s problems range from the more technical side to its usability, learnability and productivity aspects, which can deter newer programmers that are starting to get into HPC development. We have seen in the last section that usability is one of the most desired features for HPC programming and as this section will show MPI has some problems in that aspect.

First of all the MPI API is quite low-level and works purely on raw data buffers that have to be explicitly sent and received. This approach can get very tedious and it is also error-prone, especially paired with *C*’s rather lax memory-safety mechanisms.

Looking at the signature of the basic `MPI_Send` function shown in Listing 3.1 we can see that it takes a lot of arguments to send some data. It begins with taking an untyped pointer to the starting address of the buffer that is supposed to be sent. The number of elements and especially their datatype have to be manually stated in the code.

```
1 int MPI_Send(const void *buf, int count,
2         MPI_Datatype datatype,
3         int dest, int tag, MPI_Comm comm)
```

Listing 3.1: The basic MPI_Send function

We can immediately see many possible sources of errors that could arise here. It is not unlikely for a type mismatch to happen here by for example accidentally mixing up 32 and 64 bit floating point types. The danger of such kinds of mix-ups is that it would not necessarily be detected by any compiler or tools and might not even lead to a runtime crash but create wrong runtime behaviour of the program, which could be very hard to pinpoint by the programmer after the fact.

Additionally the passing of a raw void pointer comes with many possible pitfalls such as passing the wrong address by accident or pointing to an invalid memory section. Furthermore the error handling for such MPI functions can be quite tedious as well. As we can see in the signature the send function just returns an integer that corresponds to a MPI error code. Handling these error codes for every call of a MPI function adds a big amount of boilerplate code and additional manual work for the programmer which can often lead to unsafe coding practices where returned error codes get ignored completely.

Many of these problems are direct consequences of the *C* language which works on a very low level of abstraction and does not support more modern approaches like exceptions for error handling or automatic type deduction and generic function arguments. This is of course not to say that modern languages are simply better than *C* since the whole point of it is to be a very low-level systems programming language that is as 'close to the metal' as possible to give expert developers as much freedom as possible. It can however be a problem if we go back to the fact that the majority of the userbase for MPI applications are end-users that simply require an easy way to upscale their scientific applications to multiple computing nodes and therefore desire easy to use tools to facilitate this.

In an article from 2018 [Dur18] Jonathan Dursi, a scientific researcher working in HPC, criticises HPC's big focus on MPI and its slow adoption of newer and easier technologies for parallelization. One of his points states that MPI tries to appeal to both, tool developers and end-users at the same time and therefore cannot fulfill either of their needs completely. MPI's low-level design can prove tedious and complicated to end-users not only because of the just mentioned examples but also due to the fact that it lacks more high-level abstraction features like implementations of distributed data-structures. It is often the fact that data-structures like distributed arrays and matrices are manually implemented for every new HPC application. At least in theory these are concepts that should not have to be redesigned for every MPI application but should be provided by a higher level library that internally uses the low-level MPI functions. In practice however this is often not the case and the basic MPI message passing operations are instead directly added to the core program logic. There are some library solutions or

```
1 [...]
2 MPI_Send(buf_out, BUF_SIZE, MPI_UNSIGNED_LONG, target, 0,
3             MPI_COMM_WORLD);
4
5 MPI_Recv(buf_in, BUF_SIZE, MPI_UNSIGNED_LONG, source, 0,
6             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
7 [...]
```

Listing 3.2: This might deadlock dependent on buffer size and used MPI implementation

even special purpose languages like *Chapel* [CCZ07] for this kind of work but it does not seem that they are getting as widely adopted as pure MPI implementations.

On the other hand one might argue that for tool and library development MPI is not low-level enough and can prove to be too restricting for developers that need to have complete control over the actual networking and message passing technicalities to provide the best possible performance.

Another problem of MPI are the possible discrepancies between different MPI implementations that can lead to incorrect programs if the underlying MPI implementation is switched. This argument runs contrary to the portability concept of MPI which states that two standard compliant implementations can be used interchangeably and will yield the same program behaviour. However in practice this is not always the case since the MPI standard has many places where the implementation details of a function are explicitly vague and leave different options for the implementers.

A perfect example for this are the most basic MPI send and receive functions. In Listing 3.2 we have the message passing code of an application that implements a ring communication scheme, where each process sends one message to its successor and then receives a message from its predecessor, including the last process that sends the message to the first one. As we have already discussed in Section 2.1.1 it is important to know the concrete semantics of the send and receive functions here to determine whether this code produces a correct program.

If `MPI_Send` is blocking and no buffering is applied this program will deadlock because no process can move on from the send operation since the send buffer is not safe to be reused until the data has been sent, which (without buffering) can only happen after a matching receive has been posted. In practice it depends on the used MPI implementation whether this program runs correctly or not. This is due to the fact that the MPI standard defines `MPI_Send` as using the so-called *standard* mode where it is left to the implementation to decide whether buffering is used or not.

This means that the program will run fine with some implementations and deadlock with others. But it is even more unclear than that because the program behaviour on some implementations can even be dependent on the size of the data buffers that are being

sent, because some implementations only do buffering on data buffers up to a certain size. In practice such behaviours can lead to hard to find problems if the application for example runs fine with a small problem size in the test case but then deadlocks in an actual run where the problem size and therefore the data buffer size is increased.

To be fair to MPI this freedom of implementation is clearly stated in the standard and there are variations of the send and receive functions that will always enforce a certain semantic. However since these semantic inconsistencies appear in the default versions of the message passing operations it is easy to imagine that such an error can happen to a non-expert user who will probably not search through the multiple hundred page long standard MPI specifications document.

This section outlines some of the problems and challenges that a developer can see themselves confronted with when programming message passing applications with *C* and MPI. Chapter 4 will go into the technical details of some of them and propose improvements to them using a Rust based message passing implementation. It will explore how Rust's safety mechanisms and higher abstraction features can produce a more easily usable interface and how far the Rust compiler can help with detecting possible errors in parallel programs at compile time. Chapter 5 will showcase the prototype of a Rust message passing library that focuses on usability and safety/correctness aspects.

4 Correctness checking for distributed programs

This chapter will go over some specific message passing scenarios where it is easy to unknowingly create incorrect programs that will not behave as intended at runtime and will show how a message passing library can be implemented in Rust that will circumvent these sources of error as much as possible. Since we have already established that MPI is the by far most used message passing library the examples in this chapter are structured in a way that first an example MPI code that introduces an error that will not be caught at compile time is given. This is then followed by an analysis of if and how a message passing implementation using Rust's safety and correctness features could find these errors at compile time.

4.1 Memory safety using strict data ownership

One fundamental danger when working with *C* and MPI is the predominant use of raw `void` pointer references for memory management. This is not necessarily a fault of MPI but rather stems from how the *C* programming language handles memory management. Nevertheless this very low-level access to specific memory addresses which are then passed to MPI operations can lead to a multitude of errors that are not caught by the compiler and produce runtime crashes or undefined behaviour.

Correctness of message passing programs gets more complicated as soon as non-blocking operations are used. When only using blocking communication the user can at least be sure that the program flow is interrupted until the used data buffer is free to be modified again and that the message passing operations happen in an obvious sequential order. When using non-blocking operations it is no longer clear what state a used data buffer is in at a given time and the programmer has to explicitly check before using the buffer again.

In MPI the standard non-blocking communication operations are `MPI_Isend` and `MPI_Irecv` which start the communication operation but immediately return and let the rest of the program continue. It is now up to the programmer to make sure that the data buffers used by the non-blocking operations are safe again. This can be done by calling `MPI_Test` or `MPI_Wait` which either give information about the current state of the operation on the buffer or block the program until the buffer is safe again. There are however no

```
1  if(rank == 0)
2  {
3      for(int i = 0; i < BUF_SIZE; ++i)
4      {
5          buf[i] = i;
6      }
7      MPI_Isend(buf, BUF_SIZE, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &req);
8      for(int i = BUF_SIZE-1; i >= 0; --i)
9      {
10         buf[i] = BUF_SIZE + i;
11     }
12 }
13 else if(rank == 1)
14 {
15     MPI_Recv(buf, BUF_SIZE, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
       ↪ &status);
16     for(int i = 0; i < BUF_SIZE; ++i)
17     {
18         printf("rank %d: %f\n", rank, buf[i]);
19     }
20 }
```

Listing 4.1: Example of an incorrect non-blocking MPI code

mechanics in MPI that can enforce the correct use of these functions before accessing an unsafe buffer again.

In Listing 4.1 we can see an example of a program's MPI section where an unsafe buffer is modified while a non-blocking send operation is performed. In this example the process with rank 0 first fills up a data buffer and then sends it to the rank 1 process using the non-blocking `MPI_Isend` function. The data buffer is then immediately modified again by process 0 without calling `MPI_Wait`. In practice this program compiles without any warnings but the outcome is undefined. The resulting values printed out by process 1 are dependent on a lot of factors like the used MPI implementation, the underlying hardware and therefore processing and networking speed and the chosen buffer size. The results can range from process 1 receiving the correct first set of values for the buffer, the complete second set of values for the buffer or a mixed result of the first and second set of values.

This error can be fixed quite easily by adding a `MPI_Wait` call in front of the second modifications of the buffer by process 0 but the point of this example is that these kinds of errors are easily made and that there is no feedback for the programmer that indicates a possible error in this code. In a more complicated program this kind of incorrect behaviour can also be hard to debug if the program does not even crash at runtime but just sends the wrong data sometimes.

If we analyze the core of the problem here we can find that the non-blocking send

operation has no way of enforcing its policy that the data buffer is not allowed to be touched until the send operation has concluded. It would need to block any access on the buffer until the programmer has made sure that the buffer is safe again.

If we translate this behaviour to the Rust memory concepts it means that the non-blocking send operation needs to have *ownership* of the data buffer until the send operation has finished. If this were the case it would not be possible to have any other active references to the buffer during the send operation and therefore not possible to access it until the ownership has been transferred back to the rest of the program. Section 5.3 will introduce an implementation of this concept, that manages to have a similar syntax to the MPI case but also will prohibit any unsafe access on the data buffer.

4.2 Automatic datatype type deduction for messages

Another problem that comes from MPI operating on raw memory addresses is the previously mentioned need to specify the data type of the buffers manually in each function call. This can for one thing be tedious for the programmers and makes function signatures more verbose but more importantly it can also lead to incorrect program behaviour without being detected at compile time. Determining as which data type a buffer is interpreted by the MPI library by passing it as a function argument also feels contradictory to the otherwise strongly typed *C* language and this concept introduces additional pitfalls for the users.

In Listing 4.2 we can see another example of an incorrect MPI section of an application that sends the contents of a buffer from process 0 to process 1. What happens there is that the buffer is created with the `double` type, which on most modern architectures translates to 64-bit floating point values, but the message passing operations use the `MPI_FLOAT` type which in this case would translate to 32-bit floating point values.

This is clearly an error by the programmer but this scenario is not too unlikely to happen in practice. If for example the program was first developed using `float` values for the buffer but it later became apparent that more precision was needed, the buffer variable could have been switched to the `double` type, but it was forgotten that this change also implicates changing every MPI operation accordingly. The problem is that the program still compiles without warnings or errors and even executes without any runtime errors but it yields different results. When this program is run, process 0 only sends half of the expected bytes from the buffer and therefore process 1 also only receives data for the first half of the buffer. For the MPI library this is no problem since it cannot know the intent of the user and it is only asked to read from and write to legitimately allocated address spaces but for the user the resulting program will yield incorrect results.

This example not only again highlights the dangers of working with practically untyped data buffers but it also shows some of MPI's disadvantages concerning usability and productivity. MPI code is not very flexible. Due to its quite low-level approach of having

```
1  double *buf = (double*) malloc(sizeof(double) * BUF_SIZE);
2
3  for(int i = 0; i < BUF_SIZE; ++i)
4  {
5      buf[i] = 0.0;
6  }
7
8  if(rank == 0)
9  {
10     for(int i = 0; i < BUF_SIZE; ++i)
11     {
12         buf[i] = 42.0;
13     }
14     MPI_Send(buf, BUF_SIZE, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
15 }
16 else if(rank == 1)
17 {
18     MPI_Recv(buf, BUF_SIZE, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
           ↪ MPI_STATUS_IGNORE);
19     for(int i = 0; i < BUF_SIZE; ++i)
20     {
21         printf("rank %d: %f\n", rank, buf[i]);
22     }
23 }
```

Listing 4.2: Example of a communication using the wrong MPI datatype argument

to manually hard code things like a buffer's data type even a simple change like switching the type of a variable can lead to a lot of additional work and potential errors in the MPI sections of the program.

Of course this level of detail about which exact bytes are communicated gives the programmer a lot of control and can be put to good use by expert users but in the most common cases all the user wants is to send data of the underlying type of the buffer which the compiler should know anyways. It would therefore be beneficial to provide a default behaviour for message passing operations that automatically deducts the correct type of the data that is being sent and offers the transmission of ambiguous byte streams or the modifications of the original data type as optional features for when it is needed.

This behaviour is not so easy to implement in *C* since the language does not support features like function overloading and only recently the C11 standard started to incorporate basic generic programming features [Plu12]. In more modern languages like Rust or *C++* this problem can be solved more conveniently by using their generic programming concepts. Chapter 5 will show how I implemented a solution for this problem in Rust and whether it is possible to lower the verbosity of message passing function signatures even more without losing any functionalities.

4.3 Limits of compile time checks

This section will discuss categories of errors in parallel programs that cannot necessarily be caught by a standard compiler without the help of external tools. It will first be discussed what kinds of errors can and cannot be detected and then go on with showcasing common errors in message passing communication schemes that cannot purely be solved through implementing a message passing library in a certain way.

4.3.1 Limitations for purely compiler based correctness checks

One commonality between all the problems that have been previously mentioned in this chapter is that they are mostly concerned with the correctness aspects of one single process. For example in the automatic data type deduction case all correctness checks are based on the type information of the buffer that is passed to the message passing operation by a process. My proposed solution cannot take into account whether a different process will post a matching message passing operation with the same buffer data type. The situation is similar for the correct management of non-blocking communication. Even though my proposed solution implicitly makes sure that the data buffer has been completely sent or received before giving back ownership to the function caller, it is implemented in a way where the compiler does not need to understand that communication with another process will take place.

In MPI-like message passing applications the code for all possible processes that will be spawned at runtime is contained in the same source code and the control flow for different processes is often influenced by using their process-id (or 'rank') to identify which workload belongs to which process. Based on that the compiler can apply its correctness checks for every possible code path that a process might take, but it is not able to make any reasoning about the validity of the interactions between the processes via the message passing library. In essence, this means that the compiler is only able to reason about correctness for code that is local to one process of a message passing application.

This of course stems from the fact that all checks which are performed by the compiler have to be specifically implemented by the compiler developers with absolute knowledge about the instructions that are used in the program. The compiler looks for patterns in the code that either are not allowed by the language's specifications or might be known to cause bad side effects and then produces an error or a warning. For example a check for using an uninitialized variable would be implemented in a way where after the declaration of a variable all possible future code paths are inspected and searched for reads on the variable that occur before a write on it. All information that is needed to deduce this possible error is intrinsic to the language and can therefore be taken into account when designing the compiler.

In the case of a message passing library however the compiler has to reason about a user written library that is not part of the language specification. Therefore, the compiler

cannot know about for example the semantics of a pair of blocking send and receive operations. All that is known to the compiler is that they try to establish network connections and send or receive data, but it will not be able to automatically deduce information like the fact that two instances of the analyzed program will be run together and send the data between them.

To support such correctness checks for the behaviour of a message passing library at compile time one would have to modify the compiler and implement checks for the library calls that take into account external information about the supposed behaviour of the message passing operations. The other option to implement such correctness checks would be to provide a separate tool that performs static analysis on the source code and checks for problems regarding how the message passing library is used.

Such approaches do exist for either MPI in the form of the so-called '*mpi-checker*'[DKL15] which is a part of the *Clang* compiler, or for multi-threading with *pthreads* in the form of *helgrind* [Dev21]. Chapter 6 will go over both of these tools and compare their capabilities in detecting errors in parallel programs to the implicit correctness checks that are performed by my Rust implementation of a message passing library.

4.3.2 Common message passing communication errors

The previously discussed error cases were mainly concerned with the unsafe use of memory buffers or incorrect interpretations of the transmitted data. Another big problem for message passing applications however are errors in their communication schemes that lead to incorrect program behaviour.

The term 'communication scheme' refers to the way in which processes exchange messages during the runtime of a message passing application. It is concerned with what messages are exchanged and especially the possible orders in which message passing operations are posted by the processes. Reasoning about the correctness of an application's communication scheme requires most of the time direct knowledge about how the developer intends the application to work. This makes it hard to develop any general correctness checks that are able to detect possible errors in their implementations. However there are some general communication patterns that have a high likelihood of being errors on the developer's side and for which it would make sense to programmatically check for. The following will list two common ones that are nearly always produced unintentionally by the programmer.

Deadlocks

The MPI code snippet in Listing 4.3 implements a ring communication scheme where each process sends a message to the following process in the ring and then receives a message from its predecessor. We have already seen a similar example in Chapter 3 and know that this program would run into a deadlock if the blocking `MPI_Send` function does

```
1 [...]
2 next_rank = (rank < size-1) ? rank+1 : 0;
3 last_rank = (rank > 0) ? rank-1 : size-1;
4
5 MPI_Send(buf, BUF_SIZE, MPI_DOUBLE, next_rank, 0, MPI_COMM_WORLD);
6
7 MPI_Recv(recv_buf, BUF_SIZE, MPI_DOUBLE, last_rank, 0,
     ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
8 [...]
```

Listing 4.3: MPI code that causes a deadlock

not use message buffering. This is the case because all processes try to send a message at the same time and will then remain in a blocking state until a matching receive has been posted. The receives however can never be posted because they will not be reached due to the blocking behaviour of `MPI_Send`.

Scenarios like this can occur quite easily when implementing a communication scheme with blocking operations and they are not always so simple to see as in this short example code. A deadlock can be produced in many different ways, as for example when working with `MPI_Barrier` operations in the middle of a calculation for process synchronization. In practice they are not too hard to debug if the programmer has access to a debugging tool that actually supports MPI. This may not always be the case because many of these tools are proprietary and not freely available without a paid license. But even though deadlocks are not the most complicated class of errors it would be beneficial to provide correctness checks for them together with the message passing library because they can occur quite often during development.

The algorithmic detection of deadlocks such as in Listing 4.3 with static analysis methods seems quite feasible. The analyzer would have to put all blocking message passing operations in their possible orderings and simulate whether the function call for the matching message passing operation on the target process is reachable or not. Chapter 8 will discuss the possibility of providing this functionality alongside my Rust message passing implementation.

Missing communication partners

The MPI code section in Listing 4.4 showcases an error where not every receive operation has a matching send operation. This will cause a partial deadlock of the program where some of the processes will remain in their receiving state forever and the remaining processes will continue on until their next message passing operation that includes one of the blocked processes. This specific example simulates a scenario where some non-blocking send operations are started and then, after a big code section with unrelated calculations they are received with blocking operations. However, the programmer in

```
1 if(rank % 2 == 0)
2 {
3     MPI_Isend(buf, BUF_SIZE, MPI_DOUBLE, rank+1, 0, MPI_COMM_WORLD,
      ↪ &req);
4 }
5
6 // long calculation code
7 [...]
8
9 MPI_Recv(recv_buf, BUF_SIZE, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD,
    ↪ MPI_STATUS_IGNORE);
```

Listing 4.4: MPI code that does not have all matching send and receive operations

this example has forgotten that the send operations where conditional and that not every process will perform a send operation. The example also has a second error that is dependent on the number of processes that the program is run with. If it is started with an odd number of processes the last process will try to send a message to a non existing process and the MPI application will crash at runtime.

This example might seem a bit forced but in practice it is a common error when developing message passing applications to wrongly handle the rank based conditions for the message passing operations and end up with a scenario where an operation is posted that does not have a corresponding partner for the two-sided communication. Being able to detect this error class also seems very beneficial for an easier development experience of message passing applications.

The implementation of correctness checks for this kind of error shares some concepts with the detection of deadlocks and also seems feasible with a static analysis approach. It would require an analysis of the target process argument for send instructions and the source process argument of receive instructions to find out whether there are message passing operations that do not have a matching partner instruction.

5 Design and implementation of heimdallr: a Rust message passing library

This chapter will show my work on a Rust message passing library called 'heimdallr' that implements the concepts discussed in Chapter 4. The main ideas of heimdallr are to provide correctness checks at compile time and a general memory safety concept built upon Rust's fundamental ideas about memory management. The second emphasis lies on usability where I tried to keep the API as simple as possible without losing functionalities that other message passing libraries like MPI provide. In its current state heimdallr is still in a prototype phase and mainly provides proof-of-concept solutions for the previously listed problems that users may face with MPI. It is not production ready nor feature complete yet. Future goals for the library are outlined in Chapter 8.

The following sections will explain the design ideas that went into the development process and go through specific implementation details for the key features.

