DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Integration and Test of RUST Tool Support for MPI

Matthias Kübrich

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Integration and Test of RUST Tool Support for MPI

Integration und Test von RUST Tool Unterstützung für MPI

Author:	Matthias Kübrich
Supervisor:	Prof. Dr. Martin Schulz
Advisor:	Bengisu Elis
Submission Date:	October 15, 2020

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, October 15, 2020 Matthias Kübrich

Acknowledgments

I would like to thank my advisor Bengisu Elis for all the invaluable help and advice she offered me in writing this thesis, including all the work that led to its completion.

I also would like to thank my supervisor Prof. Dr. Martin Schulz, who was quick to suggest this work's very subject, which turned out to be highly interesting and enjoyable.

Abstract

With increasingly high performance expectations in modern computer platforms, parallel computing becomes more and more critical. However, implementing bug-free programs to compute efficiently in parallel is, in general, significantly more demanding than implementing their sequential counterpart.

Parallel applications have always been complicated to implement, maintain, or inspect. The Rust programming language aims to provide static programming with zero-cost guarantees for memory-safety. This technology, used to create MPI-Tools, can significantly improve any developers' workflow creating or maintaining HPC applications.

This thesis presents the Rust-Library Rmpi, which provides MPI access with memory safety, a framework implementation for the creation of PMPI -and QMPI-Tools for the programming language Rust, as well as an extensive performance analysis.

Contents

1 Introduction

1.1 Background

In modern days, parallelization is the ultimate method of accelerating computations. As good as every general-purpose processor sold today has some form of parallelization already included, be it Pipelining, SIMD-Instructions, or Multithreading. But parallelization is not confined to single processing units. Using the Message Passing Interface (MPI) enables multiple machines, connected via ethernet, to share the mutual workload by communicating with each other in synchronous and asynchronous ways. This technique allows almost unlimited scaling of computing power, as processors can arbitrarily be added to a system to increase computing power.

While not all problems can be efficiently solved in a parallel manner, hardware improvements for sequential execution cannot be expected to solve performance problems any longer. The so-called *death of CPU frequency scaling* in 2004, marked by the cancellation of a 4-GHz CPU project by Intel [6], also marked the year in which parallelization became the way forward in the competition for computation power.

However, what is often forgotten is the increased software complexity, which is inevitably added with each layer of parallelism. Parallel programming needs to be carefully approached as not to breach any of its strict rules to avoid possible race-conditions, undefined behavior, or similar errors, which are often hard to detect. This problem is mainly present in low-level languages, like C or C++, that have very permissive memory safety rules. As parallel applications often demand the utilization of every performance benefit available, those languages are often still chosen despite the possibly higher development and maintenance costs.

To inspect parallel applications, the MPI library provides an integrated mechanism called PMPI that enables the possibility of intercepting any MPI function called inside the application from an external library. Such a library can be called an MPI-Tool, and it enables inspection, debugging, or even manipulation of an MPI application without changing said application's source code in any way.

PMPI was originally only meant to be used only as a profiling interface but has since encouraged a great variety of tools with varying purposes to be developed, many of which are open-source and publicly available.[5] This tool variety prompts the need to combine functionalities by combining multiple tools. The PMPI interface, being implemented in a simple name-shift fashion, is restricted to using only one single tool because using more would lead to name collisions.

My supervisor Prof. Dr. Martin Schulz, and advisor Bengisu Elis have contributed to solving this problem through the respective implementation of PnMPI[9] and QMPI[4], two libraries that both provide a mechanism to run multiple tools with a single application in parallel.

1.2 Motivation

Despite all the functionality and abstraction MPI tools provide, be it alone or combined, used for continuous surveillance of massive parallel systems, error recovery, or outright debugging, there are some mistakes a system cannot recover from at runtime. Examples are segmentation faults caused by, e.g., reading from buffers using invalid indices, dereferencing null-pointers, use-after-free operations, and double-free operations. Many high-level languages offer memory safety mechanisms, but usually, contrariwise, coupled with garbage collection and a significant decrease in general performance. A language that aims to solve this conundrum is Go, a fast language often used in system-level programming that emphasizes memory safety and safe concurrency by keeping things less convoluted. Go, however, still works with garbage collection and cannot compete with C in performance. Rust is a language created to succeed C. It is statically compiled, has static memory management, and can achieve similar application performance as C. The benefit of Rust is its rich and stringent type system that can prevent many mistakes already at compile-time without any runtime overhead through *zero-cost abstraction*. Rust primarily focuses on safe concurrency and is therefore perfectly suited for complex and very performance demanding applications. An example is Mozilla's experimental browser engine Servo[1], which is implemented in Rust and aims to increase performance by being highly parallel.

Rust can detect all potential undefined behavior by separating code into two categories: safe-code and unsafe-code, in which safe-code is the default where it is semantically impossible to reach undefined behavior. At the same time, in unsafe-code, it's the developer's responsibility to provide memory-safety. Through Rust's strong compatibility with C, it is possible to provide a safe Rust interface on top of an unsafe library implementation written in C.

[1]For more information, see `https://github.com/servo/servo`

1.3 Goal

This thesis presents the library Rmpi, which provides safe access to MPI using Rust's principles on memory-safety and a framework for creating MPI-Tools with Rust, including two generic layer interfaces and two macros for creating environment specific tools for either PMPI or QMPI.

The main goal for the entire presented Framework is, thereby, to provide developers of MPI-Applications and MPI-Tools all advantages Rust can provide.

It includes specifically:

1. maximizing memory-safety
2. maximizing convenience for developers using the Framework
3. minimizing runtime overhead caused through abstractions by using Rust's zero-cost-abstraction mechanisms

1.4 Libraries and Compilers used in this Thesis

1.4.1 MPI

The MPI standard defines the syntax and semantics of essential MPI-functions for the programming languages C, C++, and Fortran, and there exist various implementations, many of them well-tested and open-source. However, despite their standard core definitions, differences can still be found in their respective interfaces, usually in the number of supported functions or the supported version of the MPI-standard. Therefore, this thesis focuses only on using MPI through the MPICH library (version 3.3.x), which is both open-source, popular, and free of charge.

1.4.2 Rust

The Rust compiler is being developed at a rapid pace. Every six weeks, a new minor version is released where backward compatibility generally adheres. Currently, there are two supported Rust editions: Rust 2015 and Rust 2018, which have incompatible syntax definitions. This thesis is therefore referring only to the use of Rust 1.46 edition 2018.

1.4.3 QMPI

QMPI does, at the time this paper was written, not have a versioning system. All references to QMPI will refer to the version described in Elis et al. [5] which has support for 360 different MPI functions and is compatible with the MPICH version stated in subsection 1.4.1.

2 Rmpi

The primary purpose of Rmpi is to gain access to the MPI C-Interface via Rust. The popular Rust-MPI library RSMPI[1] heavily inspires Rmpi as both have the common goal of providing a safe way to access MPI. Rmpi's main difference from RSMPI is that Rmpi aims to stay as close to the C implementation as possible. In particular, every function must have exactly one C counterpart, which it calls internally, and every provided type must be convertible to its C counterpart and vice-versa. (See section 3.3) Rmpi must also have the ability to be used from both tool environments and application environments, meaning that in a tool environment, function-calls should not result in being intercepted as this might result in an interception function intercepting itself, which would most likely lead to infinite recursions. RSMPI, on the contrary, is only designed to operate in an application environment, and access to MPI's C interface below is limited by the fact that RSMPI's types can only be converted into their C counterparts but cannot be reconstructed from them in reverse.

It would be possible to adapt RSMPI by creating a branch of its existing open-source repository[1], but the high complexity of RMPI and some of its implementation details (e.g., see section 2.1.8) favored a reimplementation.

Rmpi has support for a small subset of essential MPI functions, which is designed to be extendable. For a list of supported types and functions, see section 8.1 & section 8.3.

2.1 Implementation

2.1.1 MPI-sys

As is common practice when creating a foreign function interface in Rust, Rmpi, similarly to RSMPI, relies on a module, intended mostly for internal purposes, called MPI-sys, responsible for converting the MPI C-Header into Rust code at compile-time and providing the result as a library. Every single function in MPI-sys is marked as unsafe on principle, which is also common practice, as C functions cannot be checked by Rust's semantic rules and cannot, therefore, have any of their guarantees. Of course, the MPICH implementation has to be considered flawless to be able to use it safely.

[1]Refers to RSMPI version 0.5.4. For more information, see `https://github.com/rsmpi/rsmpi`

However, even with this assumption, no MPI function can truly be considered safe since they either require MPI being initialized or have raw-pointers as one of their input arguments, which are expected to point to a valid location. Both of those requirements remain unchecked at compile-time (and possibly at runtime, depending on the MPI implementation).

Rmpi's version of MPI-sys contains all consequential MPI types and functions in matching memory layout to their C counterpart. This way, MPI-sys can be directly linked to the MPI library in the system. The distinction to RSMPI's version of MPI-sys is that this implementation does not merely declare all `PMPI_*` functions in the same manner but aliases them with the name of their respective `MPI_*` counterparts and places them into a separate module named `pmpi`.

As can be seen in Figure 2.1, which describes the exact module layout of MPI-sys, the `pmpi` module additionally contains all the items the root module contains as well. This enables other libraries (most importantly Rmpi itself) the ability to gain a `mpi_sys` alias module that can only call non-interceptable MPI-Functions when included the following way:

```
use mpi_sys::pmpi as mpi_sys;
```

The primary purpose of MPI-sys is to provide Rmpi with raw, unrestricted access to MPI, allowing Rmpi to create a *safe* interface around it that adheres to Rust's rules for memory-safety and enforces MPI's rules for how to call its library functions. However, MPI-sys itself is meant to be used Rmpi-internally only in most use-cases.

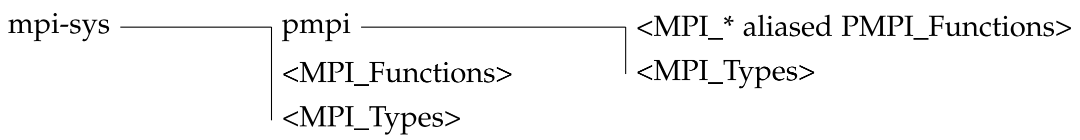


Figure 2.1: Layout of MPI-sys.

2.1.2 RmpiContext

The first and simplest rule that has to be respected when using an MPI-runtime is the necessary initialization and finalization of each process. No MPI function can be used without calling `MPI_Init()`[2] first. Furthermore, not calling `MPI_Finalize()` at the end of a connected process leads to undefined behavior.[8] The MPICH implementation

[2]Rmpi does not support `MPI_Init_thread()` and assumes MPICH is implemented with precompiler variable `MPICH_IS_THREADED` set.

constrains this further by stating that, for each process, the same thread must call `MPI_Init()` and `MPI_Finalize()`.[3]

To enforce this constraint in Rust safe code, Rmpi contains a type called `RmpiContext`, designed to provide the only safe way of accessing MPI through Rmpi. `RmpiContext` should, therefore, only be constructible and usable in circumstances where the use of MPI functions is allowed and should otherwise only be accessible through unsafe code.

An instance of `RmpiContext` can be created by calling `rmpi::init()`, which optionally returns the instance if MPI has not already been initialized (determined by `MPI_Initialized()`). This ensures that access to a RmpiContext can only be gained after MPI has been properly initialized.

To ensure that `MPI_Finalize()` is called as well at the end of the program, RmpiContext implements `Drop`, representing a destructor that automatically handles MPIs finalization when the context instance goes out of scope and is destroyed.

As MPI must only be finalized once, `RmpiContext` cannot have multiple copies of itself, leading to multiple destructions and MPI-finalizations. To ensure that there remains only one instance, `RmpiContext` purposefully does not implement the Rust traits `Copy` or `Clone`, therefore rendering duplication of the instance impossible in safe code. For similar reasons, `Send` is also not implemented, which prevents the instance from being moved to a different thread than the one it was created, while `Sync` is still implemented to allow other threads to access MPI via reference to the `RmpiContext`.

As `RmpiContext` has no other purpose than to provide safe MPI access, which is entirely possible using Rust's zero-cost abstraction, an instance of `RmpiContext` contains no runtime data and should therefore not have any performance disadvantages.

2.1.3 Communicators

The two most essential functions of `RmpiContext` are `comm_world()` and `comm_self()`, which both provide access to a type representation of an MPI-Communicator for all processes or only the current process, respectively. The Communicator type's associated functions provide high-level access to MPI's communication functions that affect all processes inside the communicator. As all of those functions are threadsafe in MPICH, communicators can be sent between threads, and so can their references. However, except for `MPI_COMM_WORLD` and `MPI_COMM_SELF`, an MPI-Communicator has to be freed using `MPI_Comm_free`. Therefore, a communicator has a `Drop` implementation, which automatically frees the communicator if necessary. To prevent multiple free calls on the same communicator, communicators do not implement `Copy`, as every referenced copy would otherwise call the free operation. However, communicators can still be cloned

[3]Defined in MPICH dokumentation: `https://www.mpich.org/static/docs/v3.1/www3/MPI_Init.html`

through the `Clone` trait which creates a deep copy of the communicator by internally using `MPI_Comm_dup()`.

While the communicator construct with its functional constraints is safe in itself, its safe use is still dependent on MPI's initialization. To ensure every communicator lives only as long as MPI is initialized, the Communicator type has a lifetime `'ctx`, which references the lifetime of its corresponding `RmpiContext`, allowing the Rust compiler to ensure that no communicator can outlive `'ctx`.

2.1.4 Process

One other thing a communicator provides is access to its contained processes through the `Process` type. The `Process` type contains a rank and a reference to a communicator. The reference to the communicator is, on the one hand, necessary to ensure that the process instance cannot live longer than the communicator it is contained in, but also provides the necessary input argument for using MPI's send, recv -and similar operations that do not affect all ranks inside the communicator and ensures the correct associated communicator is used. With those guarantees, the `Process` type offers a convenient associated function for communication with significantly less expected input arguments than their C counterparts as communicator and rank are already known and do not have to be specified anymore. See Figure 2.2 for an example where `MPI_Send()`, which has originally six arguments, is implemented using only two while still providing the same functionality.

```
impl<'c> Process<'c> {
    fn send<B: BufferRef>(&self, buffer: B, tag: Tag) -> RmpiResult
    { /* Convert types back into C representation
         call MPI_Send()
         convert return value into RmpiResult */ }
}
```

Figure 2.2: Rmpi's function equivalent to `MPI_Send()`

2.1.5 Status: Receiving Data

The MPI standard defines `MPI_Status` as a data structure that describes the status of a received message. It describes the length of the message and if its transfer had been canceled before its completion. As `MPI_Status` contains only integer data for information purposes, its use is inherently safe and can thereby be safely converted from and into Rmpi's wrapper type `Status` without any runtime cost as their memory layouts are identical. `Status` provides the functions `get_count()` and `test_cancelled()` to retrieve both provided informations.

