heimdallr: Improving Compile Time Correctness Checking for Message Passing with Rust

Michael Blesel[1(✉)], Michael Kuhn[1], and Jannek Squar[2]

[1] Otto von Guericke University Magdeburg, Magdeburg, Germany
{michael.blesel,michael.kuhn}@ovgu.de
[2] Universität Hamburg, Hamburg, Germany
squar@informatik.uni-hamburg.de

Abstract. Message passing is the foremost parallelization method used in high-performance computing (HPC). Parallel programming in general and especially message passing strongly increase the complexity and susceptibility to errors of programs. The de-facto standard technologies used to realize message passing applications in HPC are MPI with C/C++ or Fortran code. These technologies offer high performance but do not come with many compile-time correctness guarantees and are quite error-prone. This paper presents our work on a message passing library implemented in Rust that focuses on compile-time correctness checks. In our design, we apply Rust's memory and concurrency safety features to a message passing context and show how common error classes from MPI applications can be avoided with this approach.

Problems with the type safety of transmitted messages can be mitigated through the use of generic programming concepts at compile time and completely detected during runtime using data serialization methods. Our library is able to use Rust's memory safety features to achieve data buffer safety for non-blocking message passing operations at compile time.

A performance comparison between our proof of concept implementation and MPI is included to evaluate the practicality of our approach. While the performance of MPI could not be beaten, the results still are promising. Moreover, we are able to achieve clear improvements in the aspects of correctness and usability.

Keywords: Message passing · Compile-time checks · Rust · MPI

1 Introduction

Parallelization has become an essential programming technique over the last decades for applications to utilize the full resources of a computing system. In HPC, parallelization is an absolute requirement to run applications on distributed memory systems. The standard technologies used in this context today

H. Jagode et al. (Eds.): ISC High Performance 2021 Workshops, LNCS 12761, pp. 199–211, 2021.
https://doi.org/10.1007/978-3-030-90539-2_13

are message passing via MPI for inter-node parallelization in conjunction with frameworks like OpenMP or manual multi-threading for shared memory intra-node parallelization. Most of these tools are based on C/C++ and Fortran since these languages have traditionally yielded the best performance results for HPC applications and therefore make up the majority of existing HPC codebases.

Parallelization provides significant performance increases and more importantly scalability to applications but it does not come without drawbacks. The code complexity often increases heavily when parallelization is introduced into a program [11]. Additionally new classes of errors such as data races, deadlocks and non-determinism emerge from parallel code [3]. In general, modern compilers have become very good at detecting errors and providing helpful error and warning messages to the user but in respect to parallelization errors they are often still lacking. For MPI applications, not many parallelization errors are caught at compile time. Some static analysis tools such as MPI-Checker [5] exist but often manual debugging by the programmer is required. Better compile time correctness checks for message passing applications are therefore desirable.

Many of these problems can be traced back to the programming languages that are used in HPC applications. Both, C/C++ and Fortran have not been designed with parallel programming in mind. Intrinsic support for parallel programming features and the existing solutions today were either added over time to their specifications or are provided by external libraries.

Rust is a modern system programming language that focuses on memory and concurrency safety with strong compile time correctness checks [12]. One of Rust's unique features is its memory ownership concept which ensures that all data has exactly one owner at all times during a program's runtime and thereby allows the compiler to guarantee the absence of errors like data-races at compile time. This paper explores how Rust's safety mechanisms can be applied to the design of a message passing library that provides stronger compile time correctness checks than existing solutions like MPI.

2 Motivation

The purpose of the work presented in this paper is to show how a message passing library that strongly focuses on compile time correctness checks and usability can be designed in Rust. This section argues why these attributes are desirable and might even be more important than raw performance to users.

The authors of [1] have conducted a study about programming languages for data-intensive HPC applications. They combined an analysis of over one hundred papers from the HPC domain and a survey of HPC experts and concluded that the most desired and important features of programming languages in HPC are usability, performance and portability.