5.1 Using a client and daemon infrastructure

The fundamental mechanic that is needed to build a message passing library is the ability for processes to communicate with each other. The communication generally happens over some kind of network connection like a TCP stream but to make the processes be able to connect to each other at all they first need the required connection information like the addresses of each others' TCP sockets. The most straightforward solution for this would be to manually hard code the IP addresses and ports for each process into the application's source code but in practice this is not really a viable solution. Applying this method is not only tedious for the users but it is also very inflexible in regards to executing the program with varying numbers of processes and it does not scale well with the number of total processes. Therefore a better solution is needed.

A prevalent method to handle this problem is the creation of a management background process for the library that serves as a central hub for all processes. Such background services are commonly referred to as *daemons* and I decided to go this route as well with my implementation. The so-called `heimdallrd` daemon has to be run on the computing nodes and it works as the first point of contact for each process of a heimdallr application

which will from now on be referred to as clients. The daemon first collects all necessary information about an application run and then relates it back to the associated processes so that the program execution can start.

Having a daemon process that can be globally accessed by all clients now raises the question of how much the daemon should be involved in the actual program execution. It can either just be there to help establishing the initial connections between the clients or it could also be used for the actual message passing operations in the program. Possible advantages of having the daemon participate in message passing operations are that it can simplify tasks like synchronization or offering globally shared variables that can be accessed from all processes at any time. There are however also multiple drawbacks of using this method. For one it introduces more network traffic because communication is not directly client-to-client but has to be routed through the daemon, which is suboptimal for performance. Additional problems are more load on the daemon's computing node and possible performance bottlenecks where multiple clients have to wait for daemon responses because the daemon's workload is too high. Traditionally message passing libraries do not use a daemon that participates directly in the message passing operations. For the basic operations this is also true in this implementation but Section 5.4 explores how involving the daemon directly can be used to implement features like locking mechanisms and shared data structures.

Another design decision regarding the use of daemon processes is the question of inter-daemon communication. What is meant by that phrase is whether daemons on different computing nodes should interact with each other or not. Intuitively it would seem that a client that is running on a specific node does not interact with daemons that are running on a different node. If this is the case and an application is run over multiple nodes it means that the daemons have to communicate with each other at least once in the initialization phase to exchange the IP addresses of the clients on the other nodes to relate them to the client running on their node. This is not too hard to implement but it gets more troublesome when deciding to implement features where the daemon is more directly involved in the message passing application with features like the aforementioned locking or shared data structures. Therefore I decided to implement the system in a different way where for each application run exactly one daemon is responsible for all clients regardless of whether a client is running on the same node as the daemon or not. A closer discussion of the benefits and drawbacks of this approach will follow in Section 5.4.

5.1.1 Implementation of the heimdallr client and daemon system

In Figure 5.1 we can see the three central structs that make up heimdallr's core client and daemon system. It consists of the following components:

- **Daemon**: The `Daemon` struct is the central component of the `heimdallrd` daemon process. It manages all running jobs and each instance of the daemon is representative of one available computing node.

Figure 5.1: Core structs of the heimdallr daemon and client system

The fields `name` and `partition` are referring to the computing node that this instance of the daemon is running on. On a computing cluster each node runs one instance of `heimdallrd` and gives the node its identifying name through the `name` field so that the users can target on which nodes a job is supposed to be executed. The `partition` field allows grouping of nodes into specific sets. This can for example be useful if the computing cluster consists of multiple groups of nodes that contain varying hardware. In that case a separate partition can be created for the different types of nodes and when running a job it can be ensured that all processes are run on the same node type.

To be able to communicate with the clients the daemon needs to offer a point of communication to them. This is implemented via the `client_listener` field which represents an open TCP socket. It is implemented with the Rust standard library's `TCPListener` type and always waits for new connections from heimdallr clients. The IP address and port of this TCP socket need to be globally known throughout the system so that clients can find it. This is solved via a configuration file that needs to be present on the system. In this file all daemons that are currently active on the cluster are listed together with the information about their TCP sockets. When a daemon process is started on a node it is given its settings like name and partition by the user and after a successful startup process the global configuration file is updated by the daemon. For this process to work it requires the cluster to have some kind of shared filesystem so that the same configuration file can be accessed from each node. In practice this should be the case on most HPC computing clusters because it is also needed to execute the same program binary from multiple nodes.

The last field of the daemon is a `HashMap` of the currently existing jobs that are run on the daemon which are identified by a name parameter. The daemon technically supports the execution of multiple jobs at the same time on the same node but in reality this might not be very practical since it can be expected that applications which need to make use of parallelization through message passing will fully utilize each node they are run on and not leave spare computing resources for other applications.

- **Job**: The `Job` struct is used by the daemon to encapsulate all data associated to a heimdallr application that is currently running on the system. A job is defined

as one run of a heimdallr application with a given name and number of processes. The `name` parameter is necessary for the daemon to match connections from a client to the correct job in the case that multiple heimdallr applications are running at the same time.

The `size` field holds the number of processes that are participating in this run of the application. This information is sent by each client in its initial communication with the daemon during the heimdallr initialization phase. At the same time the TCP socket addresses of all clients belonging to the job are also collected and stored in the `client_listeners` field so that the daemon can aggregate and relate this information back to the clients. The details of this initialization process will be explained in the following section.

- **HeimdallrClient**: This struct is created by each client in the heimdallr initialization phase and stores all information that is needed to execute the heimdallr message passing operations from a client process.

  Each client belongs to a heimdallr job and therefore is identified by its `job_name` field that gets communicated to the daemon at the beginning of each communication. The `size` field again specifies how many processes are participating in the job and additionally each client gets a unique `id` in the range of `0` to `size-1` so that the user can distinguish between processes in the code.

  For the network communications the clients need to open TCP sockets on which they are listening for incoming connections during the applications lifetime and which are then shared with all other clients through the daemon in the initialization phase and stored in the `client_listeners` field. The exact way of how this information is shared will be made clear in the next section.

  The `HeimdallrClient` struct does not only store the relevant data for a client but also implements all available heimdallr operations in the form of methods to the struct. This design decision was made because each message passing operation needs to access information from the client struct anyways and due to the fact that Rust does not have a concept of global variables it would otherwise always be necessary to pass a reference to the `HeimdallrClient` struct to each message passing function. The currently implemented methods of the struct are not listed in Figure 5.1 because they will be introduced piece by piece during this chapter when their implementations get discussed.

The heimdallr initialization process

Each heimdallr application has to start with a call to the `init` function of `HeimdallrClient`. This function sets up all heimdallr functionality for the client and returns a `HeimdallrClient` object that can be used to access heimdallr operations. The `init` function expects a certain set of command line arguments that specify job settings like the job name and size, as well as the name and partition of the daemon that is responsible for the job.

As previously explained in its current state heimdallr functions in a way that a single daemon is responsible for a job no matter on which nodes the individual clients are located and therefore all clients have to specify the same target daemon here.

In Figure 5.2 we can see a sequence diagram of a heimdallr application's life cycle with two clients that shows the communications between clients and daemon beginning at the call of the `init` function.

When `init` is called the first task for the clients is to parse the given command line arguments and identify the name of the job, its size and the daemon that is responsible for the job. After ensuring that all needed parameters exist and are in a valid form the clients now load the heimdallr configuration file and extract the socket address of the daemon so that they can open a connection. The job information is then sent to the daemon and the client waits for a response.

On the daemon side as soon as the first connection from `Client A` has arrived it starts to create a new job because no job with the same name does exist yet. The name and size of the job are set by the daemon and the socket address of the client is stored and `Client A` is assigned the id `0`. The daemon then repeats this process for all new client connections belonging to the same job until the correct number of clients for the job to start is reached. In the given example this is immediately the case after `Client B`'s message has arrived. `Client B` is given the id `1` and the initialization of the job can now be completed. The daemon sends response messages to all clients which contain the assigned id for each client and the list of socket addresses for all clients.

After the responses of the daemon have arrived at the clients they now have all the needed information to run the actual heimdallr application's code. They have been assigned an unique id that can be used by the programmer to address a certain process and they have a list of socket addresses for the remaining processes which enables later message passing operations between processes without any more involvement of the daemon.

After both processes have finished their workload the last step is to call the `fini` function of `HeimdallrClient` to inform the daemon that the job has concluded. This step is necessary due to the fact that the daemon stores data for each job which needs the be removed after the job has terminated. Contrary to the `init` function the call of `fini` can however be automated with Rust by implementing it as the `Drop` trait for the `HeimdallrClient` struct. An implementation for this trait is automatically called when the object goes out of scope and it therefore acts similar to a destructor in languages like *C++*.

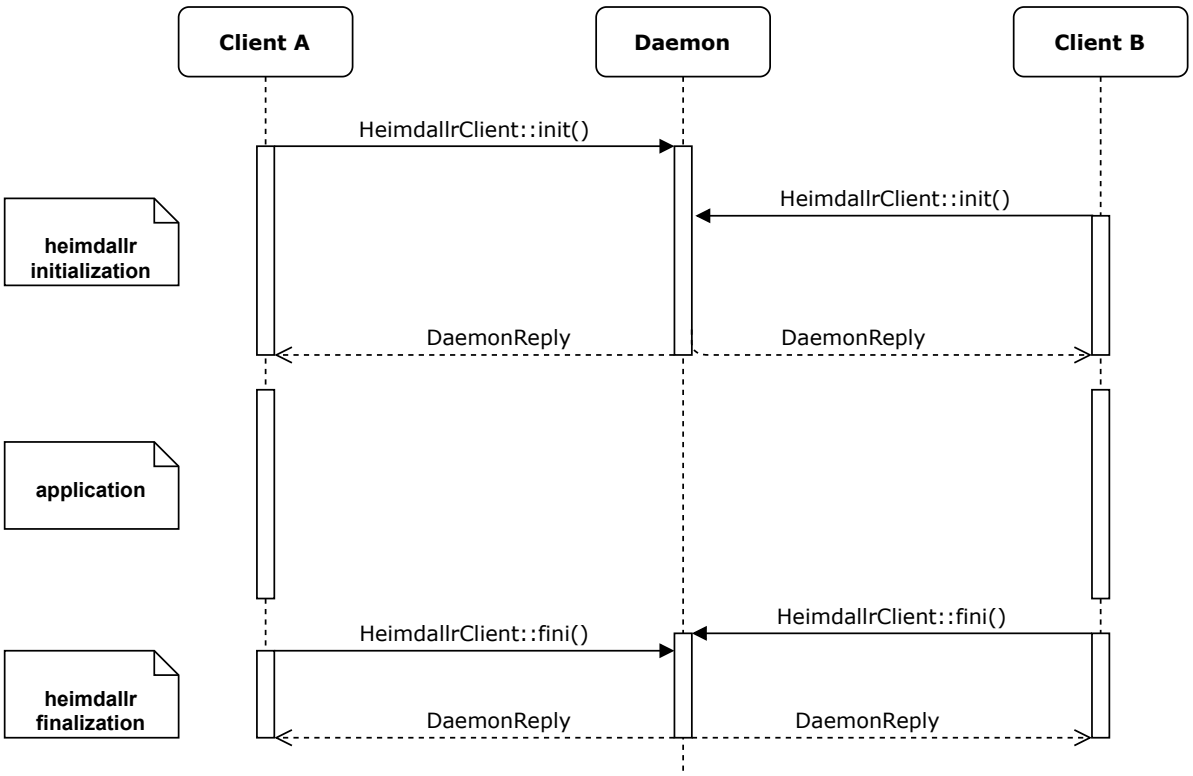

Figure 5.2: Sequence diagram of the client-daemon interaction of a heimdallr application

5.2 Transmitting messages using serialization

The process of transmitting data between processes via message passing is not as straightforward as just establishing a network connection between two processes and sending the pure data-bytes. As already discussed in Section 2.1.1 there are multiple different semantics for a pair of send and receive operations that have different requirements for the implementation of this functionality. There are also the more technical questions like how to retain the correct data type of the sent bytes and how to ensure that incoming messages to a process are matched with the correct receive calls. This section will introduce my design approaches to basic blocking and synchronous communication operations and how the data can be transmitted correctly over a TCP connection while maintaining the correct data type on the receiving process.

5.2.1 Constructing a minimal signature for send and receive operations

Because usability is one of the central design goals of heimdallr it is important to keep the basic operations like send and receive as simple as possible in their use. This means that they should have concise function signatures and very clear semantics. It is therefore important to analyze what parameters are definitely required for them and how much of

the needed information for the message passing process can be solved automatically by Rust and its compiler. The reasoning in this section refers to unbuffered messages which means that the terms synchronous and blocking can be used rather interchangeably since in the blocking case the buffer can only be considered safe again after a matching receive has been posted which allows the message to be sent.

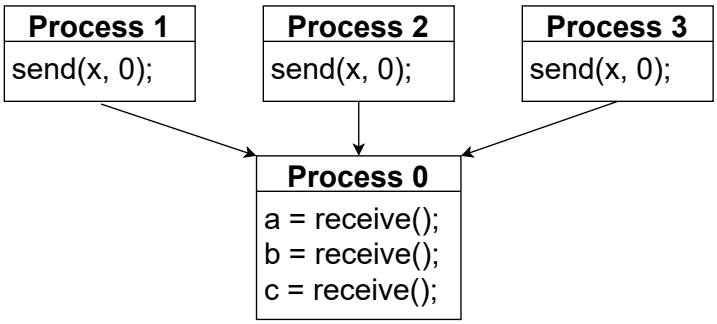

Figure 5.3: Example of multiple simultaneous messages to the same target process

When designing a synchronous send operation there are two obvious parameters. Firstly the data buffer that is being sent and secondly the id of the target process. Looking at a synchronous receive operation it might at first seem like it is not necessary to include the process id of the message source. If the sending process initiates the network connection there is no immediate need for the receiver to know from where it is coming to be able to receive the incoming data. This is in fact true for applications with simple message passing schemes where it can be assured that there will always be just one incoming message to a specific process at a time. It is however easy to construct examples where an implementation like this can lead to undefined behaviour and incorrect outcomes. Figure 5.3 illustrates the message passing scheme of an application where all processes simultaneously each send one message to the same target process which posts receive operations for the messages without stating a source process for the messages. The problem here is clear. Even though these are synchronous send and receive operations it is completely unclear which message will be received into which buffer because this purely depends on the timing of the arriving messages that might differ in every execution of the application. It is therefore necessary to always state the intended source process for a called receive operation to ensure the correct matching of messages.

The process of correctly matching multiple incoming messages to the correct receive operation can be taken one step further by also including an id-tag to the send and receive functions where the user can clearly identify messages and a receive call will only accept messages that have the correct tag. When communication is limited to synchronous operations the idea of a message id-tag becomes redundant because the messages can already be distinctly identified by having the source process parameter in the receive call. As soon as non-blocking and asynchronous operations are introduced into the system it however becomes necessary to prevent the same aforementioned problem of not being able to correctly match messages to the right receive operation. Even though we are

```
1 // The synchronous and blocking send function
2 fn send<T>(&self, data: &T, dest: u32, id: u32)
3     -> Result<(), &'static str>
4
5 // The synchronous and blocking receive function
6 fn receive<'a, T>(&self, source: u32, id: u32)
7     -> Result<T, &'static str>
```

Listing 5.1: heimdallr's basic send and receive functions

only talking about synchronous and blocking communication operations in this section this still needs to be considered because it it possible that for example a process will simultaneously send one message asynchronously and directly thereafter another one synchronously. This yields a scenario where again two messages from the same source arrive at the receiving process where they cannot be distinguished by only the id of their sending process. Therefore a message id-tag is still necessary for synchronous and blocking send and receive operations to ensure correct message matching if the library also supports asynchronous operations.

This reasoning produced the two function declarations shown in Listing 5.1 for the basic send and receive functionality of heimdallr. The first argument of both functions is the `&self` reference which stems from the fact that, as previously mentioned, both functions are implemented as member functions of the `HeimdallrClient` struct. This argument is automatically filled in by the compiler and in essence gives the functions read access to client information like its own id and the socket addresses of other processes. In the send function it is followed by a reference to the data buffer that is sent and the id of the targeted process to which the message is transmitted. As explained above the last argument in both functions is the message id-tag needed to identify received messages correctly. Looking at the receive function we can see that it is missing a reference to the target buffer where the received data is supposed to be stored. In heimdallr the received data is instead returned by the function and can be stored in a variable from there. This process moves the ownership of the received data from the receive function to the assigned variable.

In Rust the return type of a function is indicated with the '`->`' symbol. In this case both functions return values that are of the **`Result`** type which is one of Rust's central error handling concepts. This is one of Rust's unique approaches for error handling that distinguishes it from other languages. Since it is possible for the message passing operations to encounter runtime errors like not being able to establish a TCP connection or receiving incorrect network packets from which they might not be able to recover themselves they need a way to relate error information back to the caller so that this can be handled by the application. As we have already seen in *C* this is often handled by returning an integer error code as the function's return value. In other languages like *C++* or *Java* error handling is instead often done via exceptions that are thrown by the

```
1 enum Result<T, E> {
2    Ok(T),
3    Err(E),
4 }
```

Listing 5.2: Definition of Rust's Result type

erroneous function and which can be caught and handled by the function caller. Rust does not implement a traditional exception system but instead introduces the `Result` type to return error information from functions. Listing 5.2 shows the definition of `Result` and we can see that it is in fact a generic enum that has the two possible values `Ok(T)` and `Err(E)`. It is a simple wrapper around a return value that can either contain a correct return value from the function in `Ok(T)` or it contains an error message produced by the function in `Err(E)`. For new users of Rust this error handling concept can at first seem a bit strange and unusual but it comes with significant advantages for writing error resistant code which is in line with Rust's philosophy on writing safe code. Returning a `Result` value forces the programmer to take into account the possibility that the function might not have succeeded and that they need to handle the possible errors. This is enforced by having to 'unwrap' the returned `Result` value before being able to use the actual return value using one of the provided unwrap functions that force the user to decide what should happen if the `Result` instead contains an error message. I will not go any deeper into the specific error handling mechanics of Rust at this point since it is not the topic of this thesis but I found it important to provide a brief explanation for the readers to not be confused about the syntax in Listing 5.1 and other Rust code snippets that will follow in this chapter, where the `Result` type will appear regularly. In our case we can see that the send function returns a `Result` that either contains an empty tuple in the case of a successful execution or the error message produced by the function. In the case of the receive function the returned `Result` either contains the received data or an error message.

The last important piece of information that needs to be transmitted in a message passing operation is the data type of the sent data. This is handled through Rust's generic programming concepts and will be explained in the next section.

5.2.2 Transmitting data type information

Section 4.2 listed some of the problems that can arise in MPI from having to manually state the correct data type for the send and receive buffers and that it would be advantageous if the correct type for the transmitted data could be deducted automatically by the library. In *C* this concept is hard to implement but Rust and multiple other newer programming languages provide generic programming features that make this process far more achievable. Rust allows the use of generic data types which can be used to write

functions in a way where they can be mostly agnostic about the type of an argument or the specific return type.

We can see the use of generics in the function declarations for send and receive in Listing 5.1 where both of them include a generic type `<T>` for the data buffer of the message passing operation. Technically this type parameter can still be manually filled in when calling the function which would give us the same verbosity as the MPI function signatures, but in practice the Rust compiler will adopt this task whenever it is able to determine the correct type by itself. For normal use cases this will always be the case for both shown functions. In the case of the send function the user will pass a reference to the data buffer as the `data` argument which immediately informs the compiler as to what type `T` should be. For the receive function the generic type `T` is used for the return value and can be automatically deducted as long as the left-hand side of the function call is a variable of the correct type. This use of generics eliminates the need for programmers to manually state the data type for each message passing operation and thereby also eliminates some of the possible error sources illustrated in Section 4.2.

While this is a syntactical solution to the data type deduction problem the real challenge for message passing is the technical implementation of how the data that is being transmitted over a network connection as a byte stream can be transformed back to be interpreted again as the correct type in the receiving process. For MPI this does not really pose a problem because it operates on raw memory addresses and can just store the received data at the specified address without needing any internal knowledge about the type. The MPI data type argument is only needed to calculate the correct amount of bytes that are transmitted and for a check whether sender and receiver have put in the same type. In Rust however this problem gets more complicated because the language is not designed for type manipulations on raw byte data since this can often prove dangerous and lead to many kinds of runtime errors. In principal handling the transmitted data as byte arrays or vectors and casting them to the expected type on the receivers end should work relatively fine when working with basic types like integers but if user defined types are supposed to be supported by the message passing the process becomes less safe. A user defined type that holds data will take the form of a struct in Rust. The memory layouts that get produced for structs by modern compilers are not always just a concatenation of all struct members but may include forms of padding that is added by the compiler to optimize the memory layout for struct accesses. Especially when we add in the possibility of dynamically sized struct members the task of casting the raw memory representation of a struct back to be a variable of the struct type becomes more sophisticated. Therefore a better solution is needed for heimdallr.

