Universiteit Leiden

Master Computer Science

Safe, Fast and Elegant Communication in Rust

Name:	Xudong SHI
Student ID:	s3444538
Date:	[11/07/2024]
Specialisation:	Advanced Computing and Systems
1st supervisor:	Prof.dr. R.V. van Nieuwpoort
2nd supervisor:	Dr. K.F.D. Rietveld
2nd reader:	MSc. A.B. Liokouras

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

The communication system is pivotal in parallel and distributed computing. In the realm of high-performance computing (HPC), the Message Passing Interface (MPI) has emerged as a dominant framework for inter-process communication, within which, Remote Memory Access (RMA) is a module that allows efficient utilization of Remote Direct Memory Access (RDMA) networks. Traditionally, MPI programs have been primarily developed using C/C++ and FORTRAN. However, these languages bring inherent drawbacks that can hinder development efficiency and safety. Recently, with the advancement of Rust, a trend has emerged to replace the traditional roles of C/C++ with this more modern language. This project introduces RMA to the Rust ecosystem by implementing a binding that bridges the gap between its primitive form in C and Rust. Our research question is, can RMA be well integrated into Rust, making it as fast as C while enjoying the safe and modern language features Rust offers? We have implemented a prototype Rust binding for RMA, and conducted comprehensive performance evaluations to determine whether Rust, in conjunction with RMA, could maintain the performance benchmarks set by its predecessors. Our findings indicate that the Rust binding for RMA holds promise in terms of performance. Additionally, by leveraging Rust, programmers can benefit from its modern features, thereby enhancing the development experience and improving software quality significantly.

Contents

1 Introduction

Rust has been a rising star programming language prominently in system programming. It is deemed an alternative to C and C++, as its safety properties are better than those of the two older languages. However, in the domain of distributed parallel computing, MPI as one of the dominant programming models, mostly written in C/C++ or FORTRAN, has not been well integrated with Rust. With Rust, MPI developers would be able to enjoy the safety features along with the modern language ecosystem. Furthermore, there have been studies that implemented prototype components for MPI with Rust, as well as the language bindings. In addition, as one of the latest features of MPI-3 [39] standard, RMA has opened a window for developers to exploit RDMA [21] that may bring significant performance increment for the MPI applications. However, RMA remains untouched by Rust.

With eyes on a larger scope, today's computing infrastructure is inherently distributed, making communication between machines essential. Various paradigms exist for inter-process communication (IPC). Among these, message passing stands out as a widely adopted method. In message passing, two processes operate within isolated memory spaces, necessitating an external mechanism for transmitting messages. At the physical level, the Ethernet is the most widely used networking technology. On top of the physical networks, there are communication protocols that allow processes can understand each other, with TCP/IP, and UDP as the most widely used ones. Built on top of the protocols, a plethora of libraries offer communication functionalities for higher-level applications. However, these can be inefficient in performance-critical scenarios, as the transmission of messages involves OS kernel participation, which may incur interrupts that slow down communication. Parallel to TCP/IP, RDMA networks provide a more efficient communication alternative by allowing kernel and TCP/IP stack bypass, by which the message can be transmitted directly to a remote process and skip the involvement of the kernel. Thus reducing the overhead associated with the TCP/IP stack. This technology is facilitated by specialized networks such as InfiniBand [27] and RDMA over Converged Ethernet (RoCE) [54].

In the domain of HPC, most workloads are executed within computing clusters, where numerous interconnected computers function in concert despite being physically separate. Common workload patterns involve large matrices or graphs that exceed the processing capabilities of a single machine. Consequently, these problems are divided into smaller subparts, each allocated to a different machine within the cluster. This partitioning significantly enhances computational power. However, inter-computer communication is essential to coordinate these distributed tasks. Several tools have been developed to facilitate this communication, including Remote Procedure Call (RPC) frameworks like gRPC [24]. Nonetheless, given the stringent performance requirements and specific programming paradigms of HPC applications, MPI [40] remains the most widely utilized communication tool in this field.

As for programming with MPI, C/C++ and FORTRAN are the predominant languages. Due to their intrinsic language features and historical complexities, developers often face a suboptimal programming experience. Issues stemming from low-level abstractions, susceptibility to unsafe practices i.e. operating raw pointers, and cumbersome tooling frequently plague programmers. Although higher-level language bindings like mpi4py for Python [10] and Open MPI Java bindings [13] exist, their language-specific characteristics often limit their utility in

performance-critical environments [36]. There is a marked need for a system programming language that not only offers high-performance capabilities but also provides robust functionality for low-level interactivity. Rust [57], emerging as a leading system programming language, is considered a potential alternative to C/C++. It enforces safety rules such as immutability and non-nullablity at compile time checking, combined with modern language syntax, and user-friendly tooling while maintaining performance comparable to C/C++ [29]. These attributes have significantly contributed to its growing popularity.

The RMA module in MPI, also known as "one-sided communication", provides an interface similar to RDMA, enabling one process to directly transfer data to the memory region of another process. This functionality reduces the overhead associated with the traditional send receive pattern and enhances the utilization of RDMA networks [26]. However, the integration of RMA with Rust remains insufficient. Currently, to program with RMA in Rust, programmers must write unsafe C-like code, which contradicts the advantages of using Rust. In this thesis, based on the present MPI binding for Rust `rsmpi` [55], we design interfaces for RMA to further extend the library, as well as throughout evaluations to show if the modern and safety language features can be preserved along with efficient communication with RMA. With this work, rsmpi users can easily write safe and fast parallel applications with RMA. The library and evaluation programs have been open-sourced: [17, 18, 15, 16].

Therefore, our **research question** is, can RMA be well integrated into Rust, and can the speed as written in C be preserved while benefiting from the safe and modern language features offered by Rust? Concerning this question, we further propose **sub questions**:

1. At the current stage, what is the difference between programming with RMA in C and Rust?

2. How to design a Rust RMA interface, so that with this interface, users can write safe and Rust idiomatic code?

3. With the same program written in C and Rust, would there be a performance difference?

As rsmpi supports a range of essential operations but lacks RMA support, we will implement the missing RMA part for rsmpi. Our work also aims to contribute to the open-source project.

This thesis unfolds as follows: Section 2 provides the detailed background necessary for understanding this thesis. Section 3 shows the full landscape of work around the communication system and Rust. Section 4 demonstrates our methods for solving the integration problem of RMA with Rust. The performance evaluation is presented in Section 5. We also present our comments and limits regarding the work in Section 6. The thesis concludes in Section 7, with future work.

2 Background

This project is built around MPI, Rust, and RMA. The following sections will introduce these components to provide the necessary background information for understanding this thesis.

2.1 MPI

Introduced in 1994, MPI has become the standard tool for writing parallel programs in distributed memory environments, where each process operates in its own memory space and communication is achieved through message exchanges. This concept can be visualized using the example of two computers that coordinate to perform matrix calculations. Imagine a matrix is split into submatrices, with each machine handling a different submatrix. Each machine is unaware of the other's data. If one machine needs to process some rows from the other's submatrix, the necessary rows are transmitted using MPI. To transmit, there are different communication mechanisms, one is the the traditional send-receive pattern, where both the sender and receiver participate in the communication. There is also one-sided communication, as the name implies, only one side is required to participate. Furthermore, MPI is a standard, meaning that only the interface is specified. Various vendors offer their implementations of this standard. Among these, Open MPI [51] is the most widely used. Another popular implementation is MPICH [46]. Their implementations are mostly written in C.

2.1.1 Send Recv

The send and receive pair [44] is the basic and widely used mechanism for point-to-point communication in MPI. The methods used are `MPI_Send` and `MPI_Recv`. MPI follows Single Program Multiple Data (SPMD) model [11], which implies that multiple processes execute the same program, but on different data parts. A typical program that transfers data with the standard send recv is shown in listing 1:

```c
#include <mpi.h>  // Include MPI header file
#include <stdio.h>

#define ARRAY_SIZE 10

int main(int argc, char** argv) {
    // Declaration of variables to hold MPI environmental info
    // Size is how many processes to run this program
    // Rank is the ID of the process
    int rank, size;
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);
    // Obtain the rank of the current process
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // Obtain the size
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Declaration of a double array, this is local to the process.
    // Therefore, other processes have to know the contents via message passing.
    double data[ARRAY_SIZE];
    // Since MPI follows the Single Program Multiple Data (SPMD) model
    // To instruct the behaviour of a specific process,
    // use conditional branch and rank
    if (rank == 0) {
        // Rank 0 initializes the array with any value
        for (int i = 0; i < ARRAY_SIZE; i++) {
            data[i] = i * 1.0;
        }
        // Rank 0 sends the array to rank 1 by invoking MPI_Send
        // number of elements to send, element type should be specified
        MPI_Send(data, ARRAY_SIZE, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        // Rank 1 receives the array from rank 0,
        // and writes the content to its local array.
        // number of elements to send, element type should also be specified
        MPI_Recv(data, ARRAY_SIZE, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        // Should display the value as Rank 0 initialized
        for (int i = 0; i < ARRAY_SIZE; i++) {
            printf("data[%d] = %f\n", i, array[i]);
        }
    }
    // Before the program exits, every process should call this to terminate the MPI e
    MPI_Finalize();
    return 0;
}
```

Listing 1: Sample send recv in C

The program can be compiled with `mpicc` [49], e.g. `mpicc the_program.c -o the_program`. To run the program, use `mpirun` [50], by which the number of processors can be configured with `-n` option, e.g. `mpirun -2 ./send_recv_program`. In the environment of a Linux computing cluster, the program is often scheduled to be executed by Slurm [70], which is a popular workload manager.

For the method `MPI_Send`, it specifies the destination process by the `rank` argument. The number of items to send and their data type are specified by the `count` and `datatype` argument. For the above example listing 1, the sender sends the content of the array of length `ARRAY_SIZE` to the destination process rank 1, and the element type is double. Thus passing the pointer to the array, the number of elements to send, and the destination rank. The type should be specified as well, which is `MPI_DOUBLE`, as the equivalent data type to double in C. Additionally, the tag argument is an extra identifier to the message envelope that seals the message to be sent, which can be used by the receiver to identify the expected message. The above example used 0 as its tag. Last but not least, the last argument of the method is the communicator, which specifies the communication context of the send operation, which can be understood as the group of processes the sender belongs to plus MPI environmental information.

Symmetrically, the method `MPI_Recv` is being called at the other rank. It in this case also specifies a buffer to receive the incoming messages, which is also the `double array[ARRAY_SIZE]`. However, the array is local to rank 1. Next, it also specifies the number of items to receive, which can be different from the number of items to send to the sender side. Like the sender, the receiver also needs to specify the data type, the rank of the sender, the communicator, and the tag. Lastly, the receiver can specify a status argument for more details of the operation, which is omitted in the example.

Since `MPI_Send` and `MPI_Recv` is a pair of blocking calls, the invocation of send will not complete until the send buffer can be safely modified [40]. To safely modify the send buffer after sending, it must be ensured that the data to be sent previously has been placed elsewhere, either in a temporary system buffer or at the receiver buffer. As this process involves additional memory allocation and copying, it would result in inefficiency for large-size messages.

2.1.2 Remote Memory Access

RMA, also known as One-sided communication, is a set of interfaces offered in MPI. It is termed 'one-sided' contrary to the traditional send receive pattern only one party—the data source or the destination participates in data transfer. Whether it acts as the 'sender' or 'receiver' depends on the data transfer method called.

The following section introduces the core components of RMA [45]:

Window The window is the construct in the centre of RMA, which facilitates direct access to a remote memory region. When discussing the window, there are indeed two concepts around it: one is the memory region that is being opened for remote process access. Thus vividly called "window". The other is the window object [22], which acts as a handle to use the capabilities

of RMA, such as calling RMA methods. The window object exposes a region of memory of a process to a process group, which can be imagined as a set of processes. Moreover, a programmer can allocate memory but choose to not bind to a window object, thus preventing a remote process from accessing this memory region. There are multiple ways to initialize a window: allocate a memory region separately, and create a window object upon it using `MPI_Win_create`. Or let MPI allocate the memory region and window object by using `MPI_Win_allocate`. Compared to `MPI_Win_create` the window is open to remote access since its creation, memory region can also be dynamically attached or detached to the window object, hence turning on/off remote access during runtime, and this is done by `MPI_Win_create_dynamic`. Within the scope of this work, only `MPI_Win_create` and `MPI_Win_allocate` are discussed.

Communication Calls There are four types of communication calls in RMA: `Put`, `Get`, `Accumulate` and `Request-based`. Intuitively, it could be imagined that `Put` writes data to the target process's window, `Get` obtains data from the window of another process and writes it into its memory. `Accumulate` is a set of methods that perform extra operations on top of the previous two communication calls, such as sum, and subtraction. It provides a handier way to perform computation. `Request-based` is a set of methods that allows attaching user-defined methods with the previous three types of communications calls. Within the scope of this project, we only focus on `Put` and `Get`.

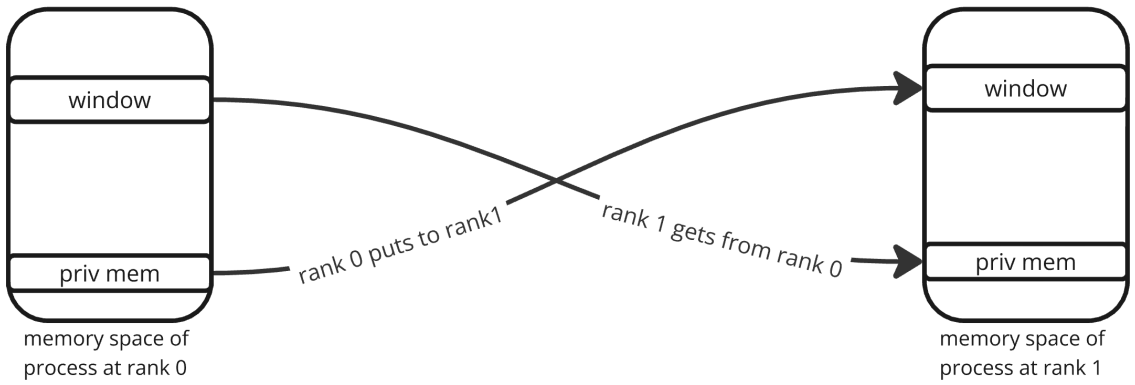


Figure 1: Illustration for Put and Get method of RMA

Synchronization Calls Synchronization is another core concept in RMA. Unlike `MPI_Send` `MPI_Receive`, the start and completion of communication calls must be explicitly synchronized. There are two synchronization modes in RMA. Given that communication involves a source process and a target process, one mode is *Active Target Synchronization*, where the target process actively participates in synchronization. The other mode is *Passive Target Synchronization*, where the target process does not engage in synchronization. If a process does not call the synchronization method, it does not engage. To choose a mode, one should consider the specific requirements of the program.