As it can be seen in Fig. 1, usability seems to be the most desired feature for many users. This makes sense when taking into account that a large percentage of the userbase of HPC systems are not necessarily parallel programming experts but rather scientist from other domains. This user group needs to develop scalable parallel applications for supercomputers to facilitate their domain specific

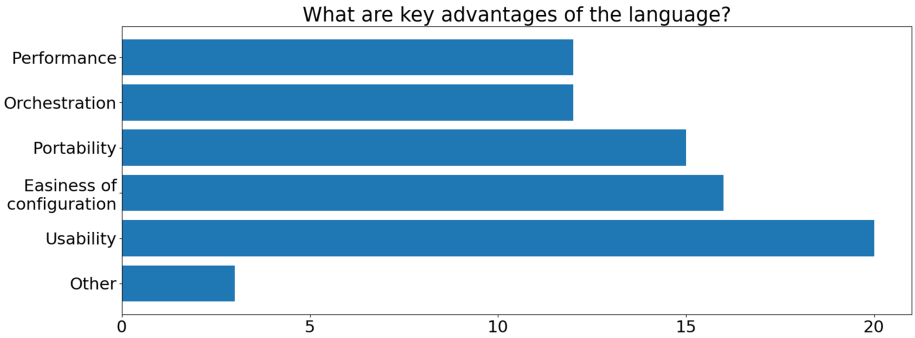

Fig. 1. The most important features of HPC programming languages (based on [1])

research. It is therefore important to provide software solutions that make this process as easy as possible.

Providing better compile time correctness checks for parallel applications can greatly improve the usability of a message passing library by reducing the need for manual debugging, which can be tedious for the users. As the next section will show, MPI and C/C++ have some significant problems in this regard.

Even though Rust is not yet as commonly used in HPC as MPI with C/C++ or Fortran, we chose it due to its fundamental safety and correctness concepts that it was designed with. The following sections will show that they fit very well to the context of message passing. Furthermore, we believe it is important to look into newer technologies that might play a part in the future of HPC. Rust seems like a good candidate since it has seen more wide spread use as a more convenient and safer alternative to C/C++ in the software industry over the last years with support from large companies such as Google, Amazon and Intel [8]. Also when looking at the next generation of HPC developers many will not be as familiar with older languages like C anymore and be more accustomed to modern languages with their design concepts and comfort features.

3 Correctness Problems with MPI

This section highlights some common erroneous coding patterns that can occur in MPI applications and which are currently not caught at compile time without the use of external static analysis tools.

Many common errors in MPI code can be traced back to the use of raw, untyped memory buffers via C's void pointers. This C-style way of working with raw memory addresses yields good performance and gives great control to the programmer, but it also harbors a lot of dangers and hinders the compiler in detecting data type related errors as the following example shows.

3.1 Type Safety Errors in MPI

MPI functions require the programmer to manually specify the data type of the passed data buffer. This not only introduces a source of errors but can also be very inconvenient for the user. Listing 1.1 presents an incorrect MPI code segment where the true type of the sent data buffer does not match the given `MPI_Datatype` argument of the send and receive functions. The true type of the data buffer is `double` but the MPI functions are given the `MPI_FLOAT` data type argument. This is clearly an error by the programmer but it is not detected by the compiler. What makes this example even more problematic is that the given code will run without a runtime crash and cause unexpected program results. This makes it a hard bug to find in a real application with a large codebase.

```
1  double *buf = malloc(sizeof(double) * BUF_SIZE);
2  [...]
3  if (rank == 0) {
4      for (int i = 0; i < BUF_SIZE; ++i)
5          buf[i] = 42.0;
6      MPI_Send(buf, BUF_SIZE, MPI_FLOAT, 1, 0,
           ↪ MPI_COMM_WORLD);
7  }
8  else if (rank == 1) {
9      MPI_Recv(buf, BUF_SIZE, MPI_FLOAT, 0, 0,
           ↪ MPI_COMM_WORLD,
10                MPI_STATUS_IGNORE);
11 }
```

Listing 1.1. Faulty MPI code that states the wrong `MPI_Datatype` argument

Errors like this can easily happen when the data type of a buffer variable has to be changed at some point during development without the programmer remembering that this also implies modifying all MPI function calls that use this buffer. This example highlights another problematic aspect of programming with MPI. Due to the quite low abstraction level of MPI operations it is very inflexible regarding changes in the code. Even a simple change like switching the type of a variable can require changes to large parts of the whole program.

