Modernization and Optimization of MPI Codes

Zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
Genehmigte Dissertation von Tim Jammer
Tag der Einreichung: 10. September 2024, Tag der Prüfung: 20. Dezember 2024

1. Gutachten: Prof. Dr. Christian Bischof
2. Gutachten: Prof. Dr.Thomas Ludwig
3. Gutachten: Dr. Emmanuelle Saillard
Darmstadt, Technische Universität Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Scientific Computing

Modernization and Optimization of MPI Codes

Accepted doctoral thesis by Tim Jammer

Date of submission: 10. September 2024
Date of thesis defense: 20. Dezember 2024

Darmstadt, Technische Universität Darmstadt

Erklärungen laut Promotionsordnung

§ 8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§ 8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§ 9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§ 9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 10. September 2024

Tim Jammer

Abstract

MPI has become the de facto standard for distributed memory computing since its inception in 1994. While the MPI standard has evolved to include new technologies like RDMA, many applications still rely on the original set of MPI operations. This thesis initially investigates the current usage of MPI. We note that developers underutilize modern MPI features, as their implementations often are not optimized. On the other hand, as many users rely on the ”old” MPI features, the MPI implementation developers have no incentive to optimize implementations for the new features. As a consequence, there is no incentive for the MPI users to learn the new features, creating a vicious cycle.

To break this cycle, this thesis explores three main approaches: (1) facilitating correctness checking tool support, (2) modernizing MPI codes with compiler based approaches, and (3) exploiting compiler knowledge to further optimize the implementation of modern MPI features.

In order to facilitate the development and improvement of tools aiding with MPI development, this thesis introduces the correctness benchmark MPI-BugBench, as a standardized benchmark to evaluate the real world applicability of such tools. Further we show that compiler-based automatic modernization methods can encourage early adoption of new MPI features with minimal programmer effort, for example partitioned operations. Lastly, compiler knowledge can be utilized in order to further optimize the performance of MPI implementations, for example in persistent operations. The use of compiler knowledge, in particular, enables modernization of existing MPI codes without the need of application developers to rewrite existing MPI codes.

Zusammenfassung

MPI ist seit seiner Einführung im Jahr 1994 zum De-facto-Standard für parallele Computersysteme mit verteiltem Speicher geworden. Obwohl sich der MPI-Standard weiterentwickelt hat, um neue Technologien wie RDMA zu integrieren, verlassen sich viele Anwendungen immer noch auf die ursprünglichen MPI-Operationen. Diese Arbeit untersucht zunächst die aktuelle Nutzung von MPI. Wir stellen fest, dass moderne MPI-Funktionen von Entwicklern selten genutzt werden, da deren Implementierungen oft nicht optimiert sind. Andererseits haben die Entwickler der MPI-Implementierungen keinen Anreiz, die neuen Funktionen zu optimieren, da viele Nutzer die ”alten” MPI-Funktionen benutzen. Dies führt dann dazu, dass die Nutzer keinen Anreiz haben, die neuen Funktionen zu erlernen, wodurch ein Teufelskreis entsteht.

Um diesen Teufelskreis zu durchbrechen, untersucht diese Arbeit drei Hauptansätze: (1) Unterstützung der Entwicklung von Korrektheitsprüfungs-Werkzeugen, (2) Modernisierung von MPI-Codes mit compilerbasierten Ansätzen und (3) Nutzung von Compilerwissen zur weiteren Optimierung der Implementierung moderner MPI-Operationen.

Um die Entwicklung und Verbesserung von Werkzeugen zur Unterstützung der MPI-Entwicklung zu erleichtern, führt diese Arbeit den Korrektheits-Benchmark MPI-BugBench als standardisierten Benchmark ein, um die Anwendbarkeit solcher Tools in der realen Welt zu bewerten. Weiterhin zeigen wir, dass compilerbasierte automatische Modernisierungsmethoden die frühzeitige Einführung neuer MPI-Funktionen mit minimalem Programmieraufwand fördern können, zum Beispiel partitionierte Operationen. Schließlich kann Compilerwissen genutzt werden, um die Leistung von MPI-Implementierungen weiter zu optimieren, beispielsweise bei persistenten Operationen. Die Nutzung von Compilerwissen ermöglicht insbesondere die Modernisierung bestehender MPI-Codes, ohne dass Anwendungsentwickler vorhandene MPI-Codes neu schreiben müssen.

Publications

The following peer-reviewed publications are part of this cumulative dissertation and can be found in Chapter 7 to Chapter 16:

- Alexander Hück, Tim Jammer, Joachim Jenke, and Christian Bischof. "Investigating the Usage of MPI at Argument-Granularity in HPC Codes". In: *Proceedings of the 30th European MPI Users' Group Meeting*. EUROMPI '23. Bristol, UK: ACM, Sept. 2023. DOI: 10.1145/3615318.3615322
  This publication is included in Chapter 7.

- Jan-Patrick Lehr, Tim Jammer, and Christian Bischof. "MPI-CorrBench: Towards an MPI Correctness Benchmark Suite". In: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. virtual: ACM, June 2021. DOI: 10.1145/3431379.3460652
  This publication is included in Chapter 8.

- Tim Jammer, Alexander Hück, Jan-Patrick Lehr, Joachim Protze, Simon Schwitanski, and Christian Bischof. "Towards a Hybrid MPI Correctness Benchmark Suite". In: *EuroMPI/USA'22: 29th European MPI Users' Group Meeting*. Chattanooga, TN: ACM, Sept. 2022. DOI: 10.1145/3555819.3555853
  This publication is included in Chapter 9.

- Alexander Hück, Tim Jammer, Joachim Jenke, and Christian Bischof. "Investigating the Real-World Applicability of MPI Correctness Benchmarks". In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W 2023. Denver, CO: ACM, Nov. 2023. DOI: 10.1145/3624062.3624091
  This publication is included in Chapter 10.

- Tim Jammer, Emmanuelle Saillard, Simon Schwitanski, Joachim Jenke, Radjasouria Vinayagame, Alexander Hück, and Christian Bischof. "MPI-BugBench: A

Framework for Assessing MPI Correctness Tools". In: *Proceedings of the 31th European MPI Users' Group Meeting*. EUROMPI 2024. Perth, Australia, Sept. 2024. DOI: 10.1007/978-3-031-73370-3_8
This publication is included in Chapter 11.

- Tim Jammer, Christian Iwainsky, and Christian Bischof. "Automatic Detection of MPI Assertions". In: *High Performance Computing. ISC High Performance 2020. Lecture Notes in Computer Science*. virtual: Springer International Publishing, 2020, pp. 34–42. DOI: 10.1007/978-3-030-59851-8_3
This publication is included in Chapter 12.

- Tim Jammer and Christian Bischof. "Automatic Partitioning of MPI Operations in MPI+OpenMP Applications". In: *High Performance Computing. ISC High Performance 2021. Lecture Notes in Computer Science*. virtual: Springer International Publishing, 2021, pp. 191–198. DOI: 10.1007/978-3-030-90539-2_12
This publication is included in Chapter 13.

- Tim Jammer and Christian Bischof. "Compiler-enabled optimization of persistent MPI Operations". In: *2022 IEEE/ACM International Workshop on Exascale MPI (ExaMPI)*. Dallas, TX: IEEE, Nov. 2022. DOI: 10.1109/exampi56604.2022.00006
This publication is included in Chapter 14.