The simplest and most straightforward way to synchronize is using `MPI_Win_fence`. It is a collective call, acting similarly to `MPI_Barrier`, by which until all processes reach the same point,

the program will not proceed. The downside of fence is overhead can be incurred if only a subset of processes of a large process group needs to synchronize. In addition, `MPI_Win_fence` is the active target synchronization mode, as all processes need to call it. In contrast, `MPI_Win_lock` is passive target synchronization. It is akin to the common locking mechanism in single-process concurrent programming. The window can be imaged as the shared resource, thus processes accessing the region need to synchronize with locks. Unlike `MPI_Win_fence`, `MPI_Win_lock` is not a collective call. Within the scope of the program, we used `MPI_Win_fence`, `MPI_Win_lock` and `MPI_Win_unlock`, which only target a single process. There are also collective locks and other active target synchronization methods, but we leave this to future work.

To demonstrate the basic usage, listing 7 does the same as the previous send recv code but written with RMA.

The program starts similarly as send recv by initializing MPI environments. At line 10, `MPI_Win_allocate` is invoked to allocate memory and open it as the window. This can be deemed as if calling malloc [31] passing the size as the first argument. The second argument, displacement, is equivalent to the size of each element of the array. Since the window is allocated as an array of double, the size and displacement are specified using `sizeof(double)`. The method returns two results: the pointer to the window, and the pointer to the window object. After allocation, the content of the window can be initialized. Before invoking `MPI_Put`, which is the communication call to write contents to the window at the target process, synchronization should be performed. Here we used `MPI_Win_fence`. Moreover, `MPI_Win_fence` is used in the form of a pair. Rank 0 as the source process, calls `MPI_Put` to write data to the destination which is rank 1, and then followed by another `MPI_Win_fence`. Finally, similar to calling `free` when at the case where `malloc` is used, `MPI_Win_free` should be called to free up the resources associated with the window. And as always, call `MPI_Finalize` to terminate the MPI environment.

As synchronization is explicitly done in RMA, the communication and synchronization are decoupled. This opens a door for more flexible communication patterns: communication calls are unblocking so that another communication call can take place right after a previous call is invoked. Although send recv supports asynchronous communication, using RMA only requires invoking the function on one side, and multiple communications can be cluttered within a pair of synchronization calls. Moreover, there are synchronization calls for different levels of granularity, such as `MPI_Win_fence` is a collective call, `MPI_Win_lock` is between two processes. The programmer can pick one based actual case. In contrast, in send recv, the receiver must specify the matching sender, thus easing the programming efforts, and the synchronization still happens between the two processes. However, the parameters of allocation and put are lengthy and obscure, which can be intimidating to programmers. For common use cases such as transmitting an entire array, always having all the arguments filled can be cumbersome and error-prone. Therefore, it can be useful to implement bindings that wrap these methods.

```c
    // Common operations omitted,
    // i.e. header file inclusion, main function, MPI initialization

    // Declaration of the pointer to memory piece and window object
    double *data;
    MPI_Win win;
    // Allocate memory and create an MPI window on every rank
    // ARRAY_SIZE * sizeof(double) as the window size
    // sizeof(double) as the displacement, i.e. the size of each element
    MPI_Win_allocate(ARRAY_SIZE * sizeof(double), sizeof(double),
        MPI_INFO_NULL, MPI_COMM_WORLD, &data, &win);

    if (rank == 0) {
        // Initialize the array on rank 0
        for (int i = 0; i < ARRAY_SIZE; i++) {
            array[i] = i * 1.0;
        }
    }

    // Synchronize, argument 0 is for optimization behavior, can be ignored.
    MPI_Win_fence(0, win);

    // Rank 0 puts data to rank 1
    if (rank == 0) {
        MPI_Put(array, ARRAY_SIZE, MPI_DOUBLE, 1, 0, ARRAY_SIZE,
            MPI_DOUBLE, win);
    }

    // Alternatively, rank 1 gets the data from rank 0
    // if (rank == 1) {
    //    MPI_Get(array, ARRAY_SIZE, MPI_DOUBLE, 0, 0, ARRAY_SIZE,
    //      MPI_DOUBLE, win);
    // }

    // Complete the RMA access epoch
    MPI_Win_fence(0, win);

    // This should be called to free up the resources
    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
```

Listing 2: Sample RMA program in C

2.2 Rust

Initiated in 2006 [7], Rust has emerged as a promising programming language poised as an alternative to C/C++ in the realm of system programming. It achieves memory safety and type safety without compromising performance — a common drawback of garbage-collected languages like Java [35]. In contrast, Rust is free of garbage collection. Moreover, while C/C++ offers optimal performance, it is notoriously prone to unsafe memory access issues. Another notable feature of Rust is its rich type system, which combines expressiveness with strict typing rules to prevent silent errors. The following subsections will introduce the features of Rust that are utilized in this thesis, laying the foundational basis for our design and implementation.

2.2.1 Cargo

Cargo [5] serves as the all-in-one package manager for Rust and is highly regarded for its modern features and convenience. Similar to Maven [1] for Java and npm [47] for JavaScript, Cargo supports a wide range of functionalities including project creation, building, running, releasing, publishing, and dependency management. Additionally, it uses TOML [63] for its configuration files, which are designed to be both human-readable and machine-readable. In contrast, the C/C++ ecosystem, due to its historical development, offers a variety of choices such as Makefile, CMake, and Bazel for building, and package managers like Conan [9]. These tools vary significantly in functionality and configuration syntax, often requiring programmers to select based on specific project needs. As a result, the adoption of package manager is not popular in C++ [41]. As a language not impeded by historical complexities, Rust was developed with modern tooling from its inception, making it an attractive option for adoption.

2.2.2 Static and Strong Type System

Rust is both strongly and statically typed, by which the compiler can prevent many errors instead of being caught during runtime. Influenced by concepts [28] such as algebraic data types from SML and OCaml, as well as Typeclasses and type families from Haskell, Rust's type system enhances its expressiveness and elevated program safety. For example, Rust supports a range of numerical types. Data types are explicitly declared by suffixing the value, for example, `3u32` denotes an unsigned 32-bit integer 3, and `6f64` indicates a 64-bit float 6. Rust does not allow implicit type coercion among primitive types: to sum an `u32` and a `f64`, one must be cast to the other using the `as` keyword. This strictness by design underscores Rust's rigorous approach to type management.

2.2.3 Trait

Trait [64] is a significant feature in the Rust type system. It defines abstract behaviours across various types. It can be comprehended as `interface` in Java, where the declaration of methods is provided, and the classes implementing the interface share similar behaviours. In Rust, types implementing the same traits share akin behaviours. For example, the `Eq` trait includes methods for comparison such as `min`, `max`. The `vector` type implements this trait, therefore different vectors can compare with each other. The numerical types inherently implement the

Eq trait, as they are naturally comparable to each other. Trait is also used in this project, and the detailed usage will be introduced in later parts.

2.2.4 Generics

Generic data types [60] is another core feature, that also commonly exists in other languages. It allows code reuse for different data types. Similar to C++ and Java, generics in Rust can be easily declared. To declare a generic type, just put `<T>` in the function signature or trait. In addition, generics can be combined with traits, meaning the generic type implements the trait, thus the methods from the trait can be called within the generic type. The trait is called **trait bound**. When substituting the generic with a concrete one, it must implement the trait. Trait bound can be declared with the `where` keyword, see Listing 3 for an example:

```
1 // This function returns a value of the same type as the vector element
2 fn do_something_with_vector<T>(vec: &Vec<T>) -> T where T: Clone
```

Listing 3: A generic function with trait bound

As can be seen in the example, the function takes a reference to a generic vector. The element of the vector has `Clone` [8] as its trait bound, where the `Clone` is a trait standing for an object that can be duplicated. Therefore, when passing a concrete type to the function, the type must implement `Clone` trait. If not, the compiler will detect this and fail the compilation.

2.2.5 Ownership

Ownership [62] is the core feature of Rust that makes it stand out among other languages. The ownership can be imaged as a variable that "owns" a piece of data. An ownership is constructed by variable binding, e.g. `let v = [1, 2, 3];`, which looks like a variable definition in C/C++. However, by this statement, the array `[1, 2, 3]` is bound to the variable `x`. It could be said `v` is the owner of the array, and **there can only be one owner of the data**. If `v` is aliased by another variable, e.g. `let x = v;`, the ownership will be transferred to `x`, and `v` becomes unavailable. Any attempt to access `v` will incur a compilation error. In contrast, if such aliasing happens in C/C++, both variables are accessible.

Here are the key concepts around the ownership system to better understand the uniqueness of Rust:

Immutability By default, a variable in Rust is immutable unless declared explicitly with `let mut`. This distinguishes Rust from C++ and Java where a variable is mutable by default and immutability is achieved by explicitly adding a modifier, which is `const` and `final` in C++ and Java respectively. This relieves the burden for programmers to maintain data immutability in programs as the compiler can guarantee that no unexpected mutations occur based on this rule.

Borrowing Like real life, an item that has its owner can be borrowed, but the borrower does not have ownership over the item. In Rust, another variable can borrow value via referencing, similar to a reference in C++. This is done by using `&`, e.g. `let y = &x;`. In this way, `y`

borrows `x`, it is a reference to `x`. As a general programming practice, this is useful to function invocations, where passing a value by reference reduces unnecessary data copies.

Immutability and borrowing are intertwined. There are rules regarding this. It can be summarized as follows: for a value,

- either multiple immutable borrows can be made, or only one mutable borrow.
- If a mutable borrow was attempted between the creation and use of an immutable reference, the compiler would fail the compilation and report this as an error. Same for vice versa.

While this significantly improves program safety, it caused a steep learning curve and the infamous "fight with the borrow checker" problem that requires thoughtful designs to solve the compilation errors [59].

By default, data cannot be mutated through a reference, unless the reference is declared with `&mut`. To create a mutable reference, the data must initially be declared as mutable with `let mut`. It is worth noting that sometimes it may confuse programmers that placing the keyword `mut` at the left-hand side (LHS) and right-hand side (RHS) of an equal sign has different semantics. `let mut x = &y;` stands for the address rather than the data, that `x` points to can be mutated. In contrast, `let x = &mut y;` allows the data that `y` owns to be mutated by `x`. But `x` cannot be mutated to point to elsewhere. There are more specifications regarding the borrowing rules which are omitted here for brevity. In summary, Rust imposes strict rules on the immutability of variables and references, thereby saving efforts in preventing unexpected mutations.

Lifetime Lifetime is the mechanism for preventing dangling references. Rust follows Resource Acquisition Is Initialization (RAII) like C++ [53], where an object is created within a scope, which is a pair of curly braces, and destructed when it goes out of the scope, the memory associated will also be freed. The destructor is implemented via the `Drop` trait [19]. Hence an object has its lifetime. A reference to an object cannot be used once the object is destroyed. In Rust, this rule is also enforced by the compiler. An example to demonstrate how Rust prevents dangling references by enforcing lifetime rules [68] can be seen in Listing 4:

```rust
// The longest function returns a reference
// to the longer one from string x and y
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

fn main() {
    let longer_str = String::from("I am longer");
    let shorter_str = String::from("short");

    let result = longest(string1.as_str(), string2.as_str());
    println!("The longest string is {}", result);
}
```

Listing 4: Rust lifetime demonstration (taken from [68])

In the example above, the `longest` function accepts two string references, determines the longer string and returns its reference. Now let's inspect the `longest` function in more detail, the `'a` is the lifetime annotation, and its single character with prime in front is Rust's syntax. The `<'a>` declares that the function has lifetime `'a`, which is akin to the way generics are declared. The function parameters and the return type are annotated with `'a` meaning they have the same lifetime. It is possible that the return value can either be x or y, but x and y are in the two branches, which are two scopes and hence different lifetimes. The compiler cannot determine the lifetime of the return value, therefore by annotating the parameters and return value `'a`, the compiler won't be confused by the lifetime of the two parameters. It is also worth noting that the lifetime annotation does not change the real lifetime of a reference. It is just a way of telling the compiler the lifetime relationship between annotated references. In the example above, the two parameters and return value have the same lifetime as perceived by the compiler since they have identical lifetime annotations, but indeed only the returned reference has the same lifetime as the reference to the longest string. The reference to the shorter string will be destroyed before the function returns.

Struct Lifetime Struct can be comprehended as its equivalence in C/C++ [12]. If there is a reference as a member of the struct, both the struct and reference should be annotated with a lifetime parameter. The usage of struct lifetime is as shown in Listing 5. Let's also inspect deep into this struct definition. The member `window_vec_ref` is a reference to a vector outside of the struct. The lifetime annotation is declared following the struct name. The Rust compiler enforces the rule that what the member references to does not go out of scope ahead of the struct itself, as this is to prevent dangling references. By attaching `'a` to the struct and the reference member, the struct object can be destructed at the earliest if either the struct object itself goes out of scope or the data that the member references goes out of scope. Thus the dangling reference problem can be prevented.

```rust
pub struct Window<'a> {
    pub window_vec_ref: &'a Vec<f64>,
}
let vector = vec![0i32; 3];
let window = Window { window_vec_ptr: &vector };
```

Listing 5: Rust struct lifetime

In summary, ownership is a distinguishable feature that makes Rust memory safe and stands out from many other mainstream languages. It is a set of rules enforced by compiler checks:

- A value has only one owner at a time.
- When the owner goes out of scope, the value will be dropped (RAII).

Other programming languages, either achieve the same with garbage collection, such as Java, and Go, or shift the responsibility to programmers such as C/C++.

2.2.6 Unsafe Rust

The unsafe feature allows programmers to perform risky operations. There are unsafe operations in C/C++, e.g. dereferencing a dangling raw pointer, which is a common error resulting in program crashes. Rust ensures memory safety at compile time by disallowing unsafe memory operations. However, programmers can perform operations that bypass the compiler's usual safety checks by using the keyword `unsafe` [67]. An example of dereferencing a raw pointer may look like Listing 6. There are more unsafe operations in this thesis. For brevity, these operations will be introduced on the spot where they first appear.

```rust
let arr = [10, 20, 30]; // initialize an array
let ptr = arr.as_ptr(); // raw pointer to the first element of arr

unsafe {
    // add offset of 2 to raw pointer and dereference it
    println!("arr[2] = {}", *ptr.add(2));
}
```

Listing 6: Rust unsafe example, operating on a raw pointer.

The dereference has to be wrapped within `unsafe` block, otherwise the compiler would recognize and report it as an error. Using `unsafe` operations allows for greater flexibility and control, albeit at the risk of compromising safety. Besides, from a perspective of design, `unsafe` transfers the responsibility for ensuring safety from the compiler to the developer. Therefore, by moving the unsafe code into a library, programmers who use the library do not have to be concerned about the unsafe code.