The task of translating a data structure into an interim format that can be translated back at runtime by a process is commonly referred to as *Serialization*. The need for a process like this arises often in software development for tasks like reading in an object from a file or exactly for our purpose of transmitting messages over a network. Therefore many different data formats and software solutions for this problem already exist and have been proven to work well. Instead of re-implementing one myself for this project I

decided on using a well known serialization crate for Rust that is called *Serde* [Ser21a]. This library supports the serialization and de-serialization of Rust data types into a variety of formats like *JSON* [sr21] or *bincode* [Ser21b]. It also does not only support the built-in types of Rust but offers an easy method of automatically implementing serialization functionalities for user defined structs by adding one line of macro code to the struct's definition in the source code.

Using *Serde* the send function firstly serializes the data buffer into a chosen format and then sends the serialized data over the network. In the receive function the type information that has been passed to it from the generic type argument can then be used to de-serialize the data and move it into the return value having the correct data type. This is especially helpful when sending dynamically sized data structures like vectors because the de-serialization process is able to infer the resulting size by itself which gets rid of the need to include a size parameter in the receive function. The receive function is able to create a data object of the correct size and can simply move the ownership out from the function to the caller without having to copy the data into a provided buffer.

5.2.3 heimdallr's implementation of the message passing process

Now that the design decisions and technical details of heimdallr's message passing process have been discussed we can take a look at an example application that puts the described concepts to use. In Listing 5.3 we see the code of a heimdallr application that sends a `String` message from process 0 to process 1. I will also use this example to illustrate the network communications that are needed for the send and receive operations to correctly implement the semantics of synchronous message passing.

In the first line of Listing 5.3 all processes create their instance of the `HeimdallrClient` struct by calling the initialization procedure that has been explained in Section 5.1.1. This is followed by the declaration of a `String` variable which will serve as the send buffer for process 0 and the receive buffer for process 1. The core logic of the example application is contained in the following match statement which assigns the different processes their workload in the program. The `match` keyword is roughly equivalent to a `switch` operator in other languages. It performs pattern matching on the value of the given parameter which in our case is the id of the current process. Interestingly the `match` operator is another feature of Rust that showcases the languages attention to correctness checking. It will only compile if all possible cases have been covered by the programmer which forces them to at least give a thought to how unexpected values of the value need to be handled instead of just ignoring or forgetting about them. In this example we expect the application to be run with exactly two processes which is why the last arm of the `match` operator assigns no work to processes with a higher id than 1. Of course the application will still work if it is launched with more processes. The spare processes will just idle and then terminate all together after the code of the other processes has been executed.

```
let client = HeimdallrClient::init(env::args()).unwrap();
let buf: String;

match client.id
{
    0 =>
    {
        buf = String::from("Message from process 0");
        client.send(&buf, 1, 42).unwrap();
    },
    1 =>
    {
        buf = client.receive(0, 42).unwrap();
        println!("Received message: {}", buf);
    }
    _ => (),
}
```

Listing 5.3: Example heimdallr application that sends one message

Inside the arms for the processes 0 and 1 we see applied use cases for the synchronous send and receive functions that have been discussed so far in this chapter. As can be seen the function signatures in the actual heimdallr implementation have remained quite concise and not much manual work is required from the programmer. Process 0 simply has to state the data buffer that is to be sent, the target process id and a message id-tag. Process 1 also states the source process of the message and the same id-tag and assigns the return value of the function to the receive data buffer. The only other code that is needed is the `unwrap` function call on the returned `Result` type from both functions. As previously explained this forces the caller to acknowledge a potential error in the message passing process which needs to be handled. To keep this example as concise as possible the error handling in this application is limited to the standard `unwrap` function which will cause the process to *panic* and terminate with an error message that indicates the location of the panic.

The main send and receive operations of heimdallr are supposed to behave in a blocking and synchronous manner. As explained in Section 2.1.1 this means that for one the functions will only return when the data buffer is safe to be used again and they will also not return before both processes have posted their message passing operation and the actual data transfer takes place. The implementation must therefore ensure that both of these conditions are met and there needs to be a specific pattern of network communications between both processes to achieve this. Figure 5.4 shows a sequence diagram of the communications that happen between the sender and receiver processes in our example application.

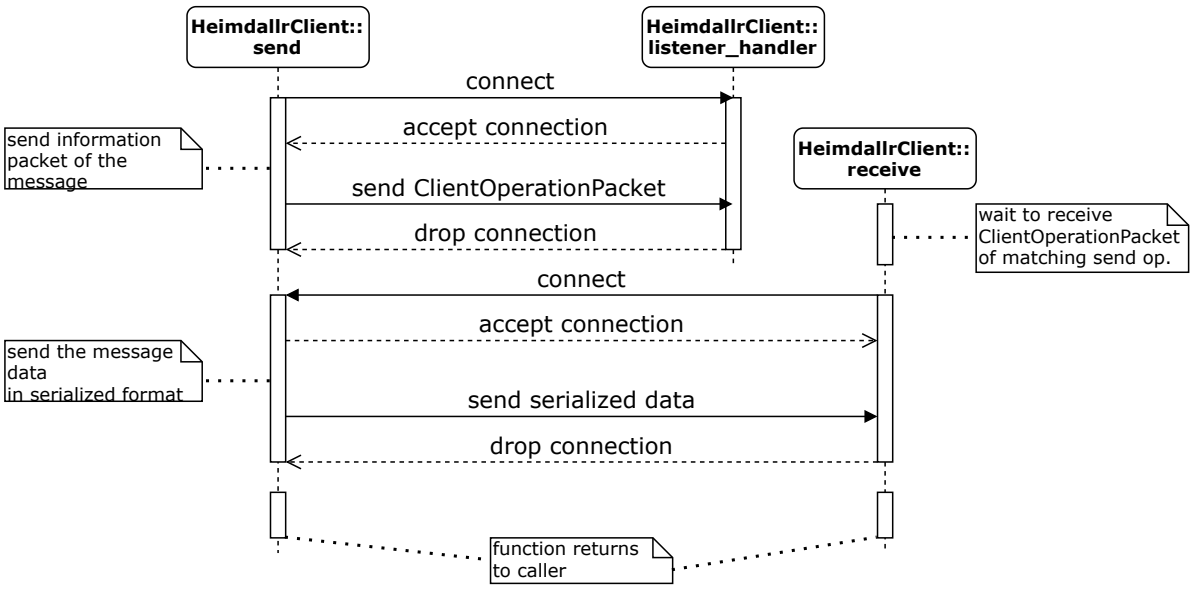

Figure 5.4: Sequence diagram of the message passing process for a send/receive operation pair

The diagram shows a timeline for both processes and illustrates all messages that are being sent over their TCP connections. Since we are dealing with a parallel program it cannot deterministically be said for our example application whether the send or the receive call will be posted first when running the program and in fact their order can differ from run to run. In Figure 5.4 the receive operation is posted shortly after the send operation but the system needs to be implemented in a way that supports all possible orders and wait times for the function pair.

The first thing that might stand out for readers of the diagram is the fact that an additional function is shown to be participating in the message exchange. This function is the so-called `listener_handler` of the `HeimdallrClient`. This function is always active as a background thread for every heimdallr client. It serves as the first point of contact for all incoming network connections to the process. The reason for its existence has to do with previously discussed meta-information that has to be sent with each message to correctly identify it and additionally with the fact that a receiving process needs to be able to handle the scenario where a send operation from another process has been posted a long time before code reaches the matching receive function call. The concrete purpose of the `listener_handler` will become clear by going through the sequence diagram step by step.

The timeline begins with process 0 calling the send function. It uses the given target process id to find the correct socket address of process 1 and starts to establish a TCP connection. On process 1 the `listener_handler` thread has been actively listening for incoming connections since the end of the heimdallr initialization phase and it accepts the connection request. Process 0 is now able to send a so-called `ClientOperationPacket`

with metadata for the message passing operation. The packet contains the id of the sender process, the message id-tag and additionally the address of another TCP socket that will be opened by the sender for the actual transmission of the message data. This information is stored by the receiving process in a `HashMap` data structure where a tuple of the sender process id and the message id-tag is used as a key to uniquely identify the message. After the packet has been sent the sender process now waits for a sign that the matching receive operation has been called by listening for connections to the newly created TCP socket for this message passing operation.

One might ask themselves why this intermediate step is necessary and why this process cannot be just handled in the receive function directly. There are two reasons for this:

1. If a scenario occurs where the send operation is posted a long time before the receive operation there would be nothing to accept the incoming TCP connection request on the receiver process which could lead to a timeout on the connection request and thereby failing the whole message passing process. This would be recoverable by making the sender process do repeated connection request until it gets a response but this also does not seem like a good solution. In the case that the receiver has crashed this could never be communicated to the sender because it would just assume that the receive has not been posted yet but still might in the future, causing the sender to endlessly try establishing a connection.

   A behaviour like this would make it nearly impossible to recover from a scenario where one process fails during runtime. Granted, for most message passing applications the failure of a single process does in fact mean that the application run has to be aborted because no usable results can be produced without all processes completing their work, but there definitely exist use cases where this is not the case and the application can be designed to recover from a node failure provided the used message passing library allows it.

   The methodology used in heimdallr causes the send function to return with an error message if the network connection to the receiving process cannot be established. This can be interpreted by the caller as either there is a problem with the network or the target process has shut down. Even if the application cannot be designed in a way to recover from this state it is still a better solution to receive an error message as a response which can dealt with and at least opens the possibility to for example enter an application shut down phase where some of the calculated data might be saved or a meaningful error log message can be output.

2. As previously discussed the receiver of a message has to check that the message source and id-tag are matching to the called receive operation to prevent that the wrong message is received. This information is sent by the sender process in the `ClientOperationPacket` that arrives before any actual data is transmitted. Analyzing this data directly in the receive function poses a problem. If the source or message id-tag do not match with the parameter of the receive function that means that the message was not intended to be received by this operation. The

receiver could now decline this message and wait for another incoming connection to find its correct message passing partner, but then the first message could get lost. Since the `ClientOperationPacket` information from the first message has now been consumed it would mean that the receiver for that message would never get it and therefore not know that its matching send operation is already waiting to send the actual data.

One way to circumvent this problem would be to send a response back to the sender of the first message to let them know that the packet has to be transmitted again. This solution can work but it has some disadvantages. In a scenario where the receiver has to wait a long time before its matching send operation is posted this whole process could repeat itself many times because the sender of the first message would get its response to resend the `ClientOperationPacket`, perform that action and the packet would again be wrongly received leading to a loop of error responses and re-sending operations. This is of course not ideal because it causes a lot of unnecessary network traffic and also more opportunities for a packet to get lost in the network.

A different solution would be to store the received `ClientOperationPacket` data somewhere in the client and to let all receive operations first look up if the packet from their matching send operation has already been received by the process. That is in essence the task performed by the `listener_handler` thread with the only difference that this whole logic has been moved out of the receive function.

The combination of these two reasons lead me to the design decision of having the `listener_handler` background thread receive all initial connections from other processes and to store their `ClientOperationPacket` data to be later requested by calls to a receive function.

In Figure 5.4 the receive operation from process 1 becomes active before the `listener_handler` thread has completely processed the message from process 0. This means that it enters a waiting period where it continually requests data from the `HashMap` data structure that matches the source id and message id-tag until an entry is found. As soon as a matching entry is found the receive function gets the address of the new TCP socket that process 0's send function is listening on and it tries to establish a connection. The need for a separate TCP stream to transmit the actual data of the message passing operation arose because if the receiver would connect to the known TCP socket of the sender it end up with a connection to the `listener_handler` thread that is running on that process to handle incoming send operations. It would need to communicate that the established connection has to be transferred to the waiting send function somehow which seemed like unnecessary work for the library and opening a new socket that will only be active for the duration of this send and receive operation pair felt like a more straightforward solution.

After the connection has been accepted by the sender process it now finally starts to send the actual data in its serialized format. Once the data transmission has concluded the network connection is closed and the send function can return to its caller. The receiver now has to perform the de-serialization process. The received data is de-serialized into a local buffer whose ownership is then moved out of the function via its return value. This process does not produce any additional overhead of copying the data into a buffer given by the caller but instead the left-hand side variable of the function call is given the ownership of the de-serialized data's memory and it can then be accessed from outside the receive function.

5.3 Non-blocking and asynchronous communication

Up until this point all introduced heimdallr message passing operations have implemented blocking semantics where the function call will not return before the data buffer is safe to be used again by the caller. While this behaviour might be enough to implement basic message passing applications there are various reason as to why non-blocking and asynchronous communication is a desirable feature in a message passing library. Foremost non-blocking communication is a great tool to increase the performance of parallel applications by letting the caller continue with calculations that are independent of the message passing process while it is executed in the background. Additionally there are message passing schemes that can only be implemented with the use of non-blocking communication.

This section will introduce the non-blocking and asynchronous message passing solutions of heimdallr, go over the problems that had to be solved during their implementation and discuss how the Rust memory safety mechanisms were used to apply correctness checks at compile time.

5.3.1 Using Rust ownership for safe non-blocking communication

Chapter 4 has already given an example of non-blocking communication and how it works in MPI. A non blocking operation immediately returns after being called and all the user knows is that the message will be transmitted at some point if there is no error. As shown with the MPI example this process leaves the data buffer of that operation in an unsafe state because any access on it from the user before the message passing operations has concluded can lead to erroneous data in the program. It is therefore important for the user to have a way of checking on the status of the message passing operation to be able to ascertain when it is safe to be used again. In MPI the `MPI_Wait` provides this behaviour by blocking the process until the MPI library indicates that the buffer is safe again. However even though the MPI standard specifies that the buffer is not to be used before a call to `MPI_Wait` or `MPI_Test` has been made this rule cannot be strictly enforced by MPI implementations. It is up to the user to adhere to this because the MPI implementation has no way of blocking the users access to the buffer.

```
match client.id {
    0 => {
        // [...]
        // buf ownership moved to send_nb function
        let nb_send = client.send_nb(buf, 1, 0).unwrap();

        // Sending a synchronous message in the meanwhile
        let sync_mes = String::from("Synchronous message");
        client.send(&sync_mes, 1, 1).unwrap();

        // buf ownership regained
        buf = nb_send.data();
    },
    1 => {
        // [...]
        // starting non-blocking receive
        let nb_recv = client.receive_nb::<Vec::<i64>>(0,0).unwrap();

        // Meanwhile receiving a synchronous message
        sync_mes = client.receive(0,1).unwrap();

        // received data's ownership moved to buf
        buf = nb_recv.data();
    },
    _ => (),
}
```

Listing 5.4: heimdallr application using non-blocking and blocking communication

With heimdallr it was my goal to implement this feature in a way where it is not possible to accidentally use unsafe buffers and doing it would lead to a compilation error. Gladly Rust's memory ownership model was the perfect fit for this problem. If the non-blocking send and receive operations are designed in a way where they take ownership of the given data buffer it is not possible for the user to access it anymore. To regain ownership of the buffer another function that semantically functions like `MPI_Wait` has to be called which blocks until the message passing operation has finished and then moves back ownership of the buffer to the caller.

Listing 5.4 shows the message passing code of an application that makes use of heimdallr's non-blocking send and receive functions. The example application sends a large buffer from process 0 to process 1 in a non-blocking manner and is able to send another, synchronous message in the meanwhile. The first statement for both processes is a call to the non-blocking `send_nb` and `receive_nb` functions. They contain two distinct differences to their previously introduced blocking counterparts:

1. The data buffer argument of the send function no longer takes a reference to the buffer but it expects to be transferred the ownership of the data. What this means

in our example is that the `buf` variable can no longer be accessed after the call to `send_nb`. Due to Rust's strict rules for references it is also not possible to have a second active reference to `buf` active in this scope to get around the fact that `buf` will be invalidated. Trying this would lead to a compilation error because Rust does not allow to move ownership of a variable that has an active reference to it in the current scope. This makes it effectively impossible for the caller to get any access to the data buffer while the non-blocking send operation is active and therefore guarantees that the contents of the buffer remain unchanged during the message passing process.

2. The non-blocking send and receive functions have a new return type called `NbDataHandle` that serves as a handle for the transmitted data buffer. For the send function this return value serves as a way to eventually get back ownership of the data buffer and for the receive function it is similarly used to obtain ownership of the received data. The `NbDataHandle` struct features a member function called `data()` that, when called, blocks the process until the message passing operation has finished and then gives ownership of the data buffer to the caller via its return value. In essence it fulfills the same role as `MPI_Wait` but with the important difference that the user is forced to use it to gain access the data buffer.

After the non-blocking send and receive functions have been called the processes in the example exchange another message but this time using blocking communication. This is included in Listing 5.4 to illustrate a possible use case for non-blocking communication where a large message is sent with non-blocking operations to allow the processes to continue work that is unrelated to the buffer that is being sent in the background. It also highlights the need of the message id-tag parameter for the message passing functions. As can be seen the two messages in the example code are given different values (0 and 1) for their id-tag parameter because otherwise there would be a possibility of process 1 not being able to correctly match the incoming messages to the right receive call. If process 0 were to reach its message passing code first it would lead to a scenario where process 1 already has both messages pending when it reaches its first receive call. This is exactly the example used in my reasoning for including the id-tag parameter even for blocking communication because without it there would now be no way of identifying from which send operation on process 0 each of the two messages originated.

The example application ends with both processes requesting ownership of the data buffer from their `NbDataHandle` variables. If the message passing operation is already finished at that point the function calls return immediately but otherwise the processes get blocked until the message has been transmitted completely.

It remains to be explained how exactly heimdallr realizes the concurrent behaviour that is necessary for non-blocking communication to work. Figure 5.5 illustrates the complete interaction between two processes using non-blocking send and receive operations. The timeline for both processes starts with a call to their respective message passing operation. To be able to return as soon as possible to their callers the actual message passing process has to be done concurrently in a new thread that is spawned by the functions. The

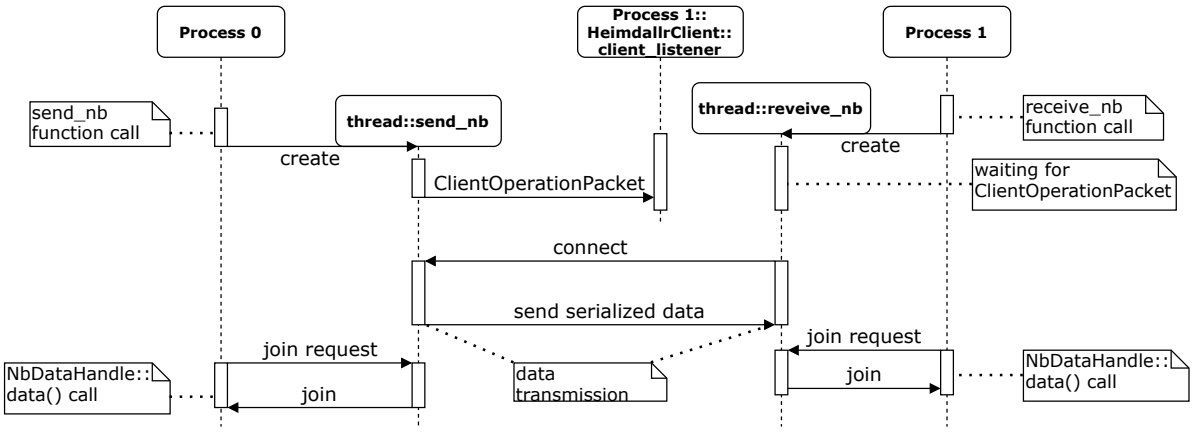

Figure 5.5: Sequence diagram of the message passing process for a non-blocking send/receive operation pair

thread gets passed all necessary information to perform the message transmission and in the case of the send function it also takes ownership of the data buffer from process 0. After the thread creation both functions return and the main threads of both processes can continue their work independently.

The actual process of sending and receiving the message is implemented in exactly the same way as described in Section 5.2.3 for the blocking send and receive operations. First the `ClientOperationPacket` is sent while the receiver waits until it finds a matching incoming message to then receive the actual data and de-serialize it.

As a last step both processes call the `data()` method of their `NbDataHandle` and thereby request ownership of the data buffers. The `NbDataHandle` contains a handle to the thread that was created by the non-blocking function and when the `data()` function is called it waits for the thread's work to finish and then joins it back into the process's main thread. After joining the threads the ownership of the data buffer is returned and the message passing operation has officially been completed, making the buffer safe to be used again.

The described functions clearly behave in a non-blocking way but an interesting question would be whether they can also be categorized as asynchronous. When only looking at the `send_nb` and `receive_nb` functions themselves one might intuitively answer this question with 'yes', but they cannot be looked at in isolation without their concluding `data()` calls that give back data buffer ownership to the caller. The ownership of the data buffers can only be given to the caller after a non-blocking operation has found its counterpart and the message transfer has concluded. Therefore I would argue that these implementations still behave in a synchronous way because they do not fully fulfill the requirements of asynchronous message passing operations. Actually implementing asynchronous operations will need some kind of buffering mechanic to allow the operation on the source process to be truly independent from the state of the matching operation on the target process.