2.1.6 Request: Safe Asynchronous Communication

One of the main benefits of Rust is its ability to detect and prevent situations that might lead to race-condition at compile time. This is realized by allowing generic lifetime parameters to be declared on types and functions, using them to declare how long certain parameters, return types or type components are allowed to live relative to each other. This feature can also be used to create a safe way of using MPI's asynchronous communication functions. The in Rmpi contained type `Request` aims to represent the MPI type `MPI_Request` only in a safe manner. Therefore a `Request` can only be generated by the respective asyncronous functions. At this point in time Rmpi only implements couterparts for `MPI_Isend()` and `MPI_Irecv()`, the same technigue can, however, also be used on MPI's other asyncronous functions like e.g. `MPI_Ireduce()`. `Request` has four associated functions:

- `free()`:
  - represents `MPI_Request_free()`
  - frees the `MPI_Request` but does not cancel the underlying operation
  - consumes the `Request`
  - this operation is inherently unsafe
- `cancel()`:
  - represents `MPI_Cancel()`
  - cancels the underlying operation
  - consumes the `Request`
- `wait()`:
  - represents `MPI_Wait()`
  - waits for the underlying operation to finish
  - consumes the `Request` if it is not persistant
  - returns a `Status` instance if not set to be ignored (see subsection 2.1.5)
- `test()`:

  - represents `MPI_Test()`
  - tests if the underlying operation has finished
  - consumes the `Request` if it is finished and not persistant
  - returns a `Status` instance if not set to be ignored (see subsection 2.1.5) and the operation has finished

When using asynchronous send or receive operations, the only time where race-conditions can occur is when accessing the corresponding buffer. When calling `MPI_Isend()`, the used buffer must not be changed afterward for as long as it takes for the asynchronous task to read the buffer. However, as `MPI_Isend()` explicitly takes a constant buffer pointer as argument, the buffer can be expected not to change during the sending process. Therefore read operations can still be safely performed. After calling `MPI_Irecv()`, on the other hand, buffers can neither be read nor written to, as long as the message has not been fully received and stored.

To ensure both conditions, each request has an attached generic lifetime that refers to the lifetime of the buffer used to create it. Therefore, an immutably borrowed buffer, used for an asynchronous send operation, can continue to be read, and its reference be copied, but it cannot be written to. On the other hand, a mutably borrowed buffer cannot be used at all after an asynchronous receive operation while still being borrowed. The borrow only ends when the `Request` is destroyed, which can be done by either of the four mentioned functions as all of them are capable of consuming the calling `Request` instance.

The functions `cancel()`, `wait()` and `test()` are safe because they make sure the operation is finished before returning, thereby adhering the lifetime restriction. `free()` cannot be implemented safely. Because after freeing the request, it cannot be known when the communication will be completed.

Request Slice: working with arrays of requests

To support operations on multiple requests, MPI defines several functions that operate on arrays of requests. Such arrays can, besides valid `Request` instances, also contain the value `MPI_REQUEST_NULL`, which marks an empty array field. Rmpi contains the type `RequestSlice` to deal with MPI operations on such arrays. `RequestSlice` is a unsized type similarly to the Rust built-in type `slice` and can be built from such a `slice` of `Request` items without any runtime cost. Only after this conversion is it semantically allowed to store `MPI_REQUEST_NULL` values. This feature is necessary as the corresponding MPI calls set awaited requests to null to mark them as removed.

Similarly, as a single `Request`, the `RequestSlice` type is constrained by a lifetime that applies to all its contained request items.

2.1.7 RmpiResult: Avoiding returns through references

It is common for functions to return data only on certain conditions. The most common scenario is a function that performs an operation that might fail but, in the case of failure, should not terminate the program entirely. An efficient way to solve this problem, which is used multiple times in MPI's C interface, is to return the value via pointer, given in one of the function's arguments. If the return condition is not met, the pointer address is never written to and remains potentially uninitialized, which, in turn, forces the developer to make sure that no read operation occurs after the call in that case in order to avoid undefined behavior.

Rust's semantic, however, disallows undefined behavior entirely, which makes this scenario only possible for primitive pre-initializable types like integers or through the use of unsafe blocks. To avoid this, Rmpi instead uses Rust's specific enum type, which can describe so-called tagged-unions. Rust enums combine an enumeration and a union, allowing the resulting value to, e.g., either contain the successfully constructed result or the error id. Because the value's variant type is now always known at runtime and is recognized by the compiler, an accidental read of an uninitialized value is impossible.

2.1.8 Safe Generic Buffer Representation

The MPI standard supports sending and receiving elements of different datatypes through arrays. This feature is very convenient as it allows the developer to avoid thinking about byte orders and data serialization and enables MPI to work across different platforms with potentially different sets of available primitive types by automatically handling necessary conversions. MPI defines multiple integer identifiers to distinguish between types common in C and Fortran. Additionally, it is possible to create more complex types by combining primitive ones into arrays or structures.

To handle buffers with dynamically typed elements, most functions in MPI have three separate arguments that define one buffer, consisting of one element type identifier, buffer pointer, and array length. As is often the case in such situations, there are many possible erroneous ways of using a buffer, leading to possible undefined behavior. The reason for this is the separation of the buffer into three highly dependent objects, which are treated as independent objects by the compiler. For example, if the length is ignored, the buffer might be accessed at an index location outside its boundary, and if the datatype is ignored, the buffer's content might be wrongly interpreted.

Rmpi Buffer 1.0

A safe buffer representation has to be combined to one single type that automatically checks and prevents incorrect access. Rust already has an unsized type called slice, which becomes a combination of a pointer and a length through any generic pointer type referencing it. However, the type of the elements in the slice remains statically dispatched and cannot be changed at runtime. RSMPI's solution to this is a trait interface called Buffer, which represents a generic buffer and is by default either a slice, a single element, or a dynamic buffer without statically known datatype.[4]

Abstracting a buffer in such a way is not entirely logical or even useful. The slice, which cannot be used as any other object without being wrapped inside a pointer due to its unknown size, is reasonable to have a buffer trait implementation as it could be used by mutable and immutable references in the same implementation. Nevertheless, the dynamic buffer type cannot be used in the same manner, as it needs to save the datatype at runtime. Therefore dynamic pointers are forced to be references themselves, and any generic implementation provided by the buffer trait would be forced to use double references for its dynamic buffer implementation.

This fact also implies that a generic buffer cannot be converted into a dynamic buffer using a generic function without borrowing. It would make it necessary for a caller that wishes to pass on a dynamic buffer as a generic buffer to create an unnecessary double reference. (See section 2.1.8) Nonetheless, the first version of Rmpi's Buffer trait is very similar to RSMPI's version (see Figure 2.3), as it was also implemented on several slice instances containing supported primitive types. The types implementing the trait had to be convertible from -and into their C representations, which would later be needed for the tool layer implementation. (See chapter 3) As the datatype itself is still statically defined in this model, it would be necessary to match a dynamically available datatype against all in the implementation supported types, which is slightly inefficient. (All supported datatypes, see) Also, this implementation cannot handle any user-implemented types that may exist on an application level.

Rmpi Buffer 2.0

To support MPI's theoretically infinite number of datatypes, it is necessary to be able to handle datatypes whose properties are only known at runtime. MPI includes the function `MPI_Type_size()` to provide this functionality by providing a way to calculate the byte length of a datatype, which in turn helps to calculate the size of buffers by using their given array length. Rmpi uses this to implement two type-dynamic buffer types, see Figure 2.4 for mutable and immutable buffers.

[4]RSMPI Documentation: `http://rsmpi.github.io/rsmpi/mpi/datatype/trait.Buffer.html`

```
Buffer {
    fn item_datatype(&self) -> MPI_Datatype;

    fn into_raw(&self) -> (*const c_void, c_int);
    fn into_raw_mut(&mut self) -> (*mut c_void, c_int);

    unsafe fn from_raw<'b>(buf: *const c_void, count: c_int) -> &'b Self;
    unsafe fn from_raw_mut<'b>(buf: *mut c_void, count: c_int)
        -> &'b mut Self;

    \\ convenience items omitted
}
```

Figure 2.3: The Buffer interface version 1.0

Immutable dynamic Buffer-Reference

```
struct TypeDynamicBufferRef<'b> {
    /// has to be alligned
    /// correctly for datatype
    buffer: &'b [u8],
    datatype: RawDatatype,
}
```

Mutable dynamic Buffer-Reference

```
struct TypeDynamicBufferMut<'b> {
    /// has to be alligned
    /// correctly for datatype
    buffer: &'b mut [u8],
    datatype: RawDatatype,
}
```

Figure 2.4: Dynamic datatype reference types

As mentioned previously, there are reasons to avoid mixing unsized types and references as buffer implementations. However, it is not possible to implement a dynamic buffer as an unsized type without avoiding unnecessary reallocations as this would force the datatype to be written, in memory, next to the actual buffer data. Therefore, a switch to a split buffer interface was necessary that only covers buffer references (see Figure 2.5), and is therefore not implemented directly on slices, but only on references to slices. This change makes it necessary to distinguish between immutable buffer references, which MPI uses as send buffers, and mutable buffer reference, used for receiving data. This distinction is necessary because of Rust's borrow-checker, which, e.g., prevents the copying of mutable references but not immutable ones. To prevent duplicate methods, every `BufferMut` is also a `BufferRef` by default. A smaller part of the original `Buffer` trait remains for buffer construction purposes but is only implemented for slice types, while dynamic buffers provide their own method, which requires the runtime datatype identifier as input.

Passing Buffers to functions using static dispatch

In Rust, there are two ways of passing on generic objects as function arguments: dynamic dispatch and static dispatch. Using dynamic dispatch means that there will be only one function symbol in the resulting library, which will receive all the information it needs at runtime. This fact can significantly reduce the library's size but has the disadvantage that the type of the generic argument has to be checked at every function call so it can be treated accordingly. However, using static dispatch will lead the compiler to create a function symbol for each possible type that might be used to call the generic function. However, this separation can be useful for optimization purposes if the type is statically known outside of the function call. Although it is useless when calling the function with a dynamically typed object.

At first, as there are potentially infinite possible MPI-Datatypes, dynamic dispatch might seem reasonable for passing on a buffer function argument. A dynamically dispatched buffer could be represented as a trait object (e.g., `&dyn BufferRef` or `&mut dyn BufferMut`), which would make it possible to pass on any type that is recognized as a buffer to be passed on behind a dynamic reference, including the dynamic buffer itself. A trait object would, in this case, partially defeat the purpose of the dynamic buffer type (see Figure 2.4), which already covers the same functionality in a more efficient although less convenient manner due to the necessary type construction that would have to precede the function call. A dynamic buffer would be more efficient than a trait object due to its representation, which is similar to MPI's C representation, and the absent need for a virtual function table[5].

[5] `https://doc.rust-lang.org/reference/types/trait-object.html`

```rust
trait BufferRef: Sized {
    fn item_datatype(&self) -> MPI_Datatype;
    fn as_raw(&self) -> (*const c_void, c_int);
    fn kind_ref(&self) -> BufferRefKind;
    fn datatype_size(&self) -> RmpiResult<c_int> {
        // ...
    }
    \\ convenience items omitted
}
trait BufferMut: Sized + BufferRef {
    fn as_raw_mut(&mut self) -> (*mut c_void, c_int);
    fn kind_mut(&mut self) -> BufferMutKind;
    \\ convenience items omitted
}

/// Only for buffers with statically known type (slices)
trait Buffer
where for<'b> &'b Self: BufferRef,
      for<'b> &'b mut Self: BufferMut,
{
    unsafe fn from_raw<'b>(buf: *const c_void, count: c_int) -> &'b Self;
    unsafe fn from_raw_mut<'b>(buf: *mut c_void, count: c_int)
        -> &'b mut Self;
    \\ convenience items omitted
}

impl<'b> TypeDynamicBufferRef<'b> {
    pub unsafe fn from_raw_dynamic(
        buf: *const c_void, count: c_int, datatype: MPI_Datatype,
    ) -> Self { /* ... */ }
}
impl<'b> TypeDynamicBufferMut<'b> {
    pub unsafe fn from_raw_dynamic(
        buf: *mut c_void, count: c_int, datatype: MPI_Datatype
    ) -> Self { /* ... */ }
}
```

Figure 2.5: The Buffer interface version 2.0

Usually, however, the datatype that is sent or received through an MPI function is known at compile-time, as types usually are in statically typed programming languages. That is why Rmpi, similarly to RSMPI, supports static type dispatching for buffers, which, nonetheless, can use the dynamic buffer types as input as they implement the buffer interface. (e.g. see Figure 2.6 for function signature example)

```
impl<'c> Process<'c> {
    fn isend<'b, B: BufferRef + 'b>(&self, buffer: B, tag: Tag)
        -> RmpiResult<Request<'b>>
    { /* Perform the operation using MPI_Isend() internally */ }
    fn irecv<'b, B: BufferMut + 'b>(&self, buffer: B, tag: Tag)
        -> RmpiResult<Request<'b>>
    { /* Perform the operation using MPI_Irecv() internally */ }
}
```

Figure 2.6: Function signatures of isend and irecv in Rmpi.

Preventing overlapping buffers

The primary mechanism in Rust that prevents race-conditions is the so-called *borrow checker*. The borrow checker makes sure that there is never more than one mutable reference to any singular value. This rule includes references to slices which are not allowed to overlap if one of them is mutable. Several communication functions in MPI support sending through or receiving into multiple buffers simultaneously, e.g., `MPI_Gatherv()` or `MPI_Scatterv()`. This feature is realized by declaring one big buffer and multiple index displacements contained in said buffer. Doing the same in Rust would be unsafe as there is no way for the borrow checker to prove that the displacements are not overlapping as they are only known at runtime. Instead, in Rmpi, a list of buffers is used to describe the displacements. As buffers are all references to slices themselves, the borrow checker can now guarantee the prevention of any overlaps where they are not allowed.

Describing Null Pointer Buffers

In many of MPI's C function definitions, it is valid to declare a buffer as absent using a null pointer. In Rust, references generally cannot be null and thereby guarantee always to reference a valid value. Rmpi, therefore, converts null pointer buffers to buffers with non-null pointer value that have size 0. This can be done without allocation as any pointer is per definition a valid reference if the referenced value is zero-sized as this fact prevents any actual read or write operations from occurring.

2.1.9 Interface for Internal Tool Usage

For Rmpi to be used in an MPI-Tool environment, it has to be possible to call MPI functions, dependent on the position in the toolchain. As any function pointer, providing a part of the next tool's functionalities, is again implemented in C, Rmpi has to be capable of using an arbitrary set of MPI functions to operate. This is achieved by providing an additional helper function for every safe wrapper function that performs the actual operation using a given generic MPI function. An example for the send function can be seen in Figure 2.7. All of those tool-generic functions are declared as unsafe as the provided MPI function has to be implemented correctly according to the MPI standard.

```
impl<'c> Process<'c> {
    unsafe fn send_with<F, B>(&self, mpi_send: F, buffer: B, tag: Tag)
        -> RmpiResult
        where B: BufferRef,
              F: FnOnce(*const c_void, c_int, MPI_Datatype,
                        c_int, c_int, MPI_Comm) -> c_int,
    { /* Convert types back into C representation
         call mpi_send()
         convert return value into RmpiResult */ }
}
```

Figure 2.7: Send function similar to Figure 2.2 which uses a provided generic version of `MPI_Send()` instead.

3 The Rust Tool Layer

To create an MPI-Tool for either PMPI or QMPI always contains the process of redefining MPI functions, the so-called interceptions, in some way which will then internally call another function that will lead to the return of its original task before its interception. Usually, this internal call either leads to the next QMPI function of the next QMPI tool or the MPI library directly. To support a generic scenario in which an interception function has to work in both PMPI and QMPI environments, a generic function must be given as input, representing the next interception.

The mpi-tool-layer library is responsible for creating such a generic environment, which can later be used for creating a dynamic library that operates as MPI-Tool.

3.1 The Raw Layer

The only way to implement a generic interface in Rust is by implementing a trait on a type. To create a dynamic library that intercepts MPI functions, we would need precisely one type with one implementation of said trait. As this type has no other reason to exist besides providing its associated implementation, it is usually a type without content as it will never be instantiated. Let us call this type *layer*, as it represents one layer of interceptions, independent of context.