- Tim Jammer, Tim Heldmann, Michael Blesel, Michael Kuhn, and Christian Bischof. "Compiler-Based Precalculation of MPI Message Envelopes". In: *High Performance Computing. ISC High Performance 2024. Lecture Notes in Computer Science*. Hamburg, Germany: Springer International Publishing, May 2024. DOI: 10.1007/978-3-031-73716-9_1
This publication is included in Chapter 15.

- Tim Jammer, Adrian Schmidt, and Christian Bischof. "Annotation of Compiler attributes for MPI Functions". In: *Proceedings of the 31th European MPI Users' Group Meeting*. EUROMPI/Australia '24. Perth, Australia, Sept. 2024. DOI: 10.1007/978-3-031-73370-3_2
This publication is included in Chapter 16.

Due to the nature of the synopsis, selected paragraphs from these publications were incorporated in the synopsis verbatim, without additional explicit labeling, in accordance with the computer science department's regulations on Cumulative Theses and Self-Citations in Theses [140].

In addition to the above-mentioned publications, the author contributed to the following peer-reviewed publications:

- Jannek Squar, Tim Jammer, Michael Blesel, Michael Kuhn, and Thomas Ludwig. "Compiler Assisted Source Transformation of OpenMP Kernels". In: *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*. virtual: IEEE, July 2020. DOI: 10.1109/ispdc51135.2020.00016
- Tim Jammer, Christian Iwainsky, and Christian Bischof. "A Comparison of the Scalability of OpenMP Implementations". In: *Euro-Par 2020: Parallel Processing*. virtual: Springer International Publishing, 2020, pp. 83–97. DOI: 10.1007/978-3-030-57675-2_6
- Tim Jammer, Christian Iwainsky, and Christian Bischof. "Survey of OpenMP Practice in General Open Source Software". In: *OpenMP: Advanced Task-Based, Device and Compiler Programming. IWOMP 2024. Lecture Notes in Computer Science*. Perth, Australia, Sept. 2024. DOI: 10.1007/978-3-031-72567-8_7
- Tim Jammer, Simon Schwitanski, Emmanuelle Saillard, Alexander Hück, Joachim Jenke, Radjasouria Vinayagame, and Christian Bischof. "Designing Quality MPI Correctness Benchmarks: Insights and Metrics". In: *Proceedings of the SC '24 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W 2024. Atlanta, GA: ACM, Nov. 2024. DOI: 10.1109/SCW63240.2024.00034

Contents

Part I.

Synopsis

1. Introduction

In the 1990s, the Message Passing Interface (MPI) Standard was designed to standardize an application programming interface (API) for distributed-memory computing. Over time, MPI has achieved widespread adoption, establishing itself as the de facto standard for distributed memory computing. Since then, the (network-)hardware of modern systems evolved considerably. Modern systems allow, for example, for new paradigms like remote memory access (RMA).

Alongside the development on the hardware side, the MPI standard evolved to reflect those changes. After the first standard in 1994 [4], several new features like MPI I/O or one-sided communication were added in version 2 of MPI in 1997 [9]. MPI 3 was released in 2012 and contains several important additions to the MPI standard (e.g., non-blocking collective operations, and one-sided communication operations) [49]. The most recent major version (4.0) was published in 2021, introducing persistent collectives and partitioned communications among other important improvements [136]. The MPI forum continues to work on the next version of the standard, currently discussing the next set of improvements like a standardized application binary interface (ABI) and improved support for accelerators [164, 165].

Many applications, however, still only use a limited feature-set from the MPI standard. In fact, most applications only use functionality that was already included in version 1 or 2 of MPI standard. The MPI usage patterns are explored in more depth in Chapter 2.

Unfortunately, the newer MPI features, such as one-sided, persistent, or partitioned communication operations, have not yet been widely adopted. One reason for the lack of adoption of modern MPI features may be a limited knowledge of new features by the MPI programmer. Programmers first need to learn the basic concepts of MPI which were already established in the first MPI version, as they form the basis of the more complex modern MPI features. This increased complexity of modern MPI is reflected in the standard document, which has grown from 238 pages for the first version of MPI [4] to 1139 pages for the most recent major version 4.0 [136].

Most applications can be expressed by only relying on the old "standard" feature-set of MPI. Apart from one-sided operations, most modern MPI operations, such as persistent or partitioned ones can be conceived as specialized versions of the already existent ones, designed to be more efficient in certain usage scenarios. But as the "standard" MPI operations are sufficient to express the desired behaviour, there is no strong incentive to further study MPI after learning the basics. Considering that the performance benefit of new features was successfully demonstrated in academia, but has not yet arrived in widespread MPI implementations, this is understandable.

On the other hand: As most users actually use the old "standard" features of MPI, it is of importance for the implementors to implement those operations as efficiently as possible. This means that there is only limited incentive for the MPI implementors to do the engineering effort required to fully exploit the potential performance benefit of modern MPI operations, since only a very limited number of users would currently benefit from it. This creates a "chicken and egg"-situation, as illustrated in Fig. 1.1.

As an additional hurdle to overcome, programming correct MPI operations can become quite challenging. In addition to the inherent complexity of distributed memory paradigm, the MPI libraries themselves only provide minimal error checking capabilities and the MPI API is quite error-prone. For example, the user needs to specify a message buffer, message length and datatype by hand. Here, the programmer can by mistake e.g. specify a wrong datatype leading to data corruption or (probably even worse) a silent error resulting in wrong results. As the usage of advanced MPI is more involved, this problem is even more relevant for advanced MPI features.

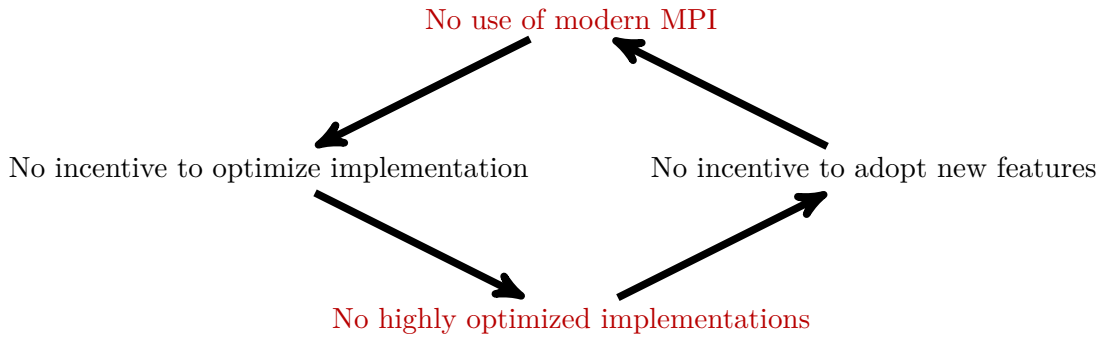


Figure 1.1.: Illustration of the "chicken and egg"-situation for adoption of new MPI features

In order to tackle this problem, this thesis makes the following relevant contributions, also illustrated in Fig. 1.2:

1. Understanding current MPI usage patterns in-depth, in Chapter 2
2. Introducing a MPI Correctness Benchmark to facilitate the development and enhancement of MPI Correctness tools that support the development of correct MPI applications, in Chapter 3.
3. Aiding developers to utilize new modern MPI features with a compiler-assisted approach, that requires no additional effort by the programmers, in Chapter 4.
4. Further showcasing the potential performance benefit from using the newer MPI features by exploiting compiler knowledge, in Chapter 5.