In conclusion I can say that this implementation for non-blocking communication definitely does offer better correctness checks than its MPI counterpart, but it also comes with some drawbacks. The main problem with giving away ownership of the data buffer becomes apparent when only parts of a data structure have to be communicated. Because of the way ownership rules work in Rust it is problematic to only give away partial ownership of an object. In a simple example where only half of the data of an array is supposed to be sent with a non-blocking send operation it would mean that for the duration of the message passing process the whole array would become inaccessible for the function caller. This can be problematic when using heimdallr with more complex real life applications and not just the kind of examples given in this chapter. Chapter 7 will analyze this problem more closely and give possible solutions and approaches on how to design the data structures of a heimdallr application to work correctly with its non-blocking operations.

5.4 Implementing safe shared memory data structures

Chapter 3 talked about one of MPI's shortcomings being a lack of built-in higher abstraction level features. As stated there a big part of the userbase for HPC message passing applications are researchers that need to make their programs more scalable with as little programming effort as possible. Up until this point I have introduced some of the core features that a message passing library needs to support to be useful to a programmer. They are however very basic in their functionality and need to be manually put together by users to construct the structures and mechanisms that are actually needed by parallel applications. What is meant by that is that there are certain patterns and data structures that occur frequently in parallel code and are universally useful to developers. Examples for that would be things like locking mechanisms that are needed to keep certain values globally synchronized over all processes of an application, or simple data structures like a n-dimensional array whose memory needs to be distributed over the processes. Especially in scientific HPC applications it is often the case that the core part of an application is a huge matrix that is split over all running processes for an iterative calculation where specific lines need to be communicated between neighbouring processes in every step.

These are the types of concepts and structures that application developers actually use to solve problems with their programs, but if limited to simple send and receive functions from the message passing library these concepts often have to be implemented from scratch for every application. It would therefore be beneficial if the library came with some built-in higher abstraction level features that can be used 'out of the box' without having to worry about the low level message passing implementations for them. This section will explore some of these features and whether it is possible to design them in a way that they are easy to use and still flexible enough that they are actually applicable for real world applications.

```
1  let counter = Arc::new(Mutex::new(0));
2
3  for i in 0..10
4  {
5      let counter = Arc::clone(&counter);
6      thread::spawn(move ||
7      {
8          let mut value = counter.lock().unwrap();
9          *value += 1;
10         println!("Value: {}, from thread {}", *value, i);
11     });
12 }
```

Listing 5.5: Use of a Rust Mutex for a counter variable

I will especially make use of the unique feature of heimdallr to let the daemon process take part in the runtime communications of an application. This is a design decision that I have not seen in other message passing libraries and it opens up unique opportunities for tasks like having shared data between processes and synchronization tasks. It does however also come with some disadvantages which will be discussed in the respective sections.

5.4.1 A shared mutex implementation

The term mutex stand for the principal of 'mutual exclusion' and is a fundamental component for parallel programming in a multi-threading context. It is used to avoid race conditions in so-called 'critical sections' of a parallel application. These are sections in parallel code where it has to be guaranteed that only one thread can access them at one time. A common use case for a mutex is to guard a shared variable like for example a 'counter variable' that can be accessed and modified from multiple threads. To avoid errors like two threads trying to modify the variable's value at the same time some kind of locking mechanism is needed.

Rust provides a Mutex implementation for use with its built-in concurrency features. An example application is shown in Listing 5.5, where a mutex is used to avoid race conditions for incrementing a counter variable via multiple threads. The mutex is created in the first line and wrapped inside a atomic reference counted pointer which is not too important for the example but needed to safely share references to the mutex between all threads later. The value that is guarded by the mutex is initialized with the value 0 and can from now on only be accessed by making a request to the mutex object.

In the next lines ten new threads are spawned inside the for loop and all are given the task of increasing the counter by one and printing the resulting value. Line 8 shows Rust's implementation of acquiring access to the shared value stored inside the mutex. Calling

the `lock()` method grants the current thread exclusive access to the `counter` variable if the mutex is not currently owned by another thread. If the mutex lock is already held by a different thread the function call instead blocks the current thread until the mutex has been released and then again tries to acquire it. If the lock acquisition is successful `lock()` returns a so-called `MutexGuard`, which in essence is a mutable reference to the value of the mutex variable. As can be seen in lines 9 and 10 the value can be accessed via the de-reference operator '`*`'. The process of releasing the lock is implemented with Rust's `Drop` trait. It can either be called manually or automatically gets called when the `MutexGuard` variable goes out of scope, which in this example would be at the end of threads code.

In a multi-threaded environment mutexes are a very important concept because they are an essential feature to safely work with shared memory. In the message passing context there normally is no such thing as shared memory since all processes have independent memory sections. However that is not to say that message passing applications could not benefit from features that simulate the behaviour of shared memory, where a value is always automatically synchronized between all processes. Normally a feature like this would be realized manually by the programmer through collective message passing operations from all processes like a broadcast, but in my opinion it is worth to provide a general implementation of such shared memory data structures in the library. This could be used in actual applications for data that would otherwise have to be broadcasted often or for things like termination condition or counter variables that always have to be synchronized between all processes. If heimdallr wants to provide some options for shared memory data structures a mutex implementation is a good starting point because as pointed out before it is vital to have some kind of locking mechanism to safely deal with such things in a parallel program.

5.4.2 Heimdallr's mutex implementation

The design goal of the heimdallr mutex was to have it interact with the user in the same way as the normal Rust mutex implementation. That means it is created with a fixed start value, is acquired by simply calling a `lock()` method and that it gets automatically released when the data handle to it goes out of scope.

The first design question that has to be answered is where the mutex's data gets stored. In a pure shared memory system this is obvious because the value can just be stored somewhere in the local memory and all threads will be able to access it. For message passing applications the question gets harder. To use the data it has to be present in the local memory of each process. That also means that there needs to be communication between the processes each time the value is modified somewhere. With standard send and receive communication this would however also mean that each access to the data from one process needs to be anticipated by all other processes because they need to actively participate in the message exchanges for locking and synchronizing the value. These requirements would not make it possible to implement the mutex data structure

with a similar behaviour to the standard Rust one. Therefore a different approach of communication is needed.

The process of exchanging messages via send and receive operations is categorized as two-sided communication. It is called that way because both parties of a message exchange need to be aware that data is being transmitted and they both actively need to call a message passing operation for the message passing process to take place. There exists a different concept called one-sided communication where a called message passing operation circumvents the participation of the target process by gaining direct access to the process's memory and either stores new data there or reads the existing data. This is mostly implemented by using so-called *Remote Direct Memory Access* technology that allows such memory accesses via the network. Newer MPI versions provide one-sided communication operations whose implementation is based on this feature.

This would be a possible solution to implement a mutex data structure that does not require participation of all processes for each access to its data. However in its current state heimdallr does not support this feature yet. Also it would be problematic to implement the requirement of locking the mutex data purely with one-sided communication without having to manually add synchronization code to the applications.

I therefore decided on a different concept which involves the heimdallr daemon. The main problem of having the mutex data spread over all processes was the method of informing all other processes when the data is modified. This would require each process to always listen for messages pertaining to the mutex. So my solution was to not always have a copy of the data on each process but to instead declare a central location where the always updated mutex data resides. In my implementation this takes place on the daemon. Each time a process wants to acquire the mutex it requests it from the daemon and if the data is changed by the process that change then gets communicated back to the daemon.

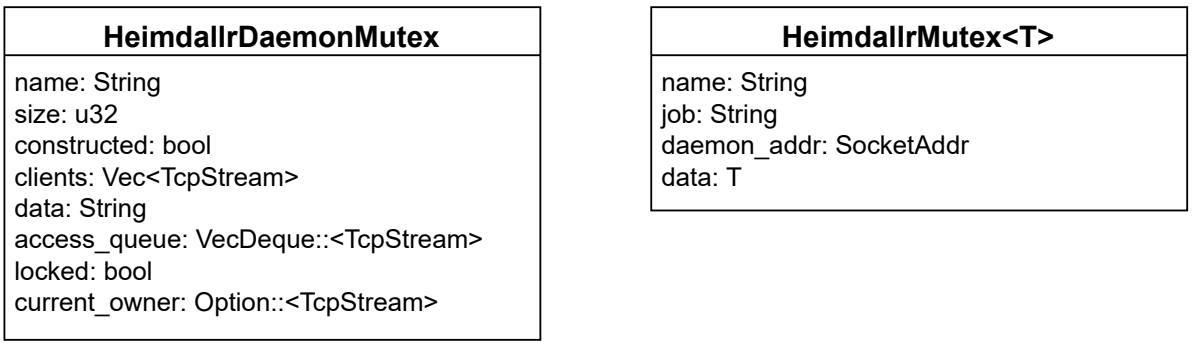

Figure 5.6: Daemon and client structs for the heimdallr mutex implementation

Figure 5.6 shows the two structs that make up the heimdallr mutex implementation.

As can be seen the client side `HeimdallrMutex<T>` is a wrapper around the mutex data. It is a generic data structure of type `<T>` which means that it is able to store arbitrary kinds

of data. The remaining fields are metadata information that is needed for communication with the daemon. The mutex requires a name that is unique to the current job, so that the daemon can match incoming request to the correct mutex data. How exactly the client-daemon interactions work will be explained soon.

The more interesting of the two structs is the `HeimdallrDaemonMutex` which is the data structure that is stored on the daemon for each existing mutex. As already mentioned to match incoming mutex requests on the daemon a combination of the job name and a specific mutex name is used. The job information is implicitly known to the daemon because the mutexes there get stored as a field of the previously introduced `Job` struct.

The next three fields: `size`, `constructed` and `clients` are used in the initialization process of a mutex. To create a new mutex in a heimdallr application a collective `create_mutex` call has to be performed by all clients. This starts an initialization phase that is similar to the initialization procedure of heimdallr itself, where the daemon waits for all processes to send an initialization message before actually creating the mutex.

It might stand out to the reader that the data field of the mutex is of the `String` type and that there is no mention of a generic type parameter in the `HeimdallrDaemonMutex` struct. This is due to the fact that the daemon is actually unaware of the mutex data's type. The data is stored in its serialized form that is transmitted over the network by the clients. This design decision has two reasons:

1. The de-serialization process requires the receiver of a message to provide the correct type information to perform de-serialization. For the message passing operations that we have seen so far this information could always be derived by the compiler from the type of the target buffer of the message passing process. In the case of the heimdallr mutex this is not possible because there is no user written receive code on the daemon side and it can therefore not know how to correctly de-serialize the received data.

   This problem could surely be circumvented for at least all basic Rust types by sending the type information separately in the message to the daemon and implementing the correct de-serialization methods based on that information. However this process does not work when the mutex contains a user defined type since it is not possible for the daemon to know about those.

2. There is actually no need for the daemon to know this information. The daemon is only used as a central storage location for the most recent state of the data and does not perform any action on it. Keeping in mind that de-serialization also produces computational overhead and that the data needs to be re-serialized to be sent back to the clients anyhow it is actually a better solution to just store the serialized data.

The remaining three fields: `access_queue`, `locked` and `current_owner` are the core components of the mutex's locking mechanism. They identify the clients via their active TCP connections to the daemon and use those to keep the mutex data updated and to serve

```
let client = HeimdallrClient::init(env::args()).unwrap();
// Initialization phase
let mut counter = client.create_mutex("counter".to_string(), 0);

{
    // Acquire phase
    let mut value = counter.lock().unwrap();
    value.set(value.get()+1);
    println!("Value: {}, from process {}", value.get(), client.id);
} // Implicit release phase
```

Listing 5.6: Use of a HeimdallrMutex for a counter variable

it to requesting clients. The particular process of how this takes place will be explained in-depth in the next section.

Listing 5.6 is an equivalent implementation to the previously shown mutex example from Listing 5.5 with the difference being that it uses the heimdallr mutex implementation in a message passing program instead of a multi-threading context. The commented lines split the example into three parts which will be used to go through the explanation of the implementation. The whole process is also illustrated as a sequence diagram in Figure 5.7.

The initialization phase

The first step of creating a `HeimdallrMutex` is for all clients to call the `create_mutex` function. As before this functionality is provided by the `HeimdallrClient` which contains all necessary information about how to connect to the daemon and how to identify the client in the sent client operation packets that will be exchanged with the daemon. The create function sends a `MutexCreationPkt` to the daemon that contains the given name of the mutex, the client id and the start data to initialize the mutex in its serialized form.

Looking at Figure 5.7 we can see that even though both clients send the same network packet the daemon reacts differently to them. When the first creation packet arrives the daemon checks if it has already started the creation of a mutex with this name for this job. This is not the case and it therefore creates a new `HeimdallrDaemonMutex` object on the daemon. Here it sets the start value of the mutex data and then it begins to collect all incoming mutex creation packets from different clients until all known clients of the job have announced their participation. We can see this step in the diagram when process 1 executes its `create_mutex` function call and the daemon enters a `register_client` method. Currently the creation is implemented as a collective operation that requires all processes of the job to participate. This can however be modified in future versions if the need for mutexes that only belong to process subsets should occur.

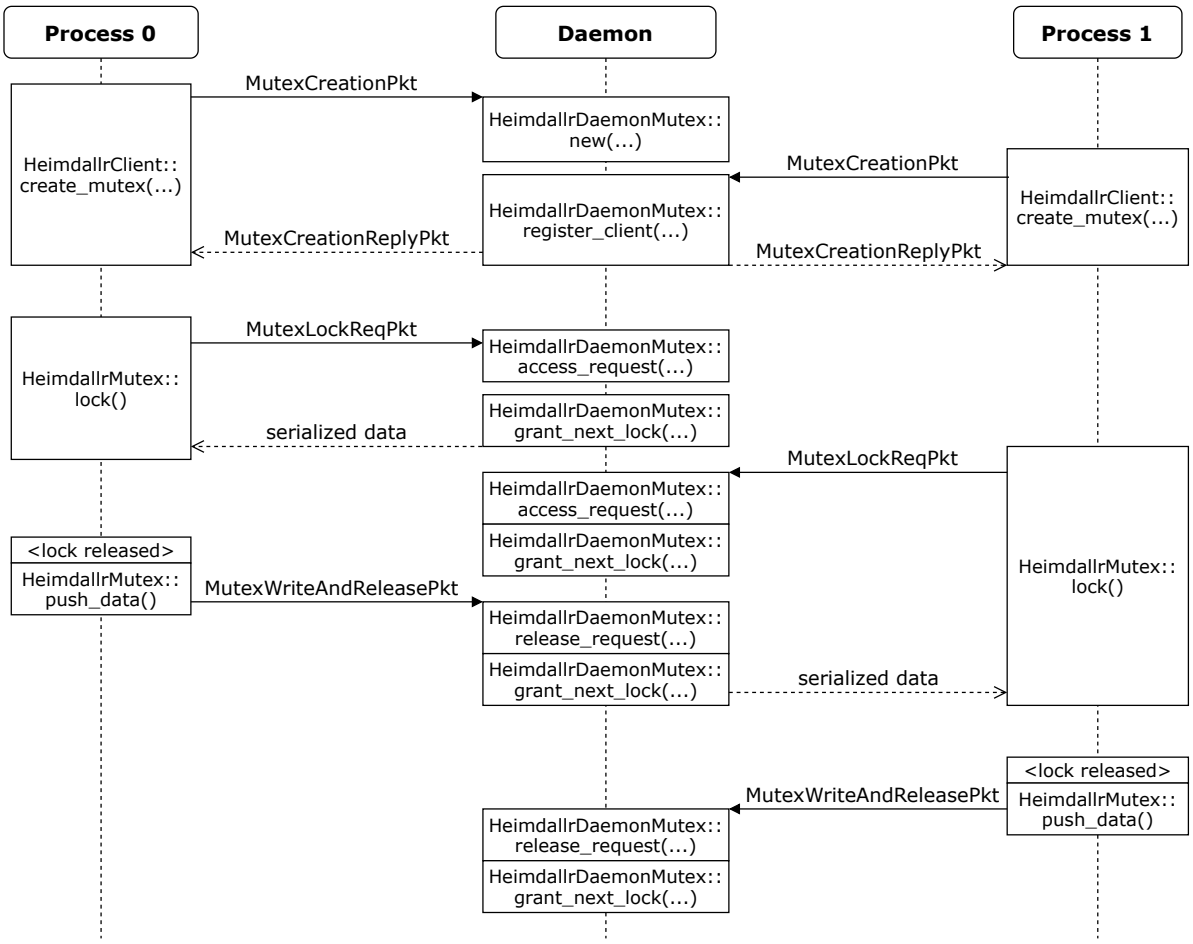

Figure 5.7: Client-daemon communications for a 2-process mutex application.

After a `MutexCreationPkt` has been received from all clients the daemon sends a reply packet that lets the function calls on the clients return from the blocking state that they have in the meanwhile been in and the mutex creation phase concludes.

The acquire phase

We then get into the first mutex acquisition phase of the program. In Listing 5.6 this is triggered by a call to the `lock()` method which requests ownership of the mutex for the current process.

A `MutexLockReqPkt` is sent to the daemon which first matches it to the correct mutex and then its `access_request` function. Here the `access_queue` field of `HeimdallrDaemonMutex` is first used. It uses the first-in-first-out (FIFO) queue semantic and stores active TCP connections to clients that have requested the mutex. The daemon then performs the necessary checks to see whether the lock request can be granted in the `grant_next_lock()` method. In Figure 5.7 the request from process 0 can be immediately granted because there is currently no other process that has acquired the mutex. Therefore the client

connection at the front of the access queue is removed and the inner state of the daemon mutex is updated to being locked.

Now that the mutex has been acquired by process 0 it is send the current state of the data in the serialized form in which it is stored in the daemon mutex. On the side of process 0 the `lock` method can correctly de-serialize the data with the type information that is stored in the client side mutex object. The method however does not return the data directly but instead, similar to the normal Rust mutex, it returns a so-called `HeimdallrMutexDataHandle<T>` struct. This serves as a wrapper around the data and implements the necessary function to later update the mutex data on the daemon.

While this is happening in Figure 5.7 process 1 has also started a lock request. The corresponding network packet arrives at the daemon and the client is inserted into the queue. This time the inner state of the daemon mutex indicates that it is currently locked and the `grant_next_lock` method denies access and does not send a reply back to the client. Therefore the `lock` method on process 1 continues to block the process until a response from the daemon has been received.

In Listing 5.6 we see that the access to the actual mutex data is handled by a pair of `get` and `set` methods. While the mutex is acquired by a client all changes to the data remain local to the process because there is no need to communicate every update to the daemon since it is not possible for any other process to get access to it at this time.

The release phase

The release phase is triggered by the `HeimdallrMutexDataHandle` being dropped by the client, either through an explicit `drop()` instruction or through the variable going out of scope. In Listing 5.6 this happens in line 10 where the current scope is left and all local variables are dropped.

This triggers the implementation of the `Drop` trait for `HeimdallrMutexDataHandle` where the daemon is informed that the lock on the mutex has been released by the current client. The client sends a `MutexWriteAndReleasePkt` that contains the updated mutex data in its serialized form. It is received by the daemon mutex's `release_request` method where the currently stored mutex data is overwritten with the contents of the packet. The TCP connection between daemon and client is then terminated and the mutex state is set to be unlocked again.

Each unlocking action of the mutex triggers a renewed call to the `grant_next_lock` which in our example finds that process 1 has been waiting in the queue. The whole process is then repeated whereby process 1 gains mutex access and is sent the newly updated data.

5.4.3 Discussion of the mutex implementation

The mutex implementation and in general the concept of of simulated shared memory is a rather unique idea to have as a built-in feature in a message passing library. It definitely provides a convenient way for automatically synchronizing data between all processes but it also has some drawbacks, especially when talking about performance.

At first I would like to talk about the correctness aspects of the shown implementation and whether there is the possibility of things like race-conditions, deadlocks or other errors. This is not the case. A race-condition on the data would only be possible if it could happen that multiple processes are granted access to the data at the same time in a scenario where two request are sent simultaneously to the daemon. This scenario is not possible due to the reason that in its current form the heimdallr daemon processes all incoming network packets in a single-threaded sequential order. Therefore, no matter the timing of incoming requests, one of them will always be given priority and other lock requests will be put in the waiting queue.

Other synchronization errors for the mutex's data might however be possible depending on the specific use cases in actual applications. If the mutex is used as a replacement operation for collective MPI operations like for example `MPI_Broadcast` the user has to be careful about reasoning when all processes will actually have received the most current state of the data. Additional process synchronization with for example `barrier` operations might be needed to safely make assumptions about at what time all processes have had their lock requests granted. An example for such a problem will be given in Chapter 7 where the heimdallr mutex is used in a more complex application to replace collective MPI operations.

The active involvement of the daemon in the message passing application also poses questions about the performance impacts of having a central process that has to communicate with all other processes. The sequential processing of all daemon requests might prove as a bottleneck for some communication schemes and slow down the overall application. It especially raises the question of whether the daemon that manages the mutex should be running on the same node as one of the regular heimdallr clients of an application. It might be the case that this would lead to load imbalance between the computing nodes if the daemon has a high workload. Chapter 7 will also explore these performance impacts in specific benchmarks and reason about the feasibility of using this shared data concept in real life applications.