Foreign Function Interface (FFI) [20] is a mechanism that allows functions from one programming language to be called from another. In this project, Rust uses the FFI to invoke MPI functions from C. The `bindgen` [58] library generates Rust bindings to the C-based MPI library, enabling Rust to use MPI. However, since these bindings originate from C, they retain C-style characteristics, such as using C's numeric types and declaring raw pointers. Functions also require raw pointers as arguments, which Rust does not consider safe. Consequently, when using the FFI, these interactions must be enclosed within an `unsafe` block to bypass Rust's safety checks. Our major goal is to wrap the FFI-generated code into a library so that the programmer will not interact with it directly.

2.3 Remote Direct Memory Access

First raised in 1993 by Hewlett-Packard engineers [4], RDMA has shown great potential to build fast networked systems. Its idea is relatively simple: with RDMA, the memory of a process can be accessed remotely without the involvement of processors, thereby significantly improving the latency and throughput. It was then adopted by MVAPICH [66, 38], another implementation of MPI. Now RDMA has been an important networking technology in distributed computing. There has been a plethora of research building new systems with RDMA [34].

To program with RDMA, one can use the InfiniBand verb library [33], which is very low level, and hardware support is needed. The communication framework Unified Communication X (UCX) [52] can provide a higher level abstraction over the verbs, thereby providing MPI high-performance networking capability [61]. HPC application developers usually do not need to program with the verbs directly, with MPI, the RDMA network can be utilized.

Conceptually, the communication method in RDMA is similar to what is in MPI [61]. As aforementioned, when transmitting messages, MPI provides send recv or one-sided access, which exists in RDMA as well. In RDMA, the sender can issue a SEND request without knowing the memory address of the destination process. The receiver needs to issue a RECEIVE request to handle the reception of the incoming data. Besides, the sender can issue a WRITE request if knowing the destination memory address, hence no involvement of the receiver is required. Similarly, the receiver can issue a READ request to fetch data from a remote process if the memory address of the data source has been known. These semantics match what was discussed previously in MPI.

Sometimes, the acronyms RMA and RDMA can be confusing due to their similarity—they differ by only one letter. Although related, these are distinct concepts [25]. RDMA stands for Remote Direct Memory Access, a network technology that enables direct memory access, bypassing the operating system's kernel and avoiding data copying during transfers, thus significantly improving network throughput. RDMA is typically implemented with specialized network adapters, such as those from Mellanox, now part of Nvidia [48], and uses dedicated network standards like InfiniBand. In contrast, RMA, which stands for Remote Memory Access, is a programming model and a subset of MPI. It was first introduced in MPI-2.0 and updated in MPI-3.0 for better exploitation of RDMA networks.

```c
    // Common operations omitted,
    // i.e. header file inclusion, main function, MPI initialization
    double *data;
    MPI_Win win;
    MPI_Win_allocate(ARRAY_SIZE * sizeof(double), sizeof(double),
        MPI_INFO_NULL, MPI_COMM_WORLD, &data, &win);
    if (rank == 0) {
        for (int i = 0; i < ARRAY_SIZE; i++)
            array[i] = i * 1.0;
    }
    MPI_Win_fence(0, win);
    if (rank == 0) {
        MPI_Put(array, ARRAY_SIZE, MPI_DOUBLE, 1, 0, ARRAY_SIZE,
            MPI_DOUBLE, win);
    }
    // Alternatively, rank 1 gets the data from rank 0
    // if (rank == 1) {
    //    MPI_Get(array, ARRAY_SIZE, MPI_DOUBLE, 0, 0, ARRAY_SIZE,
    //      MPI_DOUBLE, win);
    // }
    MPI_Win_fence(0, win);
    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
```

Listing 7: Sample RMA program in C

3 Related Work

There has been previous work bringing Rust into the HPC world.

As mentioned previously, MPI functions are exported to Rust through the bindgen-generated `mpi-sys` [43] library. While this library allows developers to write MPI applications in Rust, the use of FFI necessitates that every MPI method invocation be wrapped in an `unsafe` block. This requirement essentially negates Rust's built-in safety features. Moreover, coding with mpi-sys resembles writing procedural and low-level C-style code, causing Rust to lose not only its safety features but also the benefits of its modern language features. For example, Listing 8 illustrates calling `MPI_Win_allocate`.

```
let mut window_base: *mut f64 = ptr::null_mut();
let mut window_handle: MPI_Win = ptr::null_mut();

unsafe {
    ffi::MPI_Win_allocate(
        (vector_size * size_of::<f64>()) as MPI_Aint,
        size_of::<f64>() as c_int,
        RSMPI_INFO_NULL,
        world.as_communicator().as_raw(),
        &mut window_base as *mut *mut _ as *mut c_void,
        &mut window_handle
    );
}
```

Listing 8: Allocating a window with mpi-sys

Listing 8 resembles the C code, recalls its usage in C from Listing 7, both undergo the same process: declare two mutable pointers, one for the pointer to the window, the other for the window object. Then call the method. Particularly for Rust, the variables must be initialized, therefore null pointers are used. Furthermore, the pointers have to be cast into mutable void pointers, as there is no void type in Rust. This is neither safe nor as flexible as C because variable initialization is mandatory in Rust. Our work is to eliminate these programming difficulties.

Based on mpi-sys, rsmpi [55] brings major components of MPI into the Rust world. Common MPI operations such as environment, and send recv have been supported so that the user can program in a Rust idiomatic way instead of C-like Rust code as above. If the basic send recv program in Listing 1 is written in rsmpi, this results in the code shown in Listing 9:

```rust
use mpi::traits::*;

fn main() {
    // Initialize MPI environment
    let universe = mpi::initialize().unwrap();
    // Obtain the world communicator
    let world = universe.world();
    // Obtain the rank of the current process
    let rank = world.rank();

    // Initialize the array to pass around
    let mut data = [0.0; 10usize];
    if rank == 0 {
        for i in 0..10 {
            data[i] = i as f64 * 1.0;
        }
        // Send the array to rank 1
        world.process_at_rank(1).send_with_tag(&data, 0);
    } else if rank == 1 {
        // Receive the array from rank 0
        world.process_at_rank(0).receive_into_with_tag(&mut data, 0);
        for i in 0..10 {
            println!("data[{}] = {}", i, data[i]);
        }
    }
}
```

Listing 9: Sample send recv in rsmpi

However, RMA is not included in `rsmpi`. Besides, `rsmpi` bridges the gap between C and Rust by leveraging the type system of Rust extensively. Traits are defined for converting Rust types and their equivalence in C. Our project has borrowed these traits to convert Rust types into C and extended rsmpi to include RMA.

There is more previous work regarding Rust, MPI and RDMA.

Tronge and Pritchard [65] re-implemented the intra-node communication module of Open MPI and have achieved the result that Rust is close to C in terms of performance and better safety. The work is motivated by the intrinsic safety and handy features of Rust, by re-writing the components of MPI in Rust, not only did the software improve in safety, but also accelerated the development speed by testing and enhanced maintainability. The benchmark has shown a marginal disparity in bandwidth and latency. Meanwhile, our work is on top of MPI, focusing on the RMA part. It also shows closeness in performance and better software development experience.

Blesel et al. [3] have implemented a prototype message passing library with Rust to enhance correctness checks at compile time and elevate usability. By leveraging the Rust compiler, the

type safety problems of MPI and memory safety issues of communications are addressed with generics and ownership of Rust. The performance evaluation has shown that their implementation has an acceptable disadvantage compared to MPI, but it could be attributed to abundant development and optimizations of MPI compared to their library. Another key finding is that the safety and usability feature of Rust has greatly improved their work experience. While our work also focuses on the safety and usability of Rust in HPC, the difference is that they have developed a proof-of-concept software that parallels MPI. Our work is to bring the RMA module of MPI to the Rust world.

Gerstenberger et al. [21] have developed an RMA implementation for the MPI-3.0 specification in C. Their work was driven by the insufficient utilization of the RDMA network of existing libraries. The specification and implementation of RMA methods were discussed. As the performance evaluation, the latency, throughput and scalability were shown to be at an advantage over existing libraries. In contrast, our study focuses on the usability of RMA by introducing it to Rust, while the former provides an RMA implementation, emphasizing its performance.

Levy et al. [32] shared their experience building an embedded OS in Rust. During their work, they came across conflicts between the language features and the intrinsic properties of embedded system programming. For example, resources in OS are shared, and there is possibly more than one mutable reference to the resource needed simultaneously, which is not prohibited by the ownership rules. They proposed workarounds that either lost maintainability or memory inefficient. Another solution is sacrificing safety guarantee by using `unsafe`. In general, their work also demystified that the impact of language can be isolated in system design. Language can be crucial in the overall design. During our development work, we had a similar experience. Part of our work requires mutable borrowing of two rows of matrix which is a 2D vector, and this is not allowed by the ownership rules. We must workaround by using either `unsafe` or split the matrix into multiple smaller ones, and create mutable references thereby. Consequently, programming became more complicated. In summary, the ownership is a double-edged sword.

4 Design and Implementation

In this section, we present how we leveraged Rust language features and the `rsmpi` library to bring RMA to the Rust world. Our design guideline is exposing concise, readable, organized and Rust idiomatic interfaces to the user.

4.1 Window Definition

As introduced previously, a window object binds a memory region to a group of processes. The memory region can be allocated manually by the programmer through calling MPI's memory allocation function like `MPI_Alloc_mem` or ordinary memory allocation methods, such as an array or vector initialization. Once allocated, a memory region is exposed to the group of processes via `MPI_Win_Create` so that other processes can access it. Another way to do this is to let MPI allocate memory and expose by `MPI_Win_allocate`. The latter way is more RDMA friendly [26], as the memory allocation is done via MPI, which can be optimized for RDMA networks.

To reflect the different ways of allocating and exposing memory regions, we decided to implement two types of windows: `CreatedWindow` and `AllocatedWindow`, in the form of Rust `structs`. Here is their definition:

```
1 pub struct CreatedWindow<'a, T> where T: Equivalence {
2     pub window_vec_ptr: &'a mut Vec<T>,
3     pub window_handle: MPI_Win
4 }
5 pub struct AllocatedWindow<T> where T: Equivalence {
6     pub window_vec: ManuallyDrop<Vec<T>>,
7     pub window_handle: MPI_Win
8 }
```

The `CreatedWindow` reflects what `MPI_Win_create` does: the member `window_vec` is a reference to a vector created outside of its encompassing struct. This reflects that `MPI_Win_create` accepts pre-allocated memory. Recall `'a` stands for lifetime specifier in Rust, meaning the lifetime of the reference is bound to the vector created separately. The `mut` modifier indicates that by using this reference, the underlying vector object can be modified. `Vec<T>` is the type specifier for a generic vector. We have chosen to represent a memory region with a vector, as it is the most frequently used data structure in most scenarios. Additionally, the vector has a consecutive memory region underneath, which reflects the layout of the memory of a window. Last but not least, the trait bound `Equivalence` is used here. We would like to borrow its functionality to convert Rust types to their equivalence in C.

The other member of the struct is the `window_handle`. It is what `MPI_Win_create` and `MPI_Win_allocate` return. It is a pointer to the window object, through which operations on the window can be performed.

As for `AllocatedWindow`, it also has member `window_vec`, but this time it is a generic vector wrapped within `ManuallyDrop` [37]. Having a vector within `ManuallyDrop` will prevent the compiler from calling the destructor to the vector by default, which is part of the RAII rule of a variable as previously introduced. However, `MPI_win_free` will be called at the end of the program, which frees up the window. Hence to avoid double freeing, we use `ManuallyDrop` to only let MPI free up the memory region. Furthermore, in contrast to `CreatedWindow`, having the vector object within the struct instead of a reference to an external vector also reflects the essence of `MPI_Win_allocate`, that MPI directly allocates memory.

4.1.1 Window Initialization

Listing 10 shows the implementation of creating a window. It accepts a reference to a vector as the argument, again implying the window is created upon an existing vector. The `&self` is the syntax of associated functions of Rust, meaning the method takes a reference to the struct that the method associates to. For `create_window`, the associated struct is the MPI communicator. The communicator struct has been pre-defined in the `rsmpi` library. As Rust emulates object-oriented programming, we put this function as the associate function of Communicator to better express the idea that the window is created on each process within the communicator.

```
pub fn create_window<'a, T>(&self, vec_ptr: &'a mut Vec<T>)
        -> CreatedWindow<'a, T> where T: Equivalence {
    let mut win = CreatedWindow {
        window_vec_ptr: vec_ptr,
        window_handle: ptr::null_mut()
    };
    unsafe {
        ffi::MPI_Win_create(
            win.window_vec_ptr.as_mut_ptr() as *mut std::ffi::c_void,
            (vec_ptr.len() * size_of::<T>()) as MPI_Aint,
            size_of::<T>() as std::ffi::c_int,
            RSMPI_INFO_NULL,
            self.as_raw(),
            &mut win.window_handle
        );
    }
    return win;
}
```

Listing 10: Implementation of window creation

For easy comparison, Listing 11 shows the original function signature in C [45], it can be easily found that the parameter list is simplified into a vector reference.

```c
int MPI_Win_create(
    void *base,
    MPI_Aint size,
    int disp_unit,
    MPI_Info info,
    MPI_Comm comm,
    MPI_Win *win
);
```

Listing 11: Function signature of creating a window in C

For `AllocatedWindow`, in Listing 12, the size of the window is the only parameter required. Calling `MPI_Win_allocate` will return the pointer to the base address of the allocated memory. With this pointer, a vector can be constructed by `Vec::from_raw_parts` [69]. Similar to `create_window`, the struct is returned for external use.

```rust
pub fn allocate_window<T>(&self, size: usize)
        -> AllocatedWindow<T> where T: Equivalence {
    let mut window_base: *mut T = ptr::null_mut();
    let mut window_handle: MPI_Win = ptr::null_mut();
    unsafe {
        ffi::MPI_Win_allocate(
            (size * size_of::<T>()) as MPI_Aint,
             size_of::<T>() as std::ffi::c_int,
             RSMPI_INFO_NULL,
             self.as_raw(),
             &mut window_base as *mut *mut _ as *mut std::ffi::c_void,
             &mut window_handle
        );
        let win = AllocatedWindow {
            window_vector: ManuallyDrop::new(
                Vec::from_raw_parts(window_base, size, size)
            ),
            window_handle: window_handle
        };
        return win;
    }
}
```

Listing 12: Implementation of window allocation

Same as previously, Listing 13 shows the signature of the original C function. It could be found that the interface has been greatly shortened and the size becomes the only parameter required.

```c
int MPI_Win_allocate(
    MPI_Aint size,
    int disp_unit,
    MPI_Info info,
    MPI_Comm comm,
    void *baseptr,
    MPI_Win *win
);
```

Listing 13: Function signature of allocating a window in C

4.1.2 Window Destruction

As aforementioned, Rust adheres to RAII rules. Simply by implementing the `Drop` [19] trait, the object destructor can be defined. Listing 14 shows the implementation of dropping `AllocatedWindow`. For `CreatedWindow`, it is mostly identical, except for having a lifetime attached. When the window object goes out of scope, the `drop` method will be called, and therefore recycle the allocated memory. Recall we have used `ManuallyDrop` to construct the vector for the window, without `ManuallyDrop`, the memory will be double freed by vector destructor and `drop`. Furthermore, freeing with `MPI_win_free` adheres to MPI programming general practices, as the window is initialized by MPI. Hence, we have pinned down letting MPI free the memory instead of the vector destructor.

```rust
impl<T> Drop for AllocatedWindow<T> where T: Equivalence {
    fn drop(&mut self) {
        unsafe {
            ffi::MPI_Win_free(&mut self.window_handle);
        }
    }
}
```