The `RawMpiInterceptionLayer` trait defined in the Rust library mpi-tool-layer is an interface for creating an interception layer here defined as *raw layer*. The trait contains 360 different functions that each have one MPI function as their counterpart, and it defines itself, similarly to MPI-sys, by using the same types that are used in C as input -and return types, with the only difference being the first argument which describes the next function call (see Figure 3.1).

3.1.1 Unsafe Compromize

As mentioned in subsection 2.1.1, every MPI function defined in mpi.h has to be considered unsafe in Rust due to C's lack of any memory safety guarantees. In other words: C function calls are often only valid if certain conditions are met, which can only be confirmed by the developer. Using this logic, every function contained in the `RawMpiInterceptionLayer` trait should also be marked as unsafe as they are

```rust
unsafe trait RawMpiInterceptionLayer {
    fn init<F>(
        next_f: UnsafeBox<F>,
        argc: *mut c_int, argv: *mut *mut *mut c_char
    ) -> c_int
    where F: FnOnce(*mut c_int, *mut *mut *mut c_char) -> c_int,
    {
        unsafe { next_f.unwrap()(argc, argv) }
    }
    fn finalize<F>(next_f: UnsafeBox<F>) -> c_int
    where F: FnOnce() -> c_int,
    {
        unsafe { next_f.unwrap()() }
    }

    // 358 functions omitted
}
```

Figure 3.1: Trait defining a raw interception layer.

expected to replace the existing MPI functions. However, as all functions declared in `RawMpiInterceptionLayer` are meant to be overwritten by yet unspecified tools, it is unknown what those functions will do exactly. For example, it would be possible to write a layer that overwrites every function with an empty function that only returns `MPI_SUCCESS` without ever calling `next_f`. Such a layer would be perfectly safe because no unsafe operation is ever invoked. On the other hand, any useful tool should always invoke `next_f` in order not to break the toolchain and thereby preventing any other tool from being executed.

The `RawMpiInterceptionLayer` provides a compromise in this situation. All trait functions are marked as safe; however, it is unsafe to call the `next_f` function, which has to be considered unsafe due to internally calling an MPI function. As this unsafe operation is expected to occur, the trait function declarations are technically not implemented correctly as safe functions. To make the tool developer aware of this, `RawMpiInterceptionLayer` is unsafe in itself and is, therefore, a correct Rust implementation.

3.1.2 The Unsafe Box

Rust has no built-in way of marking the call of a statically dispatched closure as unsafe. Therefore mpi-tool-layer contains a structure called `UnsafeBox`, which generically handles this problem. An `UnsafeBox` is a wrapper around an object that is considered entirely unsafe to use. Therefore the only way to access the inside of the box, which can be done by calling a function called unwrap, is marked as unsafe. As a closure call cannot be unsafe, the effect can be simulated by wrapping it inside an `UnsafeBox`, thereby rendering any access needed to call the closure as unsafe.

3.2 The Layer Trait

The raw layer interface has few to none improvements to its C counterpart in being convenient as it uses the same types and similar method signatures for its interceptions. To use Rmpi's functionality inside an MPI-Tool, interceptions need to be provided with Rmpi's types as input and be allowed to return `RmpiResult` instead of integer results.

The `MpiInterceptionLayer`, also defined inside the mpi-tool-layer library, defines the safe version of `RawMpiInterceptionLayer` and enables more convenient high-level access to MPI. Instead of using three arguments to describe a buffer, one `rmpi::Buffer` argument is given, using static dispatch (see subsection 2.1.8), pairs of MPI-Communicator and rank are instead passed as one `rmpi::Process` and every function gains access to a reference of `RmpiContext` (see Figure 3.2).

`MpiInterceptionLayer` only supports a small subset of MPI functions. Which functions are supported is defined in section 8.1.

```
trait MpiInterceptionLayer: RawMpiInterceptionLayer {
    fn init<F>(next_f: F, args: &mut &mut [CStrMutPtr])
        -> RmpiResult<Option<RmpiContext>>
    where
        F: FnOnce(&mut &mut [CStrMutPtr]) -> RmpiResult<Option<RmpiContext>>,
    {
        next_f(args)
    }
    fn finalize<F>(next_f: F, rmpi_ctx: RmpiContext) -> RmpiResult
    where
        F: FnOnce(RmpiContext) -> RmpiResult,
    {
        next_f(rmpi_ctx)
    }

    // other functions omitted
}
```

Figure 3.2: Trait defining a safe interception layer with high-level Rmpi access.

3.3 Type Conversion

As the high-level interception layer is incompatible with either PMPI's or QMPI's function interface, types have to be converted before every interception and reconverted afterward to use the next tool in the chain. To achieve this, `MpiInterceptionLayer` is implemented on top of `RawMpiInterceptionLayer`, or expressed in Rust semantic, every type implements `RawMpiInterceptionLayer` if it also implements `MpiInterceptionLayer` (automatically). Let us call this implementation the *type conversion layer*. Although Rust does not natively support a class inheritance mechanism as many object-oriented programming languages do, `MpiInterceptionLayer` can, in this case, be seen as a subclass/subtrait of `RawMpiInterceptionLayer` which therefore can be used in the same way for creating MPI-Tool's as its raw counterpart. This relationship is illustrated in Figure 4.1.

3.3.1 Usage of Rmpi's internal Tool Call Support

The type conversion layer uses Rmpi's internal functionality for relaying high-level MPI calls to specified low-level implementation versions (tools), which is described in subsection 2.1.9. The provided `*_with()` functions in Rmpi are used to generate the high-level `next_f` argument for use in `MpiInterceptionLayer` which is implemented to reverse all type conversions in order to internally call the appropriate MPI function in `mpi-sys`. One possible scenario where two in Rust developed MPI-Tools are chained using QMPI is presented in Figure 3.3.

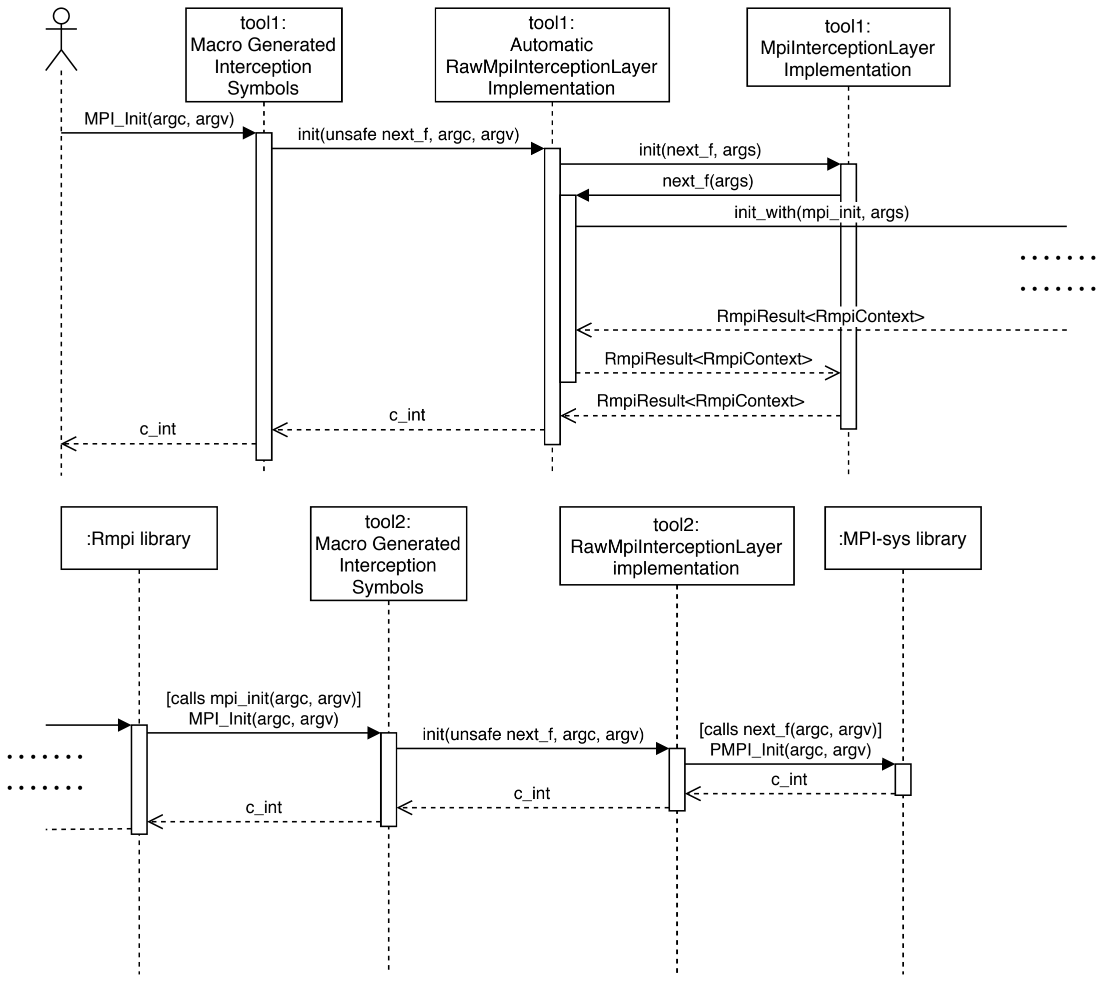

tool1: Macro Generated Interception Symbols
tool1: Automatic RawMpiInterceptionLayer Implementation
tool1: MpiInterceptionLayer Implementation
MPI_Init(argc, argv)
init(unsafe next_f, argc, argv)
init(next_f, args)
next_f(args)
init_with(mpi_init, args)
RmpiResult<RmpiContext>
RmpiResult<RmpiContext>
RmpiResult<RmpiContext>
c_int
c_int
:Rmpi library
tool2: Macro Generated Interception Symbols
tool2: RawMpiInterceptionLayer implementation
:MPI-sys library
[calls mpi_init(argc, argv)] MPI_Init(argc, argv)
init(unsafe next_f, argc, argv)
[calls next_f(argc, argv)] PMPI_Init(argc, argv)
c_int
c_int
c_int

Figure 3.3: Sequence Diagram of a Toolstack Scenario with 2 MPI-Tools.
Tool 1 is implemented using `MpiInterceptionLayer` while Tool 2 is implemented using `RawMpiInterceptionLayer`.

3.3.2 Datatype Matching

The automatic raw layer implementation is responsible for converting all types into -and from their Rmpi versions' while calling the real high-level interception function in between. However, using static dispatch for buffers prevents any simple conversion, as a different type of buffer is needed for every single mpi-datatype. This forces the conversion layer to match the datatype of any possibly given send -and/or receive-buffers and then decide which instance of the generic function call needs to be used. (See Figure 3.4)

Rust defines fixed-sized integer and floating-point types (e.g. `i8`, `u32`, `f64`) that all have matching C type definitions (e.g. `int8_t`, `uint32_t`, `double`) and matching MPI-Datatype ids (e.g. `MPI_INT8_T`, `MPI_UINT32_T`, `MPI_DOUBLE`) (except i128 and u128). However, Rust has no independent build-in types that match C's char, short, int, and similar types but rather redefines the existing types depending on the compiler's target architecture. This constraint has the repercussion that, e.g., a raw buffer with elements of datatype `MPI_CHAR` might end up as Rust type `[u8]` which, when converted backward would then have the datatype `MPI_UINT8_T`. In other words, Rust's `c_char` definition, which is meant to represent the char type of the systems current C compiler, is only a type alias and cannot be distinguished in any way from the actual type it aliases (either u8 or i8), at least not without defining a wrapper type that would exist solely for this purpose.

After experimentation, MPICH proved not to check for datatype equality across ranks and only makes sure the receiving buffer's length in bytes is not smaller than the sending buffer's. The, due to Rust, limited datatype matching does, therefore, not become a significant problem as long as all ranks run on similar architectures. See Figure 3.5 for a working MPI example that exploits this implementation detail of MPICH.

3.3.3 Why Layer 1.0 had to be abandoned

The Layer interface described until now was supposed to be the final solution to providing generic MPI-Tools. However, it has a serious flaw that can be observed in the example implementation in Figure 3.6. The layer implementation shown in this figure is, from a human perspective, valid. Nonetheless, it will not compile due to Rust's borrow checker, which cannot prove that the variable `buf` is still accessible after being passed on to the function `next_f`. The usual procedure to solve this nature's problems is to let `next_f` only "borrow" the buffer instead of owning it entirely. This adaption would lead to the closure type `F` having a buffer input type that lives shorter than the type of the variable `buf`, which would theoretically solve the problem. Unfortunately,

```
unsafe impl<T> RawMpiInterceptionLayer for T
where T: MpiInterceptionLayer,
{
  fn bcast<F>(
    next_f: UnsafeBox<F>,
    buffer_ptr: *mut c_void, count: c_int, datatype: MPI_Datatype,
    root: c_int, comm: MPI_Comm,
  ) -> c_int
  where
    F: FnOnce(*mut c_void, c_int, MPI_Datatype, c_int, MPI_Comm) -> c_int
  {
    if datatype == MPI_UINT8_T {
      let buffer = unsafe {
        <[u8] as Buffer>::from_raw_mut(buffer_ptr, count)
      };
      let rmpi_res = <Self as MpiInterceptionLayer>::bcast(
        |_rmpi_ctx, buf, root| {
          unsafe{root.bcast_with(next_f.unwrap(), buf)}
        },
        // other arguments (converted)
      );
      // return rmpi_res (converted back as integer)
    } else if datatype == MPI_INT || datatype == MPI_INT32_T {
      // (assuming int equals int32_t in C)
      let buffer = unsafe{<[i32] as Buffer>::from_raw_mut(buffer_ptr,count)};
      // ... (continue as for MPI_UINT8_T)
    }
    // match other datatypes predefined in MPI
    else { /* dynamic buffer fallback */ }
  }
  // other functions omitted
}
```

Figure 3.4: Automatic implementation responsible for converting types into their high-level representation. (This is a simplified version of the original code with all macros expanded.)

Rank 0

```
const int arr1[10] = {1,2,3};
MPI_Send(arr1, 3, MPI_INT,
         1, 0, MPI_COMM_WORLD);
```

Rank 1

```
uint32_t arr2[3];
MPI_Recv(arr2, 3, MPI_INT32_T,
         0, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
```

Figure 3.5: Using different datatype definitions for the same type.
This only works if rank 0 uses 32bit-integers.

Rust does not support generic types (in this case `F`) to be generic themselves. This limitation makes it impossible to attach a generic lifetime to `F` thereby offering no way to derive a new type `F′` from `F` with a shortened lifetime. A simple workaround to this problem would be introducing a new lifetime by wrapping every buffer in a reference. In this case:

```
buf: &'buffer_lifetime mut Buf
```

However, this declaration would seem redundant as `Buf` already defines a generic reference to a buffer.

As there is no way to implement a tool that inspects received data with the current layer interface, as can be seen in Figure 3.6, subsection 3.3.4 will introduce a changed interface that solves this problem, among others.