3.2 Memory Safety Concerns with Non-blocking Communication

Listing 1.2 shows an example use case of MPI's non-blocking communication operations. When using non-blocking operations like `MPI_Isend` from the example the function immediately returns to the caller and the message passing operation is processed in the background. This leaves the data buffer that is being sent in an unsafe state where no other part of the application should access it before making sure that it is safe to be used again. In the given example the process with rank 0 does not adhere to this and immediately after calling `MPI_Isend` it

```
1  if(rank == 0) {
2      MPI_Isend(buf, BUF_SIZE, MPI_DOUBLE, 1, 0,
3                MPI_COMM_WORLD, &req);
4      for(int i = 0; i < BUF_SIZE; ++i)
5          buf[i] = 42.0;
6  }
7  else if(rank == 1) {
8      MPI_Recv(buf, BUF_SIZE, MPI_DOUBLE, 0, 0,
9               MPI_COMM_WORLD, &status);
10 }
```

Listing 1.2. Faulty non-blocking MPI code that writes to an unsafe buffer

starts to modify the buffer. This makes the outcome of the shown code non-deterministic. This error is even harder to detect compared to the last example because the produced results may differ from run to run and the application might even yield the expected results sometimes.

The given code does not conform to the MPI specification, which states that the safety of a buffer, which has been used with non-blocking communication, needs to be verified by calling `MPI_Wait` or `MPI_Test` before accessing it again. However, MPI has no way of enforcing this rule in actual code. There will be no compile time warnings or errors if the sequence is not correct, which makes bugs caused by incorrect usage of non-blocking communication hard to track down in more complicated applications.

4 heimdallr: A Proof of Concept Message Passing Library Implementation in Rust

In this section we present our work on a proof of concept message passing library implementation in Rust called heimdallr[1]. The design of the library is focused on strong compile time correctness checks and good usability with clear semantics for all message passing operations. We explain how Rust's safety and correctness mechanics were applied to the design and implementation of basic message passing operations to achieve these goals.

4.1 Type Safety Through Generics and Message Serialization

As discussed in Sect. 3.1, MPI requires the user to manually specify the data type of given data buffers, which can lead to errors. This problem can be solved quite easily in more modern languages that provide stronger support for generic programming than C. Listing 1.3 presents the signatures of heimdallr's blocking,

[1] https://github.com/parcio/heimdallr.

```
pub fn send<T>(&self, data: &T, dest: u32, id: u32)
    -> std::io::Result<()>

pub fn receive<T>(&self, source: u32, id: u32)
    -> std::io::Result<T>
```

Listing 1.3. Function signatures of heimdallr's blocking, synchronous send and receive operations

synchronous send and receive functions. They work with Rust's generic types for their data buffer arguments, which already removes the burden of having to state the correct data type from the user and leaves it to the compiler. This may seem like a small and obvious change but it eliminates a lot of potential errors, makes the message passing code more flexible regarding data type changes and makes the function signatures more concise.