Our contributions are summarized in Chapter 6, followed by the publications that are the basis of this cumulative dissertation, presented in Chapter 7 to Chapter 16.

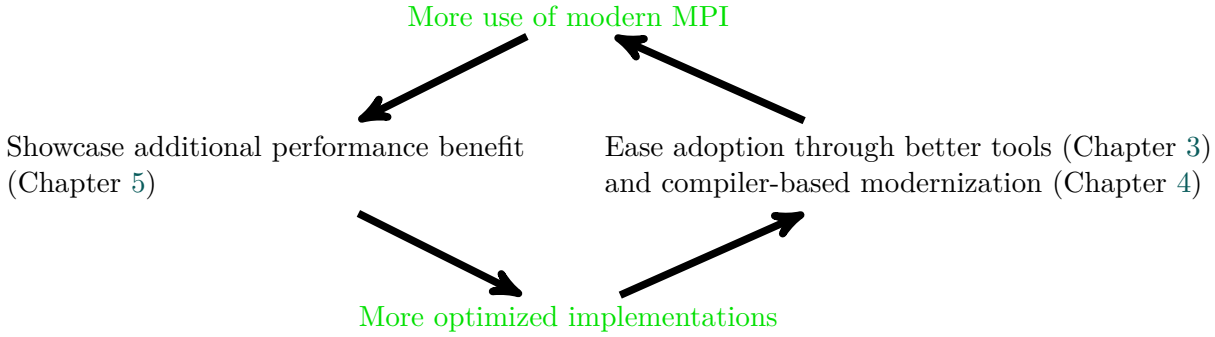


Figure 1.2.: Illustration of this thesis's contributions to address the issue shown in Fig. 1.1

2. Understanding MPI usage

In order to assess if the current MPI usage practices can be considered "outdated", an in-depth understanding of current MPI usage patterns is necessary. This also allows for an assessment which new MPI features are most important for current usage.

Due to the diverse set of MPI applications and MPI's support of multiple programming languages, gaining an in-depth understanding of current MPI usage practices is challenging.

2.1. Current understanding of MPI usage

General MPI usage has been studied before. [12, 84, 92, 101, 105, 112, 132]. The techniques used include source code analysis [84, 101, 105], developer surveys [112, 132] and tracing dynamic MPI usage on a specific high performance computing (HPC) center [12, 92].

Source code analysis has the advantage that it can be automated, as explained below, such that a large set of applications can be analyzed. Compared to tracing dynamic MPI usage, a source code analysis may however be less accurate in terms of the importance of certain operations. Some operations may be executed repeatedly in a core compute loop but show up in source code only once. However, tracing dynamic usage is specific to a particular HPC center and its user base, reducing the generalizability of the collected data, whereas a static analysis can be performed on codes from different HPC sites.

A survey has the disadvantage that it cannot be collected automatically, as the MPI programmers need to participate in the survey. If one conducts repeated similar surveys in order to research the change of MPI usage over time, the quality of the collected data might deteriorate over time, as fewer developers are willing to participate in repetitive surveys. Additionally, the collected data may be biased, not only by the set of MPI

programmers that participated, but also by their perception of the importance of different MPI features. For example, a developer may think that simple features are less important, as they do not think much about using them compared to more complex features.

Most notably is the work by Laguna et al. [105] published in 2019. It stands out, as it "use[s] the largest data sample ever employed thus far (110 unique MPI programs) to understand MPI usage; [...] [Laguna et al.] analyze the static usage characteristics of MPI whereas most previous studies focus on the dynamic characteristics of MPI usage; [they] sample the space of programs as randomly as possible, whereas the relevant previous work sample from specific projects or specific HPC centers" [105] Therefore, Laguna et al. already provides a good general overview over the used MPI functions. Unfortunately, their results still lack detail, as some MPI usage patterns are not analyzed, such as the usage of derived datatypes or matching wildcards.

2.2. In-depth understanding of MPI usage

MPI Analysis Process In order to gain a more in-depth understanding of MPI usage, also taking into account the arguments, we have implemented an extended source code analysis. This is illustrated in Fig. 2.1. Once the code to analyze is downloaded (1), we "normalize" the code by removing all comments and format the code into a defined style (2). This step makes sure that we are able to automatically analyze the source code, even if different programmers follow different style guidelines. Next, we collect all MPI calls, based on their name (3). This step is similar to the study of Laguna et al. [105]. In our study, we additionally extract and categorize the arguments for every MPI call in the source code. For instance, the first and second parameter of an `MPI_Send` operation are categorized as `BUFFER` and `NUM_ELEMENTS`, respectively. Then we resolve macro definitions (4) and try to cross-reference the arguments. This means that we look for e.g. a datatype argument being used in a type creation function, `MPI_Type_commit` and a communication operation, in order to determine the function used to create a user defined datatype. All the collected data is then stored into a database, for later analysis and visualization (5).

The details of this analysis process are presented in Chapter 7. In order to allow for easy reproducibility and re-evaluation, to check for changing usage patterns in the future, we implemented a GitHub continuous integration pipeline for the evaluation.

Insights into MPI usage patterns We used a similar set of applications as the previously mentioned study by Laguna et al. [105], in order to be able to compare our results

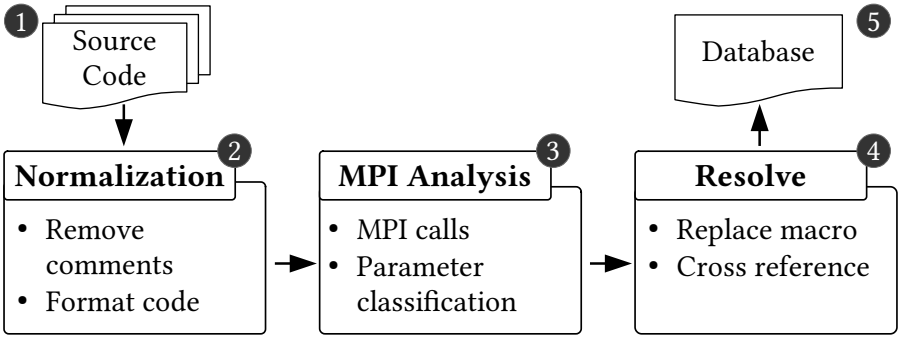


Figure 2.1.: Our analysis tool chain consists of the following steps. (1) Downloaded HPC codes are analyzed per source file (Fortran, C, C++). (2) The code is normalized by removing code comments and formatting the code. (3) Matched MPI calls are classified w.r.t. our MPI feature categories. For each call, we also classify the call parameters, e.g., `BUFFER`. (4) The data set is post-processed by resolving macro `define` statements before cross-identifying named handles. (5) The database contains all MPI calls with cross-resolved symbols.

and see if MPI usage patterns have evolved since Laguna et al.'s study. Our study was able to mostly reproduce the results of Laguna et al., this means that we detected no significant change regarding MPI usage. The usage frequency of different MPI features is shown in Fig. 2.2. We see that all applications use at least one collective operation, with `MPI_Allreduce` being the one most commonly used. For point-to-point operations, we see that most operations are non-blocking. But other communication operations such as persistent or partitioned operations (see Section 4.2) are not commonly used.