3.3.4 Using dynamically typed Buffers (Layer 2.0)

It turns out that, as useful as static datatype dispatch for buffers may be in the Rmpi library, it has serious disadvantages when implementing a high-level layer:

1. Every intercepted call needs a datatype matching (see subsection 3.3.2), even if the tool does not care about the datatype.
2. It cannot be reliably used on a system with differing architectures. (See subsection 3.3.2)
3. Rust's borrow checker has not enough information to allow access to buffers after having been passed on as argument. (See subsection 3.3.3)

Point 1 might have been a minor performance disadvantage, and point 2 could easily be fixed by only allowing the use of fixed-sized integer types. However, point 3 is a

```
 1 impl MpiInterceptionLayer for MyLayer {
 2     fn recv<F, Buf>(
 3         next_f: F, rmpi_ctx: &RmpiContext,
 4         buf: Buf, src: Process, tag: Tag, status_ignore: bool,
 5     ) -> RmpiResult<Option<Status>>
 6     where F: FnOnce(&RmpiContext, Buf, Process, Tag, bool)
 7                  -> RmpiResult<Option<Status>>,
 8           Buf: BufferMut,
 9     {
10         let res = next_f(rmpi_ctx, buf, src, tag, status_ignore);
11         println!("first␣byte:␣{}", buf.as_bytes()[0]);
12         res
13     }
14 }
15
```

Figure 3.6: Implementation example for Layer 1.0 that is not accepted by Rust's borrow checker.
After switching line 10 and 11 the code would compile.

serious problem that, without significantly changing the type conversion process, can only be solved awkwardly.

To fix all three concerns, static typing for MPI-Datatypes needed to be removed entirely in the layer construct. By using dynamic buffer references as input, there is no longer any need for datatype matching when converting types as the MPI-Datatype can just be taken as-is into the dynamic buffer and be possibly inspected later by the tool implementation. Therefore, e.g., `MPI_INT` will not be transformed into, e.g., `MPI_INT32_T` and the MPI call will therefore again work on heterogeneous architectures, although the tool implementation will still not be able to distinguish those types without falling back to the raw implementation.

Most importantly, the dynamic buffer types have their own respective lifetimes attached (see Figure 2.5) which solves the problem in Figure 3.6, presented in Figure 3.7.

```rust
fn recv<F>(
    next_f: F, rmpi_ctx: &RmpiContext, mut buf: TypeDynamicBufferMut,
    src: Process, tag: Tag, status_ignore: bool,
) -> RmpiResult<Option<Status>>
where F: FnOnce(&RmpiContext, TypeDynamicBufferMut, Process, Tag, bool)
          -> RmpiResult<Option<Status>>,
{
    // borrow buf mutably using as_mut()
    let res = next_f(rmpi_ctx, buf.as_mut(), src, tag, status_ignore);
    // the borrow is over, so buf can be used again
    println!("first␣byte:␣{}", buf.as_bytes()[0]);
    res
}
```

Figure 3.7: This implementation compiles using the layer interface 2.0, unlike in Figure 3.6

3.4 Prevention of Infinite Recursions

When implementing an MPI tool, it will most likely become necessary to call MPI functions from inside interceptions. Calling MPI functions in the same way an application would, would naturally lead to that call being intercepted. This phenomenon might not be a problem in many cases but can potentially lead to infinite recursions when functions are intercepted in an endless cycle.

To conveniently prevent such mistakes, Rmpi contains, similarly to MPI-sys (see subsection 2.1.1), a module named `pmpi_mode` which contains everything the root module contains, only every call gets redirected towards `PMPI_*` functions. The mpi-tool-layer package reexports this module as a fake Rmpi alias module that can only call non-interceptable MPI calls. In a tool implementation it is expected to be imported the following way:

```rust
use mpi_tool_layer::rmpi;
```

4 The Tool Creator Macro

A generic tool, as described in chapter 3, cannot be used on its own with either the PMPI or QMPI tool interface. A dynamic library (or possibly static for PMPI) is needed with certain symbols that provide the required functionality in both cases. Of course, the most convenient solution would be to provide both tool functionalities in the same dynamic library. As the current version of QMPI[5] is implemented as a PMPI tool in itself, it is not possible to link to QMPI from a PMPI-Tool without creating conflicting symbols. Therefore, any compiled tool can only be either a QMPI-Tool or a PMPI-Tool but not both at the same time.

The layer interface (see chapter 3) has the purpose of providing a generic way of creating tools, independent of the context in which they would be used. Therefore the needed function symbols to create the target tool still have to be implemented. This would mean in PMPI's case that all of the 360 supported MPI-functions would be defined, which would then call the provided layer by using the corresponding PMPI-function as *next* function (`next_f`). Similarly, for QMPI, a list of QMPI-interceptions has to be defined in which the next function will be determined by QMPI's `QMPI_Table_query` function (see Elis et al. [5]), which will again be used to call the generic layer.

As any layer interface implementation already contains all necessary details for defining an MPI-Tool, the target adaptation can be performed automatically. My framework contains the two tool creator macros `install_pmpi_layer` and `install_qmpi_layer` for that very purpose. Both macros automatically create symbols for all possible interceptions that might have been defined in the specified layer, which happen to be 360 functions. (See Figure 4.1) This is possible as any function not defined in a layer is predefined as an empty interception that only calls the provided next function on being triggered, which, ultimately, has the same effect as having no interception. This automation makes it very easy to implement an empty Rust tool, as shown in Figure 4.2.

To understand the usefulness of such a macro one can look at the "very_simple_tool" example that can be found in the QMPI repository[1], and which represents the implementation of an empty QMPI-Tool in C. The difference in effort is obvious as the very_simple_tool is implemented in a file containing over 3000 lines of code. Of course, much work can be spared in situations where code repeats itself in recurring patterns,

[1] `https://raw.githubusercontent.com/caps-tum/qmpi/a2c9033f20e7614d3c8b8a0134b9654b4bddd340/tool_examples/very_simple_tool/very_simple_tool.c`

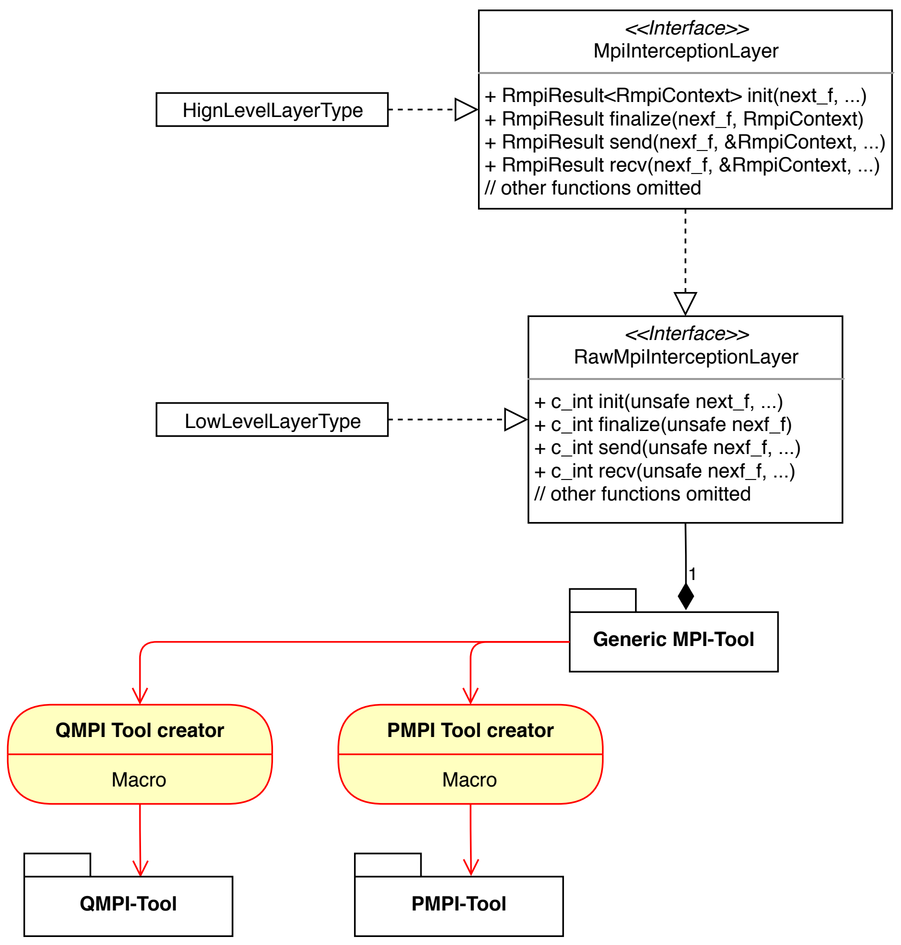

<<Interface>>
MpiInterceptionLayer
+ RmpiResult<RmpiContext> init(next_f, ...)
+ RmpiResult finalize(nexf_f, RmpiContext)
+ RmpiResult send(nexf_f, &RmpiContext, ...)
+ RmpiResult recv(nexf_f, &RmpiContext, ...)
// other functions omitted
HignLevelLayerType
<<Interface>>
RawMpiInterceptionLayer
+ c_int init(unsafe next_f, ...)
+ c_int finalize(unsafe nexf_f)
+ c_int send(unsafe nexf_f, ...)
+ c_int recv(unsafe nexf_f, ...)
// other functions omitted
LowLevelLayerType
1
Generic MPI-Tool
QMPI Tool creator
Macro
PMPI Tool creator
Macro
QMPI-Tool
PMPI-Tool

Figure 4.1: Relationships between the layer interface and the resulting MPI-Tools. Note that Rust is not UML compatible and that this only represents the functionality of the implementation but not its internal structure. (Interface types do not exist in Rust)

(a) Empty generic layer

```
struct MyEmptyLayer;
impl MpiInterceptionLayer for MyEmptyLayer {}
```

(b) creating tool symbols for PMPI

```
install_pmpi_layer!(MyEmptyLayer);
```

(c) creating tool symbols for QMPI

```
install_qmpi_layer!(MyEmptyLayer);
```

Figure 4.2: Implementation of an empty MPI-Tool for either QMPI or PMPI

like in this example, by using common techniques like repeatedly copying code snippets or by creating scripts for assistance in code generation. However, long programming files also increase the probability of mistakes being introduced while it becomes more difficult to find said mistakes in a very long file. For very_simple_tool.c this means that every MPI interception function has to use the correct function-id for calling the next function, has to use the correct way for QMPI for querying the next function call, and has to make sure to pass all function arguments to the next function in the correct order. In contrast, using the tool-creator-macro combined with the layer interface, it can be guaranteed that the correct next function is called in every interception as only that function is provided as argument. Also, providing function arguments in the wrong order is, in most cases, not possible as the next function definition `next_f` is statically typed and, therefore, can only be called using the correct types in the correct positions. Lastly, all details concerning QMPI or PMPI are handled automatically by the respective macro and can, therefore, not be wrongly implemented in any tool using that macro (assuming the macro itself is mistake-free).

In conclusion, using a macro, thereby having to worry about such mistakes only once, can vastly improve code quality for any future tools written in Rust.

5 Performance Evaluation

The goal of parallel applications is to gain performance by using multiple processing units in parallel. Therefore, performance is always a significant concern for such applications as it can quickly happen that, after evaluation, a parallel application turns out not to be faster than its sequential counterpart, after all, resulting in many working hours being wasted. Considering this, it is essential to ensure that the resulting performance overhead created through the layer interface and Rmpi is low enough not to affect typical MPI applications significantly.

For evaluation purposes, this chapter uses the four HPC benchmarks *ASC Sequoia AMG 2006* (AMG), *High Performance Conjugate Gradient* (HPCG), *Co-Design for Molecular Dynamics* (CoMD) and *OSU Micro-Benchmarks* (OMB). Different scenarios will be measured using both PMPI and QMPI. All benchmarks were run on a skylake machine in the CAPS Cloud provided by the TUM chair of computer architecture & parallel programming.[1] The machines each contain one Intel Xeon Silver 4116 processor, containing 12 cores and 24 computation threads. Therefore, all following benchmarks were run on a single skylake node using 24 processes unless otherwise specified.

5.1 ASC Sequoia AMG 2006

AMG is an algebraic multigrid solver capable of solving linear systems on unstructured grids in parallel. All following AMG benchmarks in this thesis will use the options `-laplace -n 85 85 85` which creates a three-dimensional grid over a unit cube for solving a 3D-Laplace equation by dividing the data into chunks of equal sizes for parallelism.[1]

As can be seen in Figure 5.1, a significant amount of time of execution time in AMG is spent on MPI calls, especially the three blocking collective calls `MPI_Allgather()`, `MPI_Allreduce()` and `MPI_Waitall()` which take up ca. 1.216 sec. of execution time for an average runtime of ca. 10.685 sec. (see Table 5.4). The MPI initialization also takes up a significant amount of computing time. This fact has to do with the small runtime that resulted from the chosen configuration and, due to being a one-time operation, can be expected to be less significant for long-running applications.

[1]`Formoreinformation,seehttps://www.in.tum.de/caps/hw/caps-cloud/`

The most often called functions are `MPI_Isend()` and `MPI_Irecv()` with 35276 calls each over all ranks (see Table 5.1). As both functions are non-blocking operations, their nontheless insignificant performance impact is not surprising.

While AMG does support execution using OpenMP, the feature has been disabled for all benchmarks as MPI is already configured to occupy all logical processor cores.

5.2 High Performance Conjugate Gradient

HPCG is a solver for the Poisson differential equation using the Conjugate-Gradient algorithm and a symmetric Gauss-Seidel preconditioner on a 3d regular grid. For discretization operations, a 3x3x3-stencil is used. For communication, it uses a tiny number of MPI functions, consisting only of blocking, wait, and allreduce -and non-blocking receive operations. One `MPI_Bcast` call per rank is also used for initialization.[3]

For all following benchmarks, HPCG will be configured to use a 48x48x48 cube for computations. HPCG has, by default, a timed benchmark portion that is configured to run a fixed amount of time. However, this portion has been deactivated for this thesis as it would affect the accuracy of measuring the time needed to run from start to finish.

HPCG wasts less time using blocking operations than AMG, as can be seen in Figure 5.2, but the functions `MPI_Send()`, `MPI_Wait()` and `MPI_Allreduce()` still delay coputations, on average, for ca. 1.088 sec. for one 11.953 sec. run (see Table 5.4). Again, `MPI_Init()` takes up a significant amount of processing time due to the short runtime duration. The absolute initialization delay, however, remains at roughly the same duration of ca. 0.8 sec.

MPI calls are made with high frequency. The most often called functions are `MPI_Send()`, `MPI_Irecv()` and `MPI_Wait()` with 450048 calls each in total.

HPCG is the only benchmark used that is written in C++ instead of C. It, like AMG, has also support for OpenMP, which is, again, not enabled for benchmarking.

5.3 Co-Design for Molecular Dynamics

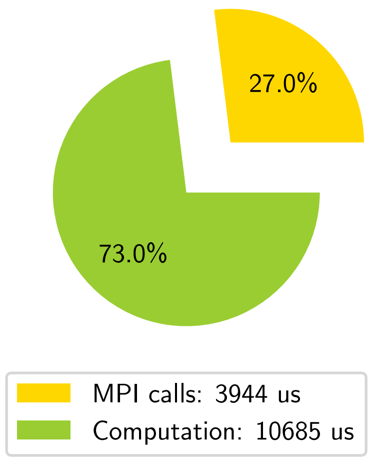
CoMD is a reference implementation, developed to represent the typical workload of a molecular dynamics simulation.[7] In all tests the following options are going to be used:

```
-e -i 4 -j 3 -k 2 -x 80 -y 40 -z 40
```

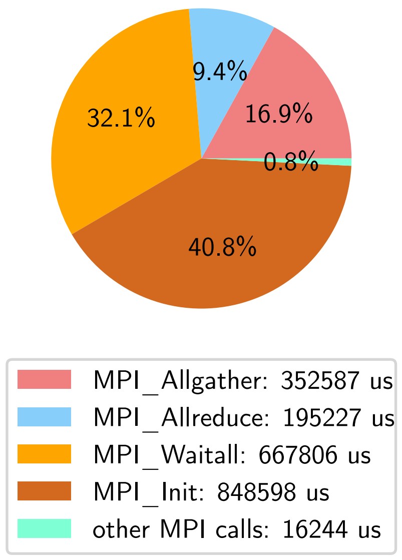
This will create a 3D 80x40x40-grid split into 4, 3, and 2 parts in x, y, and z directions, respectively, resulting in 24 sub-grids. One sub-grid is distributed to every rank. The

Table 5.1: Record of how often MPI functions were called using AMG.