Overall I do think that this approach does look useful for message passing. The introduced mutex implementation can be seen as a fundamental example for operations that include the daemon in their work and in my opinion it could be continued to implement more complex operations. It might for example be useful for collective operations that gather data from all processes and then do further processing on it before relaying a result back to the clients. The described locking mechanics might also be useful for more involved shared data structures like n-dimensional arrays that can automatically handle the necessary message passing to allow processes access to data that is stored on a

different node without having the user manually implement the transmission of data. This is however speculation on future work for heimdallr and nothing that I worked on in the scope of this thesis.

6 Related work

This chapter will introduce multiple projects that relate to the topics of this thesis and compare them to my approach. These include static analysis software for parallel programs, modern HPC languages and Rust implementations for parallel programming.

6.1 Static analysis for MPI and multi-threading

Section 4.3 mentioned the existence of tools that perform correctness checks for parallel programs that cannot be done by the default compiler. This section will first highlight one tool that implements this for MPI programs. Secondly it will look at a tool that performs correctness checks for multi-threaded applications that use *pthreads* and discuss whether the checks performed by it are relevant in a message passing context.

6.1.1 mpi-checker

The *mpi-checker* [DKL15] project is a static analysis tool for MPI applications that provides correctness checks for many of the MPI error classes that have been previously discussed in this thesis. It is built with the LLVM compiler's [LA04] *C* and *C++* frontend *Clang*. The correctness checks are conducted on the so-called *Abstract Syntax Tree* (AST) of the analyzed program. The AST is a representation of the program that gets created in the early stages of compilation. It models the source code in a tree data structure in which instructions like if-statements or compare operations are represented as nodes in the tree and the variables that are used by them as leaf nodes. During the compilation process this program representation is then given to the later stages of the compiler to eventually produce an executable binary. The AST representation is very suitable for static analysis methods because it clearly models the possible code paths that an application can take at runtime and what variables are used in the program's operations.

The *mpi-checker* is built to be aware of the message passing functions that are provided by a MPI library and through the use of the program's AST also their possible uses by multiple processes during runtime, which makes it able to reason about some correctness aspects of MPI applications. The following will go through the listed feature set of the *mpi-checker* and compare its implemented checks to the built-in correctness features of heimdallr to figure out how many of them are already covered by my implementation.

- **Type mismatch**: The type mismatch check looks for incorrect MPI datatype arguments in MPI function calls. As previously discussed this check is necessary because MPI requires the user to manually specify the type of the used buffer as an argument for all functions and a mismatch here can lead to the wrong amount of transmitted bytes. Since C is a statically typed language the type information for the buffer will always be clear to the compiler and a static code analysis can catch this type of error quite easily.

  I have already explained the reasons where this design decision for MPI stems from and how a language with more advanced generic programming concepts can solve this automatically by having the data type of the buffer be deduced during compilation and instead of the data type argument giving the functions a generic data type `<T>` for the buffer argument. Therefore this error class is already handled by my heimdallr implementation and there is no need for a external static analysis approach.

- **Incorrect buffer referencing**: This is a very small check that also stems from the fact that MPI takes pure `void` pointer arguments for the data buffers. It can happen in *C* that the user gets confused with more complicated data structures like for example a 3-dimensional array that might be of the type `double***` and therefore does not pass a correct pointer to the begin of a memory section that contains the actual values of the array but instead passes an address that points to a memory section that just contains pointers to the actual data value sections. This error is not caught by the compiler because the passed pointer gets casted to the `void` type, which in essence just stands for an arbitrary memory address and this can lead to incorrect data being sent. The mpi-checker is able to look for these cases and point out that the passed buffer argument does not point to valid data values.

  In my Rust implementation this error is not possible to happen because the message passing function clearly expect the buffer to be of the correct type, which can either be a reference to the generic type `<T>` for the blocking operations, or a move of the data buffer with the type `<T>` for non-blocking operations.

- **Unmatched point-to-point call**: This check handles the second example given in Section 4.3.2. It looks for send or receive operations that do not have a matching partner, which would cause a process to block indefinitely in the case of blocking operations. The *mpi-checker* however has some limitations here because the used algorithm is only able to determine what instructions belong together if the variables passed as the `rank` argument have the same names and only addition and subtraction operations on their values are allowed. If the limitations are met the *mpi-checker* is able to clearly state whether each message passing operation in the program has a partner operation.

  As previously stated this correctness check cannot be implemented into heimdallr in a way where the default compiler is able to perform it. The Rust compiler is

however built upon LLVM and there are crates for working with the AST of Rust programs. It would therefore definitely be possible to provide similar static analysis methods as the *mpi-checker* in heimdallr's future. The methods and algorithms used by *mpi-checker* would not have to be altered much to make them work with heimdallr message passing operations that semantically behave very similar to their MPI equivalents. The usage of a `rank` argument to identify the target or source of a message is identical to MPI and a static analyser for heimdallr should therefore be able to achieve the same results as `mpi-checker` can for MPI applications.

- **Unreachable call**: The 'unreachable call' check is build upon the previous check and goes one step further in also analyzing whether both of the partner operations of a message transmission can actually be reached in an application run and the processes do not get stuck in previous blocking operations. It is essentially a check for deadlocks caused by the use of blocking MPI operations, like the ones shown in the first example of Section 4.3.2.

  Same as for the last check this can also not be detected during compilation of a heimdallr application. However the same reasoning as above can be applied here and it should be possible to recreate this feature of *mpi-checker* in a heimdallr specific static analysis tool.

- **Non-blocking communication checks**: The *mpi-checker* also provides a group of correctness checks for the use of non-blocking MPI operations like `MPI_Isend`. It is able to detect multiple successive non-blocking operations on the same data buffer, the absence of a `MPI_Wait` and the occurrence of an `MPI_Wait` for a message that was never sent.

  The elimination of such errors was the main focus of heimdallr's implementation of non-blocking communication and all of the above errors are caught at compile time when using heimdallr. Due to the requirement of moving the data buffer's ownership into the non-blocking functions it is not possible to reuse the data buffer until an operation equivalent to `MPI_Wait` has been called by the user. Therefore this correctness check is already built into the design of heimdallr and no static analysis tools are needed here.

Overall we can see that many of the error classes that the *mpi-checker* detects are already taken care of by the basic heimdallr implementation. This shows that heimdallr is successful in getting rid of some of MPI's dangers as it was intended to do. The other described error classes are of a more complex nature and need methods like static analysis to be detectable. In my opinion it would not be possible to design a message passing library in a way that these errors could be automatically detected at compile time without compiler checks that explicitly know about the message passing library.

Adding such features to the Rust compiler would in theory be possible, but it would also mean that users of the message passing library need to use a modified compiler version that is distributed with the added correctness checks because it seems highly unreasonable to expect such changes to be merged into the official compiler. Therefore

the best solution would in my opinion be the future development of a standalone static analysis tool for heimdallr that provides these correctness checks.

6.1.2 helgrind

Helgrind [Dev21] is part of the *valgrind* tool suite. It performs correctness checks for *C*, *C++* and *Fortran* applications that use *pthreads* for parallelization. It provides a variety of different checks for the detection of parallel programming errors and therefore would seem to be a good candidate for a feature comparison to heimdallr's correctness checks.

When looking deeper into the actual types of correctness checks that *helgrind* offers one can see that it heavily focuses on the analysis of shared data usage. It has data race detection algorithms that can for example find multiple threads accessing the same unprotected shared data variables, which can lead to nondeterministic program behaviour. Furthermore it also includes different checks for the correct usage of the *pthread* mutex data structures. These can detect errors like mutex lock patterns that can lead to deadlocks or early de-allocation of memory that is belonging to a mutex which is still active in a different thread.

Most of these error classes have in common that they can only occur because the threads work on a shared memory system and can therefore indirectly interfere with the work of a different thread. There is a fundamental difference in these errors compared to the error classes for message passing applications. Therefore many of *helgrind*'s correctness checks cannot really be applied to a message passing context, since actual shared data does not exist in such applications.

The closest comparison to heimdallr would be to look at the mutex data structure introduced in Section 5.4. Some examples for *helgrind*'s checks for correct mutex usage include 'unlocking a not-locked mutex', 'unlocking a mutex held by a different thread' and 'de-allocation of memory that belongs to a locked mutex'. These types of errors are not possible to occur with the current mutex implementation of heimdallr.

For the cases of incorrectly unlocking a mutex there is no way for a heimdallr client to send a request to the daemon in a way where the daemon would assume that the request came from a different client. Due to the unlocking mechanism being implemented via the *Drop* trait of the mutex's data handle it is also only possible to send exactly one unlock request for a client that has acquired the mutex.

Incorrect modifications of the mutex's underlying memory are also by design not possible with heimdallr because the actual data is not stored in the clients but on the daemon. For each acquisition of the mutex the given client is only handed a local copy of the data, which will be sent as an update to the daemon once the lock has been released. Therefore this problem is also not a concern.

Overall it can be said that the kinds of parallelization errors that can occur are quite different between a multi-threading and a message passing application. Therefore not

many correctness check concepts from the multi-threading domain can be directly transferred to message passing. The way message passing applications are sharing data by essentially sending a copy to another process in and of itself eliminates most of the dangers that shared memory systems have to deal with. If a message passing library wants to introduce the behaviour of shared data, like the heimdallr mutex example, it by design has to take these problems into account and provide solutions like locking mechanisms.

Some of the problems of shared memory data races might occur if the message passing library provides one-sided communication. In that case it would be possible for a process to unknowingly have some of its local memory modified and some additional correctness checks would be needed to make sure that no data races can happen. If the library however stays limited to two-sided communication each process will always be in control of its local data and has to actively partake in its modification.

6.2 Chapel

As we have previously discussed there are many specific requirements for languages or libraries for parallel HPC applications. Since distributed computing is a niche domain in the broader context of programming most languages cannot afford to have some of these required features built into them fundamentally. This is especially the case for mechanics like message passing, which is the basic reason why HPC programmers are reliant on libraries like MPI.

The *Chapel* [CCZ07] project tries to address these problems from a different perspective by not developing a new *C* or *Fortran* library for HPC but instead designing a whole programming language that has data distribution and parallel features as its foundation. The *Chapel* development team has identified many of the same problems with current 'status quo' of HPC programming and they state similar goals as I do in the motivation of this thesis. In a talk [Cha19] at the *HPC Knowledge Meeting '19* the following goals for *Chapel* where stated:

- *Chapel* aims to be as **programmable** as Python
- *Chapel* aims to be as **portable** as C
- *Chapel* aims to be as **scalable** as MPI
- *Chapel* aims to be as **fast** as Fortran
- *Chapel* places importance on static analysis for correctness checking.

These motivations and goals closely resemble my set goals for using Rust and heimdallr in HPC applications. The first two points in essence talk about *usability* and *simplicity*. Rust also puts importance on being easy to use and as discussed in Chapter 5 one of my design goals was to provide a simple and clear programming interface for all heimdallr

```
1 config const numMessages = 100;
2 const MessageSpace = {1..numMessages} dmapped Cyclic(startIdx=1);
3
4 forall msg in MessageSpace do
5   writeln("Hello, world! (from iteration ", msg, " of ", numMessages,
6           " owned by locale ", here.id, " of ", numLocales, ")");
```

Listing 6.1: Example for *Chapel*'s automatic data distribution for arrays

functions. While it is true that *C* is very portable and runs on essentially all current architectures this is not necessarily always true for big *C* applications. As already stated it can often be a lot of work to get *C* programs with many external dependencies to build and run correctly on a different system. This is one aspect of portability in which Rust with its *Cargo* tool for dependency management and as a build system clearly shines.

The next two points are about performance and scalability. *Chapel* aims to deliver the same or even better performance as *C* programs using MPI. Based on some example benchmarks on the *Chapel* project's homepage this does seem to often be the case. This was of course also a goal of my heimdallr implementation in this thesis. I will discuss the performance and scalability aspects of heimdallr more in Chapter 7.

The point of correctness checks is of course the central point of this thesis and my main goal with heimdallr and I have shown that Rust and heimdallr have much to offer in this regard.

Listing 6.1 shows an example *Chapel* program that uses its data parallelism and data distribution features. In line 2 a distributed array is declared with use of the `dmapped` keyword. This automatically causes *Chapel* to distribute the data evenly between all processes that the application is started with. The array is then iterated over by all processes. In the for loop only the processes that own the current index in the array are activated to print the output for this iteration of the loop.

This example showcases a major feature of *Chapel* by being able to distribute the array data automatically in the background. Implementing this example with a common message passing library would take a lot more effort and more verbose code and it should be easy to imagine developers of HPC applications could benefit from such an easy parallel programming model.

Overall it can be said that *Chapel* focuses on many of the same topics as I do in this thesis and also does accomplish them very well in many cases. However there are also some distinct differences between the two approaches and going with the concept of developing a whole new programming language for HPC does come with some drawbacks.

Chapel heavily focuses on specific use cases like scientific calculations based on distributed arrays or matrices. This might be a good fit for many HPC applications but it does not cover all aspects for which message passing can be used. Hiding the underlying message

passing operations completely does take away some of the control from the user and might not make *Chapel* the best choice for other kinds of applications that are not based upon mathematical calculations on distributed data.

Another problem with *Chapel* being an independent language is a smaller support of external libraries and solutions for problems that are not directly involved with the core calculations of an HPC application. One of the big benefits of general purpose languages like *C* and Rust is that they have huge communities of developers from all kinds of domains that develop libraries or Crates for nearly all imaginable use cases. Examples could be topics such as data visualization or I/O APIs for many different file systems and data formats. With a special purpose language like *Chapel* the assortment of available libraries will be a lot smaller in comparison.

Additionally for newer programmers in the field of HPC using something like *Chapel* means to learn a whole new language. For MPI and heimdallr there is a good chance that programmers that come in from different domains might already be familiar with the basic language from a different context which means that the barrier to entry is a lot lower.

6.3 Similar Rust projects

Since this thesis is about message passing in Rust it should be mentioned that other Rust projects exist that tackle this issue. The one that comes closest to what I have worked on in this thesis is called *rsmpi* [rsm21]. It is a Rust wrapper around some supported *C* MPI implementations. The library consists of two main parts. The first is a very thin wrapper around the actual MPI functions which are included as *external C* code. The second part is the surrounding Rust library that gets exposed to the user and grants access to the underlying MPI functions in a more Rust-like style.

The *rsmpi* project provides a lot of MPI's feature set like all different two-sided communication operations and many of MPI's collective operations. It currently does not provide features like one-sided communication and MPI-I/O operations.

Some of *rsmpi*'s features are quite similar to the functionalities of heimdallr but there are also some significant differences. For one the syntax of *rsmpi* can get quite complicated and verbose as can be seen even in the primary example on the crate's Github page [rsm21]. Listing 6.2 shows a small excerpt of the code. As can be seen the syntax is quite involved and would probably not lend itself well for programmers without much prior MPI experience.

The main difference to heimdallr however is the fact that in the end *rsmpi* programs are still compiled into regular MPI applications. This for one means that a compatible MPI implementation is needed to build and run the programs and it also limits the developers to the feature set of MPI.

```
[...]
mpi::request::scope(|scope| {
    let _sreq = WaitGuard::from(
        world
            .process_at_rank(next_rank)
            .immediate_send(scope, &msg[..]),
    );

    let (msg, status) = world.any_process().receive_vec();
[...]
```

Listing 6.2: Small excerpt from the main example in *rsmpi*'s documentation

With heimdallr I first of all tried to provide a simple and safe to use message passing library that is not bound to some of MPI's problems. By having full control over the actual process of sending message over a network I am able to also explore some features that would not be possible with MPI. One example for this is the role of the heimdallr daemon and features like the mutex implementation shown in Section 5.4. Not being dependent on MPI will enable me in the future to explore even more message passing concepts that are currently not well supported by MPI like *fault-tolerance*, or to implement features like *dynamic process management* in a different way to MPI. More discussion about these topics follows in Chapter 8.

7 Evaluation

This chapter will do an evaluation of my message passing implementation in Rust. I will first use heimdallr to parallelize a small scientific application and compare it to an equivalent *C* application with MPI. I will not only discuss the pure performance results but also go into the process of translating the application and talk about necessary design changes that need to be done to compared to the MPI version of the program.

It will then go on to do some micro benchmarks comparisons between heimdallr's message passing operations and their corresponding MPI versions and discuss the results with a focus on how they speak to the viability of using heimdallr instead of MPI in actual applications.

The chapter will close out with a general discussion about my work with heimdallr in this thesis and whether I was able to accomplish the set goals of this thesis with my implementation.

7.1 Parallelizing a real world application with heimdallr

Chapter 5 has shown some small code examples for heimdallr's message passing operations but it remains to be shown that they can actually be applied to the parallelization of real world applications. This is what I am going to do in this section by taking a C application called *partdiff* that is normally parallelized with MPI and by translating it to Rust with the use of heimdallr's message passing functionalities. Since I tried to emphasize the usability aspects of my message passing implementation it should not be more complicated to use it compared to MPI, but it can be expected that some design changes need to be made to the application because of the differences between the languages *C* and Rust and also some key differences between heimdallr and MPI.

7.1.1 The partdiff application

The *partdiff* application is a partial differential equation solver that solves the Poisson equation. The program is used for teaching at the University of Hamburg's Scientific Computing group. There, it is used as an example application for students to learn how to parallelize scientific applications with MPI. This will not go into the mathematical background of what problem the application actually solves but will explain the technical

```
1 for (int i = 1; i < N; i++)
2 {
3     for (int j = 1; j < N; j++)
4     {
5         [...]
6         star = 0.25*(M[i-1][j] + M[i][j-1] + M[i][j+1] + M[i+1][j]);
7         [...]
8     }
9 }
```

Listing 7.1: C code for stencil method

implementation of the used algorithms and data structures to give the reader an idea about how the parallelization process for the application needs to be done.

The base *partdiff* is written in pure *C* and does not feature any parallelization code. The application can be configured with multiple command line arguments that specify parameters like the problem size, what algorithm should be used for the calculations and the program's termination condition (more on that later). Two possible algorithms for solving partial differential equations are supported by partdiff but this chapter will only focus on one of them, which is called the *Jacobi* method.

The core data structure of *partdiff* is a quadratic 2-dimensional matrix that the calculation algorithm iterates over for many steps to incrementally solve the problem. It uses a so-called 'stencil' method to calculate new values for each cell of the matrix per iteration. This process is visualized in Figure 7.1 and the code example in Listing 7.1. The name 'stencil' comes from the fact that a specific shape is moved over the matrix and all cells that are included in the shape are used to calculate the resulting value of the currently selected matrix cell. The stencil method has been identified to be a common pattern in parallel computing [ABC+06], which makes the *partdiff* application a good example benchmark for this thesis. In Listing 7.1 we can see that for *partdiff* the stencil is composed of the four neighbouring cells, which get added up and then averaged to calculate a result value.

The Jacobi algorithm uses two same sized matrices for its calculation, one input and one output matrix. In each iteration of the algorithm the stencil method is used to calculate new values from the input matrix's cells and the results for each cell are then stored at the same location in the output matrix. After each iteration it then changes the 'input' and 'output' status of both matrices so that for the next one the direction is switched. This method of using two matrices allows for an easier parallelization later on because the order in which all stencil calculations on the matrix are done does not influence the resulting values.

The other option of only using one matrix and immediately storing the results for each cell in itself would necessitate the same order of calculations in each iterations to yield

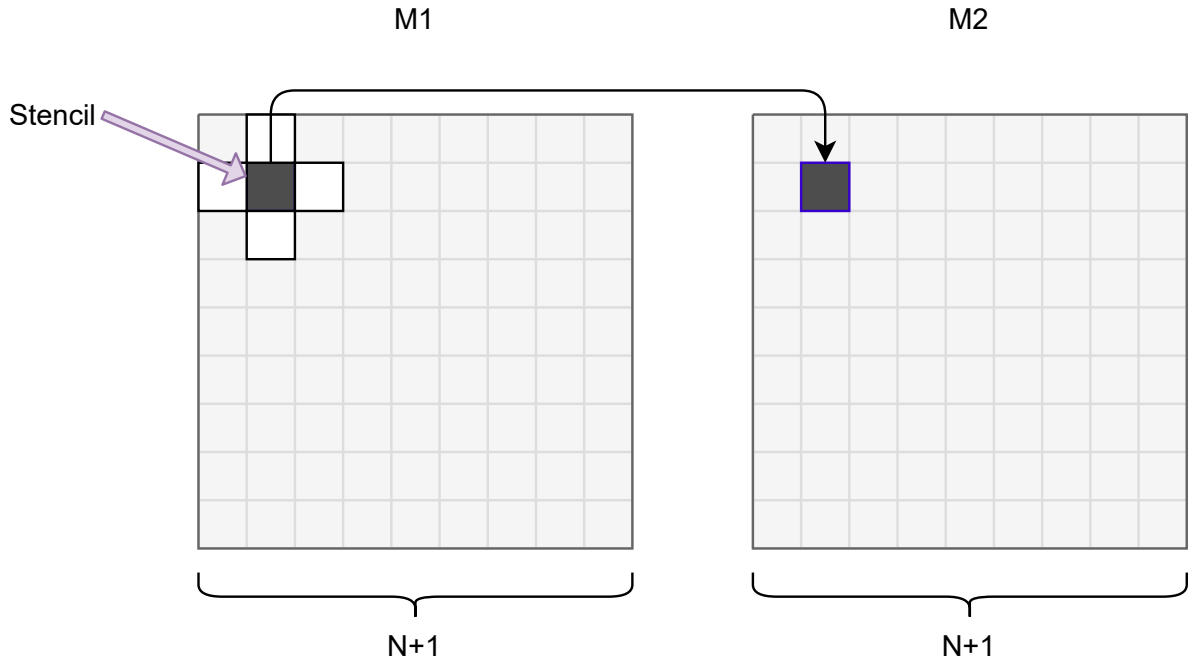

Figure 7.1: Stencil method for Jacobi algorithm

deterministic and reproducible results due to the fact that the result of the stencil method for one cell influences the result for all of its neighbours. *Partdiff* does also feature an algorithm that uses this method but it leads to a harder parallelization scheme and for reasons that will be explained later I chose to focus on the Jacobi variant for this demonstration.