Listing 14: Implementing Drop trait for AllocatedWindow

4.2 Communication Calls

Besides window initialization and destruction, communication calls are the next major part. These calls are window-type agnostic, i.e. they can be invoked regardless of created or allocated. For better code organization, we have designed a trait to include these methods and let two types of Windows implement this trait. Listing 15 presents a snippet of the trait for demonstration:

```rust
pub trait Communication <T> {
    fn put_from_vector(&self, origin: &Vec<T>, target_rank: usize);
    fn get_from_vector(&self, origin: &mut Vec<T>, target_rank: usize);
    fn put(&self,
        origin: &Vec<T>, origin_disp: usize, origin_count: usize,
        target_rank: usize, target_disp: usize, target_count: usize
    );
    fn get(&mut self,
        origin: &mut Vec<T>, origin_disp: usize, origin_count: usize,
        target_rank: usize, target_disp: usize, target_count: usize
    );
}
```

Listing 15: Trait for Communication Call

For easy comparison, Listing 16 presents the original function signature in C [45]:

```c
int MPI_Put(
    const void *origin_addr, int origin_count,
    MPI_Datatype origin_datatype, int target_rank,
    MPI_Aint target_disp, int target_count,
    MPI_Datatype target_datatype, MPI_Win win
);
int MPI_Get(
    void *origin_addr, int origin_count,
    MPI_Datatype origin_datatype, int target_rank,
    MPI_Aint target_disp, int target_count,
    MPI_Datatype target_datatype, MPI_Win win
);
```

Listing 16: Equivalent in C

In our trait, `put_from_vector` and `get_from_vector` are the custom methods for pragmatic use in common scenarios, `put` and `get` are the ones resembling their original signature. We propose the `put_from_vector` and `get_from_vector` as a convenient way of transmitting a vector to other ranks, which is a common action in communications. Without this, the programmer has to invoke `MPI_Put` method which has a lengthy and obscure parameter list. Besides, it could be found the pointer `origin_addr` from C has become the generic vector `origin` in Rust. Although a pointer can be created upon a vector, we have opted to leave the vector reference type in the interface. The reason is that pointer is a fundamental concept in C, powerful but can be dangerous. Using reference is Rust idiomatic, therefore we have shifted the conversion from the user to our implementation. In addition, compared to the original interface, there is a new argument `origin_disp`. We propose this argument since the original one requires passing in a pointer. In C++ the pointer can point to an address other than the start of the vector, but the reference must point to the start. We have decided to propose the origin displacement and shift the work of creating a pointer from a reference along with

the pointer arithmetic to our library. Furthermore, the datatype is explicitly passed in as an argument in C. We can doubtlessly leverage Rust generics and predefined types in `rsmpi` to make it Rust idiomatic, as we had in window initialization.

For the implementation detail, to improve code reuse, we defined a function `common_put` as shown in Listing 17 private to the library. It explicitly requires a raw pointer instead of a reference as the trait above. Hence the trait methods can be implemented as Listing 18.

```
fn common_put<T>(
    origin: *const T, origin_count: usize,
    target_rank: usize, target_disp: usize, target_count: usize,
    window: ffi::MPI_Win
) where T: Equivalence {
    unsafe {
        ffi::MPI_Put(
            origin as *const c_void,
            origin_count as c_int,
            // Since T has trait bound Equivalence defined in rsmpi,
            // it can use equivalent_datatype().as_raw() to convert to C type
            T::equivalent_datatype().as_raw(),
            target_rank as c_int,
            target_disp as MPI_Aint,
            target_count as c_int,
            T::equivalent_datatype().as_raw(),
            window
        );
    }
}
```

Listing 17: Implementation of common_put

```
fn put_from_vector(&mut self, origin: &Vec<T>, target_rank: usize) {
    common_put(
        origin.as_ptr(), origin.len(),
        target_rank, 0, origin.len(),
        self.window_handle
    );
}
```

Listing 18: Implementation of put_from_vector

For the `put` method, it could be found in Listing 19 that the `origin` vector is converted to a pointer with the `as_ptr` method, and forwarded `origin_disp` with `add` method. This is the code to move the pointer forward that was mentioned previously. Since `add` is unsafe, it must be wrapped within `unsafe`. After that, `common_put` can be invoked.

```rust
fn put(
    &self,
    origin: &Vec<T>, origin_disp: usize, origin_count: usize,
    target_rank: usize, target_disp: usize, target_count: usize
) {
    // call add to move pointer forward, unsafe
    let origin_addr = unsafe { origin.as_ptr().add(origin_disp) };
    common_put(
        origin_addr, origin_count,
        target_rank, target_disp, target_count,
        self.window_handle
    );
}
```

Listing 19: Implementation of put

4.3 Synchronization Call

Same as a communication call, a synchronization call is window type agnostic - hence we define its trait as shown in Listing 20, whereas Listing 21 shows the original function signature in C.

```rust
pub trait Synchronization {
    fn fence(&self);
    // group: a group of processes
    fn post(&self, group: &UserGroup);
    fn start(&self, group: &UserGroup);
    fn complete(&self);
    fn wait(&self);
    fn exclusive_lock(&self, rank: Rank);
    fn shared_lock(&self, rank: Rank);
    fn unlock(&self, rank: Rank);
}
```

Listing 20: Trait for Synchronization Call

```c
int MPI_Win_fence(int assert, MPI_Win win);
// assert argument: accepts different pre-defined values
// this argument is to instruct MPI to do different optimizations
// 0 is the simplest value to make the function work
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win);
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win);
int MPI_Win_complete(MPI_Win win);
int MPI_Win_wait(MPI_Win win);
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win);
int MPI_Win_unlock(int rank, MPI_Win win);
```

Listing 21: Synchronization calls in C

The function signature of the synchronization call is simpler, and so is our trait. The parameters become their equivalent in Rust. The window becomes the `&self`, again implying the methods are invoked on the window object. The `assert` arguments are currently hard coded as 0 in our implementation, as it is the simplest value to let the method work. As for the scope of the work, we have restrained to 0. Moreover, for the original `MPI_Win_lock`, the `lock_type` argument accepts two values: one is `shared_lock`, the other is `exclusive_lock`. Out of readability, we have shifted the types of locks into function names. The lock method now becomes two separate methods: `exclusive_lock` and `shared_lock`. Listing 22 presents the implementation of `exclusive_lock` for demonstration.

```rust
fn exclusive_lock(&self, rank: Rank) {
    unsafe {
        ffi::MPI_Win_lock(
            // lock type
            ffi::MPI_LOCK_EXCLUSIVE as c_int,
            // target rank
            rank as c_int,
            // assert
            0,
            // window object
            self.window_handle
        );
    }
}
```

Listing 22: Implementation of `exclusive_lock`

Now with our library, a simple RMA program that transmits a double vector as Listing 23 can be rewritten into Listing 23

```
// Import and enclosing main function omitted
let universe = mpi::initialize().unwrap();
let world = universe.world();
let rank = world.rank();

let mut window_base: *mut f64 = ptr::null_mut();
let mut window_handle: MPI_Win = ptr::null_mut();

unsafe {
    ffi::MPI_Win_allocate(
        (vector_size * size_of::<f64>()) as MPI_Aint,
        size_of::<f64>() as c_int,
        RSMPI_INFO_NULL,
        world.as_communicator().as_raw(),
        &mut window_base as *mut *mut _ as *mut c_void,
        &mut window_handle
    );
}
let mut window_vector = ManuallyDrop::new(
    unsafe {
        Vec::from_raw_parts(window_base, vector_size, vector_size)
    }
);
unsafe {
    ffi::MPI_Win_fence(0, window_handle);
}
if rank == 0 {
    unsafe {
        ffi::MPI_Put(
            window_base as *mut c_void,
            window_vector.len() as c_int,
            f64::equivalent_datatype().as_raw(),
            1,
            0,
            window_vector.len() as c_int,
            f64::equivalent_datatype().as_raw(),
            window_handle
        );
    }
}
unsafe {
    ffi::MPI_Win_fence(0, window_handle);
}
unsafe {
    ffi::MPI_Win_free(&mut window_handle);
}
```

Listing 23: Transmitting a double array with `mpi-sys`

```
// Imports and enclosing main function omitted
let universe = mpi::initialize().unwrap();
let world = universe.world();
let rank = world.rank();

let mut win: AllocatedWindow<f64> = world.allocate_window(vector_size);

win.fence();
if rank == 0 {
    win.put_from_vector(&win.window_vec, 1);
}
win.fence();
```

Listing 24: Transmitting a double array with our library

In summary, our design aims to ease the burden of programming with RMA which is low-level and unsafe. We extract the common use case, wrap the low-level library, and expose a simpler interface to the user. Users can program with high-level Rust constructs, such as vector and Rust types, rather than handling the pointer to vector, or casting Rust types into C types, as these efforts have been shifted to the library. Furthermore, we introduced a destructor to our library, which is an important property of RAII and thereby frees up the window automatically.

5 Evaluation

In this section, we conducted a series of experiments, including micro benchmarks, application level performance and programmability to evaluate the library.

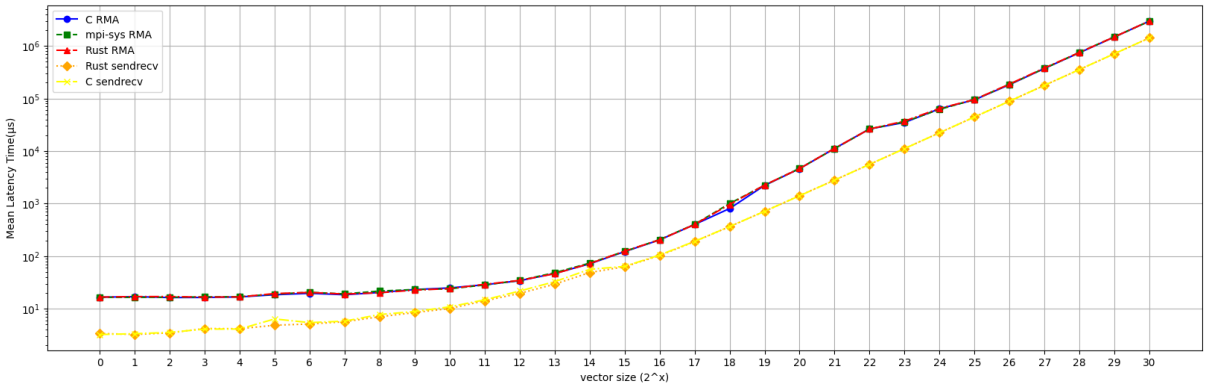
Unless explicitly stated, the experiments were performed on the Distributed ASCI Supercomputer DAS-6 [2] at the Leiden University site, where each node has a 24-core AMD EPYC-2 (Rome) 7402P CPU, 128 GB memory, interconnected by 100 Gbit/s InfiniBand which supports RDMA network. The network supports two modes: using InfiniBand (IB) directly and IP-over-InfiniBand [30] (IPoIB) which transmits IP packets on top of InfiniBand. In addition, the programs were compiled with the highest optimization option: `-O3` for C and `--release` for Rust. The MPI version is Open MPI 4.1.1, which implements the MPI-3.1 standard, with ucx version 1.11.2. GCC version 9.4.0, cargo version 1.74.0, rsmpi 0.5. For plots, the error bar is neglected for slight standard deviations.

5.1 Micro Benchmarks

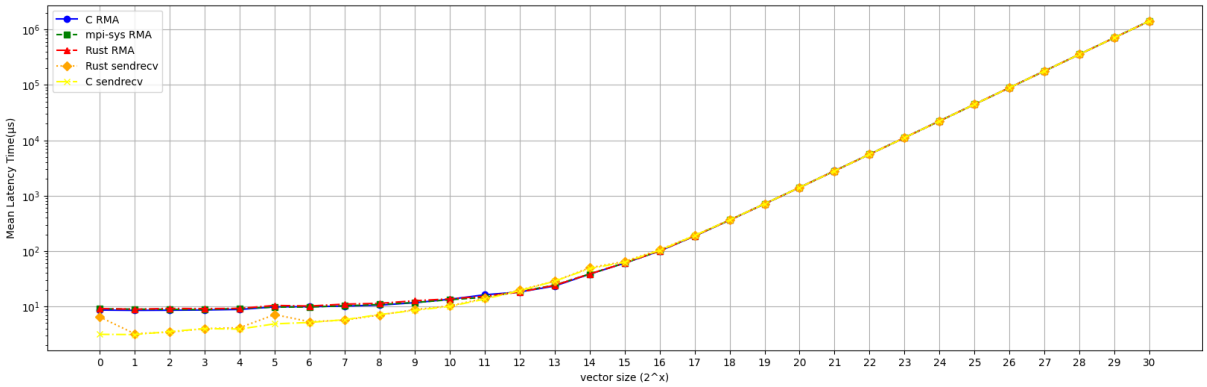
We first run micro-benchmarks to evaluate the latency of communication. The programs are One-to-One Ping Pong and One-to-Many Ping Pong.

5.1.1 One-to-One Ping Pong

We first performed a ping-pong latency test with two nodes to test out the latency. The experiments were performed against the growing size of a vector of `f64` type, which is equivalent to `double` in C. The vector size increases by a power of 2 for optimal cache utilisation. The metric is the Round Trip Time (RTT) of a vector. The maximum size can reach 2^{30} as limited by MPI itself. During each run of different sizes, the vector was allocated before Ping-Pong begins and destroyed when the program exits. The round trip was repeated 12 times, and the times with minimum or maximum results were removed before doing statistical analysis. We also conducted benchmarks on two different network settings to provide more insight into the performance: using InfiniBand directly and IPoIB. The program is written in five versions: send recv and RMA in C, for Rust, it is `mpi-sys` (the unsafe C-like library by FFI), our RMA library, and send recv in `rsmpi`.