MPI function	call count
`MPI_Isend`	35276
`MPI_Irecv`	35276
`MPI_Allgatherv`	168
`MPI_Allgather`	336
`MPI_Allreduce`	3744
`MPI_Waitall`	17208
`MPI_Comm_size`	21072
`MPI_Comm_rank`	21288
`MPI_Init`	24
`MPI_Finalize`	24



(a) Total distribution on rank 0

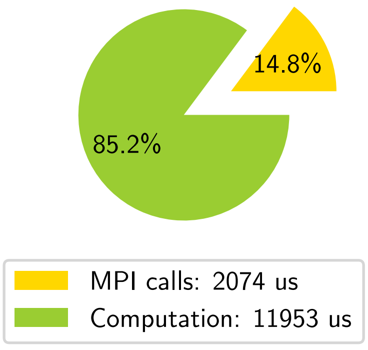


(b) MPI calls split, rank average

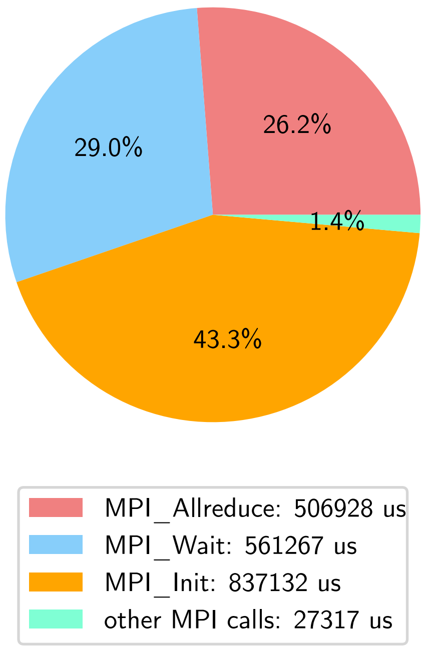
Figure 5.1: Impact of MPI calls on AMG
results are averaged over 100 iterations

Table 5.2: Record of how often MPI functions were called using HPCG.

MPI function	call count
`MPI_Send`	450048
`MPI_Irecv`	450048
`MPI_Wait`	450048
`MPI_Allreduce`	13248
`MPI_Wtime`	86568
`MPI_Bcast`	24
`MPI_Comm_size`	42216
`MPI_Comm_rank`	42240
`MPI_Init`	24
`MPI_Finalize`	24



(a) Total distribution on rank 0



(b) MPI calls split, rank average

Figure 5.2: Time consumption split for HPCG

simulation is set to run for the default value of 100 iterations but is treated as one single benchmark.[2]

Figure 5.3 shows that this benchmark only uses `MPI_Sendrecv()`, `MPI_Allreduce()` and `MPI_Bcast()` for communication, which takes up ca. 0.339 sec. from the average runtime of 11.688 sec (see Table 5.4) which is much less than in either AMG or HPCG. The total delay duration of `MPI_Init()` remains ca. 0.8 sec. Its impact relative to all other MPI operations is, however, higher than for AMG or HPCG.

The most called functions are `MPI_Sendrecv()` and `MPI_Get_count()` with 29088 calls each in total (see Table 5.3).

CoMD also has implementation versions for OpenCL and OpenMP. However, only the pure MPI version is used for all benchmarks.

5.4 OSU Micro-Benchmarks

OMB is a collection of small benchmarks for MPI, UPC, and OpenSHMEM, which all cover a single HPC functionality, in MPI's case often represented by one or two MPI functions.

The only benchmark used in this collection is the MPI latency benchmark, which performs a ping-pong test using a message of a specific size to measure how long the exchange takes. The latency benchmark has been configured to run between two connected skylake nodes, with each node running on one separate process.

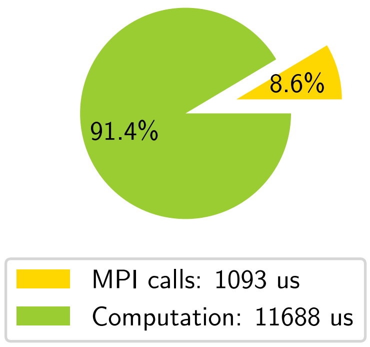
The delay of one ping-pong operation depends, of course, heavily on the message size being used. However, for small messages ($< 2^10$), the difference is bearly measurable. For increasing message sizes, the delay grows with linear proportions. Figure 5.4 shows the different delay times for message sizes that are powers of 2 on a logarithmic scale (which is why the delay time seems to grow exponentially, but it does not).

5.5 Methodology

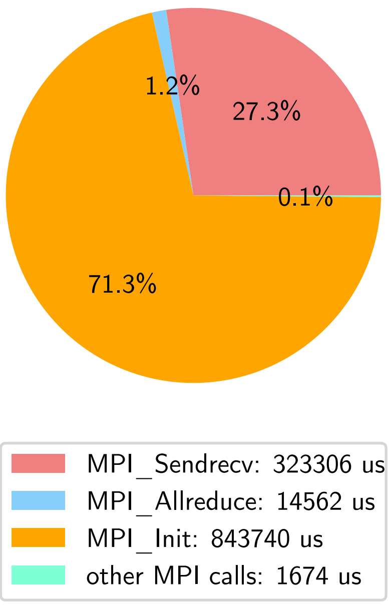
To measure the overhead of using a Rust Tool compared to using either an MPI-Tool implemented in C or having direct MPI access, this paper will mimic the evaluation procedure described in Elis et al. [5]. First, baseline measurements will be created for the most simple scenarios. Those baselines will then serve as orientation for any of the following benchmark that, if valid, should always have a longer or equal runtime than its corresponding baseline.

Table 5.3: Record of how often MPI functions were called using CoMD.

MPI function	call count
`MPI_Sendrecv`	29088
`MPI_Allreduce`	624
`MPI_Bcast`	192
`MPI_Comm_size`	24
`MPI_Comm_rank`	48
`MPI_Get_count`	29088
`MPI_Init`	24
`MPI_Finalize`	24



(a) Total distribution on rank 0



(b) MPI calls split, rank average

Figure 5.3: Time consumption split for CoMD

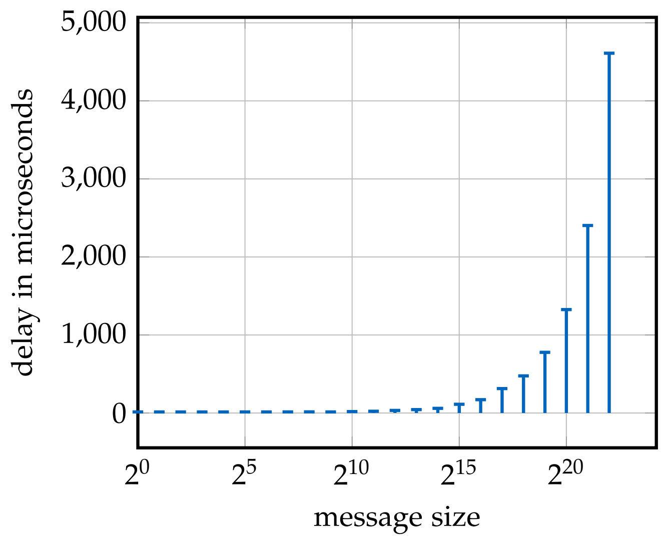


Figure 5.4: Latency benchmark run between 2 skylake nodes

5.5.1 Empty Tool

To evaluate the overhead caused by the layer interface's abstractions, it is necessary to measure the overhead caused by a tool that does nothing else than redirect all MPI calls without doing anything else. Such a tool can be called an empty tool. Empty Rust tools are implemented as demonstrated in Figure 4.2. For comparison purposes, the *very simple tool* (which is an empty tool written in C) in the QMPI repository will be used as an empty tool implemented in C.

The drawback of evaluating a "useless" implementation with no real purpose is that the compiler's program optimizer would most likely remove those parts of the implementation that are "useless" too obviously. Avoiding to call the trait functions of an empty layer implementation would be such an unwanted optimization that could happen if the layer itself is inlined and then optimized. This would possibly lead to creating the same dynamic library, independent of which layer interface is used, and would therefore destroy any differentiation in the resulting measurements. To avoid such unwanted optimizations, all empty-tool implementations and QMPI itself are compiled with zero optimization. However, one must note that the obtained results will only give us an indication of the interface's complexity and its impact on performance but does not prove its general performance in real-life scenarios where optimization is used.

5.6 Single Tool Test

Similar to the *zero-tool test* in Elis et al. [5], this test measures the performance difference of adding one empty Rust tool through the PMPI interface to having direct access to MPI. For comparison purposes, QMPI, with empty tool-stack, is also measured as it is also a PMPI-Tool. As the layer interface is only meant as assistance for developers, it is expected to have a very low performance overhead to counter the benefits it provides.

The measurements in Table 5.4 describe the baseline results for AMG, HPCG and CoMD. Each benchmark is divided into five instances compiled individually to either link to MPI directly, one of three empty Rust Tool implementations using different layer traits internally, or QMPI, which is executed without any inserted tools. In turn, each benchmark has been modified to provide three measurement values via stdout after approximately running for 10 seconds. Those values describe first, the entire runtime of the benchmark, starting from just before `MPI_Init()` until just after `MPI_Finalize()`, while second and third describe the runtime just after `MPI_Init()` until just before `MPI_Finalize()`, the second being the average runtime of all rank while the third represents the maximum rank runtime.

MPI-Applications are notorious for having varying runtimes, which is caused by the need for communication between ranks, which, especially for blocking communications, depends heavily on the state of the underlying scheduler, and that might, e.g., have temporarily suspended a sending task and thereby unwittingly lengthened the corresponding receive operation. This can also be observed in Table 5.4, where the results seem to contradict expectations, e.g., the Rust raw-layer implementation seems to be faster than direct access to MPI for AMG and HPCG. The benchmarks have all been run 100 times, with every presented measurement representing the average runtime. Still, this noise reduction technique is not enough to measure the difference between single empty tool implementations. The only deduction that can be made out of Table 5.4 is that any overhead created through the Rust layer interface is insignificant when using only one single tool.

5.7 Tool-Stack Test

In Elis et al. [5], the *Empty Tool Test* is used to see the resulting overhead of adding one additional tool to QMPI in correlation to the number of tools that are already present. When visualizing the resulting runtime per tool-stack size in a graph, it shows, on average, a gradual linear increase in runtime per increased number of tools. This correlation seems to hold even when using different benchmarks, and only the slope increase changes. Therefore, this test can also be used to accurately compare an empty

Table 5.4: Tool Baseline benchmark results
every value is the average of 100 iterations

	MPI only	PMPI Rust (Raw Layer)	PMPI Rust (High Level Layer)		QMPI (Empty Stack)
			1.0	2.0	
AMG	10.685 s	10.652 s	10.802 s	10.802 s	10.794 s
without init()					
rank avg.	9.830 s	9.783 s	9.933 s	9.950 s	9.928 s
rank max.	9.831 s	9.784 s	9.934 s	9.951 s	9.930 s
HPCG	11.953 s	11.931 s	11.948 s	12.092 s	11.949 s
without init()					
rank avg.	11.112 s	11.087 s	11.103 s	11.235 s	11.106 s
rank max.	11.114 s	11.090 s	11.105 s	11.238 s	11.109 s
CoMD	11.688 s	11.765 s	11.708 s	11.704 s	11.773 s
without init()					
rank avg.	10.840 s	10.917 s	10.857 s	10.847 s	10.924 s
rank max.	10.841 s	10.918 s	10.857 s	10.847 s	10.924 s

Rust Tool (low-level and high-level layer interface) implementation with a counterpart implementation in C. An inefficient layer implementation, e.g., can be identified by its steeper increase in runtime per added tool compared to the graph of a more efficient implementation.

5.7.1 Removal of Lone Outlier

It is natural for timing measurements to have, from time to time, extremely unexpected results. This can happen through, e.g., unexpected system events, poor process scheduling, or similar. Measurements that are too far away from the measured values next to them in the graph are therefore not displayed in any of the following plots.

5.7.2 Empty Tool Test

The tool-stack test results using empty tools can be seen in Figure 5.5, Figure 5.6 and Figure 5.7.

Every benchmark is divided into three measurements. First the overall runtime of the benchmark starting from just before `MPI_Init()` until just after `MPI_Finalize()`. As could be seen in Figure 5.1, Figure 5.2 and Figure 5.1, MPI's initialization can be expensive. However, for long-running applications, this is often not relevant. There-

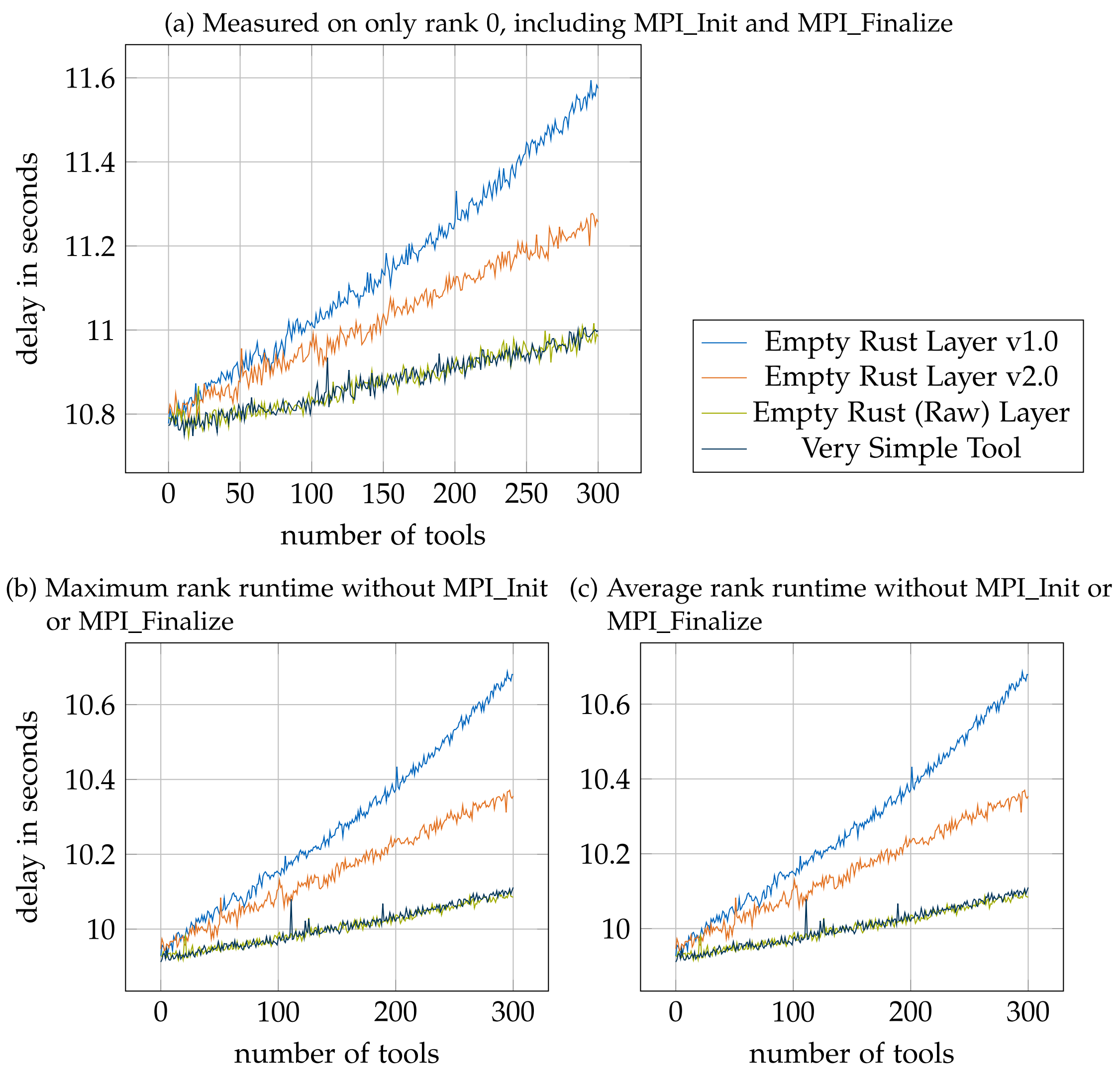


Figure 5.5: AMG Toolstack Benchmark
(3 outliers removed)