The whole stencil process is performed for many iterations by the application until a termination condition has been met, which then terminates the algorithm and the calculation results are output. *Partdiff* features two different termination conditions. The first one is very simple and just takes a set number of iterations for the algorithm as an input parameter. The second condition is based on the precision of the calculation. For each iteration it monitors the amount by which the result for a cell differs from its previous value and calls that difference the *residuum*. It then identifies the *residuum* with the highest absolute value as the most amount of change that has happened to the matrix in this iteration and calls it the *maxresiduum*. As the iterations go on the value of the *maxresiduum* will go down because the algorithm over time causes the matrix to converge to an almost stable state where only very minute changes happen to the cells. With the precision termination condition the user can set a value for which the algorithm terminates when it exceeds the *maxresiduum* value of an iteration. If this is the case it means that the values in the resulting matrix are precise enough for the user and can be given as output.

The results of *partdiff* are a print out of a subset of the matrix and the value of the *maxresiduum* of the last iteration. The meaning of these values does not need to concern us here but it can be used to compare the parallelized implementations to the results of the default one to make sure that the program still produces the same results after the parallelization process.

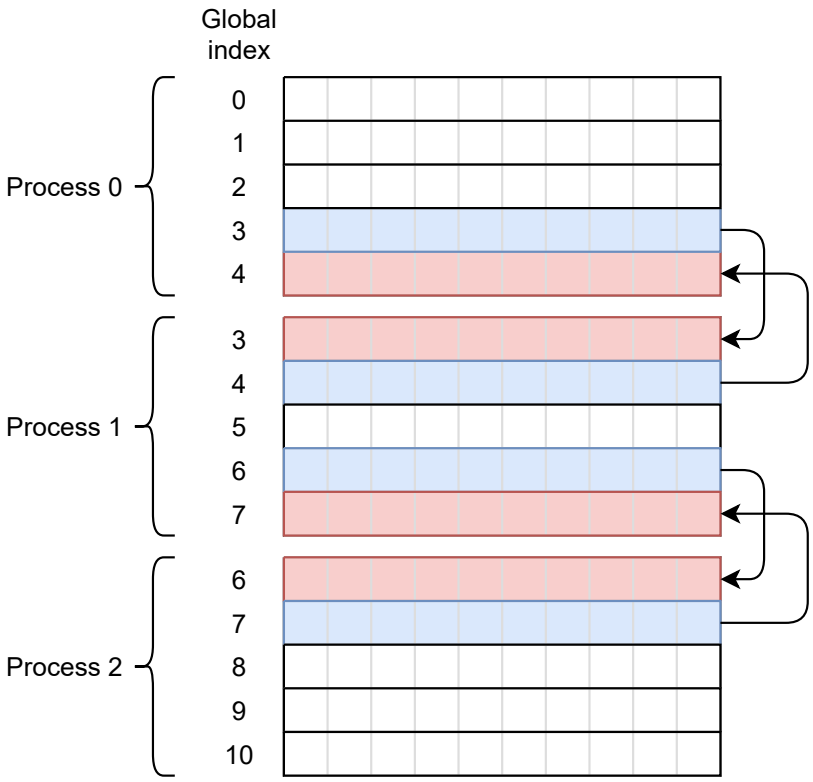

Figure 7.2: Data distribution for parallel partdiff

The parallelization process for the Jacobi version of *partdiff* is relatively straightforward. The goal is to have each iteration of the algorithm done in parallel by giving each process a part of the matrix to perform the stencil calculations on. The best model for this is to split the matrix into multiple blocks which all contain the same amount of lines if possible. If the matrix size cannot be divided evenly by the number of processes the remaining lines are distributed evenly over some of the processes. A visualization of the data distribution can be found in Figure 7.2 where an example for an application run with three processes is given.

When looking at this illustration two facts should stand out. Firstly the processes need to keep the global line index for their local matrices and secondly some of the lines are duplicated over two processes. The first requirement comes from the fact that within the algorithm some additional calculations are done that use the line indices of the global matrix, forcing each process to know what part of the overall matrix they are managing. The second fact is more interesting and pertains to the parallelization process.

As we have seen the stencil method always requires one value from the previous line for its calculation. When we look at the first line of the local matrix of a process that is not process 0 that means that for its calculations it will require information about the last line of the local matrix of the previous process. This is where message passing comes into play. Before the start of an iteration the processes need to exchange the data of their fringe lines with each other to provide them with the correct values of the last iteration. This data exchange is also highlighted in Figure 7.2 with the directed arrows at the right side that also indicate the direction of the message passing operations. The process of iterating over the local matrices always starts on their second line and ends on their second last line. The fringe lines are only read from and contain the previously received data from the neighbour processes.

This is actually all the message passing that is required for the algorithm to work correctly with the termination condition for a set number of iterations. The termination after precision however requires more work. At the end of each iteration a global *maxresiduum* has to be determined, which requires additional message passing and synchronization efforts. How this can be implemented will be discussed in the next section when I compare my heimdallr solution for this with a MPI one.

This should give the reader a basic understanding for the *partidff* application and the required parallelization efforts. The message passing scheme is not too complicated but the *partdiff* is a good example for real life scientific applications, that often work with comparable algorithms that iterate over big distributed matrices. It is therefore a good benchmark case to test heimdallr's performance with varying problem sizes and process counts.

7.1.2 Translating partdiff to Rust and heimdallr

The basic translation of *partdiff*'s sequential *C* code to Rust was quite uneventful. Most of the *C* concepts can be directly translated, such as the basic mathematical calculations and the structs used in the original code. It can be said that some of Rust's advanced features made things like for example command line argument parsing a bit more convenient but I will not go into such detail here since the thesis is not about directly comparing the feature set of these two programming languages.

The most important change in the basic code is the implementation of the central matrix data structure. The differences will be discussed here in detail, because the memory layout of the matrix and how it can be accessed is one of the most important factors for the performance of the program. Its performance must therefore be taken into account later when comparing the two versions of the application in respect to the used message passing library.

In its *C* version *partdiff*'s matrix gets allocated as one continuous chunk of memory. Addressing specific lines and their individual fields is then implemented by using pointer arithmetic on the base address of the matrix and storing pointers to the begin of each line

in an additional array. The fact that all values of the matrix are stored in a continuous memory block has a huge performance impact when iterating over the data in the same order as it is arranged in memory. This is due to the fact that such a memory access pattern produces way less cache misses on the CPU because one cache-line will contain multiple values of the matrix that are needed after each other and also CPU and compiler optimizations that can understand the program's memory access pattern and prefetch the following matrix values into its cache before they are needed.

To get comparable performance results in Rust this concept has to be recreated, but in Rust heap memory allocation works a bit different than in *C*. To achieve the same memory layout I chose to use a Rust vector that also contains all of the matrix's values. For being able to correctly access specific matrix cells I then implemented the '`[]`' index operator trait for the struct that holds the matrix vector with a 2-dimensional index parameter. This yields a syntactically comparable matrix to the *C* version but it still has a big performance drawback. By default a Rust vector performs boundary checking for each access on one of its elements. This is in line with Rust's attention to correctness but will impact the performance of a program that constantly accesses the vector immensely. I therefore had to turn this feature off by introducing a small section of *unsafe* code to the index functions of the matrix that ignores boundary checks for accessing the data. This was absolutely required to get comparable performance results to the *C* version of *partdiff*.

After these changes what remained was to substitute the MPI calls from the original with my own heimdallr message passing functions. In Listing 7.2 a comparison between the MPI and heimdallr versions of the line exchange at the start of an iterations is shown. Both versions use non-blocking send operations to exchange the lines to avoid possible deadlocks when all processes are simultaneously sending their lines. The first and the last process only need to send and receive one line each because they do not have a preceding or succeeding process. Even though the semantics of both versions are equal there are some interesting syntactical differences that can be discussed here.

Firstly the heimdallr version is a lot more concise because of the fewer arguments that are needed for the message passing functions. The design decisions behind this have already been explained in Chapter 5 and the simpler syntax here is a clear benefit of heimdallr. The second difference are the `data()` method calls on the non-blocking send operation's data handles, which are equivalent to `MPI_Wait` calls for `MPI_Isend`. In the MPI version of the code these instructions appear later in the loop but due to some limitations of heimdallr's non-blocking operations they have to appear at this point. I will go into detail about these limitation that were discovered while implementing the heimdallr *partdiff* version in Section 7.3 and just state here that in this case the `data()` calls have to appear in the same scope as their corresponding non-blocking send operations.

The last significant difference is another change to *partdiff*'s matrix data structure that was necessary for the heimdallr version to work correctly. Due to the fact that heimdallr's non-blocking operations take ownership of the data buffer it would mean that the whole matrix would be inaccessible during the message passing process because it is not possible

```
1  if(proc_next <= size-1) {
2      MPI_Isend(&Matrix_In[chunk_size-2][0], N+1, MPI_DOUBLE,
           ↪ proc_next, 0, MPI_COMM_WORLD, &req_up);
3      MPI_Recv(&Matrix_In[chunk_size-1][0], N+1, MPI_DOUBLE, proc_next,
           ↪ 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
4  }
5  if(proc_before >= 0 ) {
6      MPI_Isend(&Matrix_In[1][0], N+1, MPI_DOUBLE, proc_before, 0,
           ↪ MPI_COMM_WORLD, &req_down);
7      MPI_Recv(&Matrix_In[0][0], N+1, MPI_DOUBLE, proc_before, 0,
           ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
8  }
9  ------------------------------------------------------------------------
10 if rank < size-1 {
11     nb1 = client.send_nb(m_in.last1, proc_next, 2).unwrap();
12     m_in.last2 = client.receive(proc_next, 1).unwrap();
13     m_in.last1 = nb1.data();
14 }
15 if rank > 0 {
16     nb2 = client.send_nb(m_in.first2, proc_before, 1).unwrap();
17     m_in.first1 = client.receive(proc_before, 2).unwrap();
18     m_in.first2 = nb2.data();
19 }
```

Listing 7.2: Comparison between MPI and heimdallr message passing of matrix lines

to just give away the ownership of some of the data of the underlying vector. Additionally in its current form heimdallr can only send complete object which would also not be very feasible for this use case. Therefore I modified the matrix data structure in a way that all lines that will be used in message passing operations are stored as separate line vectors. This allows the program to only give ownership of a specific line to the heimdallr operations and to receive data into a specific line as well. This behaviour can for example be seen in the first send operation where the data buffer argument is made up of the `m_in.last1` field of the matrix that corresponds to the second to last line of the local matrix.

This data structure design will have some negative performance impacts on the heimdallr version because the whole matrix is no longer stored as a single continuous memory block, but with a large enough problem size it should not have too much of an impact. It does however show, that working with heimdallr compared to MPI does require a bit of a different program and data structure design for being able to get the benefits of the built-in correctness features.

The last interesting comparison between the MPI and heimdallr versions of the program is how they implement a solution for the previously described termination after precision condition. The problem that needs to be solved is that at the end of an iteration the global *maxresiduum* of the iteration needs to be determined. In a parallelized version of

```
if (options->termination == TERM_PREC || term_iteration == 1) {
    MPI_Allreduce(MPI_IN_PLACE, &maxresiduum, 1, MPI_DOUBLE, MPI_MAX,
        ↪ MPI_COMM_WORLD);
}
-----------------------------------------------------------------------
if (options.termination == TerminationCondition::TermPrec)
    | (term_iteration == 1) {
    {
        let mut mr = global_maxresiduum.lock().unwrap();
        match *mr.get() {
            r if r < maxresiduum => mr.set(maxresiduum),
            _ => (),
        }
    }
    client.barrier().unwrap();
}
```

Listing 7.3: Comparison between MPI and heimdallr implementation of updating *maxresiduum*

the program each process will have a local value for the *maxresiduum* which needs to be exchanged with all other processes to figure out the true result for this iteration and whether the algorithm should terminate.

Listing 7.3 shows the implementations for this process for the two *partdiff* versions. MPI provides a specific solution for this problem in the form of the `MPI_Allreduce` function. This operation collects the value of a variable for all processes and then performs the selected reduce operation on it and returns the result back to all processes. In this case it performs a reduction over the maximum of all *maxresiduum* values which causes all processes to receive the correct value.

For the heimdallr version I could have implemented a similar function either by using the standard message passing operations or by adding a specific `allreduce` function to the library to achieve the same behaviour as the MPI version. However I decided that this scenario would be very suitable to test out the shared memory mutex implementation shown in Section 5.4. This is exactly the kind of use case where such a data structure would be useful and it gives me the opportunity to benchmark the performance of my mutex implementation in a practical example.

In Listing 7.3 we can see that in the heimdallr version a global *maxresiduum* variable exists that is of the `HeimdallrMutex` type. All processes try to acquire it and then overwrite the contained *maxresiduum* value if their local version is larger, which leads to the same reduction behaviour as in the MPI example. This process however requires an additional `barrier` call at the end for synchronization purposes to make sure that no process can go on in the code before all other processes have been granted access to the *maxresiduum*. In its MPI counterpart this synchronization implicitly happens via the `MPI_Allreduce`

because of it being a collective operation that only returns for all processes after the reduction has concluded.

The heimdallr version additionally requires another read of the *maxresiduum* value later on when the actual termination check on the value is performed. I will analyze the actual performance impacts of this implementation in Section 7.1.4 alongside the corresponding benchmark results and also discuss possible optimization to the mutex implementation for use cases like this.

7.1.3 Benchmark results for termination after iterations

For the comparison of the termination after iterations method I chose to do a so-called *weak scaling* benchmark. This benchmark takes a parallel application and tests its scaling behaviour for proportionally increasing process counts and problem sizes. If the number of processes for the application run is doubled so is the problem size. In the case of *partdiff* this means that the matrix size is scaled up accordingly to the number of used processes in MPI or heimdallr.

I chose this type of benchmark because it is very applicable for a real world use of message passing libraries. Most often the purpose of parallelizing large applications is not to simply reduce the runtime of a sequential application, but to enable the application to handle bigger problem sizes for which one computing node does not have enough resources. Therefore the a message passing library becomes necessary to distribute the application over a computing cluster.

In a best-case scenario the application scales well, which in the context of this benchmarks would mean that the runtime for two processes is equal to the runtime with hundreds of processes. The basic benchmark results therefore indicate how good the scaling behaviour of the parallel implementation of the application is. To get comprehensive results for heimdallr's performance I therefore compared *partdiff*'s heimdallr implementation to its MPI version by running them with the same process count and matrix size configurations on the computing cluster of the 'Scientific Computing' group of the University of Hamburg.

In Figure 7.3 we can see the results of the benchmark. All benchmarks were run three times and the resulting runtimes have been averaged in the shown graphs. All benchmarks in this chapter have been run on identical computing nodes with the following hardware:

- 4xAMD Opteron(tm) Processor 6344, with 48 cores
- 128GB RAM
- 40 GBit/s InfiniBand network (using TCP with InfiniBand)

The existing InfiniBand network was used with TCP for MPI and heimdallr.

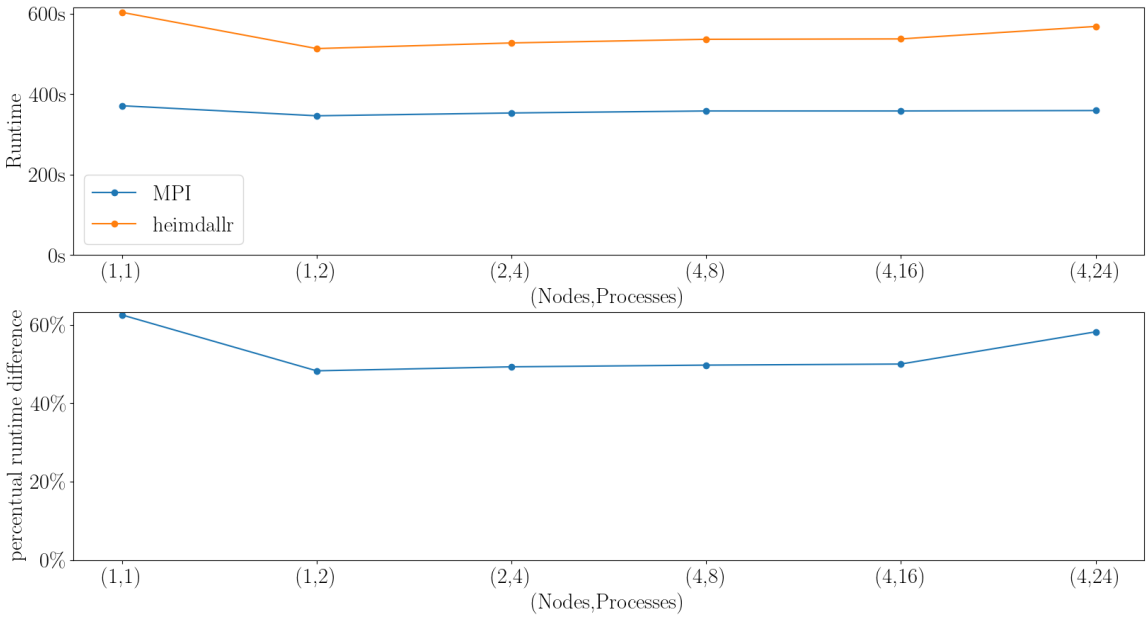

Figure 7.3: Benchmark result comparison for MPI and heimdallr versions of partdiff

Figure 7.3 shows two graphs. The first illustrates the total runtimes for both *partdiff* versions with the number of used computing nodes and overall process count indicated on the x-axis. It also includes a runtime value for the unparallelized versions of the application which do not use any message passing operations.

The first obvious fact from the results is that there is a big difference in runtime between the two sequential *partdiff* versions. The Rust implementation performs significantly worse than the *C* one. This could be due to a variety of reasons. The main one is probably the difference of the matrix data structure in both implementations. It seems that the matrix accesses in the Rust version are not as well optimized compared to the original, which means that it produces a lot more cache-misses and therefore yields a worse performance. This probably stems from the fact that each access on the matrix of the heimdallr version includes some minor calculations to transform the 2-dimensional index into the correct 1-dimensional index into the corresponding vector. This makes it harder for the compiler to perform the same kind of optimizations as the *C* compiler is able to do and might lead to lower chances of the CPU doing the correct cache prefetching operations. This is however hard to prove without deeply analyzing and comparing the created assembly code of both programs. Since the topic of this thesis is not a performance comparison between the two languages I simply accepted the difference in base runtime of the two implementations here and will go on with the analysis of the benchmark results keeping this fact in mind.

The next interesting observation is that both programs yield faster runtimes for running the program in parallel even though the problem size is adjusted appropriately in such a way that each process has roughly the same amount of data in its matrices. One would expect that with the addition of message passing code some overhead is introduced to the application that would lead to worse performance for the algorithm working on the same amount of data as in the sequential version. Because the shown results are averaged over multiple runs it is unlikely that this behaviour stems from measuring variance. This might also be the result of less cache misses by the CPU. Even though the amount of data is the same for every process the individual matrix line get longer with a bigger problem size because the data is distributed in blocks of lines. Therefore it could be possible that the CPU has a better chance of correctly recognizing the application's memory access patterns and is able to perform better optimization procedures.

After discussing these two obvious features of the benchmark results we can come to the actual comparison of heimdallr's performance compared to MPI. What we want to compare with this benchmark is whether heimdallr's message passing implementation can compete with MPI in the context of a real world application. Since the basic runtime of both *partdiff* versions varies by a lot we need a way to eliminate this factor if we want to purely reason about the performance of the message passing operations. The second graph in Figure 7.3 shows by what percentage the heimdallr program was slower for each program run. If this number would stay constant over all data points it would mean that heimdallr shows the same scaling behaviour as MPI. In other words that would mean that heimdallr adds the same amount of overhead to an application as MPI does.