Table 2.1.: MPI minimum version requirement.

	MPI 1.0	MPI 2.0	MPI 3.0	MPI 4.0
# applications	35	35	26	0

That modern MPI features are not commonly used yet is also reflected by the version of MPI required by the analyzed applications, which is shown in Table 2.1. Notably, since Laguna et al.'s study in 2019, no analyzed application has employed the most recent version MPI 4.0, which was published in 2021.

[1]For example: `MPI_Wait` is defined in the point-to-point chapter but can also be used solely with non-blocking collective operations.

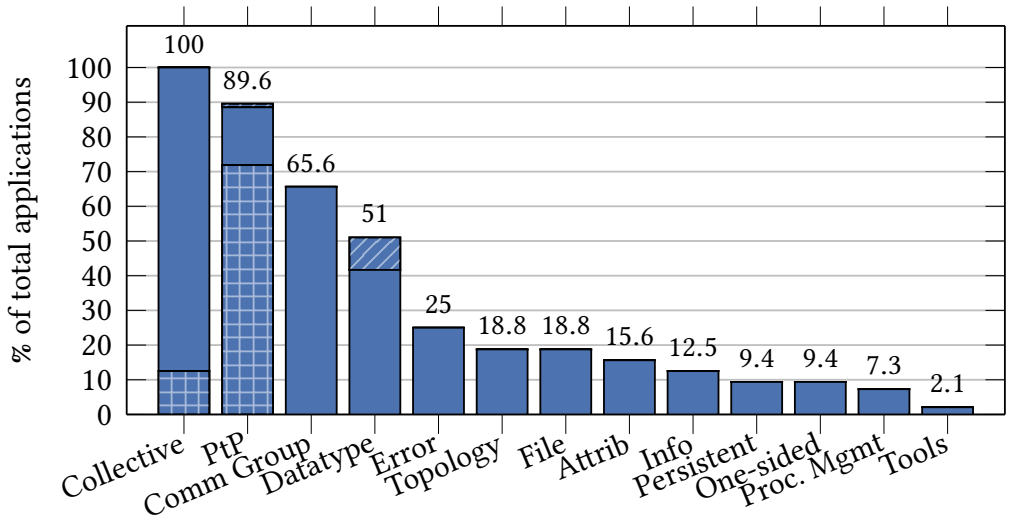
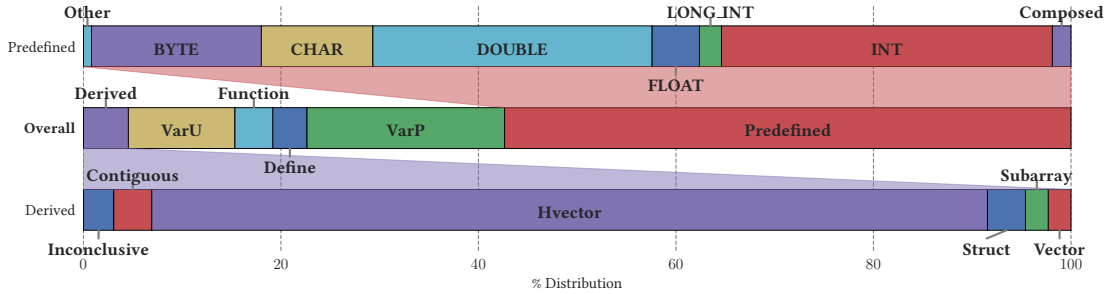


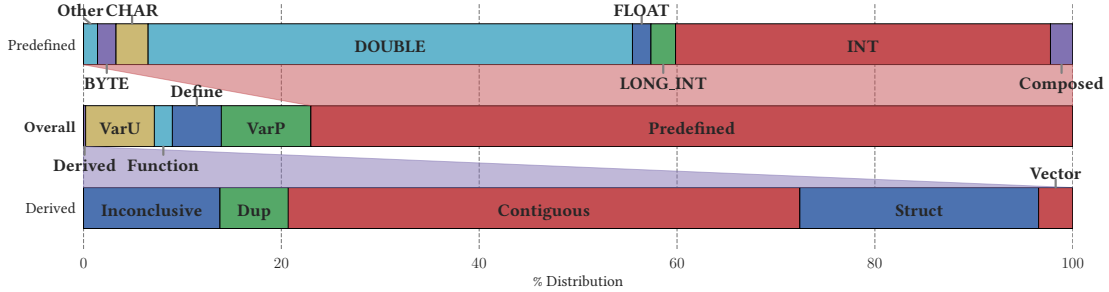
Figure 2.2.: MPI feature usage. The squared areas represent contributions of non-blocking functions. For point-to-point (PtP) and datatype MPI calls, the cross-hatched areas are contributions defined by the standard that we do not consider as *core* functionality for our analysis[1]. For PtP, this includes request handling, which is also used for collectives. For datatypes, this includes functionality that does not construct or duplicate a type.

So our study shows that general MPI usage patterns have not changed since Laguna et al.'s [105] study. In addition, however, our refined code analysis approach is also able to extract more in-depth knowledge about MPI usage patterns. An example can be seen in Fig. 2.3. One trend we observe is that in collective operations, `double` and `int` datatypes are more prevalent than the other types, compared to point-to-point or file-I/O operations. Derived datatypes, however, are rarely used in collective operations, their use is larger in file-I/O. `MPI_BYTE` is also less used in collective operations but very often used for file-I/O operations.

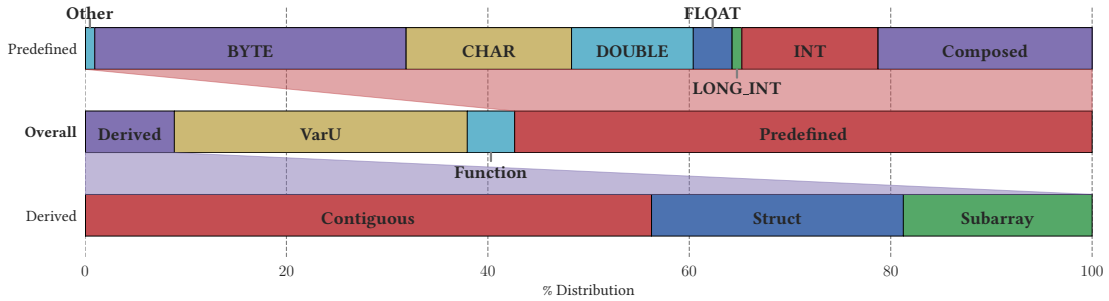
We also note that wildcard usage is quite limited: only $\approx 3\%$ of point-to-point operations use `MPI_ANY_SOURCE` and only $\approx 1\%$ use `MPI_ANY_TAG`. This means that the assertions concerning lack of wildcard usage, explained in Section 4.1, will apply to many applications. The full results of our study, including additional relevant figures, are presented in Chapter 7.



(a) *PtP* datatype usage distribution.



(b) *Collective* datatype usage distribution.



(c) *File* datatype usage distribution.

Figure 2.3.: Categories and their datatype usage distribution. For each subfigure, (1) the middle bar shows the overall datatype distribution, (2) the top bar shows predefined datatypes only, and (3) the lower bar shows derived datatype handles only. *VarU* and *VarP* are variables we cannot cross-reference, while the former also indicates derived datatype constructor usage in codes. Datatypes from *Function* calls and preprocessor *Defines* are listed separately. *Composed* describe types like `MPI_COMPLEX`.