(a) Measured on only rank 0, including MPI_Init and MPI_Finalize

(b) Maximum rank runtime without MPI_Init or MPI_Finalize

(c) Average rank runtime without MPI_Init or MPI_Finalize

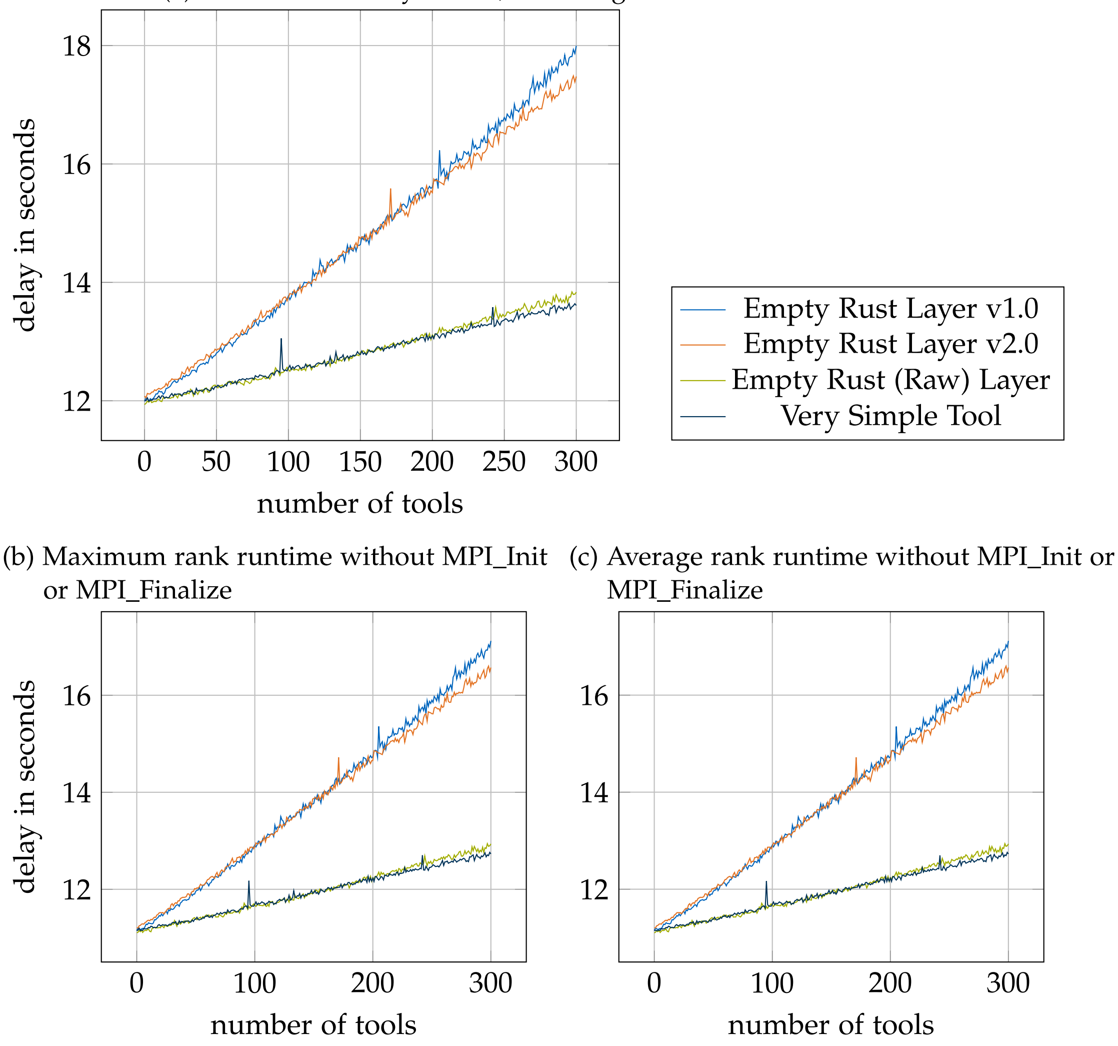


Figure 5.6: HPCG Toolstack Benchmark

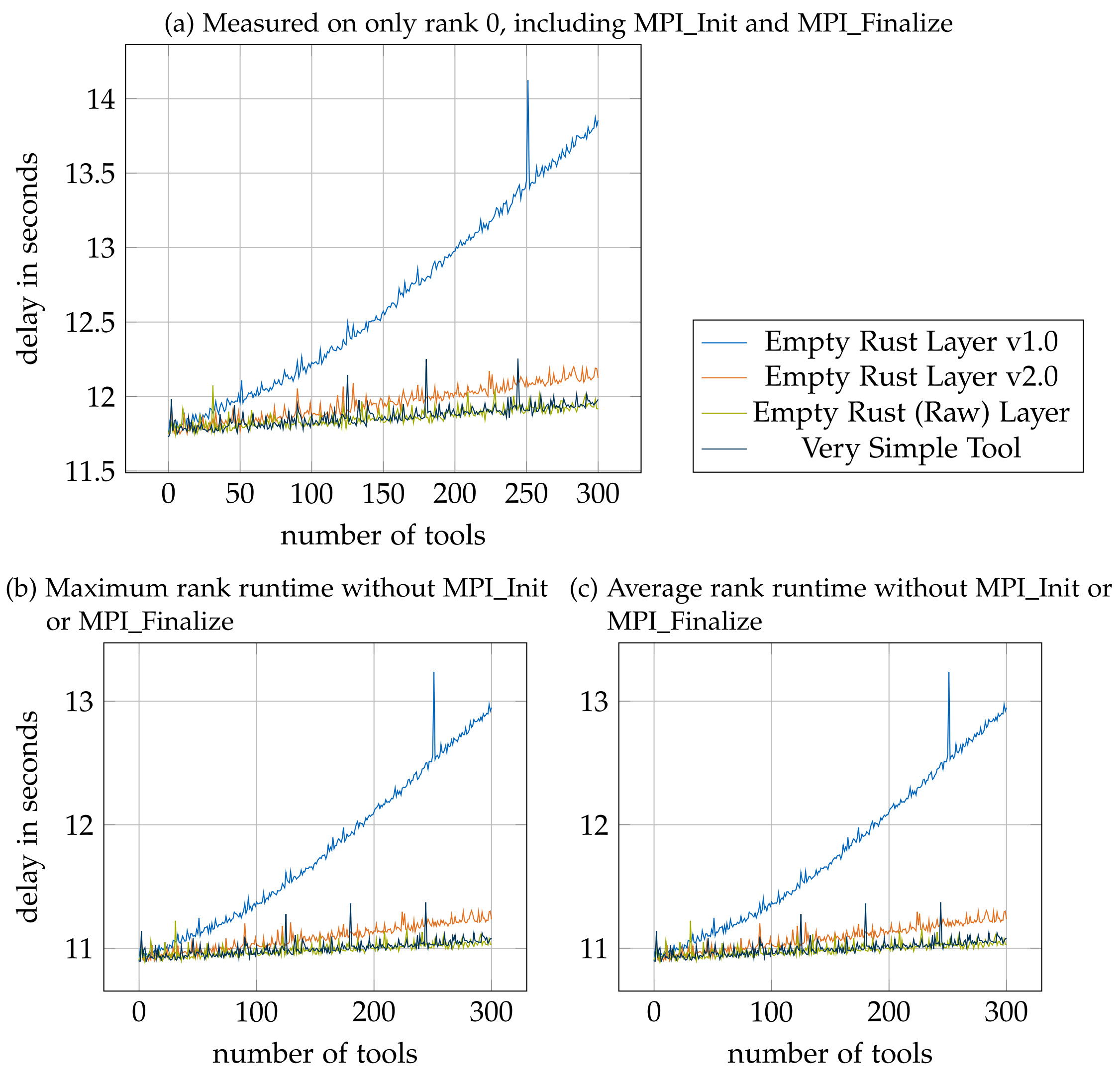


Figure 5.7: CoMD Toolstack Benchmark
(1 outlier removed)

fore, each benchmark has a second and third measurement for the timespan starting immediately after `MPI_Init()` and ending shortly before `MPI_Finalize()`. The second measurement measures the average rank runtime, while the third the maximum rank runtime.

As it turns out, no distinct differences between those three types of measurements can be observed for any of the benchmarks. All ranks seem to be enough in sync that the average and maximum rank runtime turn out to be nearly the same. The general overhead from calling `MPI_Init()` and `MPI_Finalize()` can be observed for every benchmark but is constant enough as not to affect the observed slope increase significantly.

The tool implementation with the least overhead is the Rust raw-layer implementation and the very-simple-tool. The very-simple-tool can be observed to be slightly faster with HPCG but generally has the same performance as the Rust raw-layer.

Both implemented versions of the empty high-level layer perform noticeably worse than their raw counterpart. Version 2.0 still proves to be significantly faster than its predecessor, although, the results vary. The most significant improvement can be observed with CoMD where version 2.0 comes close to a raw-layer implementation's performance. The HPCG benchmarks, on the other hand, measures barely any improvements between the two versions.

The AMG benchmark turns out to be the least sensitive. The difference between the best and worst runtime is only ca. 0.6 sec. for 300 tools. HPCG, on the other hand, is very sensitive to tool overhead and measures a runtime difference of ca. 4 sec. when using 300 tools.

5.7.3 Latency Benchmark

The benchmark results described to this point have all been run on one single processor. This has the benefit of reducing the noise level to a level where it can be distinguished from the actual measurement. HPC applications are, however, meant to run on massively parallel systems that usually consist of a large number of connected nodes. The diagram in Figure 5.4 describes a benchmark performed across two skylake nodes connected via ssh tunnel using local ipv4 addresses. To maintain a low noise level, the OMB latency benchmark, described in section 5.4, is used, which is small and simple enough to produce accurate results still. The exact measurement is starting just before the Send operation of rank 0 until just after the echoed message's receive operation, again on rank 0.

The tool-stack test is performed in the same way as before but for three different message sizes (0 bytes, 2^{10} bytes, and 2^{15} bytes).

Similar to the baseline measurements illustrated in Figure 5.4, no differences appear in the measurements between message size 0 and 2^{10}, independent of layer implementation. Again, as in the baseline, the increased message size of 2^{15} leads to a generally increased ping-pong delay time. However, this does not change the gradual increase in runtime due to increased tool-stack size, again, independent of tool implementation.

The very-simple-tool can, again, be observed to perform slightly better than the Rust raw-layer implementation. Both high-level layer versions perform, again, significantly worse. The 2.0 version is, however, significantly more efficient when over 150 empty tools are used.

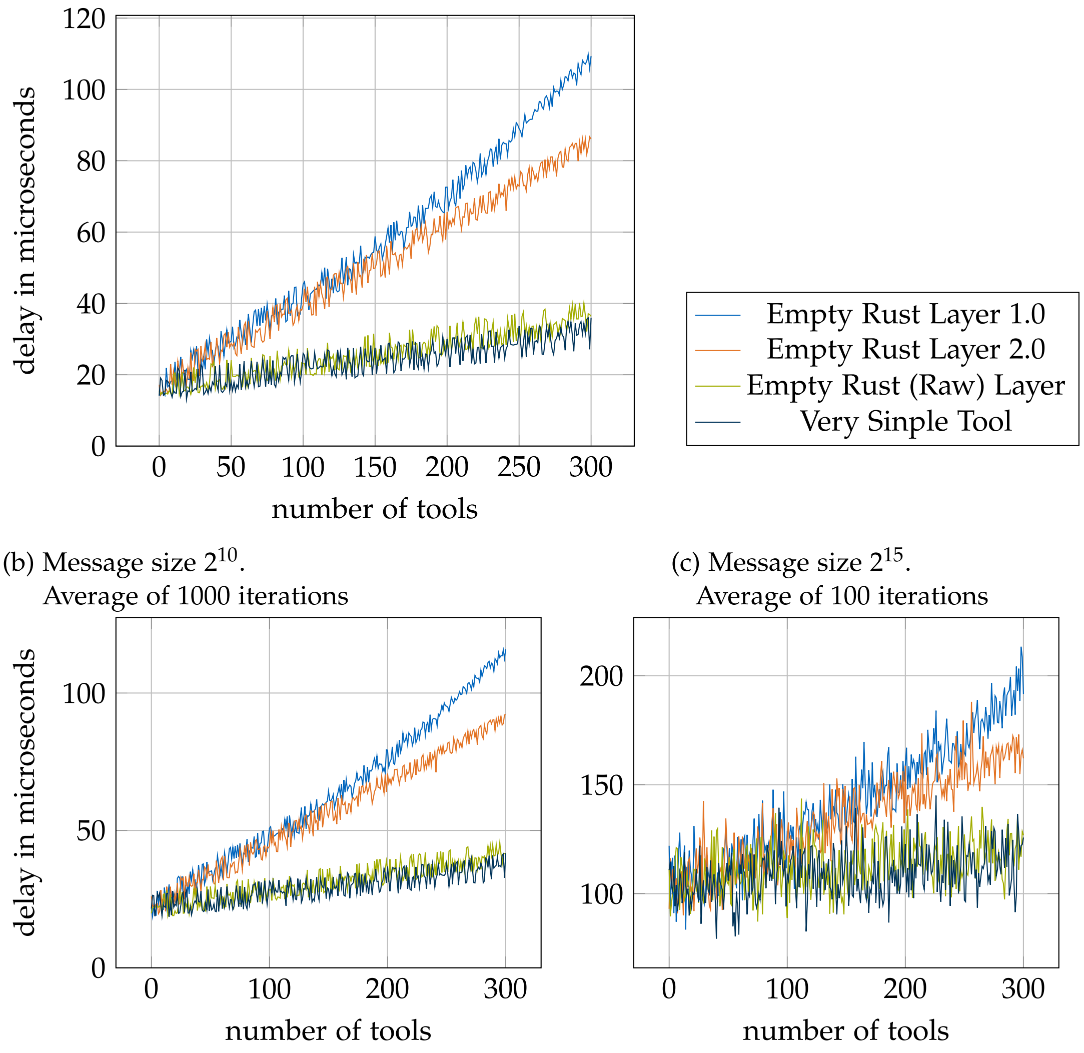
5.7.4 Function Call-Counter

While unoptimized, the empty tools accurately show their internal complexity in benchmarks. They cannot necessarily represent the performance in a real-world scenario where program optimization is activated. The function call-counter tool represents a minimal real-world scenario using the low-level layer interface as a backend for the implementation. The tool works as a set of atomic counters, which count the number of occurred function calls for each function. When `MPI_Finalize()` is called, the tool provides the generated counter data as output (e.g. see Figure 5.9). For comparison purposes, a C implementation with the same functionality is also provided. As the program optimizer cannot entirely eliminate the function counter mechanism, both implementation are being compiled with optimization flag -O3.