Leaving the local data buffer type deduction up to the compiler however only solves half of the problem of type safety for message passing. The other aspect is to make sure that all processes that are participating in a message exchange agree about the type of the message's data. For reasons that are discussed in Sect. 6.3, this problem is hard to solve at compile time. Therefore, in its current state, heimdallr is only able to detect errors of this kind at runtime. To make sure that the data type of a received message is interpreted correctly, heimdallr uses serialization. This adds some computational overhead to the message passing procedure when compared to working on raw byte streams, but it can ensure that the message data type cannot be misinterpreted on the receiving side. heimdallr makes use of the established Rust serialization crate Serde [9] and uses the bincode [7] serialization protocol. This also allows users to easily send custom made types if they implement Serde's *Serialize* and *Deserialize* traits. These traits can be automatically generated by Rust if the user simply adds a `#[derive(Serialize, Deserialize)]` statement to the declaration of a custom type. This feature is a nice step up in usability compared to the steps that are needed in MPI to send user defined types.

4.2 Ensuring Buffer Safety for Non-blocking Communication

One of Rust's unique selling points are its compile time guarantees for memory safety without needing a garbage collector. The central feature that allows the compiler to achieve this is called *Ownership*. All data that is allocated in a Rust program has to have exactly one owner. If the owner variable goes out of scope the memory is automatically deallocated. This concept can be applied very well to the previously described problems with unsafe data buffers for MPI's non-blocking operations. Listing 1.4 presents the signatures of heimdallr's non-blocking send and receive functions. They are very similar to the blocking versions from the last section but contain two significant changes. Instead of a

reference to the data buffer the non-blocking send function takes ownership of the buffer from the function caller. This means that after the function call has returned the caller no longer has access to the buffer. Modifying it like in the MPI example from Listing 1.2 would lead to a compilation error due to accessing data whose ownership has been moved into the send function. This protects the buffer while the message passing operation is processed in the background. For retrieving ownership of the data buffer the non-blocking send function returns a data handle type, which provides member functions comparable to `MPI_Wait`.

```
pub fn send_nb<T>(&self, data: T, dest: u32, id: u32)
    -> std::io::Result<NbDataHandle<std::io::Result<T>>>

pub fn receive_nb<T>(&self, source: u32, id: u32)
    -> std::io::Result<NbDataHandle<std::io::Result<T>>>
```

Listing 1.4. Function signatures of heimdallr's non-blocking send and receive operations

As we can see, the workflow of using non-blocking communication in heimdallr is more or less the same as in MPI but Rust's ownership concept allows the library to actually enforce the rules of having to verify the safety status of data buffers before being able to access them again. This approach works very well from the perspective of safety but it does have some drawbacks. Data ownership can only be moved for entire objects. Many HPC applications contain a core data structure such as a matrix of which only certain parts need to be communicated via message passing. Using heimdallr's non-blocking operations in such a scenario would mean that the whole data structure becomes inaccessible until the ownership has been retrieved. This is not acceptable for algorithms that need to work on other parts of the data while the non-blocking message passing takes place in the background. Therefore, using heimdallr might require some restructuring of distributed data structures where the parts of the data that are used in message passing are isolated as separate objects. This might impact the performance negatively for reasons such as worsened cache-locality.

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

Listing 1.5. Simplified example of a use pattern of heimdallr's non-blocking communication that produces compilation errors

Listing 1.5 showcases a second problem with the usage of heimdallr's non-blocking operations. In the first match statement of the example in line 2 the process 0 uses a non-blocking send operation to transmit a buffer to process 1. In the second match statement in line 7 the ownership of the buffer is requested back from process 0 and both processes try to access it by printing the buffer's contents. Getting the ownership of the data back works well for process 0 but this example does not compile because of process 1's access on the buffer variable in line 10. Rust's borrow-checking algorithm is not able to correctly analyse the control flow of this example program and instead detects that process 1 might not have ownership of the buffer variable because it was moved in the previous match statement by process 0.

This is problematic because the code pattern given in Listing 1.5, where a non-blocking operation that is only executed conditionally based on the process ID and then later concluded in a similar conditional block, appears frequently in message passing applications. Up until this point we could not find a generalizable solution for this problem. Depending on the context most often the code can be restructured in a way that the compiler will accept it in the end, but doing so will add more complexity to the code and and make it less readable. We hope that future improvements on the Rust borrow checker algorithm like those presented in [6] will fix this problem but currently workarounds in the code need to be used to implement these kind of message passing code patterns.