3. Aiding Developers to write correct MPI Applications

Programming a distributed memory application with many different processes is quite complex. The programmer not only needs to implement correct application logic on one process but additionally needs to properly synchronize all processes, distribute the problem to different processes without major load imbalance, etc. All of this has to be achieved without the need for excessive communication, such that the communication overhead stays reasonable. In order to achieve this, the programmer needs to learn the semantics of the different operations MPI offers, in order to use them correctly. Apart from the inherent complexity of programming a distributed application, where many things happen in parallel, the API of MPI itself is also error-prone. For example, in an `MPI_Send`, the user explicitly has to specify the number of elements to send and their datatype. This may be error-prone and lead to an additional maintenance burden, as the user may specify the wrong datatype. Consider, for example, an application that is using double precision numbers and the programmer wants to try single precision, in order to evaluate if a performance increase justifies a loss of precision in the given application. In this case, the programmer may easily miss changing the datatype from `MPI_DOUBLE` to `MPI_FLOAT`, which leads to a mismatch between the buffer size and the size of the message to be sent. This can cause an invalid memory access, as illustrated in Listing 3.1.

In order to help the programmer to correctly use MPI, several MPI correctness tools have been developed, which are discussed in the next section. Such correctness checking tools are of particular importance to MPI, as the error detection capabilities of the MPI libraries are often quite limited for performance reasons. The MPI standard explicitly states that ”text [in the standard document] that states that errors *will* be handled should be read as *may* be handled” [136], as ”the set of errors that are handled by MPI is implementation-dependent” [136].

```
1 float* buf = malloc(10 * sizeof(float));
2 // initialize the message buffer
3 MPI_Send(buf, 10, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
```

Listing 3.1: Example: mismatch between send buffer (float) and MPI datatype (double).

3.1. MPI correctness tools

In general, two different kinds of tools for checking for programming errors in MPI programs have been developed: static and dynamic ones.

Static Correctness Checking tools analyze the source code, or some other representation, e.g., the LLVM intermediate representation (IR), of the MPI program, without executing it.

One such tool is MPI-Checker [59], some checks of which are now available within clang-tidy.[1] It performs its analyses on the program's abstract syntax tree built by the Clang compiler. Hence, it can detect type violations, e.g., passing a `long int` argument where an `int` is expected. Since the tool has knowledge about the MPI semantics, it can also detect certain mismatching argument combinations. Another static analysis tool, PARCOACH (PARallel COntrol flow Anomaly CHecker) [57, 76] builds a control-flow graph (CFG) from the LLVM IR, in order to detect errors related to control flow, such as collective operations not being called from all ranks. PARCOACH also has capabilities to detect data races for RMA operations.

Dynamic Correctness Checking tools provide special libraries that are loaded when executing the target application. These libraries wrap the regular MPI function calls and intercept them at runtime. Hence, the tools first perform the error checking and then continue with invoking the original MPI implementation for the actual communication. The MPI standard supports this interception via PMPI. The most prominent state-of-the-art dynamic tools are MUST (Marmot Umpire Scalable Tool) [42] and ITAC (Intel Trace Analyzer and Collector) [153], although ITAC announced end of support for future MPI development in 2021.

Static analysis tools have the advantage of catching errors early, before execution, and can therefore often provide faster feedback. On the other hand, for certain complex situations in large scale applications, the capabilities of a static tool to detect all possible

[1] clang-tidy, available as part of Clang/LLVM: https://clang.llvm.org/extra/clang-tidy

errors may be limited. Therefore, static tools are often focused on a specific aspect of correct MPI usage. Dynamic tools have the disadvantage that they typically rely on the "observability" of an error. As complex distributed application can exhibit a high degree of non-determinism, this may impair the tools' error detection rate, such as when a deadlock is avoided by chance rather than proper synchronization. This issue is especially relevant in Hybrid MPI+X applications, as discussed in Section 3.5.

3.2. Previous Practice of Correctness Tool Evaluation

Correctness Tools are evaluated with self-written test-cases. As tools have different focus, it is likely that the given test-cases used for evaluation are developed with this focus in mind. While comparative evaluations exist, the focus on certain MPI errors makes it difficult to determine which tools are best suited for specific MPI operations in an application. Additionally, as the test-cases are most often provided by the tool developers themselves, a general assessment of how they transfer to real world usage patterns is missing so far. We will first discuss the general types of MPI errors possible (Section 3.3), before discussing the idea of a correctness benchmark in Section 3.4 and how it can be utilized to evaluate the real world applicability of MPI correctness tools.

3.3. Categorization of MPI Errors

Different kinds of programming errors in MPI may lead to different (non-deterministic) failures at runtime. In general, MPI errors can be categorized into three different categories:

1. **Single call errors**: These errors are only related to local MPI functions and can be detected by only analyzing the parameters of a given MPI function.

2. **Process-local errors**: These errors often consist of an inconsistency between the local context of a process and the parameters of a given MPI call in that process. Thus, the detection of these errors requires analysis of local process information.

3. **Multi-processes errors**: These errors result from the interplay of multiple application processes, such as a deadlock.

Single call errors

1. Invalid Parameters: This category contains invalid parameters in an MPI call, such as a negative value for a rank.

Process-local errors

2. Resource Leak: Any improper destruction of MPI resources (e.g., datatype, request, communicators) leads to a resource leak.
3. Initialization of MPI: Wrong initialization or finalization of MPI can lead to errors, e.g., if messages are sent before MPI is initialized.
4. Request Lifecycle: Request lifecycle errors occur if, e.g., a wait to complete a nonblocking operation is missing.
5. Local Concurrency: A local concurrency error occurs when a process accesses a memory region asynchronously read or written by MPI. This type of error can be encountered with nonblocking or one-sided communication. An example would be using a message buffer before the nonblocking operation completes.
6. Epoch Lifecycle: An epoch lifecycle error occurs when MPI RMA operations are wrongly synchronized by, e.g., mixing different RMA synchronization modes (fences and locks), or performing an RMA operation outside an access epoch.

Multi-processes errors

7. Message Race: Wildcard receive calls can lead to non-deterministic message matching with potential senders, possibly leading to deadlocks.
8. Parameter Matching: Parameter matching corresponds to MPI calls matched with incompatible arguments. An example is a collective operation, where the processes do not agree on a root, which can result in a deadlock.
9. Call Ordering: Wrong ordering of MPI calls can lead to a call mismatch, e.g., when all processes call a receive operation before any process calls a send operation, resulting in a deadlock.
10. Global Concurrency: Global concurrency errors occur if two or more processes access the same memory region (at least one access is a write). An example would be two concurrent `MPI_Put` operations accessing the same memory region at the target process.

For detailed code examples of the different error classes, we refer to Chapter 8. Some literature considers a deadlock as its own error class [16, 35, 51, 79, 80]. In our classification, we consider a deadlock to be a symptom of another error, like a message race.

3.4. MPI Correctness Benchmarks

In order to compare and evaluate different correctness tools, a correctness benchmark suite can be used. Such benchmarks also exist for other programming models besides MPI. For example DataRaceBench [85], for testing Open Multi-Processing (OpenMP) data race detection tools, or RMARaceBench [149], for data race detection tools with RMA operations, such as the Shared Memory Programming Model (SHMEM) [123] or the Global Address Space Programming Interface (GASPI) [81].