Surprisingly, the results in Figure 5.10 and Figure 5.11 indicate that the Rust implementation is more efficient than the C implementation. Most noticeable is the effect with HPCG, which has already in previous benchmarks proven to be very sensitive to small alterations. For AMG and CoMD the difference becomes only clear after adding more than 100 tools to the stack. Although being quite noisy, the latency benchmark indicates the same, except for message size 2^{15}, which generates too much noise to be interpreted.

5.7.5 Bandwidth Counter

To create a similar real-world scenario as in subsection 5.7.4, a new tool needs to be created using the high-level layer (version 2.0). The bandwidth counter also consists of a set of atomic counters. This time, the functionality of the buffer interface is used to count the amount of data sent and received in the program across ranks. To be compatible with CoMD, reduced elements are also counted. All three counter categories are split into distinguishing between the different integer and floating-point datatypes the respective programming language supports. An example output can be seen in Figure 5.12 This time, the Rust and C implementation do not match precisely in



(a) Message size 0.
Average of 1000 iterations

(b) Message size 2^{10}.
Average of 1000 iterations

(c) Message size 2^{15}.
Average of 100 iterations

Figure 5.8: OMB Latency Toolstack Benchmark

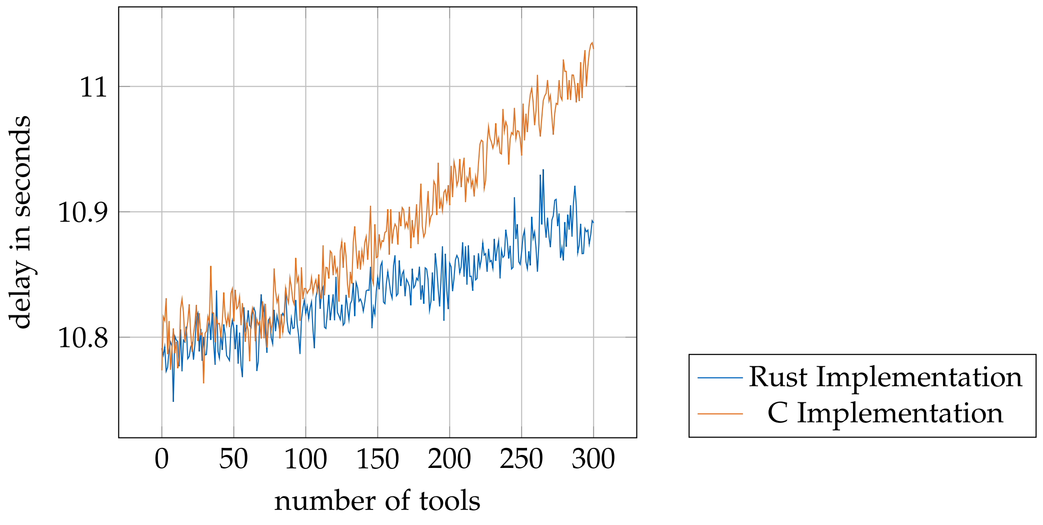
```
MPI_Barrier: 48
MPI_Comm_rank: 2
MPI_Comm_size: 2
MPI_Finalize: 2
MPI_Init: 2
MPI_Recv: 321180
MPI_Send: 321180
MPI_Wtime: 48
```

Figure 5.9: Function Counter Tool Output of one OMB Latency Run Using Default Configuration

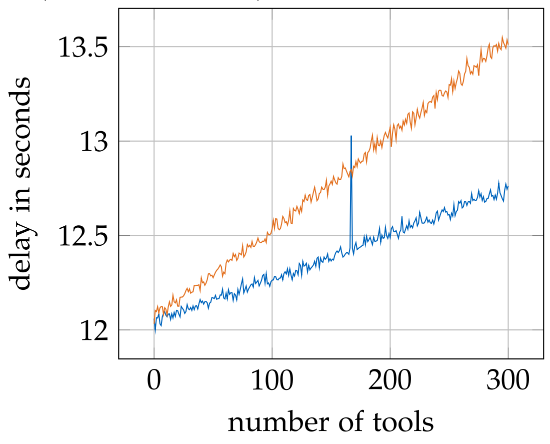
functionality because of their different set of provided integer types. The general way of counting buffer elements remains, however, the same.

The C and Rust bandwidth counter implementations perform similarly. The latency test, presented in Figure 5.8, barely detects any difference in performance. Only for message size 2^{10} can the Rust implementation be observed to be slightly slower. Results are similar for CoMD, as shown in Figure 5.14, which only detects the C implementation being slightly slower after a tool-stack size of over 200.

(a) AMG with optimized function counter tool.
(2 outliers removed)



(b) HPCG with optimized function counter tool.
(1 outlier removed)



(c) CoMD with optimized function counter tool.

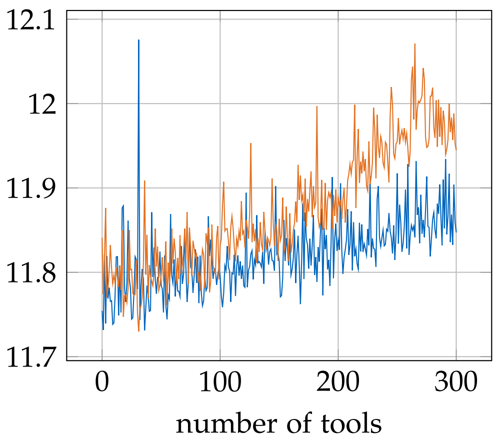


Figure 5.10: Toolstack Benchmarks with optimized function counter tool.

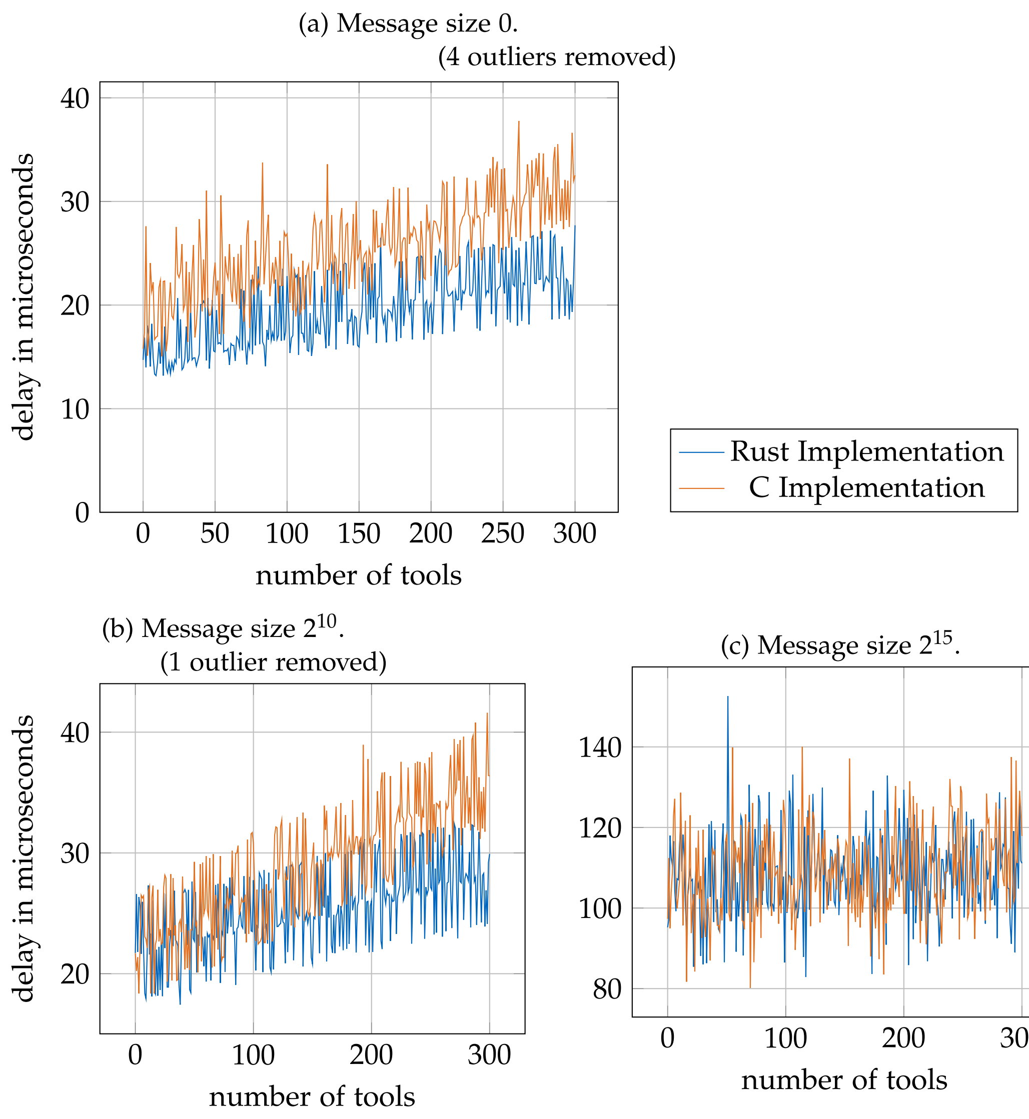


Figure 5.11: OMB Latency Toolstack Benchmark with optimized function counter tool.

```
8-bit Intergers sent : 17242829080
8-bit Intergers received : 17242829080
```

Figure 5.12: Bandwidth Counter Tool Output of one OMB Run Using Default Configuration

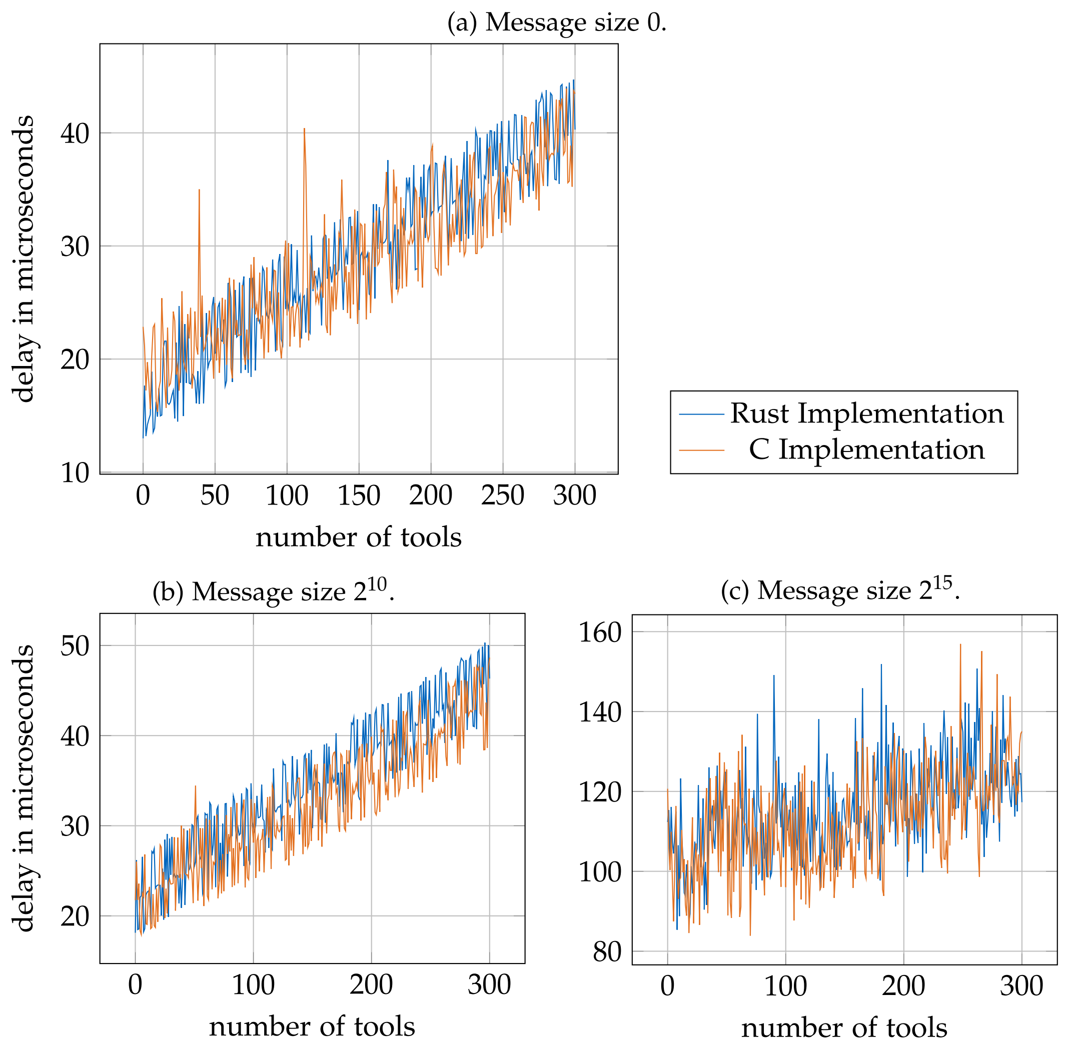


Figure 5.13: OMB Latency Toolstack Benchmark with optimized send/receive/reduce counter tool.
All benchmarks consist of 300 datapoints, each one the average of 1000 measurements.

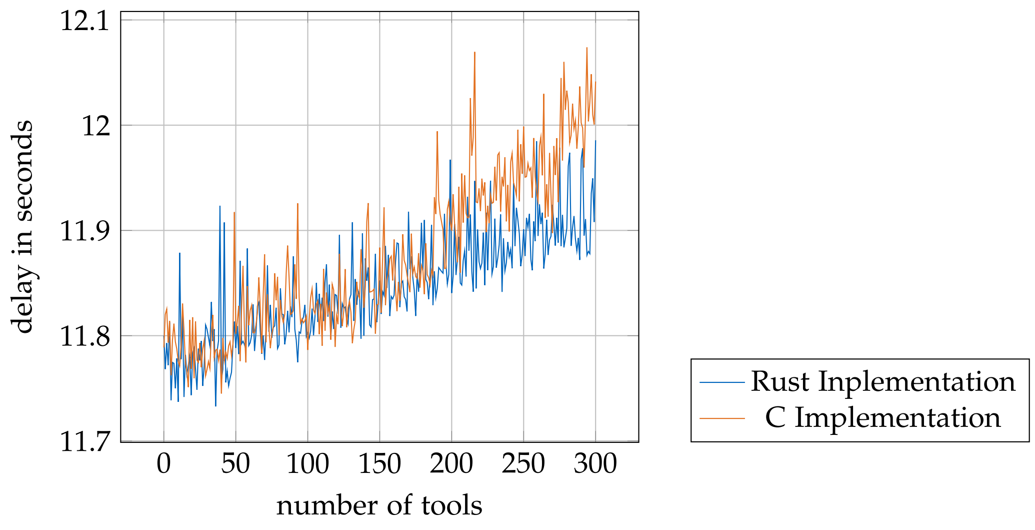


Figure 5.14: CoMD Toolstack Benchmark with optimized send/recv/reduce counter tool
(4 outliers removed)

6 Conclusion

Rmpi and the layer interface successfully provide a memory-safe way of implementing MPI-Applications and MPI-Tools. The existing implementations of the function call-counter and the bandwidth counter prove that useful implementations can be realized, and Rmpi's object-oriented approach ensures that it can be done with minimal learning effort.

chapter 5 shows that significant performance differences can be experienced with only slightly altered implementations providing the tool-stack contains a large number of tools. However, it also shows that the differences between Rust and C implementations remain negligible for a small number of tools.