When we look at the graph we can see that this is not exactly the case. Starting from the second data point, which is where message passing is first used, we can see that for each following data point the heimdallr implementation gets comparatively slower. This means that heimdallr does not show the same scaling behaviour as MPI. As the process count grows it looks like heimdallr has more significant overhead compared to MPI. A trend can also be observed towards the end of the benchmark results. At first the decrease in scaling performance grows very slowly and seems to keep itself in reasonable margins. At the step from 4 nodes and 16 processes to 24 process however a stronger decrease in performance manifests itself. This leads to the assumption that with ever growing process counts heimdallr's performance will also degrade further.

There are two possible explanations for this. For one the basic amounts of overhead for every heimdallr message passing operation compared to their MPI counterparts could add up for more processes and therefore have a more visible effect on the programs runtime. In essence, this would mean that heimdallr's message passing operations are always slower than MPI's. Section 7.2 explores this further by showing individual benchmark comparisons for the implemented message passing operations.

The second possible reason for this performance behaviour could come from the fact that the node count for both of these data points remains at 4, which means that there is a lot more intra-node communication for the last data point. Heimdallr's message passing operations currently do not take into account whether a message is transmitted to a

different node or not. It is likely that the used MPI implementation is aware when a message is sent between two processes on the same node and has an optimized way of transferring the data if that is the case. This is also a possible optimization that can be explored for heimdallr in the future.

Overall this benchmark shows that heimdallr is not yet on par with MPI when comparing performance. However this result could be expected since heimdallr is still in its prototype phase and MPI implementations have been tuned for decades of years regarding their performance. In my opinion the results are not too discouraging. Even though heimdallr does not scale as well as MPI the benchmark only shows a relative performance loss of around 10 percent when going from running *partdiff* with 2 processes compared to with 24. It is not ideal but also not as significant as to say that heimdallr is not practically usable. There is still a lot of potential for optimizing heimdallr's performance, which will be discussed more in Chapter 8.

7.1.4 Benchmark results for termination after precision

The benchmarks of *partdiff* running with the termination after precision condition mainly tried to figure out how the heimdallr implementation using a mutex would perform compared to the MPI version using an `MPI_Allreduce` for synchronizing the *maxresiduum* variable.

The benchmarks in this section differ from the ones shown in the last section in multiple ways. First of all I did not do a weak scaling benchmarks again. This is mainly because for the termination after precision it is not really possible to get comparable results for the different data points by increasing the problem size accordingly to the process count. This is due to the fact that the number of iterations that is needed to fulfill the termination condition varies heavily depending on the problem size and it would therefore not be possible to accurately compare the runtime of two program runs with different problem sizes.

The other change made to the *partdiff* settings is that it uses slightly different calculations inside the stencil algorithm. *Partdiff* allows the user to turn on a interference function that is used on every stencil result. This function causes a higher local workload in each iteration and therefore adds runtime to the program and slightly offsets the overhead of message passing operations. In the previous benchmark this function was turned on because it more closely represents the behaviour of generic real world applications where more calculations than just computing the stencil value take place in an iteration.

For the termination after precision condition benchmark I had to turn the interference function off because for larger problem sizes I was not able to find a set of input parameters that produced a program run that did not either terminate immediately after a few iterations or did not terminate at all. Turning the interference function off results in the program spending far less time in one iteration and it therefore increases the impact that message passing overhead has on the application.

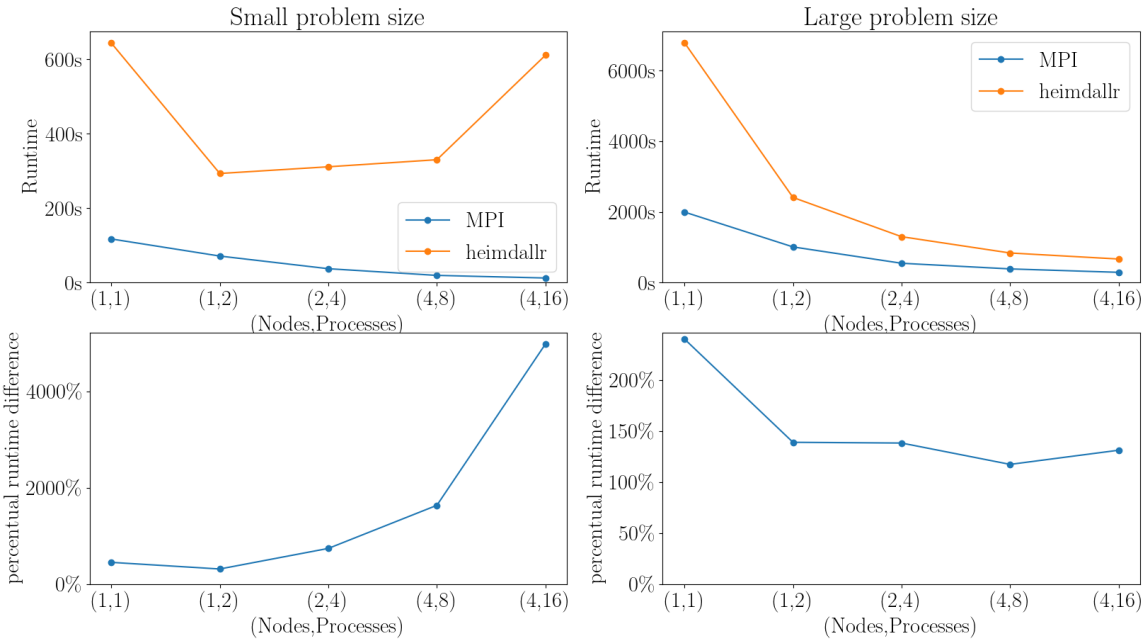

Figure 7.4: Benchmark results for termination after precision condition

Figure 7.4 shows two benchmark results that compare the performance of my heimdallr implementation and the default MPI version. All benchmarks were again run on the same hardware mentioned in Section 7.1.3. The first column ran *partdiff* with a relatively small problem size and increasing process counts. For the second column I quintupled the problem size and ran the program with the same node and process count configurations.

Like in the previous benchmark we can again see that the heimdallr implementation is noticeably slower, even in the sequential case without any message passing. This effect is a lot larger for this benchmark. While the heimdallr version for termination after iterations had less than twice the runtime of the MPI version it goes up to around 3 to 5 times the runtime for this benchmark. I would attribute this difference for the sequential case to the change of turning the interference function off which highlights possible performance problems with the accesses to the matrix data structures.

Since the benchmarks from Figure 7.4 keep the problem size constant for all runs one would hope for a scaling behaviour where adding more processes leads to a decrease in runtime. In the optimal case the runtime would be halved when the process count is doubled. Looking at the 'small problem size' benchmark we can see that this is already not quite the case for the MPI implementation. There is some noticeable overhead for the MPI message passing but the scaling behaviour can in my opinion still be regarded as good.

Looking at the results for heimdallr this is not the case. Even though the jump from no parallelization to two processes looks promising it can be seen that the total runtime even increases when more processes are added. This means that the heimdallr implementation for termination after precision does not scale for the given *partdiff* parameters. Using the heimdallr mutex causes a lot of overhead and waiting time for the processes. This is due to the fact that a lot of mutex accesses are performed by the program. It is not only necessary for each process to acquire the mutex once to update its value. It has to be acquired a second time in the iteration for all processes to receive the final *maxresiduum* value for that iteration. The waiting times introduced by this communication scheme have a far worse effect than the single `MPI_Allreduce` operation in the MPI version.

Performing the benchmarks for this case also revealed a second problem with the heimdallr implementation. A lot of TCP connections have to opened by the application to facilitate the client-daemon communication for the mutex. Depending on the overall calculation time of one iteration and the process count a problem would sometimes occur where the benchmark system ran out of available ports for new TCP connections. This does not mean that there are more active TCP connections than available ports however. The mutex implementation always correctly closes its TCP connections after a message has been sent.

The actual reason for this problem comes from the way TCP connections are implemented in the Linux network stack. After a TCP connection is closed it enters a so-called `TIME-WAIT` state. This blocks the port of the closed connection to be reused for a specified time frame. By default this time limit should be configured to be 60 seconds on most Linux distributions. The purpose of this waiting state is to avoid scenarios where a delayed network packet to a closed connection is received by a newly established connection that got assigned the same port.

In the case of heimdallr this problem could occur in the 'small problem size' benchmark when the process count got too high. In that case each process only had to work with a very small local matrix, which means that the mathematical calculations of the algorithm where performed in a small time frame. This caused each iteration to be very fast and therefore a huge amount of network communication regarding the mutex in a short time frame, triggering the described error.

Seeing these problems one could come to the conclusion that heimdallr's mutex implementation is not very usable in practical applications, but after identifying the causes of its bad performance I decided to run another benchmark with a significantly larger problem size. The results for that can be seen in the second column of Figure 7.4.

We can see that increasing the problem size yielded a much better scaling behaviour for the heimdallr *partdiff*. Ignoring the absolute runtime difference between both program versions the two line graphs look very similar and resemble the type of result that was expected for this benchmark. The 'percentual runtime difference' graph even shows that adding more processes starts to close the performance gap between both programs. The total runtime difference is still in the range of 120 to 150 percent but it does grow smaller

the more processes are used.

The reasons for this dramatic difference between the two benchmarks is the fact that a larger problem size increases the time that each iteration has to spend on its mathematical calculations. This for one means that the message passing operation's time share of an iteration is smaller and therefore does not have such a high impact on the overall performance. Additionally longer calculation times for each process also lead to fewer scenarios where all processes arrive at their mutex acquisition requests at the same time. This means that there is an overall reduction of idle time for the processes because some may still be on the algorithms mathematical calculations when the first processes are ready to update the *maxresiduum* value.

In conclusion I think that the presented results in this section do show that heimdallr's mutex can be a viable tool to be used in practical message passing applications. The 'small problem size' benchmark does show that it is not the most efficient way to handle the synchronization of the *maxresiduum* value in the *partdiff* application, but it does provide other benefits. Testing the performance on small problems will reveal the pure performance differences between the two message passing libraries but it is also not a very realistic scenario to use message passing in such a way. Using message passing parallelization on program runs that can easily be done on one node is not ideal and the typical use case should be for problems that handle a large amount of data and therefore require parallelization over multiple nodes.

For the larger problem size the correct scaling behaviour could be observed. In applications that do not spend most of their overall runtime with executing message passing operations it seems to be feasible to use this feature of heimdallr if it does provide benefits for the developer. One of the strengths of the mutex is that it provides a familiar concept of shared memory for data synchronization for programmers that might only have experience with working on shared memory systems.

It also does provide more functionality than the compared `MPI_Allreduce`. If we for example imagine a scenario where not every process has to access the *maxresiduum* variable in every iteration the heimdallr mutex would be very suitable to use. This would not be as easy to implement with collective MPI operations which require all processes to participate in the operation. It is of course still possible to implement such behaviours with MPI but the heimdallr mutex would make such scenarios trivial to implement. With such examples in mind I would say that heimdallr does provide some benefits for developers by giving them a broader variety of features to implement parallel applications.

```
1  FOR 0 to 500 DO
2  {
3      IF rank == 0
4      {
5          SEND(buf, 1);
6          RECEIVE(buf, 1);
7      }
8      ELSE IF  rank == 1
9      {
10         RECEIVE(buf, 0);
11         SEND(buf, 0);
12     }
13 }
```

Listing 7.4: Pseudo code for micro benchmark applications

7.2 Micro benchmark comparisons of heimdallr to MPI

The benchmarks discussed in the previous sections show that there is an increased overhead caused by using heimdallr message passing compared to MPI. The two *partdiff* implementations already have a significant performance difference when run without any message passing. Due to this it is hard to clearly derive from the previous benchmarks how much of an impact the actual runtime difference between the two message passing libraries has. This section will show a performance comparison between heimdallr and MPI with very simplistic benchmarks applications that purely do message passing without any additional computations in between.

Table 7.1 shows measurements for two semantically equivalent message passing applications with one using heimdallr's blocking and synchronous send and receive operations and the other one using the corresponding MPI operations. The same hardware as in Section 7.1.3 was again used. Listing 7.4 shows a pseudo code version of the benchmarks communication scheme. Overall the benchmark is transmitting 1000 messages containing a data buffer which gets increased in size for each run of the benchmark. The direction in which the messages are sent alternates between process 0 sending to process 1 and vice versa for 500 iterations, yielding a total of 1000 messages. The benchmark was run on two separate computing nodes and the results are averaged over three runs for each data point.

	100B	1KB	5KB	10KB	50KB	100KB	1MB
MPI	0.0289s	0.0293s	0.0356s	0.0470s	0.1158s	0.2007s	1.277s
heimdallr	0.1971s	0.3414s	0.9740s	1.7901s	8.1541s	16.068s	157.60s
serialization	0.0073s	0.069s	0.338s	0.69s	3.40s	6.81s	72.88s

Table 7.1: Measurements for 1000 messages sent with increasing data size

Table 7.1 shows a severe runtime difference between the two message passing libraries for all chosen data sizes. In the following I will try to explain what causes this comparatively bad performance of heimdallr's send and receive operations.

For the lower buffer sizes it might be the case that establishing the two needed TCP connections and the lookup operation in the receive function for finding the correct incoming message could add additional overhead that does not occur in MPI. This process might just be better optimized in MPI and therefore enable a faster transmission of small messages. But even if this is the case this procedure does just add a small runtime overhead and the time needed for these operations is constant and independent of the message size.

Looking at the results for larger messages shows that the size of the runtime difference grows by over one magnitude from MPI being around ten times faster for a 100Byte message to it being over a hundred times faster for a message size of 1MB. Therefore there must be another factor with a big performance cost in heimdallr that causes this runtime behaviour. The actual speed of transferring data over the network is equal for both applications since the benchmarks where performed on the same computing nodes and using the same network connections.

My explanation for this is the data serialization process that is performed by heimdallr for every message before it is sent over the network. As explained in Section 5.2 all data of a message gets translated into a serialized format for the receiver to be able to safely de-serialize it back to the correct data type and memory layout. This process does take computation time which increases with the size of the data buffer. For MPI this process is not needed because it transmits the data as a pure byte stream that gets written directly into the memory of the receiving process.

The overhead introduced by this does not seem too bad for small messages but as we can clearly see from Table 7.1 it does become a big problem for large messages. These results show a big problem for heimdallr to be used in large scale applications. In practice the size of messages in HPC applications that are run over tens or even hundreds of nodes will often exceed even the maximum message sizes measured in Table 7.1. As we can see from the results this would lead to unacceptable runtimes for heimdallr applications when compared to MPI. Therefore this is a problem that must be solved in future versions of heimdallr for it to have a place in real HPC applications.

To illustrate what kind of performance influence the serialization and de-serialization can have I added a third row to the table. These measurements were done with a separate application that performs 1000 serialization and de-serialization operations of the given data size. The results show that indeed the serialization process takes up a significant portion of the send and receive operations runtime. When these results are subtracted from the heimdallr results there still remains a big difference in performance but they do bring the two measurements much closer to each other. It should also be noted that in an actual heimdallr application the serialization times could even be larger. The way the serialization is done in the third benchmark is very straightforward and it does not read

the data from an incoming TCP stream. Furthermore heimdallr also has to take care of the additional serialization of the metadata packets that are exchanged between two clients during a message passing operation.

One possible way to reduce the serialization overhead would be to switch out the currently used serialization format for a better performing counterpart. In its current form heimdallr serializes the message data into the *JSON* format. This choice was made in the beginning stages of development because it is the best supported format of the used Rust serialization library *Serde* and it provided very useful features such as *zero-copy de-serialization*. However there are multiple different Rust crates for serialization and many different formats exist that promise faster serialization and de-serialization. Some of these formats provide features for serializing user defined types and some do not.

Andre Bogus [Bog20] has collected benchmarking results for different serialization formats for which Rust APIs exist. The results show that there should be significant performance improvements possible for heimdallr by exchanging the currently used *JSON* serialization format with a more performant solution.

For optimal performance results and still being able to support all possible data types a more complex implementation of the serialization process will have to be implemented, which can chose the optimal serialization format according to the given data type of the message. This is definitely possible and will be one of the first runtime optimizations for future versions of heimdallr.

However the fact remains that any serialization will always add a certain amount of overhead. This means that even with the optimal data format heimdallr's performance will still be behind MPI. If the overhead could be reduced to acceptable margins this would however not rule out the use of heimdallr if its other features can offer significant benefits over using MPI. It might also be possible to perform the message passing procedure for some of Rust's default types without any serialization, which could in theory yield performance results very close to MPI. This will also have to be explored in the future.

Overall the results in this section show that on a performance level heimdallr is not ready yet to be scaled up to huge HPC applications that send large messages. There is however still a lot of potential in future performance optimizations that could close the performance gap to MPI significantly.

```
let client = HeimdallrClient::init(env::args()).unwrap();
let mut buf = vec![0;5];
let nb: Option<heimdallr::NbDataHandle<_>>;

match client.id {
    0 => {
        nb = Some(client.send_nb(buf, 1, 0).unwrap());
    },
    1 => {
        buf = client.receive(0,0).unwrap();
        nb = None;
    },
    _ => nb = None,
}

match client.id {
    0 => {
        buf = nb.unwrap().data();
        println!("Client {}\n buffer: {:?}", client.id, buf);
    },
    1 => {
        // THIS DOES NOT COMPILE!
        println!("Client {}\n buffer: {:?}", client.id, buf);
    },
    _ => (),
}
```

Listing 7.5: Not compiling example for problems with heimdallr's non-blocking communication

7.3 Problems with heimdallr's non-blocking operations

During the *partdiff* implementation with heimdallr I identified a significant problem with the usage of heimdallr's non-blocking operations. They can fulfill their basic task of transmitting the data in the background but I found limitations on where in the code the `NbDataHandle` that is returned by a non-blocking operation can be used. It seemed that the Rust borrow checker was producing errors as soon as the `.data()` on the data handle was moved out of the same scope of the corresponding non-blocking send operation. This would be problematic for many scenarios in which one would use non-blocking communication, since the whole purpose of it is that the `.data()` method should be able to be called in a later part of the application.

For *partdiff* the optimal use of the non-blocking send operations would be to start the exchange of lines at the beginning of the iteration and only conclude the operations at the end of the iteration. However this was not possible without a compiler error for my heimdallr implementation and I already mentioned this when discussing Listing 7.2 where the compared MPI implementation did not have its `MPI_Wait` calls at the same

```
error[E0382]: borrow of moved value: `buf`
     let mut buf = vec![0;5];
         ------- move occurs because `buf` has type `Vec<i32>`, which
             ↪ does not implement the `Copy` trait

              nb = Some(client.send_nb(buf, 1, 0).unwrap());
                                       --- value moved here

              println!("Client {}\n buffer: {:?}", client.id, buf);
                                   value borrowed here after move ^^^
```

Listing 7.6: Compiler error for Listing 7.5

place as the heimdallr version had its `.data()` calls.

Explaining why this error occurs on the *partdiff* implementation would not be ideal because it contains a lot of code unrelated to message passing and would therefore give very verbose examples. In Listing 7.5 I tried to isolate the problem as good as possible and give a minimal example application on which I will analyse the source of the problem.

Listing 7.5 shows the message passing code of an application that uses a non-blocking send operation to send a buffer from process 0 to process 1. It is made up of three sections. In the first lines we can see the initialization of heimdallr and declaration of the needed variables. This is followed by two **`match`** statements which take care of the message passing process. This example should be understood as a simulated communication scheme of a real application where in between the two **`match`** sections other calculations can take place while the send operation is executed in the background.

In the first **`match`** block process 0 starts to send the buffer in a non-blocking fashion and process 1 posts a blocking operation that receives the message. In the second **`match`** block process 0 then concludes the send operations by calling the `.data()` method on the send operation's data handle and thereby takes back ownership of the buffer. Both processes then print out the content of the buffer and the application terminates.

This program does not compile and instead returns the error message shown in Listing 7.6. This is a good example for the types of compiler warning and error messages that the Rust compiler produces. If the reader has a bit of experience with Rust it is a very clear and readable message that exactly indicates what is wrong. The compiler says that the `println!` statement tries to borrow the `buf` value, which has previously been moved in the `send_nb` function call.

Interestingly the print statement in question is however not the one for process 0 which actually moved the buffer into the send function. The compiler sees a possible error for process 1 accessing the buffer, even though it never calls the `send_nb` function. If the print line for process 1 is removed the program compiles and runs without any problems. This is on the one hand good news because it shows that the concept of moving the

ownership to the buffers data back to the caller via the `.data()` method does work as planned. But on the other hand it highlights a bigger problem that is hard to solve.

What actually causes the error here is that the Rust compiler's borrow-checker assumes that it is possible that process 1 could have also executed the `send_nb` operation which would have move its local `buf` variables and it can therefore no longer access it. The reason for this is probably that the `client.id` value is in theory mutable and could have changed between the two `match` statements. If this were the case it would be possible for process 1 to have also started the send operation and to have given away ownership of the `buf` variable.

Fixing this error proved to be difficult and I was not yet able to solve it completely. For the small example in Listing 7.5 there is a possible solution by simply adding a second buffer variable with a different name that is used to receive the message from process 0. This would solve the problem of being able to correctly access the received data but it does not address the root of the problem. Process 1 would still remain unable to access the `buf` variable for the rest of the program. Also this solution requires twice the memory, which would be very problematic and inefficient in actual applications that work with large message sizes.