5 Related Work

When looking at the traditional approach of using MPI with C/C++ or Fortran, static analysis methods are the most promising for detecting errors in the code or communication scheme of parallel applications. The MPI-Checker [5] project is a static analysis tool built upon LLVM's C/C++ compiler frontend Clang. It is able to detect some process local errors such as type mismatches between the true data type of a buffer and the stated `MPI_Datatype` and incorrect buffer referencing where the passed void pointer does not point to valid data. Furthermore, it is able to detect some common errors in the communication scheme of applications. This encompasses deadlock detection and missing communication partners for point-to-point message passing. Additionally it can analyze and detect some errors with MPI's non-blocking communication such as missing `MPI_Wait` calls. As this paper has shown, many of these error classes are already caught automatically at compile time by our Rust implementation.

One reason why MPI applications are so error-prone is that end users have to use the low-level MPI operations to parallelize their applications. From a usability standpoint, a higher abstraction level as well as easy-to-use parallel data structures and algorithms would be preferable. The desire for such solutions is apparent when looking at the popularity of OpenMP, which provides such features for a multi-threading context. The Chapel [4] project is one example for such a solution. Chapel is a special purpose language that is designed for scientific computing in an HPC context. It provides automatically distributed

data structures and algorithms to the user with a syntax that is comparable to writing sequential code. Most of the parallelization logic is hidden from the user, which avoids possible parallelization errors. This approach has great usability advantages but being its own special purpose programming language, the barrier of entry is higher compared to using general purpose languages like C or Rust.

MPI bindings for Rust exist in the form of *rsmpi* [10], which supports a subset of MPI's operations containing all two-sided communication functions and most collective operations. The syntax of the message passing operations is in a more Rust-like style that looks quite different from traditional MPI code. Improvements like automatic data type deduction for buffers and some guarantees for better handling of `MPI_Request`s for non-blocking communication are present, where the latter is handled quite different than in our implementation. For users who are not familiar with MPI code, the rsmpi syntax might prove a bit challenging.

For our work, we decided against an approach that is reliant on an existing MPI implementation. Building our Rust message passing library from scratch allowed for a more concise interface and clearer semantics of message passing functions. Furthermore, we were able to experiment with some ideas that do not exist in MPI, such as shared distributed data structures and a central daemon process that can participate in the message passing at runtime.

6 Evaluation

In this section we perform an evaluation on our work with heimdallr. In Sects. 6.1 and 6.2 the performance of heimdallr is compared to equivalent MPI applications. All measurements for the benchmarks in this sections where done on identical computing nodes with the following specs:

- 4x AMD Opteron Processor 6344 (48 cores total)
- 128 GB RAM
- 40 GBit/s InfiniBand network (using TCP over InfiniBand)

For a fair comparison, MPI and heimdallr both used TCP over InfiniBand, because heimdallr currently only works with TCP. All results presented in this section are averaged over three separate benchmark runs.

6.1 Performance Comparison on a Realistic Application

This benchmark uses a realistic scientific application called partdiff that was developed for teaching purposes at the University of Hamburg's Scientific Computing group. It solves a partial differential equation by continuously iterating over a distributed matrix with a stencil operator. This type of application can be categorized as a *structured grid* approach and belongs to the so-called *seven dwarfs of HPC* [2], which makes it a good benchmark candidate to compare both message passing implementations. The original partdiff is written in C and