However, due to MPI's incredible size and complexity, both Rmpi and the high-level interception layer both support only a small subset of MPI functions, and much development and improvement are still possible in this area, and this thesis presented only a small portion of what could be done with MPI and Rust together.

7 Possible Future Improvements for Rmpi and the MPI-Tool-Layer

Rmpi, in combination with the MPI-Tool-Layer, are both powerful libraries that can simplify the process of creating MPI-Tools immensely. However, the implementations are neither complete nor cover MPI's full functionality, nor are they perfect solutions. There are multiple possible improvements, most of them only becoming apparent after actively using the library. The following sections will mention a few problems with the current implementation and suggest possible solutions.

7.1 The layer parameter `next_f` is too restricted

Rust's semantic has the powerful capability of specifying constraints very precisely. This is when safety is considered, is a very positive thing. However, sometimes this can lead to definitions being specified as too restricted. An example is the `next_f` parameter in the raw -and high-level layer interface, which specifies a generic function that can strictly be called only once. This makes sense as it is the expected behavior of an MPI-Tool first to have arbitrary pre interception instructions, then call the next tool's function with possibly modified arguments and lastly perform some arbitrary post interception instructions. It is, of course, still possible to call other MPI functions, but those are always redirected to their respective `PMPI_*` function and are not intercepted by any following tools. It is also possible not to call `next_f` and thereby breaking the toolchain.

An entirely impossible scenario is illustrated in Figure 7.1. The illustrated *Tool 1* reimplements `MPI_Send()` and `MPI_Recv()` by calling their respective asyncronous counterparts combined with `MPI_Wait()`. Doing the same thing using the layer interface would lead to *Tool 2* being skipped in the execution process. This could be prevented by providing, instead of just a single *next function*, an entire function table for every interception.

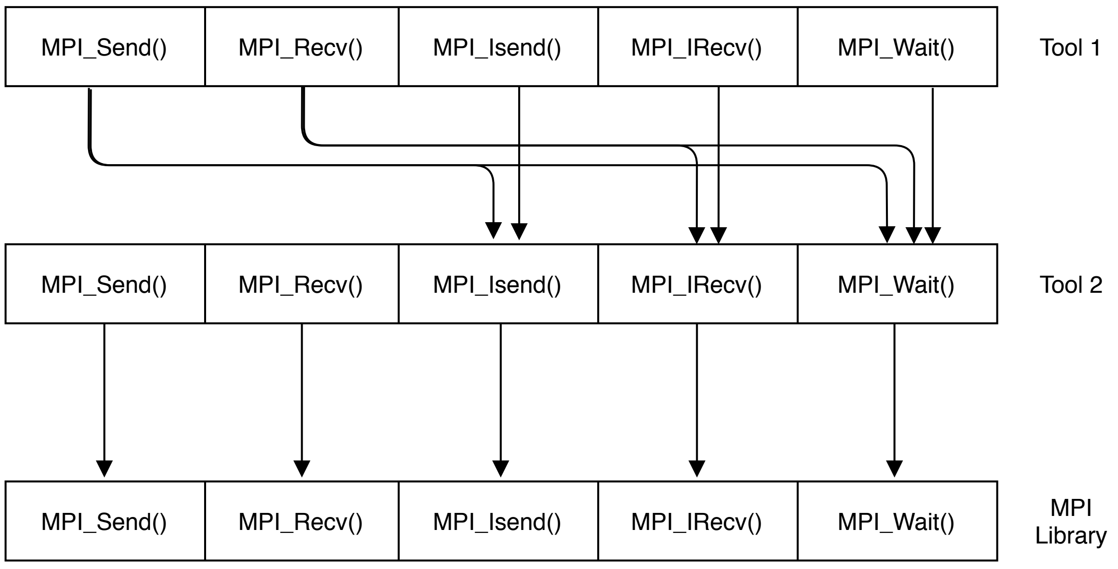


Figure 7.1: Representation of a Toolstack with 2 MPI-Tools. Every arrow represents a function call.
Tool one could be implemented with QMPI but not with the current implementation of the MPI-Tool-Layer.

7.2 Splitting MPI Functions to increase Static Information

For some MPI functions, it is not entirely sensible to convert them into one single Rust equivalent. In Figure 7.2, Rmpi's `MPI_Recv()` wrapper implementation can be seen. Like its C counterpart, it optionally creates a status instance describing the received message, but only if the `status_ignore` flag did not request this operation to be skipped. The Rust compiler will force the developer to check which type of output has been generated. This can be bothersome as it is most likely known at compile-time if the status will be needed or not. (e.g. see Figure 7.3)

```
impl<'c> Process<'c> {
  fn recv<B: BufferMut>(
        &self, buffer: B, tag: Tag, status_ignore: bool,
    ) -> RmpiResult<Option<Status>>
    { /* Convert types back into C representation
         call MPI_Recv()
         convert return value into RmpiResult */ }
}
```

Figure 7.2: Rmpi's function equivalent to `MPI_Recv()`

```
match process.recv(&mut buffer, 0, false) {
    Ok(Some(status)) => { /* message received successfully */ }
    Ok(None) => { /* this never happens */ }
    Err(error) => { /* message failed to be received, handle error.. */ }
}
```

Figure 7.3: In this scenario it is known that a status will be generated by `recv()`. Therefore one of the three match cases is useless, despite being necesary for compilation.

Splitting `recv()` into two separate functions, one that creates status instances and one that does not, would solve that problem in this particular case. This procedure might, however, when used on multiple different functions, generate a significant amount of unnecessary complexity in the layer interface, which might lead to performance disadvantages for MPI-Tools. Such adjustments, therefore, have to be performed with consideration.

7.3 Support for sending arbitrary Rust structures over MPI

Rust provides a powerful macro system. One aspect of it is the possibility of automatic implementation using so-called derive attribute-macros. As discussed in section 2.1.8, static dispatch can be very convenient and efficient. However, Rmpi only supports a fixed number of datatypes for the use of static dispatch in buffers. Figure 7.4 represents a potential future use-case of Rmpi where `MpiDatatype` describes a macro that generates the appropriate implementation necessary in order to be able to send `Color` instances directly through MPI without byte-serialization.

```
#[derive(MpiDatatype)]
struct Color {
  r: u8, g: u8, b: u8
}
```

Figure 7.4: Theoretical way of automatically generating an implementation on how to create a matching MPI-Datatype.

7.4 Layer Chaining

As long as `MpiInterceptionLayer` does not implement all 360 functions supported in `RawMpiInterceptionLayer` and the current QMPI version, tools using this layer will not be able to intercept unsupported functions in any way. A static layer chain would be a simple solution to this problem, enabling the chaining of multiple layer implementations inside a single tool. The generic type definition presented in Figure 7.5 could be used to provide an automatic implementation of an MPI-layer that contains two layers. This way, a high-level layer could be chained with a low-level layer as a fallback for unsupported functions.

```
struct LayerChain<Layer1: RawMpiInterceptionLayer,
                  Layer2: RawMpiInterceptionLayer>;
impl<Layer1, Layer2> RawMpiInterceptionLayer for LayerChain<Layer1, Layer2>
    where /* layer constraints */
{ /* layer chain implementation */ }
```

Figure 7.5: Type Defining a Generically Chained Layer Pair

8 Appendix

8.1 MPI functions supported by Rmpi and MPI-Tool-Layer

Note that `RawMpiInterceptionLayer` supports all 360 MPI-Functions that the current version of QMPI supports as well.[5]

MPI function	Rmpi function (`rmpi::Xxx`)	`MpiInterceptionLayer` function
`MPI_Init()`	`init()`	`init()`
`MPI_Initialized()`	`initialized()`	`initialized()`
`MPI_Finalize()`	`RmpiContext::finalize()`	`finalize()`
`MPI_Finalized()`	`finalized()`	`finalized()`
`MPI_Wtime()`	`RmpiContext::wtime()`	`wtime()`
`MPI_Wtick()`	`RmpiContext::wtick()`	`wtick()`
`MPI_Barrier()`	`Communicator::barrier()`	`barrier()`
`MPI_Group_incl()`	`Group::incl()`	`group_incl()`
`MPI_Group_free()`	`Group::free()`	`group_free()`
`MPI_Comm_size()`	`Communicator::size()`	`comm_size()`
`MPI_Comm_rank()`	`Communicator ::current_rank()`	`comm_rank()`
`MPI_Comm_create()`	`Communicator ::create_subset()`	`comm_create()`
`MPI_Comm_free()`	`Communicator::free()`	`comm_free()`
`MPI_Abort()`	`Communicator::abort()`	`abort()`
`MPI_Send()`	`Process::send()`	`send()`
`MPI_Bsend()`	`Process::bsend()`	`bsend()`
`MPI_Ssend()`	`Process::ssend()`	`ssend()`
`MPI_Rsend()`	`Process::rsend()`	`rsend()`
`MPI_Isend()`	`Process::isend()`	`isend()`
`MPI_Ibsend()`	`Process::ibsend()`	`ibsend()`
`MPI_Issend()`	`Process::issend()`	`issend()`
`MPI_Irsend()`	`Process::irsend()`	`irsend()`

MPI_Recv()	Process::recv()	recv()
MPI_Irecv()	Process::irecv()	irecv()
MPI_Sendrecv()	Process::sendrecv()	sendrecv()
MPI_Bcast()	Process::bcast()	bcast()
MPI_Get_count()	Status::get_count()	get_count()
MPI_Buffer_attach()	buffer_attach()	buffer_attach()
MPI_Buffer_detach()	buffer_detach()	buffer_detach()
MPI_Wait()	request::Request::wait()	wait()
MPI_Waitany()	request::RequestSlice ::waitany()	waitany()
MPI_Waitall()	request::RequestSlice ::waitall()	waitall()
MPI_Test()	request::Request::test()	test()
MPI_Testany()	request::RequestSlice ::testany()	testany()
MPI_Testall()	request::RequestSlice ::testall()	testall()
MPI_Request_free()	request::Request::free()	request_free()
MPI_Cancel()	request::Request::cancel()	cancel()
MPI_Gather()	Process::gather()	gather()
MPI_Gatherv()	Process::gatherv()	gatherv()
MPI_Allgather()	Communicator::allgather()	allgather()
MPI_Allgatherv()	Communicator::allgatherv()	allgatherv()
MPI_Alltoall()	Communicator::alltoall()	alltoall()
MPI_Alltoallv()	Communicator::alltoallv()	alltoallv()
MPI_Reduce()	Process::reduce()	reduce()
MPI_Allreduce()	Communicator::allreduce()	allreduce()
MPI_Scan()	Communicator::scan()	scan()
MPI_Scatter()	Process::scatter()	scatter()
MPI_Scatterv()	Process::scatterv()	scatterv()
MPI_Type_size()	datatype::RawDatatype ::size()	-
MPI_Type_free()	datatype::Datatype::free()	-
MPI_Type_dup()	datatype::Datatype ::duplicate()	-
MPI_Test_cancelled()	Status::test_cancelled()	-
MPI_Comm_dup()	Communicator::duplicate()	-

`MPI_Comm_group()`	`Communicator::group()`	-

8.2 Supported predefined MPI datatypes in Rmpi

Custom MPI-Datatypes can be allocated at runtime, but many primitive C and Fortran types have predefined definitions. Rmpi only supports a subset of the C definitions and none of the Fortran ones.

Note that the pointer sized integer types `usize` and `isize` also have the same datatype definition as the matching fixed-sized datatype on the same platform. However, they never appear in conversions from C buffers in tool environments.

MPI-Datatype	Corresponding Rust Type
`u8`	`MPI_UINT8_T`
`u16`	`MPI_UINT16_T`
`u32`	`MPI_UINT32_T`
`u64`	`MPI_UINT64_T`
`i8`	`MPI_INT8_T`
`i16`	`MPI_INT16_T`
`i32`	`MPI_INT32_T`
`i64`	`MPI_INT64_T`
`c_float`	`MPI_FLOAT`
`c_double`	`MPI_DOUBLE`
`CppBool`	`MPI_C_BOOL`
`Complex<c_float>`	`MPI_C_FLOAT_COMPLEX`
`Complex<c_double>`	`MPI_C_DOUBLE_COMPLEX`
`LongInt`	`MPI_LONG_INT`
`DoubleInt`	`MPI_DOUBLE_INT`
`ShortInt`	`MPI_SHORT_INT`
`TwoInt`	`MPI_2INT`
`LongDoubleInt`	`MPI_LONG_DOUBLE_INT`

8.3 Rmpi's Version of Supported MPI types

All important type definitions in Rmpi have one or more counterparts in MPI's C definition. All pairs can be converted in both directions. Conversions are usually performed with associated functions `from_raw()` and `into_raw()`.

Rmpi Type	Corresponding MPI Type(s)
`RmpiResult`	`int` (function result)
`Error`	`int` (non `MPI_SUCCESS` function result)
`&[<primitive type>]`	`const void*`, `int`, <datatype statically known>
`&mut [<primitive type>]`	`void*`, `int`, <datatype statically known>
`TypeDynamicBufferRef`	`const void*`, `int`, `MPI_Datatype`
`TypeDynamicBufferMut`	`void*`, `int`, `MPI_Datatype`
`Communicator`	`MPI_Comm`
`Process`	`MPI_Comm`, `int` (rank)
`RmpiContext`	— (`RmpiContext` only defines a semantic constraint)
`datatype::Datatype`	`MPI_Datatype`
`Group`	`MPI_Group`
`MpiOp`	`MPI_Op`
`request::Request`	`MPI_Request`
`&RequestSlice`	`const MPI_Request*`, `int`
`&mut RequestSlice`	`MPI_Request*`, `int`
`Status`	`MPI_Status`
`Tag`	`int` (send/recv tag)

List of Figures

List of Tables

Bibliography

[1] *AMG Summary v1.0.* `https://asc.llnl.gov/sites/asc/files/2020-09/AMG_Summary_v1_7.pdf`. [Online; accessed 1-October-2020]. Sept. 2020.

[2] *CoMD: A Classical Molecular Dynamics Mini-app*. `http://ecp-copa.github.io/CoMD/doxygen-mpi/index.html`. [Online; accessed 14-October-2020].

[3] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. *HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems*. Tech. rep. UT-EECS-15-736. Knoxville, Tennessee: Electrical Engineering and Computer Sciente Department, Nov. 2015.

[4] Bengisu Elis. "Design, Implementation and Testing of a new Profiling Interface for MPI." MA thesis. Technische Universität München, 2018.

[5] Bengisu Elis et al. "QMPI: A next generation MPI profiling interface for modern HPC platforms." In: *Parallel Computing* 96 (Aug. 2020). DOI: `10.1038/nature13570`.

[6] Igor Markov. "Limits on fundamental limits to computation." In: *Nature* 512 (Aug. 2014), pp. 147–54. DOI: `10.1038/nature13570`.

[7] Jamaludin Mohd-Yusof, Sriram Swaminarayan, and Timothy C. Germann. *Co-Design for Molecular Dynamics: An Exascale Proxy Application*. `https://www.lanl.gov/orgs/adtsc/publications/science_highlights_2013/docs/Pg88_89.pdf`. [Online; accessed 4-October-2020].

[8] *MPI: A Message-Passing Interface Standard, Version 3.1*. `http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`. [Online; accessed 21-September-2020]. June 2015.

[9] M. Schulz and B. R. De Supinski. "A Flexible and Dynamic Infrastructure for MPI Tool Interoperability." In: *2006 International Conference on Parallel Processing (ICPP'06)*. 2006, pp. 193–202.