(a) Ping-Pong direct IB

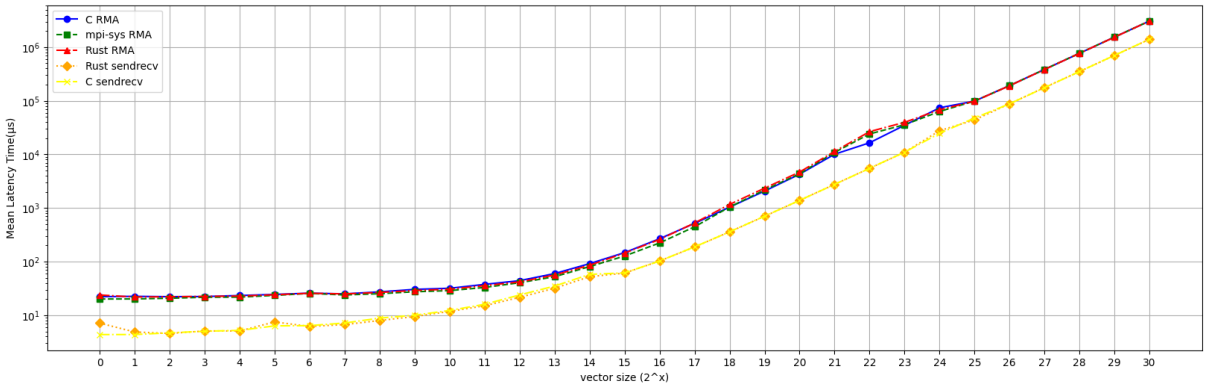


(b) Ping-Pong IP over IB

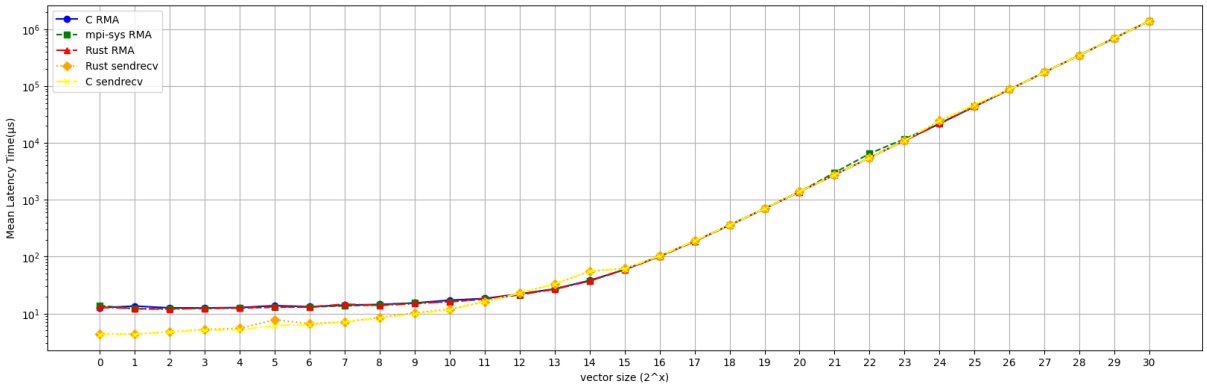
Figure 2: Ping-Pong latency test on InfiniBand (axes in log scale)

As shown in Figure 2, for directly using InfiniBand, send recv has marginal lower latency than the three variants of RMA, while the results for the RMAs are overlapped. For IPoIB, send recv has slightly lower latency at a small vector size, then overlaps with RMAs at a larger vector size. We believe this is due to the sophisticated communication method since the early MPI version, send recv has been highly optimized, while RMA was first introduced in MPI-2.0, and renovated in MPI-3.0 [21]. However, the Rust versions are the same as C for the RMA itself. This demonstrates that Rust RMA preserves the performance of C.

Besides testing on InfiniBand, we also conducted a test on DAS-6 at the VU Amsterdam site, which employs RDMA over Converged Ethernet (RoCE) [54] as the network.



(a) Ping-Pong direct RoCE



(b) Ping-Pong IP over RoCE

Figure 3: Ping-Pong latency test on RoCE (axis in log scale)

From Figure 3 can also be found that with RoCE, the latency pattern stays the same as with InfiniBand. This demonstrates that the library is portable in terms of performance.

To provide further insights into the performance at small sizes, we present the latency data of a double array with only one element in Table 1, i.e. $x = 0$ in the graphs above. The table shows that send recv achieves the best latency regardless of language. For the RMA versions, Rust has preserved the same latency as C. This trend is the same across the four network settings, proving that our library performs as well as its C equivalent. However, the key performance factor is the method.

	C RMA	mpi-sys RMA	Rust RMA	Rust sendrecv	C sendrecv
Direct InfiniBand	16.5469	16.5165	16.6580	3.3772	3.2281
IPoIB	8.6179	9.0889	9.1757	3.209	3.1412
Direct RoCE	22.2126	20.0955	23.8849	4.3002	4.3689
IP over RoCE	12.5936	13.8471	12.9895	4.3993	4.2769

Table 1: Latency transmitting a double array of size 1 (microseconds)

Moreover, we have conducted a comprehensive test against different network settings to find an optimal one to use the library. Figure 4 exhibits that the optimal network is IPoIB, meanwhile, IP over RoCE is only slightly slower than the former at a small vector size and merged at larger sizes. It may seem counterintuitive that both direct network settings are defeated by IPoIB. Indeed, this is strongly related to the implementation detail of MPI. Open MPI itself is built

upon lower-level network libraries, such as ucx, and Infiniband verbs. Their proper usage of networks may be vital to the real performance of MPI [23, 14].

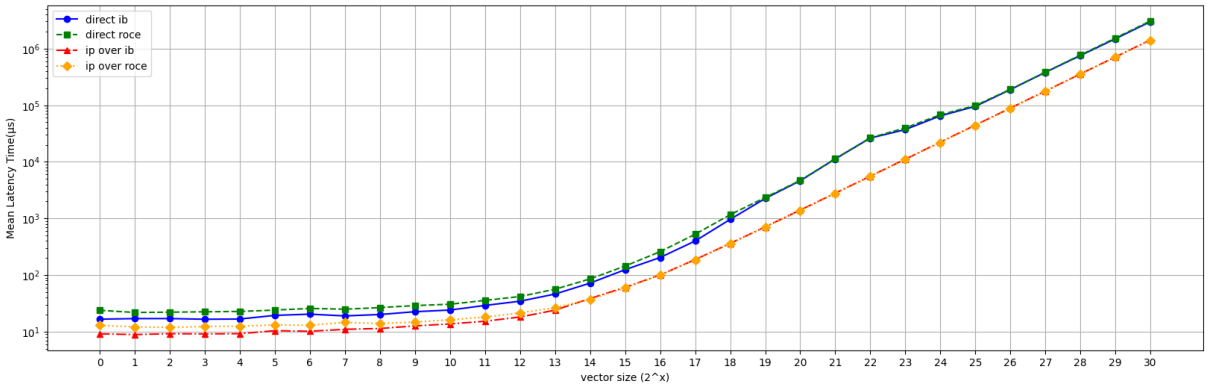
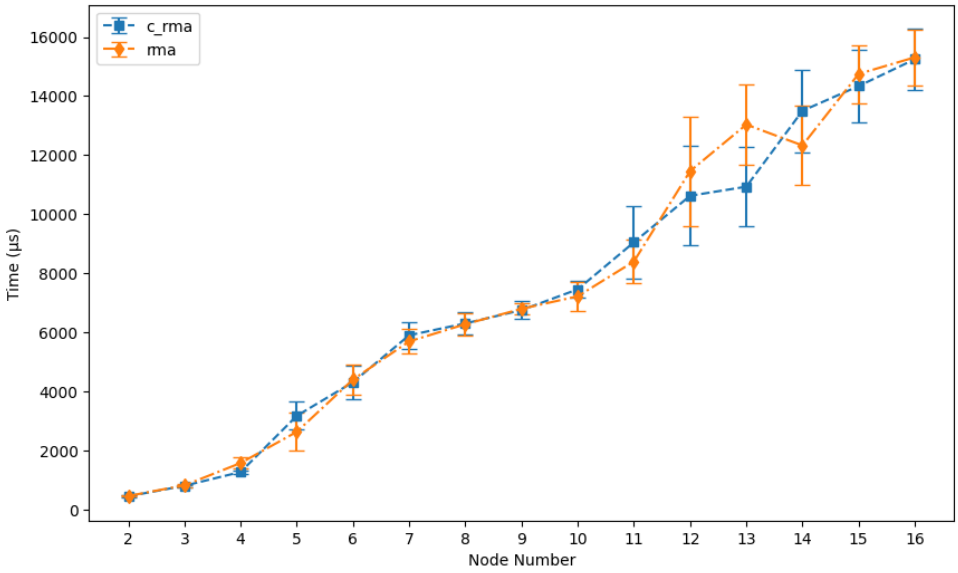


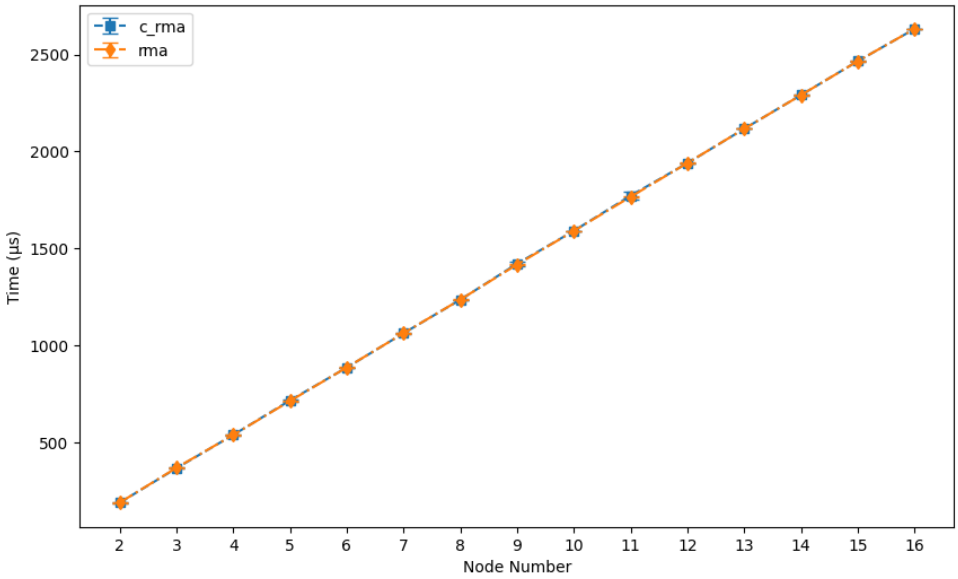
Figure 4: Ping Pong InfiniBand vs RoCE (axes in log-scales)

5.1.2 One-to-Many Ping Pong

One-to-Many Ping Pong is one process that sends a message to multiple processes, and all these processes send the message back to the source process. Since there are no collective methods in RMA, the source process sends messages one by one within a loop. RTT is used as the metric. The previous benchmark has shown negligible performance distinction between different RMA flavours. This experiment only focuses on the comparison between C RMA and Rust RMA using our library to provide further performance insights.

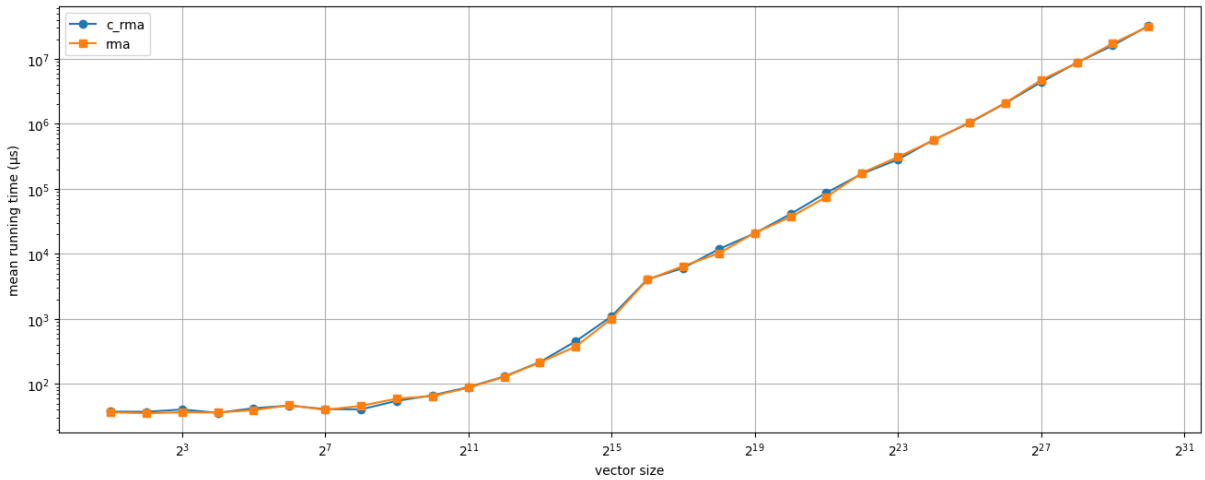


(a) 1-to-many Ping Pong direct InfiniBand

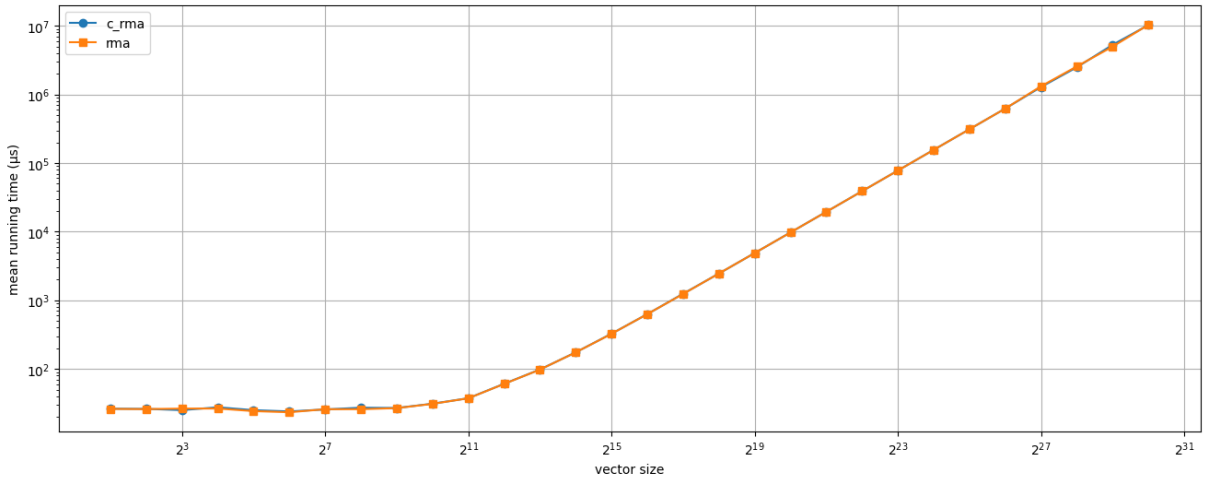


(b) 1-to-many Ping Pong IP Over InfiniBand

Figure 5: 1 to many Ping Pong by node number



(a) 1-to-many Ping Pong direct InfiniBand



(b) 1-to-many Ping Pong IP Over InfiniBand

Figure 6: 1 to many Ping Pong by vector size (axis in log scale)

From both latency tests in Figure 5 and 6, it could be found that the library has achieved identical latency as C, regardless of the network setting. For simple programs such as Ping Pong, Rust has achieved the same performance as C.