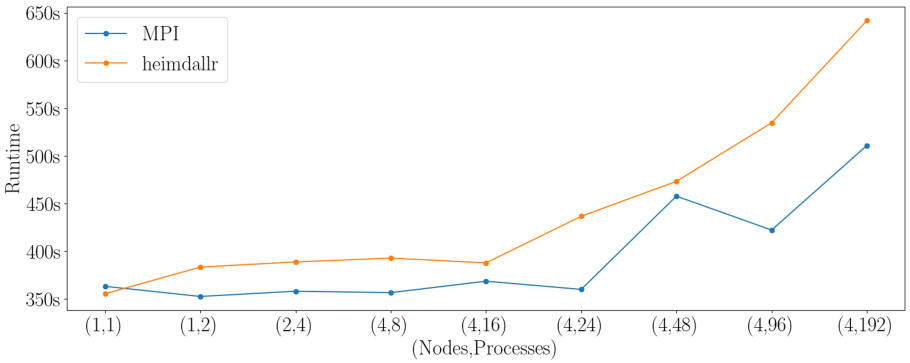

Fig. 2. Weak scaling benchmark for MPI and heimdallr versions of the partdiff application with increasing (#Nodes,#Total processes)

parallelized with MPI. We implemented an equivalent Rust version that uses our heimdallr library for the message passing aspects.

Figure 2 presents a runtime comparison of the two partdiff versions for different configurations of computing nodes and process counts. The weak scaling behaviour of both applications was examined. The results show that from a performance standpoint heimdallr is beaten by MPI. However, this was expected since the performance of MPI implementations has been optimized for decades and for our proof of concept implementation pure performance was not the main focus. The runtime differences can mainly be attributed to two factors. Firstly, the serialization of message data will always produce more overhead than MPI's approach of sending raw byte streams that are written directly into memory at the receiver's end. Secondly, since we only had four computing nodes available the later data points spawn a lot of processes on the same nodes. This leads to a lot of intra-node communication, for which MPI features optimizations that are not yet implemented in heimdallr.

Overall, for most data points heimdallr does not perform significantly worse than MPI and we believe that the provided benefits from better compile time correctness checking and its usability advantages make up for the performance differences.

6.2 Micro Benchmarks for Message Passing Operations

While the previous benchmark shows heimdallr's comparable performance in the context of a realistic application, it does not expose the direct performance difference between the individual message passing operations. Table 1 refers to a micro benchmark that only measures message passing performance without any other unrelated computations. It exchanges 1,000 messages in total between two processes on different computing nodes. Multiple measurements with increasing message sizes were done for the blocking send and receive operations of MPI and heimdallr. The results show that heimdallr's message passing operations have a

Table 1. Performance comparison of heimdallr and MPI for 1000 sent messages between two computing nodes with increasing data size

	$1000 \times 100\,\text{KB}$	$1000 \times 1\,\text{KB}$	$1000 \times 10\,\text{KB}$	$1000 \times 100\,\text{KB}$	$1000 \times 1\,\text{MB}$
MPI	0.0257 s	0.0284 s	0.0456 s	0.2116 s	1.2426 s
heimdallr	0.1661 s	0.1822 s	0.3650 s	2.1253 s	19.303 s
Serialization time	0.0012 s	0.0119 s	0.1192 s	1.1795 s	12.056 s
Serialization time share	0.7%	6.5%	32.7%	55.5%	62.5%

significant computational overhead compared to their MPI equivalents. A significant part of this is due to the serialization of the message data that is performed by heimdallr. In the second table row the serialization and deserialization time of the data buffers has been isolated. It turns out that the time spent in the serialization procedure is above average compared to the message size. This is a trade-off in heimdallr between safety/usability and performance. Serialization allows for transmitting nearly any user defined type and also gives runtime error checks for type correctness of message passing operations but will always require additional computations for the transformation of the transmitted data.

6.3 Limitations of Compile Time Correctness Checks for Message Passing Applications

All compile time correctness checks of heimdallr have in common that they only consider errors that are local to one process of the parallel heimdallr application. Parallelization errors such as deadlocks caused by blocking operations or missing communication partners for send/receive operations are not caught by the compiler. This is due to the fact that the compiler does not know about the broader context of the message passing code. Without knowledge of the actual parallel execution configuration in which heimdallr library calls will be communicating with each other, there is no way to detect these types of errors at compile time.