The idea of such structured benchmarks to evaluate and compare different tools is not unique to the field of HPC. Examples are benchmarks to evaluate static analysis tools for web security [97], as well as benchmarks for design pattern [48] or clone detection tools [58]. The Rigorous Examination of Reactive Systems (RERS) challenge [168] provides a forum to compare solvers for various program verification tasks. It consists of sequential Linear Temporal Logic (LTL) [1] and reachability solvers, and, more recently [96], parallel LTL and Computation Tree Logic (CTL) [2] solvers.

For MPI, the correctness benchmarks MPI-CorrBench (COBE) (Section 3.4.1), MPI Bugs Initiative (MBI) (Section 3.4.2) and MPI-BugBench (MBB) (Section 3.4.3) were developed.

3.4.1. MPI-Corrbench

MPI-CorrBench (COBE) was developed as the first MPI correctness benchmark for structured evaluation of MPI correctness checking tools. COBE contains 510 small-scale unit-test like tests with 202 correct and 308 incorrect codes in C programming language. The correct test cases have been extracted from an MPI library implementation test set. The incorrect codes, on the other hand, have been handwritten. Each incorrect test contains a brief description of the error. In addition to the small scale unit tests, COBE also includes three well-known (C/C++) HPC mini-apps with manually introduced specific errors, to test the tools with larger codes.

3.4.2. MPI-Bugs Initiative

The MPI Bugs Initiative (MBI) [134] was developed by Laurent et al. independently of MPI-CorrBench. The collaborative integration of COBE and MBI resulted in MPI-BugBench, a part of this thesis, as discussed in Section 3.4.3.

The MPI Bugs Initiative [134] contains 1668 small scale unit-test like codes in C programming language, including 682 correct and 986 incorrect codes. In contrast to COBE, MBI generates the testcodes using several templates for the various MPI categories. The templates contain placeholder tokens that get replaced by (in-)correct usage of MPI calls to generate the final test case. Each test defines a header that describes the intent of each code and specifies how to execute and evaluate it.

3.4.3. MPI-BugBench

MPI-BugBench (MBB) evolved from MPI-CorrBench and MPI Bugs Initiative as a collaborative effort. It improves the generation of test-cases, in order to test for more possible real world scenarios. This means that, in contrast to both previous benchmarks, MPI-BugBench contains multiple different examples of the same error. For example, a datatype mismatch between `float` and `int` as well as a mismatch between `float` and `double`. The reason is, that for a correctness tool, the mismatch between `float` and `int` may result in a message of the same size, whereas the mismatch between `float` and `double` results in a message of a different size. A correctness tool therefore may need to implement different strategies to detect the same error (e.g. datatype mismatch) in different manifestations. In order to enhance the real world applicability, the tools' implementation is therefore tested with several manifestations of the same error.

This, however, leads to a very high number of test-cases: MPI-BugBench can generate more than 4 million test-cases. As such exhaustive testing requires a lot of runtime, the number of test-cases to generate can be reduced using different levels. Level 1 contains only one particular example for each error. Level 2 contains only errors in usage patterns found in the real world, whereas level 3 contains errors in all possible usage patterns. Level 2 and 3 are additionally segmented into 2.1 and 2.2. 2.1 includes erroneous combinations of MPI usage, such that each usage pattern is used at least once, whereas 2.2 contains all possible erroneous combinations. MPI-BugBench is designed with the usage level 2.1 "Sufficient coverage of real world usage patterns" as the default level leading to ≈ 2000 test-cases, in order to strive a balance between efficiency and a thorough tool evaluation.

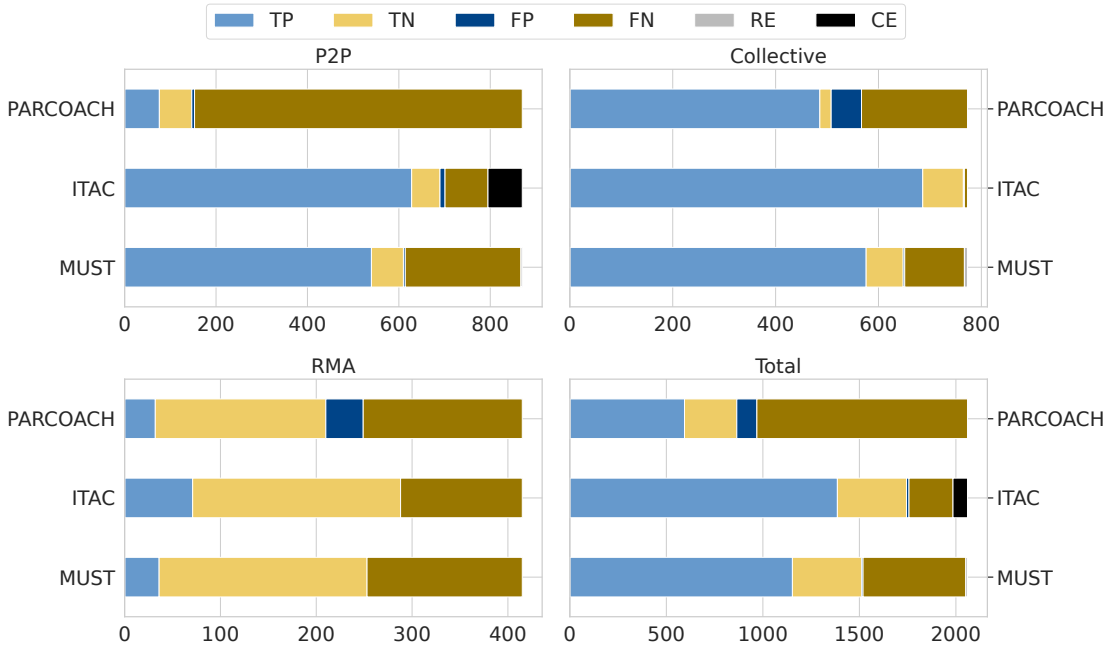
Testing the state-of-the-art correctness tools with MPI-BugBench shows that the tools are able to detect most of the errors present. Fig. 3.1 shows the capabilities of MUST, ITAC, and PARCOACH to detect the errors present in MPI-BugBench. A tool can either correctly detect an error (true positive, TP), wrongfully detect an error, when there is none (false positive, FP), correctly report no error for a correct program (true negative, TN), or wrongfully report no error for an erroneous program (false negative, FN). Additionally, some test-cases lead to compiler errors (CE) for ITAC, as MPI 4.0 features are not supported by the used MPI implementation. In rare cases, the tool itself leads to a runtime error (RE). Runtime errors of the tool are only counted on correct test-cases, as the error present may lead to a crash by itself already, and it is therefore hard to decide whether the tool leads to a crash or the actual error.

In Fig. 3.1, we see that many errors are already detected by the correctness tools. Nevertheless, we see that the tools are not perfect. For example, MUST and ITAC are not able to detect some erroneous usages with less-frequently used MPI calls or datatypes. ITAC does not support all MPI 4.0 features and fails compilation in those cases. The low performance of PARCOACH on point-to-point operations is expected, as it is not designed for those operations, it focuses on collective and RMA operations instead. Detailed results are presented in Chapter 11.