5.2 Application

In this part, we have implemented the Successive Over-Relaxation (SOR) [42], an iterative method of solving linear equations. A pseudo-code of which is shown in Algorithm 1:

Algorithm 1 Synchronous SOR

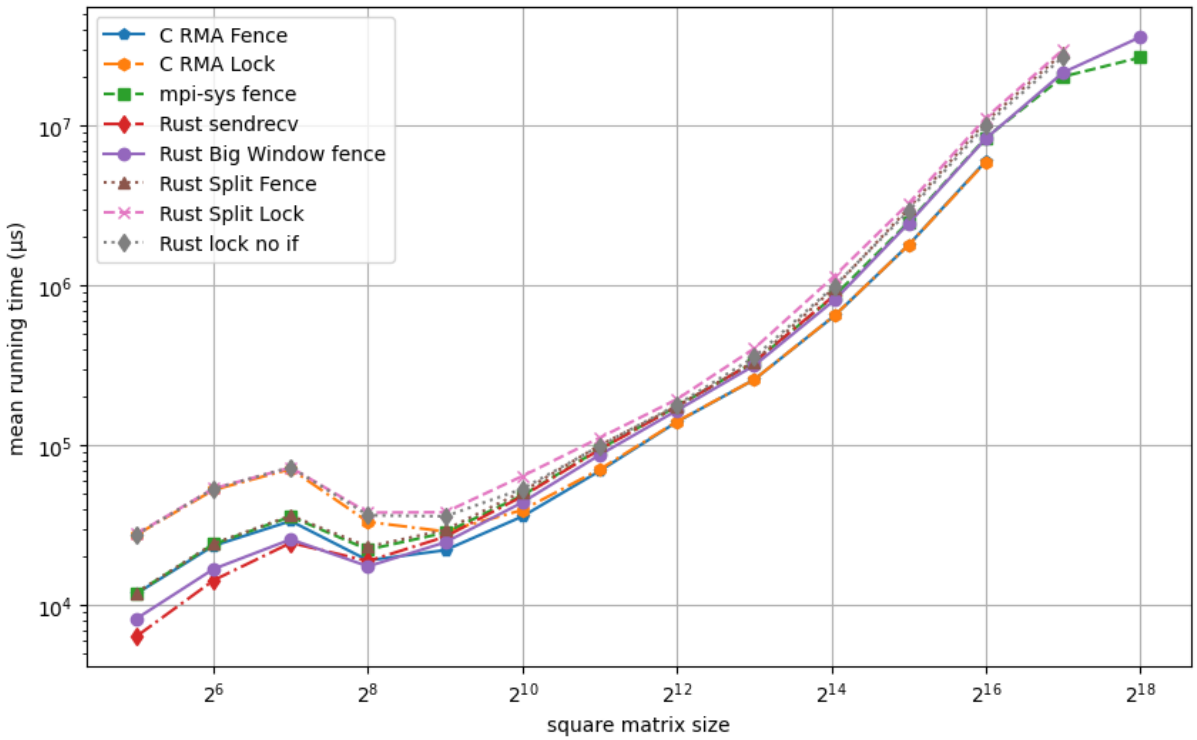
```
Initialize MPI environment (size, rank, etc.)
Decompose matrix into processes by calculating row number with rank
Allocate local submatrix as 2D vector for each process
Initialize matrix values
while condition do
    Send one row to rank − 1
    Send another row to rank + 1
    Perform computation on the local submatrix
end while
Terminate MPI environment
```

The implementations are synchronous, within which for each process, the computation process takes place after communication completes. Unless stated, the `fence` was used as the synchronization method for RMA, and `Put` as the communication call. Three different implementations have been written, they are: 1. C RMA; 2. Rust RMA with mpi-sys; 3. rsmpi send recv. Additionally, due to the ownership rule of Rust, when using our library, it is not allowed to open up two windows on two rows from the same 2D vector respectively. As shown in Figure 7, it looks like it is only two rows of the matrix were borrowed. Indeed, it is the matrix itself that was borrowed. For the workaround solution, we must either: 1. for each process, split its local matrix into parts so that the windows can be initialized on the parts. Therefore we named it "Split"; 2. Use a 1D vector, and indexing as if a 2D vector by conversion, so that only one mutable borrow is needed. The window is initialized using the whole vector. As the matrix can be huge, we named it as "Big Window". For Split, we have implemented two versions: one using `fence` as the simplest way of synchronization, and the other is `(un)lock` for fine-grained synchronization. Moreover, due to the implementation detail of Split, several conditional branches are frequently used during the computation process. To evaluate the impact of if branches on the program, based on the `(un)lock` Split, we have another version with if branches eliminated, which does not affect the result. And we name this "Rust lock no if".

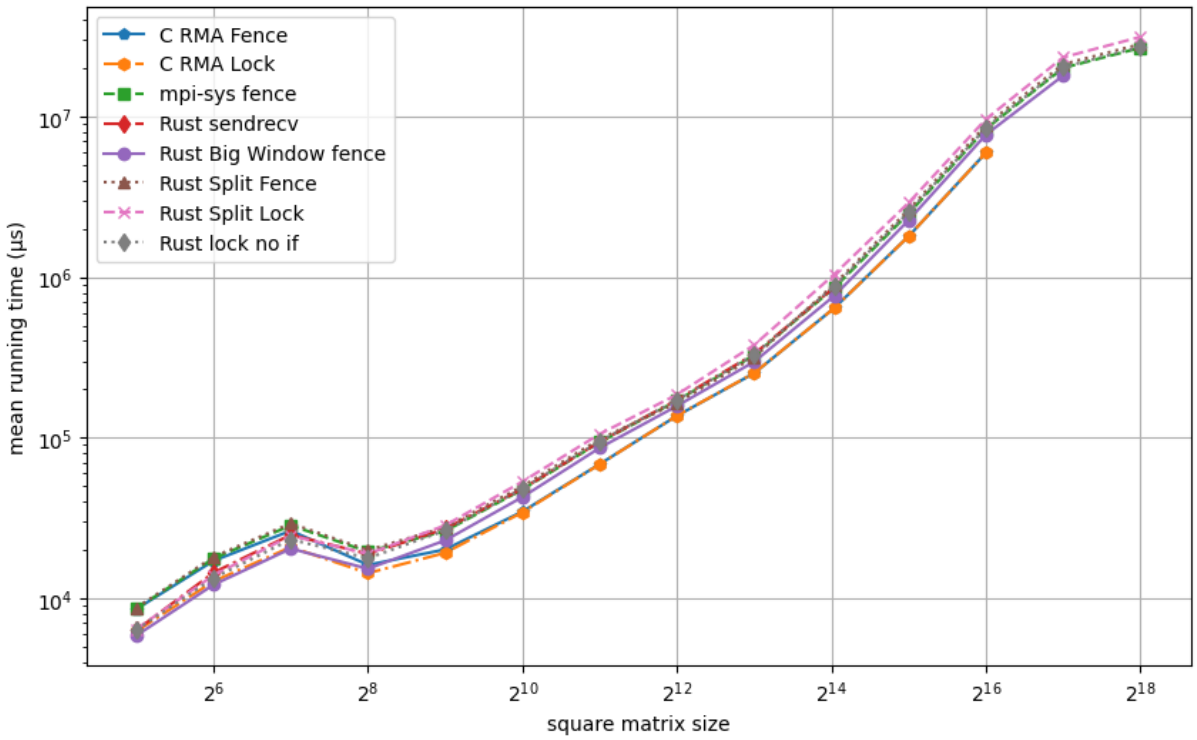
```
error[E0499]: cannot borrow `matrix` as mutable more than once at a time
   --> src/sor_rma_raw.rs:123:56
    |
122 |     let window_local_ub = world.create_window(n_col, &mut matrix[local_ub]);
    |                                                       ------ first mutable borrow occurs here
123 |     let window_row_0 = world.create_window(n_col, &mut matrix[0]);
    |                                                     ^^^^^^ second mutable borrow occurs here
...
228 | }
    | - first borrow might be used here, when `window_local_ub` is dropped and runs the `Drop` code for type `CreatedWindow`
```

Figure 7: Double mutable borrow fails

For each square matrix size, the program is repeated 12 times. The minimum and maximum results were excluded. The number of nodes to run the program is 8. Each element in the matrix is `f64` in Rust and `double` in C.



(a) SOR direct InfiniBand



(b) SOR IPoIB

Figure 8: SOR performance on growing matrix size (axis in log scale)

Given Figure 8, it is worth noting that, the max matrix size that can be reached is 2^{18}. The program will either throw insufficient memory or exceed the time limit to run a job in the cluster which is 15 minutes. Moreover, most significantly, the advantage of RMA compared to

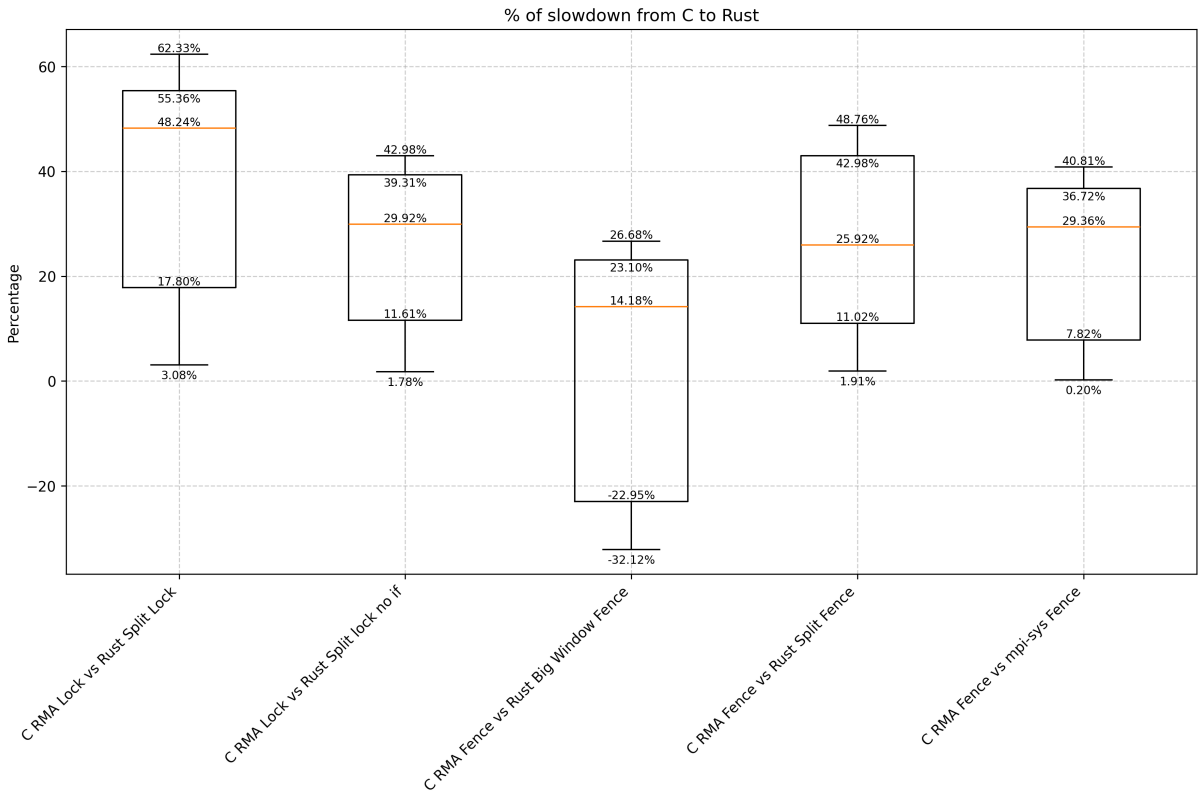


Figure 9: % of slowdown from C to Rust, IPoIB

send recv has shown: given 15 15-minute jobs running time limit, for both network settings, the max matrix size that can be completed for send recv is 2^{14}. Contrarily, all RMAs have reached larger size and completed 12 repetitions. We believe this is due to the additional memory copies of send recv having resulted in considerable overhead, even though only two rows from the matrix participate in the communication. The C RMA is slightly faster than the Rust RMAs given a matrix size larger than 2^8. However, C RMA could not proceed over 2^{16} due to the process being killed by the Linux OOM Killer. As for the different RMA variants, there is no significant performance disparity, especially with IPoIB. Similar to Ping Pong latency tests, running directly on InfiniBand, RMAs are slower than send recv given the small matrix size, especially since the locks are noticeably slower. Furthermore, we present the percentage by which Rust is slower than its C counterpart in Figure 9. The matrix sizes that C RMA lacks are excluded. For Rust Split Lock, since it is a suboptimal implementation, its slowdown is more significant than other variants. For Rust Big Window Fence, its minimum slowdown is negative meaning it is even faster. Moreover, its mean and max are noticeably smaller than other comparisons. It can be observed from Figure 8b that Rust Big Window is initially faster than C RMA Fence. We believe this can be attributed to addressing a 1D vector being faster than a 2D vector for small matrix sizes, as the former presents the matrix with a 1D vector. For other variants, they exhibit acceptable slowdown. The median and max are capped by 30% and 50% respectively. In summary, it can be concluded that our library has achieved a variable performance discrepancy to C. The optimal implementation can be even faster, but the opposite of sub-optimal.

Besides testing the time, we have conducted evaluations regarding scalability. For each pro-

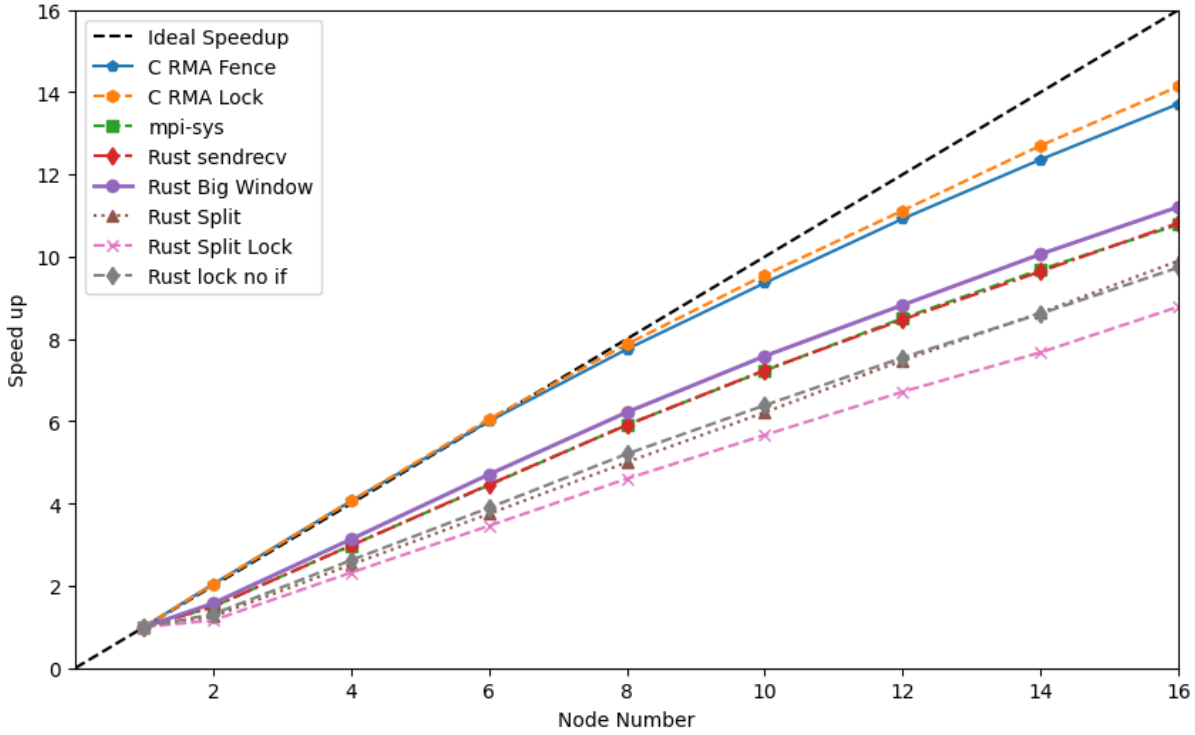
gram, the number of nodes starts from 2 to 16, and strides by 2. The matrix size is identical: 2^{14} across all experiments. The program that all variants are compared against is a sequential implementation in C.