At this point, there are two possible solutions to include such correctness checks. Firstly, an external static analysis tool like MPI-Checker [5] could be developed for heimdallr. Secondly, direct modifications to the Rust compiler could be made to make it aware of the logic and semantics of message passing applications. The first solutions seems more feasible at the current time. With a static analysis approach that is aware of the parallel context of SPMD applications and the semantics of the used message passing library more thorough correctness checks for process interactions could be deployed.

7 Conclusion and Future Work

This paper shows that the Rust programming language is very applicable for message passing applications. Our heimdallr implementation is able to provide a

much safer environment for parallel programming by leveraging Rust's compile time correctness guarantees. Compared to MPI, the performance of heimdallr is lacking behind, but not by unacceptable margins. In addition, our implementation is not production ready yet but a proof of concept to demonstrate the benefits of concepts such as *Ownership* for message passing.

Even though improvements in memory and type safety were made, compile time correctness checks for the validity of an application's communication scheme are still missing and would require the help of external static analysis tools or improved compiler support. We plan to integrate static analysis checks as mentioned in Sect. 6.3 into the build process of heimdallr applications.

This work shows that improvements in correctness checking of parallel message passing applications can be achieved without the need for direct compiler modifications but a more complete solution would require awareness by the compiler about the context of message passing. Rust's correctness checks for multithreaded code are able to detect errors such as data-races because the compiler is aware of the semantics of such concurrent programs. Integrating comparable correctness checking procedures for SPMD parallelization schemes directly into the Rust compiler might yield even stronger correctness guarantees for message passing applications and presents itself to be promising follow-up research on this topic.

Since most of the existing HPC infrastructure today is built upon MPI and C/C++ it would be interesting to further explore whether Rust's safety concepts such as ownership could be applied there retroactively via the use of static analysis and source-to-source translation methods.

It also seems feasible that at least some of the correctness features of heimdallr could be implemented in modern C++ bindings for MPI by using templates for the type safety aspects and smart pointer types such as `unique_ptr` for data buffer protection in the context of non-blocking communication.

References

1. Amaral, V., et al.: Programming languages for data-intensive HPC applications: a systematic mapping study. Parallel Comput. **91**, 102584 (2020). https://doi.org/10.1016/j.parco.2019.102584
2. Asanović, K., et al.: The landscape of parallel computing research: a view from Berkeley. Technical report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006). http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html
3. Ba, T.N., Arora, R.: Towards developing a repository of logical errors observed in parallel code for teaching code correctness. In: EduHPC@SC, pp. 69–77. IEEE (2018)
4. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007)
5. Droste, A., Kuhn, M., Ludwig, T.: MPI-checker: static analysis for MPI. In: LLVM@SC, pp. 3:1–3:10. ACM (2015)

6. Matsakis, N.D.: An alias-based formulation of the borrow checker. http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/ (2018). Accessed on Mar 2021
7. bincode org: Bincode. https://github.com/bincode-org/bincode (2021). Accessed on Mar 2021
8. R, B.: "Rust is the future of systems programming, C is the new assembly": intel principal engineer, Josh Triplett. https://hub.packtpub.com/rust-is-the-future-of-systems-programming-c-is-the-new-assembly-intel-principal-engineer-josh-triplett/ (2019). Accessed on Mar 2021
9. serde rs: Serde. https://serde.rs/ (2021). Accessed on Mar 2021
10. rsmpi: rsmpi. https://github.com/rsmpi/rsmpi (2021). Accessed on Mar 2021
11. Vanderwiel, S.P., Nathanson, D., Lilja, D.J.: Complexity and performance in parallel programming languages. In: HIPS, p. 3. IEEE Computer Society (1997)
12. Yu, Z., Song, L., Zhang, Y.: Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software. ArXiv: CoRR abs/1902.01906 (2019)