Real world applicability of Correctness benchmarks

In order to assess the real world applicability of a MPI correctness benchmark, we propose a scoring mechanism, that calculates the percentage of real world usage patterns covered in the correctness benchmark. For this MPI usage pattern scoring, we consider: (1) The MPI call, (2) the *Datatype* and *Reduction Operator*, (3) the *Count* (4) and if wildcards, such as `ANY_TAG` are used; (5) for collective operations, the root *Rank* is also important for the usage pattern.

The score for each such usage pattern is based on its frequency in our real-world dataset. As the dataset contains operations with varying arguments, a 100% score requires all operations for each MPI feature category to be present with all arguments in the correctness benchmark. Consider a dataset with six scatter operations and four broadcast operations. If all six scatter operations fully match, but only one out of four broadcast operations match (e.g., one uses `MPI_FLOAT` and the others use `MPI_DOUBLE`), the score of the benchmark would be 70% for collectives. The full details of this scoring workflow are explained in Chapter 10 and the dataset of the real world usage pattern is described in Chapter 2.



CE: Compilation Error RE: Runtime Error, TP: True Positive, TN: True Negative, FP: False Positive, FN: False Negative

Figure 3.1.: Tool evaluation using MPI-BugBench for each MPI feature category.

The scores achieved by the MPI correctness benchmarks COBE, MBI and MBB are shown in Fig. 3.2. We see that the real world coverage score of MPI-BugBench (75% overall) is better than for the other correctness benchmarks. The current lack of coverage for the *Types* category is explained by a surprisingly high number of conversion functions like `Type_f2c`, which the benchmark currently does not cover as Fortran is not supported. For the `other` category, the lack of coverage is mostly due to MPI I/O which is currently not covered by MBB. This high coverage can be achieved by generating multiple instances of the same error, such that all different MPI usage patterns are covered, as explained in Section 3.4.3. This means that generation of test-cases for MBB is guided by our real world dataset (see Chapter 2), such that errors of the usage patterns that occur in the real world are generated. But one can replace our dataset of the general MPI usage patterns, with a dataset of the usage patterns for a specific application. This allows for a tailored evaluation of the correctness tools, if they support precisely the usage patterns used in any given application.

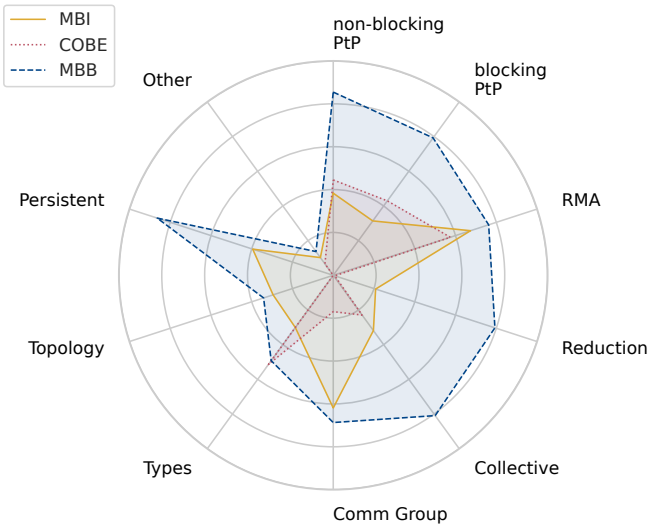


Figure 3.2.: Coverage of real-world MPI usage patterns by COBE, MBI, and MBB.

3.5. Correctness-Checking for Hybrid MPI Applications

As discussed in Chapter 2, MPI is often used in conjunction with other programming models (MP+X) such as OpenMP (MPI+OpenMP). This leads to more complexity, as the developer not only needs to understand the semantics of each programming model individually, but additionally needs to be aware of the interplay of both models. This may lead to even more complicated errors. Therefore, tools support in these situations is strongly desirable. As explained in the next section, a combination of a *pure* MPI correctness checking tool with a *pure* OpenMP tool is not sufficient to detect all erroneous usages resulting from the interplay of MPI and OpenMP.

3.5.1. Categorization of Hybrid Errors

The categorization of possible MPI errors was discussed in Section 3.3. In this section, we first discuss the categories of OpenMP errors that are relevant for MPI usage, before detailing the *new* classes of errors arising from the interplay of MPI and OpenMP.

OpenMP related errors

In previous work [24, 56, 70], OpenMP issues and classifications thereof have been presented. OpenMP errors can stem from, e.g., syntactical problems like misspelling a compiler directive or semantic defects that may manifest as, e.g., a deadlock during execution. We briefly summarize the classification of relevant OpenMP errors based on Münchhalfen et al. [56]. We only include those classes of errors that are relevant when combining OpenMP with MPI. There exist other errors, like the "violation of the single entry single exit principle" of an OpenMP parallel region, that do not add additional complexity when combining with MPI.

Uninitialized memory is a more general concern, as using uninitialized values may lead to undefined behaviour. Münchhalfen et al. [56] originally denoted *unallocated* memory, in our opinion, however, uninitialized memory is more general. In particular, the data sharing semantics of OpenMP allow making variables private to each thread inside a parallel region (the variable is, thus, local to the thread's stack). Any prior initialization outside the parallel region is ignored, and it is the user's responsibility to initialize the value inside the parallel region, [122, Section 2.21.3 C/C++]:

> "The new list item is initialized, or has an undefined initial value, as if it had been locally declared without an initializer."

Wrong assumption about parallelism may cause issues, e.g., expecting the number of threads is always greater than two for a parallel region. This can lead to deadlock, for example, when one OpenMP task is "stuck", waiting for a particular event to happen, but no other threads exist to trigger that event.

Race conditions describe scenarios where the result of an operation depends on the order of execution in a multi-threaded program. Typically, it can be defined as two or more threads accessing a shared resource, where at least one of these threads executes write access. Hence, this scenario, when the shared resource is not protected by synchronization routines such as an OpenMP *lock*, results in undefined behaviour.

Deadlock is a program state that can not make *progress* anymore. Typically, a wrong order of locking for mutual exclusion can lead to deadlocks of participating threads.

MPI	OpenMP	Hybrid
	Assumption about parallelism	Wrong initialization or completion
Concurrency Message race	Data race	Data race
Erroneous program flow	Deadlock	Wrong order of MPI calls
	Uninitialized memory	Wrong memory usage

Table 3.1.: Comparison of OpenMP and MPI errors and resulting errors in hybrid programs

Hybrid Errors

Table 3.1 compares relevant classes of MPI and OpenMP errors and the resulting errors in hybrid programs.

For initialization or completion of a multi-threaded MPI program, one needs to use the proper threading level. MPI defines four threading levels:

`MPI_THREAD_SINGLE` Each process has only one executing thread, it behaves the same as a standard single-threaded MPI program using `MPI_Init`.

`MPI_THREAD_FUNNELED` Only the thread which initialized MPI executes MPI calls.

`MPI_THREAD_SERIALIZED` Any thread can call MPI, but only one thread at a time.

`MPI_THREAD_MULTIPLE` All threads can make MPI calls concurrently.

In order to determine the appropriate threading level, one needs to understand the semantics of the used OpenMP constructs. For example, if all MPI calls are executed inside a `pragma omp master`, `MPI_THREAD_FUNNELED` is a sufficient threading level. But if all MPI calls are inside a `pragma omp single` instead, at least `MPI_THREAD_SERIALIZED` needs to be specified.