From Figure 10, it could be found that results by the two network settings look similar, except send recv, mpi-sys, Rust split fence and split no lock are closer in IPoIB. This again demonstrates that IPoIB is better for running RMA programs. C RMA has achieved the best scalability among these lines, while Rust Split Lock has the worst. For the Rust RMAs with `fence`, Big Window showed more ideal speed up than other versions. These behaviours can be attributed to the synchronization as the major factor. In the Big Window implementation, there is only one window, therefore a matching pair of `fence` invocation is sufficient. On the contrary, other RMAs have two windows, which requires two pairs of `fence` to be called. Moreover, the more nodes, the larger the synchronization overhead. As for Split Lock, since locking does not guarantee the local matrix of each process has been updated before computation starts, a barrier after communication calls becomes necessary, which brings extra overhead besides the locks. Additionally, the "no-if" implementation of Rust split lock achieved better scalability than if-s, which has exhibited that excessive conditional branches also slow down the program. In addition, Table 2 shows the ratio of speed up from sequential to 16 nodes. It matches Figure 10 well. Last but not least, the programming language being used can have impacts. Due to their design differences, what is feasible in C may not be available in Rust. The workarounds bring overheads. This part will be discussed further in detail in the Discussion section.

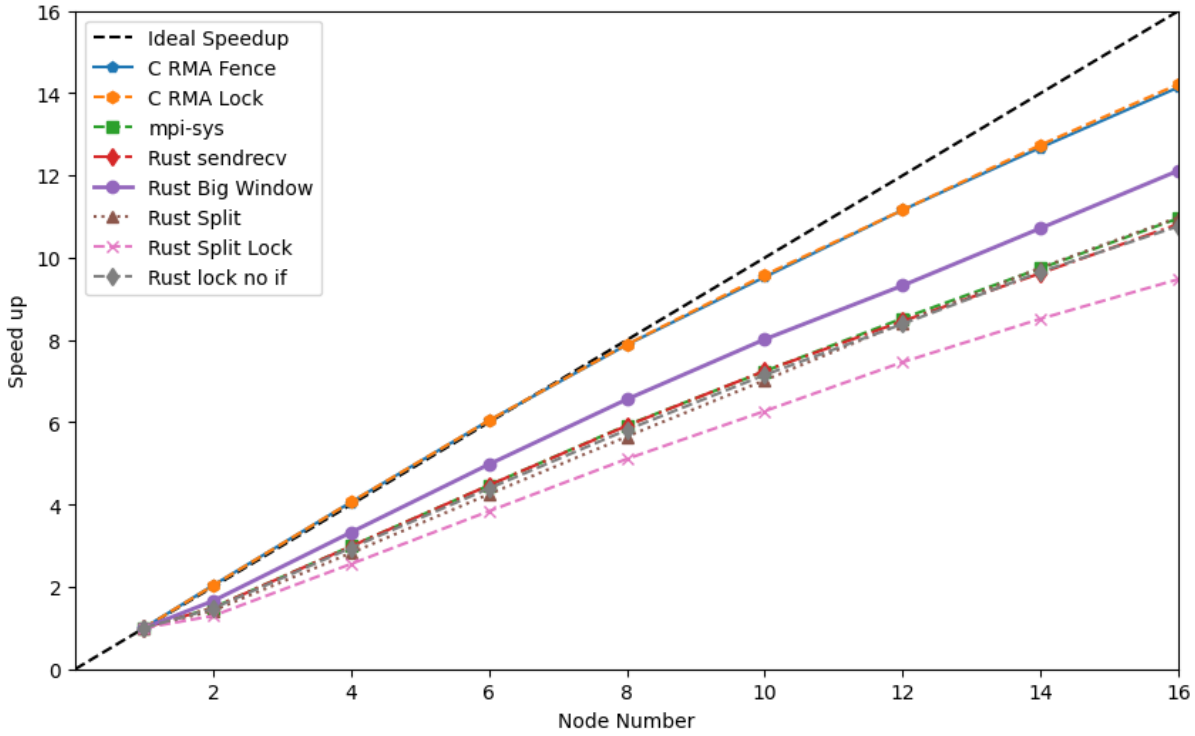
	C RMA Fence	C RMA Lock	mpi-sys	Rust sendrecv
Direct IB	13.71	14.14	10.78	10.82
IPoIB	14.14	14.21	10.96	10.81

	Rust Big Window	Rust Split	Rust Split Lock	Rust lock no if
Direct IB	11.21	9.903	8.784	9.739
IPoIB	12.11	10.98	9.479	10.77

Table 2: Speed up ratio sequential to 16 nodes



(a) Speed up with direct InfiniBand



(b) Speed up with IPoIB

Figure 10: SOR speed up

5.3 Programmability

We have evaluated the programmability of the library. The metrics used are intuitive: the number of lines of code and unsafe blocks, as well as readability.

Recall initializing a window of double with `MPI_Win_allocate` from `mpi-sys` as shown in Listing 25:

```
let mut window_base: *mut f64 = ptr::null_mut();
let mut window_handle: MPI_Win = ptr::null_mut();

unsafe {
    ffi::MPI_Win_allocate(
        (vector_size * size_of::<f64>()) as MPI_Aint,
        size_of::<f64>() as c_int,
        RSMPI_INFO_NULL,
        world.as_communicator().as_raw(),
        &mut window_base as *mut *mut _ as *mut c_void,
        &mut window_handle
    );
}
```

Listing 25: Allocating a window with mpi-sys

With our library, this can be simplified as Listing 26:

```
let window: AllocatedWindow<f64> = world.allocate_window(size);
```

Listing 26: Allocating a window with Rust RMA library

Recall calling `MPI_Put` to write contents to remote process in Listing 27:

With our library: this can be easily done with the following in Listing 28:

```
window.fence();
window.put_from_vector(&mut source_vec, target_rank);
window.fence();
```

Listing 28: Put with RMA Rust library

We also take the total lines of code for each program into account. It was counted roughly: to keep the readability of code, we preserve the empty lines between logical blocks, e.g. initializing variables, and doing computation, where code aims for the same purpose clusters. New lines exist at a lengthy argument list of method invocation, e.g. `MPI_Put`.

```
unsafe {
    MPI_Win_fence(0, window);
    MPI_Put(
        source_vec.as_mut_ptr() as *mut c_void,
        source_vec.len() as c_int,
        RSMPI_DOUBLE,
        target_rank,
        0,
        source_vec.len() as c_int,
        RSMPI_DOUBLE,
        window
    );
    MPI_Win_fence(0, window);
}
```

Listing 27: Put with mpi-sys

C RMA Fence	C RMA Lock	mpi-sys Fence	Rust sendrecv	C sendrecv
201	203	210	153	142
Rust Big Window fence	Rust Split Fence	Rust Split Lock	Rust lock no if	
171	181	178	215	

Table 3: Lines of code of each program

From Listing 25 vs 26 and 27 vs 28, it can be told that the lines of code are significantly reduced. Besides, the complex details of RMA methods, which consist of several arguments and the esoteric conversion of types from Rust to C, such as line 10 in Listing 25, are encapsulated and hence eliminated for the user. It is further demonstrated in Table 3 that total lines of code are reduced with our library while readability is preserved. Moreover, `Unsafe` block is also eradicated at the user's code. In addition, the code becomes more readable, adhering to object-oriented style: the RMA method is invoked via the window object, and the window allocation method is invoked with `world`. The semantics also map to their original form: a window is initialized with a communicator, and the methods are invoked by passing the window object.

6 Discussion

In this section, we present the experience and thoughts about the work. We give the benefits and drawbacks of Rust and our library, which can be useful for future programmer's reference:

Here are some concrete benefits of Rust:

- *Installation and configuring Rust is straightforward*. The installation can be easily done by following the instructions from the official website. And the process is mostly just by one line of command or an installation package. In contrast, for configuring C++, depending on specific needs, different compilers of C++ might be chosen, which in turn their prerequisite action needed.

- *Cargo is a very powerful tool*. It contains almost everything for the full life cycle of development. From initializing a new template project, building and running, testing and releasing, dependency management, and code linting, all can be done easily with a single command. During our work, we did not come across any tooling issues. Surely our project has simple dependencies, but to introduce our customized dependency, a line of simple config in `Cargo.toml` also does the work. This has greatly sped up the development efficiency, allowing us to focus on the problem itself.

- *Rich Language Features*. Needless to mention the ownership rules that greatly contribute to program safety, Rust also supports object-oriented programming, which has been utilized in our work. Besides, it has many modern language features: different kinds of looping, a powerful type system, and a vector type that supports initialization in various ways, etc. For example, we constructed vectors with `from_raw_parts` method.

We also give our thoughts regarding the drawbacks of Rust and our library:

- *Ownership can be limiting*. As previously mentioned in Figure 7, we had to use workaround solutions to pass the borrow check, as a result, indexing became more complicated and lowered memory efficiency.

  In addition to the 2D vector, in Figure 11, line 140 is trying to call `get` method to write to its window with data from its successor rank. The destination of the `get` method is the vector `huge_window.window_vec`. The invocation requires creating a mutable reference to `huge_window`, although we are referring to the `window_vec` within. Besides, an immutable reference to `huge_window` also must be created and used, because `get` method is invoked on it, recalling the first parameter of `get` is `&self`. However, the compiler has recognized this and raised an error. It turns out the `get` method cannot be used for writing to the window which invokes the `get` method. The error message shows that mutable borrow happens between the creation and use of immutable borrow, which is not allowed by the ownership rules. Although using `get` to write to the calling process's window is not a standard use but not discouraged by MPI reference [40], `Get` method is intended to write to the memory region that is not in the window. Through our practical use, writing to Windows also works. Considering this, as a workaround, we encourage users to use `unsafe` directly. In summary, ownership is a trade-off - better safety but less flexibility.

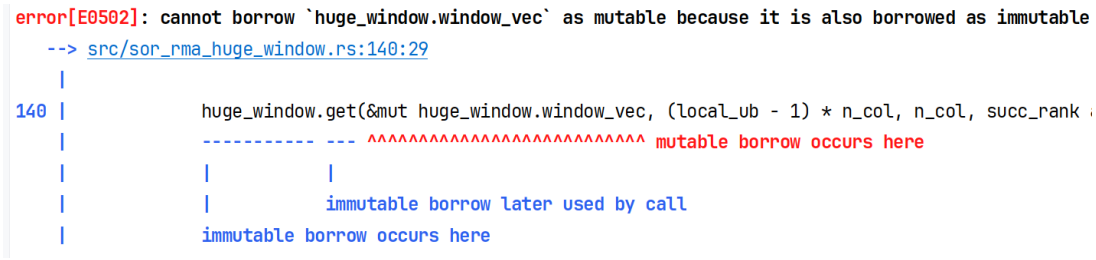


Figure 11: Immutable borrows when mutable borrow exists

- *Safety is bounded in the context of MPI*. The safety of Rust only takes effect in the context of a single process. However, within MPI, the concurrent issues, such as synchronization, should still need to be addressed manually by the programmer.
- *Rust programming style contradicts MPI*. It is common in MPI to declare a variable first, then pass it into an MPI method and have values assigned. However, accessing an uninitialized variable is not allowed in Rust. The workaround can be declaring a mutable variable, and assigning it random values first. For a safe programming practice, the mutable variable should be avoided as much as possible. This is what the `rsmpi` and our library have contributed to.
- *Performance is heavily influenced by program design*. By "program design", it means the implementation detail due to language features. For Split SOR, due to complicated indexing, we implemented two versions: one with multiple conditional branches to read the expected row, and the other eliminated conditional branches by explicitly setting the row number. The former is relatively easier to implement, while the latter is more complex. As a result, the latter outperforms. The conditional branch is the superficial difference that resulted from the ownership rules. This showed that programming language can impact implementation details, which in turn influence the performance.

Last but not least, we present our lesson learned during the process of learning Rust and designing the interface:

- *Learn the "hard" parts of Rust first* Designing the interface is a comprehensive process. Our library contains these elements of Rust: borrowing, lifetime, trait, unsafe, pointer, reference, struct, and generics. Therefore it is necessary to understand these essential concepts first to design a generally applicable interface.
- *Exploit the Rust type system* We as programmers from mainstream languages like C++ and Java, it is likely to write Rust code with an old mindset. Rust borrows ideas from functional languages with a powerful type system such as Haskell, SML [28]. Therefore compared to the elements that Rust is famous for, such as the borrow checker, its type system is also worth a deep dive. In RMA, the `fence` method requires to be paired, `unlock` can only be invoked given `lock` has been called, etc. In our thesis, these constraints are currently addressed during runtime. However, with the Typestate pattern [6], these can be shifted to compile-time checks. To implement the Typestate pattern, intermediate familiarity with the type system is necessary. It makes use of the phantom type [56], a powerful type that only participates in compilation but not in runtime. To conclude, the type system is also a crucial element of Rust, by leveraging which a more robust and versatile API can be designed.

7 Conclusion and Future Work

We have implemented a prototype binding to bring the unsafe and C-like RMA interface from mpi-sys into the Rust world. We have leveraged the open-source project rsmpi for common MPI operations. Based on the MPI specification, we have designed the Rust interface of the RMA operations in the form of Rust traits. The three core components in RMA: window initialization, communication and synchronization calls, have been partially supported, so that programmers can write RMA programs without touching `unsafe` blocks. For window initialization, two kinds of windows were supported: created and allocated. For communication calls, besides having methods adhere to the original one from the specification, several methods with simplified parameters are designated to address the common use case such as passing a whole vector. For these interfaces, Rust generics have been used, and the conversion from the generic type to the MPI datatype is achieved by leveraging the rsmpi library. As a result, with our binding, programmers can write RMA code in the Rust idiomatic way, thereby greatly improving the code safety, maintainability and readability.