A complete solution to this problem would require a way to make clear to the compiler that the id of a process does not change at runtime. In my opinion there are two possible ways to make the application from Listing 7.5 run in its shown form and they both in some way would require advances in the Rust compiler's borrow-checking algorithms.

1. One way to indicate to a compiler that a variable will not change at runtime is to mark it as a constant value. Sadly Rust does currently not support a feature for marking member values of a struct as always immutable. If this was possible the `id` field of the `HeimdallrClient` struct could be marked as immutable and only once defined in heimdallr's initialization step. This should at least in theory give the compiler the needed information to understand that if a process takes the `match` arm for value 1 once it will always take this arm in the future.

2. The other option is that the compiler has to understand from the given code that no modification of the `client.id` value takes place between the two `match` statements in Listing 7.5. This would make it clear that process 1 in both cases takes the `match` arm with the condition 1 and never does move its ownership of the `buf` variable. With my limited knowledge about the implementation of compilers it does seem possible to me that future versions of the Rust compiler could be able to make this deduction. However this is not currently the case and the root cause of the shown error remains to be a fact for heimdallr.

In conclusion this is a significant problem for heimdallr's non-blocking communication and limits its usefulness. Having the message passing operations inside conditional statements for the id of a process is a pattern that will occur in most message passing applications. As explained there are possible solutions to implement heimdallr applications in a way where the programmer can work around the error. For example, it might often be possible

to introduce additional buffer variables for each process or to devise other programming patterns that can move the non-blocking communication operations out of conditional scopes. However this is no ideal way and would defeat the goal of heimdallr being easy to use and understand.

There might be a possible solution to this problem that does not rely on changes to Rust or better algorithms for the compiler's borrow checker but I was not yet able to find one that keeps the improved safety and correctness checks for non-blocking operations and also eliminates the core problem shown in this section.

8 Future Work

This chapter will discuss future ideas for heimdallr and talk about some interesting correctness checking concepts that could not be explored in this thesis. I will first talk about some straightforward features that are currently still missing from my implementation and optimizations that have to be done to make it more competitive with MPI. The possible future work is also not limited to the heimdallr library itself but also includes ideas for external tools that can support heimdallr and provide more complex correctness checks for parallel programs. The chapter will end with some more speculative ideas that could be interesting to explore in the future but have not necessarily been mentioned before in this thesis.

8.1 Improvements to heimdallr

Heimdallr's current feature set is comparatively small to all the functionalities that MPI can provide. I mainly focused on the basic features that are necessary for a message passing library and have not yet implemented a lot of the convenience functions that MPI offers, like for example collective operations like `MPI_Allreduce` or `MPI_Broadcast`. The reason for this, aside from development time, is that my main concern with the thesis was to explore how heimdallr could be designed in a way that offers as many safety and correctness features as possible. The correctness features shown in the previous chapters happen at the lowest level of the message passing process and therefore writing about the implementation of for example a broadcast operation that internally just uses basic send and receive functions would not have been of much interest to the readers in that aspect.

It is however important for users of heimdallr to provide a comparable set of convenience operations to MPI. Since one of my goals with heimdallr was a good usability and productivity such operations definitely need to be implemented in the future to avoid the problem for users of having to re-implement basic message passing features that are generally applicable to parallel programs themselves. For collective operations like *broadcast* or *reduce* it would also be interesting to explore whether they can be implemented in a better way by including the daemon in the process. The standard way of only client-to-client communication should be compared to an implementation that uses the daemon to figure out whether this approach can provide some performance or usability benefits.

MPI does offer a lot of different versions of the *send* and *receive* functions. They provide a lot of combinations of the previously discussed semantics for message passing. This poses the question of whether this should be mirrored in heimdallr. This thesis has only introduced two of these behaviours for heimdallr. First the standard blocking and synchronous send and receive version and secondly a non-blocking but still synchronous version. To provide truly asynchronous message passing the introduction of message buffering would be needed. This is the only way to actually decouple a send operation completely from its receive counterpart.

It is however questionable how much need there is actually for all these different versions of two-sided communication in actual applications. Message buffering for example has the severe downside of doubling the memory usage of the message data buffer. In big HPC applications this memory overhead and the accompanying performance loss caused by copying the data can often not be afforded. This means that the only sensible use case for message buffering is to use it only for small messages. Even then it is often not necessary that the message data is buffered and the same effect for the application could be achieved by using normal non-blocking communication. The rare use case where the send buffer actually has to be modifiable again immediately by using buffering can also be easily solved by the user manually creating a copy of the data and thereby doing the buffering process themselves.

The large variety of possible send and receive operations that MPI provides can also easily confuse the user and some of the existing functions only have very niche use cases. As an example the `MPI_Rsend` function in its specification states that the user has to make sure that the target has to already have posted a matching receive operation when `MPI_Rsend` is called. Otherwise the standard states the result of this function call as undefined. This is a very specific requirement which is hard to fulfill in a parallel application and it should be even harder to prove the correctness of programs that use this operation.

Keeping all of these factors in mind it might be the better solution for heimdallr to keep the variety of its send and receive operations small but therefore their semantics very clear to the user. One blocking and one non-blocking version of these operations should be enough to solve nearly all scenarios that require message passing and it keeps the library easy to use.

Chapter 7 has shown that there are currently two main problems with my heimdallr implementation. The first being a significantly worse performance compared to MPI. This has to be improved upon in future versions of heimdallr for it to be usable in practice. As we have seen the serialization process for messages has a significant influence on heimdallr's performance. The first optimization step there will be to try different serialization formats that can provide faster serialization and especially de-serialization. This will however not completely remove the existing overhead. A second approach would be to add support for the transmission of the unmodified data of a message in the form of byte streams over the network connection. As discussed this process will not work for all Rust data types but it should be possible for many of the basic types

and probably also for some collection types like a vector. If this can be implemented it should reduce the computational overhead for message passing operations significantly and make the performance gap between heimdallr and MPI much smaller.

It would not be very realistic to expect exactly equal or even better performance than MPI but reducing the overhead to more acceptable levels should be enough to make heimdallr usable in practice. Especially when taking into account the benefits of heimdallr that have been shown in this thesis.

8.2 External tool support

Section 4.3.2 showed some common errors that can occur when developing parallel applications. As explained, these types of errors cannot be detected by an unmodified compiler that is unaware of the semantics of the used message passing library. The core feature of heimdallr is supposed to be good correctness checks that make the development of parallel applications easier on the user. It would therefore be beneficial if heimdallr would also provide more sophisticated correctness checks that can detect the presence of errors in the communication scheme of an application. Chapter 6 introduced the *mpi-checker* tool, which performs static code analysis to detect these kinds of errors for MPI programs.

As described, the tool is built upon the *LLVM C*-frontend *Clang*. The Rust compiler also uses *LLVM* as its compiler backend and there are crates [rl21] that allow access to the Rust *AST* that is created by the compiler. This should allow the development of a similar static analysis tool to *mpi-checker* that performs the same correctness checks for message passing operations. Since the behaviour of heimdallr's *send* and *receive* operations is quite similar to their MPI versions the algorithms used by *mpi-checker* should be transferable to also work with heimdallr. Implementing a comparable tool would strengthen heimdallr's ability for correctness checking and is a task that should definitely be undertaken in the future.

The usage of this tool could also be incorporated into the build process of heimdallr applications by providing a wrapper around the rust compiler. This would automatically grant the users better correctness checks without the work of having to run and maintain the static analysis tool themselves.

Another supporting tool for heimdallr that is needed is an equivalent of the *mpirun* command. In its current state each process of a heimdallr application has to be started manually and given the correct parameter for the job run. This process includes manually stating the correct partition and node of the daemon that is responsible for the job and giving the correct process count value to each execution of an applications binary. For easier handling of heimdallr applications a tool should exist that can take care of this process by spawning the correct amount of processes distributed over the selected computing nodes. Some of this work can already be automated if the used computing

cluster provides a job-scheduler but it would be better if heimdallr came with a tool for this job which streamlines the parallel execution of applications out of the box. Implementing this would not be a huge amount of work and would allow users a much more convenient experience with heimdallr.

Other ideas like adding the ability to inspect and debug heimdallr applications at runtime also come to mind but this would be a large project. Developing debuggers that are able to work with distributed programs over multiple nodes is a very complex undertaking and even for MPI there are only a hand full of debuggers and most of them are paid, proprietary software. This goal therefore does not seem too realistic for the immediate future. What could rather be done is adding an optional logging feature to heimdallr, which can be switched on when there are problems with an applications. Supporting this feature with a tool that can visualize the resulting log data seems like a more appropriate goal for a better debugging experience.

8.3 More complex future ideas

One aspect of message passing that has not been discussed in this thesis is the topic of one-sided communication. Transmitting messages via a pair of send and receive functions is commonly referred to as two-sided communication because both processes have to actively participate in the exchange of data. One-sided communication on the other hand only requires one process to call an operation. The operation then uses so-called '*Remote Direct Memory Access*' (RDMA) [Mil17] to access the memory of another process over the network while bypassing the participation of the other process completely.

Having this type of message passing introduces new categories of errors to parallel applications. With two-sided communication it is always clear when a process receives a message that modifies some of its local data. With one-sided communication this is no longer the case because the receiving process's data can be accessed and changed without it having to actively receive the incoming data. Good process synchronization methods and practices have to be followed to ensure a deterministic parallel application without any data races when using one-sided communication. On the upside one-sided communication allows for more freedom in accessing the local data of another process and when deployed correctly it can produce performance increases [SAW$^+$] for an application.

One-sided communication is definitely a good feature for message passing libraries and could be implemented into heimdallr in the future. However ensuring the correctness of parallel applications that contain one-sided communication and designing the corresponding message passing operations in a safe way would be a very complex task and could probably be the topic of its very own thesis.

One other big topic for message passing that pertains to safety and correctness is *fault-tolerance*. For huge applications that are run over tens or even hundreds of nodes it is important how the application reacts to the failure of a process when for example one

computing node crashes or network errors occur. The simplest and most wide spread solution is that the whole application shuts down with an error as soon as one of its processes fails. This is the standard behaviour for MPI. However this might not be the optimal solution because this means that a program run with thousands of processes that have been running for many hours can be destroyed by a single failing process. On big computing clusters this does not only waste time but also a lot of money in the form of wasted energy on an application run that does not produce any results.

Better solutions for this are therefore sought after in the HPC community. What makes this problem so complex is the fact that it is very heavily intertwined with how the specific applications work. As [GL04] states: "*fault-tolerance* is a property of a message passing program coupled with a message passing library implementation". The authors highlight the fact that no library can automatically make message passing programs fault-tolerant, since this is mainly a property of the application itself. In many cases the failure of one process means that parts of the computing data are lost forever which makes it impossible to continue the calculations. Even if the message passing library does not automatically terminate the whole application after a process is lost there is often no way for the program to recover and would have to be shut down anyways. Recovering a partially crashed parallel application requires a combination of fault-tolerance features of the message passing library and application specific code to manage a partial crash.

[GL04] shows a few approaches how *fault-tolerance* can be achieved by the developers of MPI applications. A common practice is the use of *checkpointing*. The application will write out data checkpoints in regular intervals. A checkpoint could for example be all the current data from an algorithms data structures and the accompanying information about the state of the application at the time of the checkpoint. If the application then later crashes at least not all progress is lost but it can be restarted on the checkpoint data.

A more MPI specific concept is the use of *intercommunicators*. These are sub-groups of the processes of an MPI application. All communication is limited to be between the processes of the same *intercommunicator*. This means that the failure of a process only has to lead the failure of the other processes in the same group because the rest of the program can be unaffected by it. This method can be used for suitable applications, where for example many small tasks are given to individual *intercommunicator* groups of processes. If one of them fails the task can be reassigned later but the whole application does not have to fail.

Enabling developers to write their applications with process failure in mind does however require certain features from the used message passing library. Firstly there needs to be a way for the processes of an application to recognize that another process has crashed. This can be solved with good error handling procedures for the message passing operations. Heimdallr does currently return error messages for all of its functions in the case that the message passing has failed. This could be improved upon in the future to give the users as many tools as possible to write their application in a failure tolerant way.

A second feature that would be helpful for *fault-tolerance* is the ability to add new processes to a running application during runtime. Currently the process count for a heimdallr application has to be announced at the start of a job and there is no way to later add more processes, but this functionality could be added in the future. Dynamic process spawning for MPI has been discussed for a long time as for example in [GL95] and [FD00]. At the current time it is implemented with the `MPI_Comm_spawn` operation, which spawns new MPI processes in an *intercommunicator*. Similar solutions for heimdallr would be possible, but with use of the heimdallr daemon it might even be possible to allow for more simple solutions.

Dynamically adding or removing processes from a heimdallr application would not only be helpful for *fault-tolerance* but it would also open up the possibilities for a lot more different types of applications that could be developed with heimdallr. The fundamentals for this approach already exist because of heimdallr's daemon. The daemon keeps all information about currently running jobs and it sounds feasible to expand on this concept and to use it for the dynamic addition of new processes to a running job.

Overall compile time correctness checking for parallel message passing applications is a broad and very complex topic with many possible approaches. A more distant future goal would be direct compiler support for these features. As this thesis has shown many improvements to correctness checking can be made by specific design decisions for the message passing library and external static analysis tools, but the ultimate step would be a compiler that has all these features built-in.

9 Conclusion

This thesis has discussed the design and development of a Rust message passing library for HPC that is focused on correctness and usability. As I have stated my motivations for this come from some of the current shortcomings with MPI and *C*, which are the standard tools used in the development of parallel HPC applications. Even though these technologies have stood the test of time in many aspects they do come with some drawbacks in regards to their usability and safety for non-expert users. In Chapter 3 I have shown that most of the userbase for programming languages and message passing libraries are not necessarily experts in parallel programming but more likely researchers that need this technology to solve their domain specific problems with large parallel applications. It is therefore a good idea to research methods and tools for parallel programming that make it easier on the user and can provide better correctness checks than the existing solutions in this space. Accomplishing this goal could not only increase the productivity of HPC programmers but also open up the world of parallel programming to more people by lowering the barrier of entry.

Chapter 4 worked out some of the most common types of errors in MPI applications and gave suggestions on how a more modern message passing approach that uses Rust's memory safety model and the rigorous correctness checks of its compiler could largely avoid these errors. Chapter 5 then introduced the design for my Rust message passing library heimdallr. With my implementation I was able to provide compile time checks for many of the previously described problems with MPI while also giving a more simplified interface to the users.

Problems of buffer- and type-safety could be solved with Rust's ownership model and generic programming concepts. I was also able to implement a usage pattern for non-blocking communication that can enforce the safe handling of used data buffers. This was again achieved by relying on Rust's memory ownership model which lend itself very well for this purpose.

Furthermore I used the opportunity of building a new message passing library from ground up to also explore some different concepts that do not exist for MPI. In Section 5.4 I implemented message passing operations that include the heimdallr daemon into the execution of parallel programs. The discussed mutex data structure is a first step in providing simulated shared memory operations for distributed memory systems. Regardless of some potential performance problems this concept should be more familiar to developers that come from a domain where parallel programming only takes place in shared memory systems. In my opinion I have shown that this concept can be very

convenient for specific use cases where a value has to be kept synchronized over multiple processes at all time. The locking functionality of heimdallr's mutex implementation could in the future also be used as a basis to provide more complex data structures such as automatically distributed arrays that do not require manual message passing from the user to give processes access to the local memory of other nodes.

Chapter 7 showed that the current version of heimdallr can already be used to parallelize real world applications. I took a small HPC application that was using *C* and MPI and translated it into a Rust application that uses heimdallr for message passing. I explained some of the differences between MPI and heimdallr and that some modifications to the software design are necessary when using heimdallr. This mostly pertains to the layout of a parallel applications data structures. As we have seen the problem of partially moving the ownership of some data came up. The Rust ownership model does not allow the moving of only parts of an object's data. Therefore heimdallr applications and especially their data structures have to be designed with this knowledge in mind.

The *partdiff* example showed that heimdallr is able to produce semantically equivalent parallel programs to MPI but the accompanying benchmarks also revealed some problems with its performance. Even though it was never a realistic goal to achieve equivalent performance to MPI with the work done in this thesis the amount of additional overhead from heimdallr's message passing operations is concerning. Performance is not necessarily the most important factor for users in a message passing library as per the study [ANG+20] cited in Chapter 3, but it does become more significant as the performance gap between to available solutions grows.

The different *partdiff* benchmarks show that it is very use case specific how much slower heimdallr applications are compared to equivalent MPI implementations. For some of the measured cases heimdallr's performance might be acceptable for a user but the micro-benchmarks of the *send* and *receive* operations show that they do not scale very well with growing message sizes. As explained a large part of the computational overhead stems from the used serialization process. If this overhead can be decreased significantly in the future with my proposed solutions heimdallr would become much more usable in practice than it is in its current form.

Overall this thesis clearly showed how the safety and correctness aspects of message passing can be improved upon by different design decisions and the use of a more modern programming language. However my solutions do come with some drawbacks and more future work is required to make heimdallr 'production ready' for real HPC applications. As discussed in Chapter 8 there a many different topics that could be explored with heimdallr in the future.

As a general conclusion it can be said that the work in this thesis looks promising but there is still a lot of potential for future improvements.

Bibliography

[ABC+06] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[ANG+20] Vasco Amaral, Beatriz Norberto, Miguel Goulão, Marco Aldinucci, Siegfried Benkner, Andrea Bracciali, Paulo Carreira, Edgars Celms, Luís Correia, Clemens Grelck, Helen Karatza, Christoph Kessler, Peter Kilpatrick, Hugo Martiniano, Ilias Mavridis, Sabri Pllana, Ana Respício, José Simão, Luís Veiga, and Ari Visa. Programming languages for data-intensive hpc applications: A systematic mapping study. *Parallel Computing*, 91:102584, 2020.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley Sons, Inc., Hoboken, NJ, USA, 2004.

[BCJ01] Rajkumar Buyya, Toni Cortes, and Hai Jin. Single system image. *The International Journal of High Performance Computing Applications*, 15(2):124–135, 2001.

[Bog20] Andre Bogus. Rust serialization: What's ready for production today? `https://blog.logrocket.com/rust-serialization-whats-ready-for-production-today/`, 2020. last accesses February 2021.

[BSD05] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, page 17, USA, 2005. USENIX Association.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[Cha19] Brad Chamberlain. Chapel Comes of Age: A Language for Productivity, Parallelism, and Performance. `https://chapel-lang.org/presentations/ChapelForHPCKM-presented.pdf`, 2019. last accesses February 2021.

[Dev21] Valgrind™ Developers. Helgrind: a thread error detector. https://valgrind.org/docs/manual/hg-manual.html, 2021. last accesses February 2021.

[DKL15] Alexander Droste, Michael Kuhn, and Thomas Ludwig. Mpi-checker: Static analysis for mpi. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[Dur18] Jonathan Dursi. HPC is dying and MPI is killing it. https://www.dursi.ca/post/hpc-is-dying-and-mpi-is-killing-it.html, 2018. last accesses February 2021.

[FD00] Graham E. Fagg and Jack J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[For21] MPI Forum. MPI Forum. https://www.mpi-forum.org/docs/, 2021. last accesses February 2021.

[GL95] W. Gropp and E. Lusk. Dynamic process management in an mpi setting. In *Proceedings.Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 530–533, 1995.

[GL04] William Gropp and Ewing Lusk. Fault tolerance in message passing interface programs. *The International Journal of High Performance Computing Applications*, 18(3):363–372, 2004.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[Mil17] James Alan Miller. Remote Direct Memory Access (RDMA). https://searchstorage.techtarget.com/definition/Remote-Direct-Memory-Access, 2017. last accesses February 2021.

[Plu12] Thomas Plum. C11:TheNewCStandard. http://isocpp.open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3631.pdf, 2012. last accesses February 2021.

[rl21] rust lang. rustc-ast. https://github.com/rust-lang/rust/tree/master/compiler/rustc_ast, 2021. last accesses February 2021.

[rsm21] rsmpi. rsmpi project. https://github.com/rsmpi/rsmpi, 2021. last accesses February 2021.

[Rus21] Rust. Rust lang: The Rust programming language. https://www.rust-lang.org/, 2021. last accesses February 2021.

[SAW+] Hongzhang Shan, Brian Austin, Nicholas J. Wright, Erich Strohmaier, John Shalf, and Katherine Yelick. Accelerating applications at scale using one-sided communication.

[Ser21a] Serde. Serde.rs. `https://serde.rs/`, 2021. last accesses February 2021.

[Ser21b] Servo. Bincode. `https://github.com/servo/bincode`, 2021. last accesses February 2021.

[sr21] serde rs. serde-rs/json. `https://github.com/serde-rs/json`, 2021. last accesses February 2021.

[Str95] Bjarne Stroustrup. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., USA, 1995.

[TOP20] TOP500. TOP500 November 2020. `https://www.top500.org/lists/top500/2020/11/`, 2020. last accesses February 2021.

List of Figures

List of Listings

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Master of Science Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum	Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum	Unterschrift