We also have conducted a series of benchmarks to show our library has a marginal performance cost. The Ping-Pong latency test has demonstrated that with our library, a C-equivalent latency can be achieved even under different network settings. Furthermore, we also implemented different versions of SOR and benchmarked them in different network settings. It shows that language and implementation strategies can make a noticeable impact on performance. The implementation strategies use different synchronization methods and data layouts. As a result, C and lock synchronization achieved the best performance, while Rust versions are acceptably slower.

During the work, we also gained Rust's development acquaintance. The overall experience is positive, except the ownership rules can be hindering. The workaround regarding the ownership also complicated the program slightly. But generally speaking, adopting Rust is recommended. Particularly in HPC, where performance is critical, Rust has strived for a balance between performance and development experience. The language feature and tooling have made a constructive impact on our work. In conclusion, the **answer** to our research question is: RMA can be well integrated into Rust, hence enabling programmers to use RMA safely and elegantly in Rust. The C-like performance can be preserved, but depends on program design. Furthermore, to the sub-problems:

1. Currently to program with RMA in Rust, the programmer has to write C-like unsafe code. Our library helps mitigate this issue by hiding unsafe blocks in the library.

2. The interface can be designed in an object-oriented way, using trait and generic in Rust. Besides, it should not only attempt to resemble the original interface in C but also new methods for common use cases.

3. Our evaluation has shown that Rust has achieved variable loss compared to its equivalence in C. A small program like Ping-Pong has a neglectable loss but an intermediate program like SOR can depend on program details.

The future work could be the complementary set of the library to the whole RMA. For window initialization, the current library only covers creation and allocation. Other types of windows, such as one allowing dynamic memory attachment, can be the next. Methods used for querying

window attributes can be included as well. The accumulate functions [40] and request-based can be implemented for communication calls. For synchronization calls, they are relatively simpler. The current implementation has hard-coded all `assert` arguments as 0, as it is the default one to work. It can be extended to allow passing arguments other than 0 so that the methods can be fully fledged. Last but not least, all the work is based on a fork of the open-source project rsmpi, we have contacted the source maintainers to seek their ideas. As hinted in the response, the Typestate pattern [6] can be incorporated to enhance compile-time checks, such as guaranteeing `fence` must be paired, thus further improving the safety of the interface.

Acknowledgement

Thanks to my supervisor Professor Rob van Nieuwpoort for his guidance and support throughout the project. Also, a shoutout to my daily supervisor Badia Liokouras for her meticulous reading of my work and feedback. Same to my second supervisor, Professor Kristian Rietveld. I would also like to acknowledge Dr.Stijn Heldens for his help on Rust. His advice has been an important factor in the motivation of the work. I genuinely appreciate the opportunity to study further into HPC and programming models. I enjoy the thoughtful design and the aim of providing handier tools.

Another big thanks to my parents at home, and my friends for their unwavering support. To my old friend Kuo Hsü Sen, this is also a reciprocal acknowledgment of his thesis.

References

[1] Apache Maven. *Apache Maven Project*. https://maven.apache.org/. Accessed: 2024-04-20. 2024.

[2] Henri Bal et al. "A medium-scale distributed system for computer science research: Infrastructure for the long term". In: *Computer* 49.5 (2016), pp. 54–63.

[3] Michael Blesel, Michael Kuhn, and Jannek Squar. "Heimdallr: Improving compile time correctness checking for message passing with Rust". In: *High Performance Computing: ISC High Performance Digital 2021 International Workshops, Frankfurt am Main, Germany, June 24–July 2, 2021, Revised Selected Papers 36*. Springer. 2021, pp. 199–211.

[4] Greg Buzzard et al. "An implementation of the Hamlyn sender-managed interface architecture". In: *ACM SIGOPS Operating Systems Review* 30.si (1996), pp. 245–259.

[5] *Cargo Guide*. https://doc.rust-lang.org/cargo/index.html. Accessed: 2024-04-20. 2024.

[6] Cliff Clifford. *The Typestate Pattern in Rust*. https://cliffle.com/blog/rust-typestate/. Accessed on July 15, 2024. 2019. URL: https://cliffle.com/blog/rust-typestate/.

[7] Clive Thompsonarchive. *Rust: World's Fastest Growing Programming Language*. https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/. Accessed: 2024-04-20. Feb. 2023.

[8] *Clone in std::clone - Rust*. Accessed: 2024-06-17. 2024. URL: https://doc.rust-lang.org/nightly/std/clone/trait.Clone.html.

[9] Conan Developers. *Conan 2.0: C and C++ Open Source Package Manager*. https://conan.io/. Accessed: 2024-06-17. 2024.

[10] Lisandro Dalcin and Yao-Lung L. Fang. "mpi4py: Status Update After 12 Years of Development". In: *Computing in Science Engineering* 23.4 (2021), pp. 47–54. DOI: 10.1109/MCSE.2021.3083216.

[11] Frederica Darema. "The spmd model: Past, present and future". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 8th European PVM/MPI Users' Group Meeting Santorini/Thera, Greece, September 23–26, 2001 Proceedings 8*. Springer. 2001, pp. 1–1.

[12] *Defining and Instantiating Structs*. Accessed: 2024-06-17. 2024. URL: https://doc.rust-lang.org/book/ch05-01-defining-structs.html.

[13] "Design and implementation of Java bindings in Open MPI". In: *Parallel Comput.* 59.C (Nov. 2016), pp. 1–20. ISSN: 0167-8191. DOI: 10.1016/j.parco.2016.08.004. URL: https://doi.org/10.1016/j.parco.2016.08.004.

[14] devreal. *Comment on "Is it possible for the target to service MPI_Get's in parallel?"*. GitHub issue comment. Accessed on July 11, 2024. GitHub, Nov. 2021. URL: https://github.com/open-mpi/ompi/issues/9508#issuecomment-952532348.

[15] DonFlat. *fust*. Accessed: 2024-07-06. 2024. URL: https://github.com/DonFlat/fust.

[16] DonFlat. *one_many_ping_pong*. Accessed: 2024-07-06. 2024. URL: https://github.com/DonFlat/one_many_ping_pong.

[17] DonFlat. *rsmpi*. Accessed: 2024-07-06. 2024. URL: https://github.com/DonFlat/rsmpi.

[18] DonFlat. *sor*. Accessed: 2024-07-06. 2024. URL: https://github.com/DonFlat/sor.

[19] *Drop in std::ops - Rust*. Accessed: 2024-06-17. 2024. URL: https://doc.rust-lang.org/std/ops/trait.Drop.html.

[20] *FFI - The Rustonomicon*. https://doc.rust-lang.org/nomicon/ffi.html. Accessed: 2024-06-17. 2024.

[21] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. "Enabling highly-scalable remote memory access programming with MPI-3 one sided". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–12.

[22] William Gropp et al. *Using advanced MPI: Modern features of the message-passing interface*. MIT Press, 2014.

[23] William D Gropp and Rajeev Thakur. "Revealing the performance of MPI RMA implementations". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2007, pp. 272–280.

[24] gRPC. *gRPC - A high performance, open-source universal RPC framework*. https://grpc.io/. Accessed: 2024-04-20. 2024.

[25] Torsten Hoefler. *What are the real differences between RDMA, InfiniBand, RMA, and PGAS?* https://htor.inf.ethz.ch/blog/index.php/2016/05/15/what-are-the-real-differences-between-rdma-infiniband-rma-and-pgas/. Accessed: 2024-04-20. May 2016.

[26] Torsten Hoefler et al. "Remote memory access programming in MPI-3". In: *ACM Transactions on Parallel Computing (TOPC)* 2.2 (2015), pp. 1–26.

[27] InfiniBand Trade Association. https://www.infinibandta.org/. Accessed: 2024-04-20. 2024.

[28] *Influences*. https://doc.rust-lang.org/stable/reference/influences.html. Accessed: 2024-06-17. 2024.

[29] Nikolay Ivanov. *Is Rust C++-fast? Benchmarking System Languages on Everyday Routines*. 2022. arXiv: 2209.09127 [cs.PL]. URL: https://arxiv.org/abs/2209.09127.

[30] Vivek Kashyap. *IP over InfiniBand (IPoIB) Architecture*. Request for Comments 4392. Accessed: 2024-06-17. 2006. URL: https://www.rfc-editor.org/rfc/rfc4392.

[31] Michael Kerrisk. *malloc(3) - Linux manual page*. Accessed: 2024-06-17. man7.org. 2023. URL: https://man7.org/linux/man-pages/man3/malloc.3.html.

[32] Amit Levy et al. “Ownership is theft: experiences building an embedded OS in rust”. In: *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. PLOS ’15. Monterey, California: Association for Computing Machinery, 2015, pp. 21–26. ISBN: 9781450339421. DOI: 10.1145/2818302.2818306. URL: https://doi-org.ezproxy.leidenuniv.nl/10.1145/2818302.2818306.

[33] linux-rdma. *RDMA core userspace libraries and daemons*. Accessed: 2024-06-17. 2024. URL: https://github.com/linux-rdma/rdma-core.

[34] Shaonan Ma et al. “A Survey of Storage Systems in the RDMA Era”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 4395–4409. DOI: 10.1109/TPDS.2022.3188656.

[35] Jason Maassen, Thilo Kielmann, and Henri E Bal. “GMI: Flexible and efficient group method invocation for parallel programming”. In: *Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*. Citeseer. 2002, pp. 1–6.

[36] Jason Maassen et al. “Efficient Java RMI for parallel programming”. In: *ACM Transactions on Programming Languages and Systems* 23.6 (2001), pp. 747–775.

[37] *ManuallyDrop in std::mem - Rust*. Accessed: 2024-06-18. 2024. URL: https://doc.rust-lang.org/std/mem/struct.ManuallyDrop.html.

[38] Rick Merritt. *What Is RDMA and RoCE and How Did They Fuel Mellanox’s Fast Networks?* Accessed: 2024-06-17. 2020. URL: https://blogs.nvidia.com/blog/what-is-rdma/.

[39] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. Technical Report. Message Passing Interface Forum, Sept. 2012. URL: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[40] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. Nov. 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[41] André Miranda and João Pimentel. “On the use of package managers by the C++ open-source community”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC ’18. Pau, France: Association for Computing Machinery, 2018, pp. 1483–1491. ISBN: 9781450351911. DOI: 10.1145/3167132.3167290. URL: https://doi.org/10.1145/3167132.3167290.

[42] Sparsh Mittal. “A study of successive over-relaxation method parallelisation over modern HPC languages”. In: *International journal of high performance computing and networking* 7.4 (2014), pp. 292–298.

[43] *mpi-sys - Rust Package*. https://crates.io/crates/mpi-sys. Accessed: 2024-06-17. 2024.

[44] *MPI: A Message-Passing Interface Standard Version 4.0*. Chapter 3, Point-to-Point Communication, p.2. MPI Forum. 2020. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[45] *MPI: A Message-Passing Interface Standard Version 4.1*. Chapter 12, One-sided Communication. MPI Forum. 2021. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[46] *MPICH: High Performance and Widely Portable MPI Implementation*. https://www.mpich.org/. Accessed: 2024-04-20. 2024.

[47] npm, Inc. *npm: Node Package Manager*. https://www.npmjs.com/. Accessed: 2024-04-20. 2024.

[48] NVIDIA Corporation. *Title of the Mellanox Content*. https://www.nvidia.com/en-us/mellanox/. Originally published by Mellanox Technologies, now part of NVIDIA Corporation. Accessed: 2024-04-20. 2024.

[49] Open MPI. *mpicc - Open MPI compiler wrapper*. https://www.open-mpi.org/doc/v4.0/man1/mpicc.1.php. Accessed: 2024-04-20. 2024.

[50] Open MPI. *mpirun - Open MPI's orterun, mpirun and mpiexec wrapper command*. https://www.open-mpi.org/doc/current/man1/mpirun.1.php. Accessed: 2024-04-20. 2024.

[51] *Open MPI: Open Source High Performance Computing*. https://www.open-mpi.org/. Accessed: 2024-04-20. 2024.

[52] OpenUCX Project. *OpenUCX: Unified Communication X*. Accessed: 2024-06-17. 2024. URL: https://openucx.org/.

[53] *RAII*. https://doc.rust-lang.org/rust-by-example/scope/raii.html. Accessed: 2024-06-17. 2024.

[54] RoCE Initiative. https://www.roceinitiative.org/. Accessed: 2024-04-20. 2024.

[55] *rsmpi: MPI bindings for Rust*. https://github.com/rsmpi/rsmpi. Accessed: 2024-06-17. 2024.

[56] *Rust By Example: Phantom Type Parameters*. Accessed: 2024-07-21. 2024. URL: https://doc.rust-lang.org/rust-by-example/generics/phantom.html.

[57] *Rust Programming Language*. https://www.rust-lang.org/. Accessed: 2024-04-20. 2024.

[58] *rust-bindgen: Automatically generates Rust FFI bindings to C (and some C++) libraries*. https://github.com/rust-lang/rust-bindgen. Accessed: 2024-06-17. 2024.

[59] Let's Get Rusty. *How to fight Rust's borrow checker... and win.* Accessed: 2024-07-04. 2023. URL: https://youtu.be/Pg07HQJ0tvI?si=FeIGmhCKMv3-YCZm.

[60] *Syntax - The Rust Programming Language*. Accessed: 2024-06-18. 2024. URL: https://doc.rust-lang.org/book/ch10-01-syntax.html.

[61] Konstantin Taranov, Fabian Fischer, and Torsten Hoefler. "Efficient RDMA Communication Protocols". In: *arXiv preprint arXiv:2212.09134* (2022).

[62] The Rust Programming Language Authors. *Understanding Ownership*. The Rust Programming Language Book. 2024. URL: https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

[63] TOML. *TOML: Tom's Obvious, Minimal Language*. https://toml.io/en/. Accessed: 2024-04-20. 2024.

[64] *Traits - The Rust Programming Language*. Accessed: 2024-06-18. 2024. URL: https://doc.rust-lang.org/book/ch10-02-traits.html.

[65] Jake Tronge and Howard Pritchard. "Embedding Rust within Open MPI". In: *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, pp. 438–447.

[66] Ohio State University. *MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE*. Accessed: 2024-06-17. 2023. URL: `https://mvapich.cse.ohio-state.edu/`.

[67] *unsafe - Rust*. `https://doc.rust-lang.org/std/keyword.unsafe.html`. Accessed: 2024-06-17. 2024.

[68] *Validating References with Lifetimes*. Accessed: 2024-06-17. 2024. URL: `https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html`.

[69] *Vec::from_raw_parts in std::vec - Rust*. Accessed: 2024-06-18. 2024. URL: `https://doc.rust-lang.org/std/vec/struct.Vec.html#method.from_raw_parts`.

[70] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple linux utility for resource management". In: *Workshop on job scheduling strategies for parallel processing*. Springer. 2003, pp. 44–60.