Data Races can happen at the MPI or OpenMP level. This means that data races can also happen between MPI and OpenMP. An example is the use of a non-blocking operation, where the buffer gets overwritten by another thread, while the operation is in progress. A *pure* OpenMP tool that does not know about the MPI semantics may not detect this error, similar to a *pure* MPI tool, that does not handle multiple different threads.

The ordering of MPI calls is also more complex when combining with OpenMP. In the single threaded case, the order of MPI calls is well-defined. However, between multiple threads in an MPI process, there are no strict ordering guarantees. The standard specifies that for concurrent MPI operations, the "outcome will be as if the calls executed in some order, even if their execution is interleaved" [136, Section 11.6.1]. The user is responsible for avoiding orderings that lead to erroneous program behaviour, such as deadlocks.

Wrong memory usage can be the result of not being aware that e.g. the `private` directive for variables does not create private value copies for each thread, instead they are put on the thread's local stack as if "it had been locally declared without an initializer", see [122, Section 2.21.3]. Hence, communicating and using such a value without explicit initialization inside the parallel region may lead to undefined behaviour in C/C++.

3.5.2. Hybrid MPI-Corrbench

Based on the errors that can occur due to the interplay of OpenMP and MPI, we developed Hybrid-Corrbench with 103 test-cases. As the interplay of MPI and OpenMP can lead to nondeterministic ordering of operations, we executed each test-case at least ten times, with various interleavings, some that make an erroneous ordering likely and others where it is unlikely. We tracked if an erroneous ordering was observable in a specific interleaving by using vector clocks. We note that a high quality correctness tool should be able to distinguish between a correct ordering ensured by proper means of synchronization and one that occurs "by chance", as this indicates a lack of proper synchronization. This means that although the error may not always be observable during execution, all our test programs are erroneous according to the MPI standard.

The results of MUST and ITAC for the cases of wrong ordering is shown in Fig. 3.3.

Overall, the tools perform adequately given the difficulty detecting conflicting MPI communication calls in a nondeterministic multi-threaded environment. But there is still room for improvement: MUST misses cases where a `MPI_Probe` is used instead of the correct `MPI_MProbe` (cases 80 and 81). Unfortunately, the runtime influence of MUST prevents the error from being observable in these cases.

In some cases (75 and 82), we see that MUST is able to detect that the combination of MPI operations may cause a deadlock, even if we did not observe it during execution. For ITAC it is not possible to detect any errors that are not observable. This is a major drawback, a developer may experience a correct ordering during testing with the error

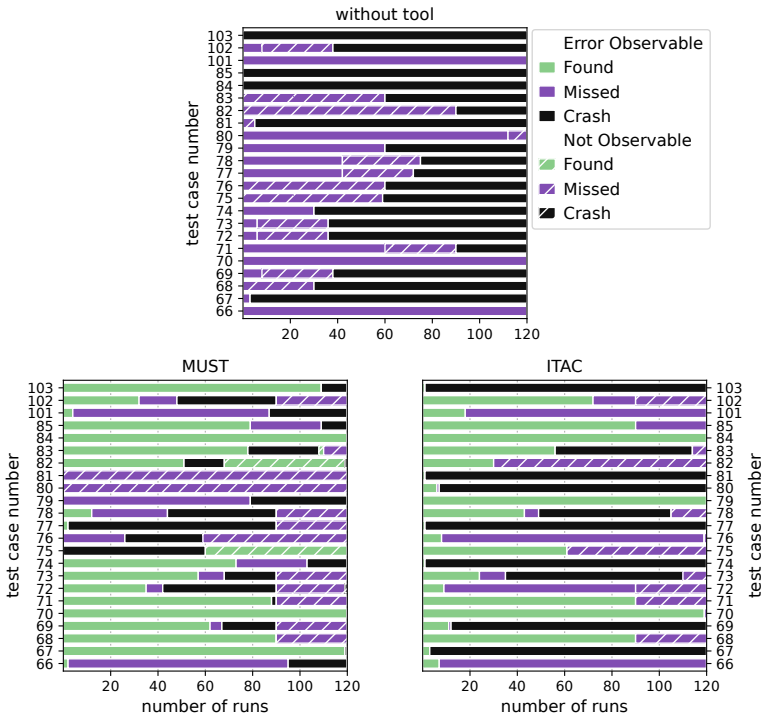


Figure 3.3.: Error detection results for cases where the wrong ordering of calls leads to conflicts. Each bar represents one test case; the x-axis denotes the number of test case executions (with varying runtime settings). The upper bars show the execution without the tool, while the lower bars show the tool-enabled executions. The textured bar section distinguish executions where no wrong call ordering was observable during execution (by chance), whereas the plain bar section denotes an execution where the interleaving of the threads resulted in an erroneous call ordering.

only manifesting in production, making it cumbersome to find the error. But the support for detecting errors that do not manifest themselves during execution is also limited in MUST.

In summary, this shows that a *pure* MPI tool as well as a *pure* OpenMP tool are not sufficient to detect all errors. Detailed results are presented in Chapter 9.

4. Aiding Developers with the usage of Modern MPI features

One way to increase the use of modern MPI features are compiler-based approaches, where the compiler performs the code transformation instead of the user. This way, the user does not need to learn the more complicated modern MPI features, while still being able to utilize - at least part of - the performance benefit.

Compiler-based approaches to transform MPI applications for better performance have been explored before [27, 34, 74, 78, 126]. The focus of this research effort was the conversion from blocking to non-blocking operations. Apart from that, compiler-based approaches can also be utilized to transform a program using a different programming model, such as OpenMP, into an MPI program. The author of this thesis has contributed to CATO (Compiler Assisted Source Transformation of OpenMP Kernels) in this regard [129].

This chapter will demonstrate that compiler based approaches can also be applied to the most recent version of the standard: MPI 4.0. The MPI Forum characterizes MPI 4.0 as follows: "The largest changes [for the 4.0 version of the MPI standard] are the addition of large-count versions of many routines to address the limitations of using an int or INTEGER for the count parameter, persistent collectives, partitioned communications, an alternative way to initialize MPI, application info assertions and improvements to the definitions of error handling" [136]. In this chapter, we focus on the usage of application-info assertions and partitioned communications, as these concepts are entirely new to MPI.

4.1. MPI Assertions

In version 4.0 of the MPI standard, several application info assertions were introduced, in order to allow for a more optimized implementation in certain conditions. These assertions can be specified for a communicator to inform the MPI library about specific usage patterns that may facilitate a more optimized implementation. For instance, the `allow_overtaking` assertion informs the library that it is allowed to relax strict message ordering, enabling more optimized implementations, as the usual ordering guarantees of the MPI standard are not needed for correct execution.

In the remainder of this section, we discuss the different assertions included in the MPI standard and in which cases the compiler is able to determine that this assertion can be specified for the application, while preserving the application's correctness.

4.1.1. `no_any_tag` and `no_any_source` Assertions

MPI provides `MPI_ANY_TAG` and `MPI_ANY_SOURCE` as wildcards in receive operations to match any incoming message, rather than just specific ones. However, Dang et al. [71] show that matching *unexpected* messages is particularly costly. Using wildcards results in all messages being treated as *unexpected*, which negatively impacts matching performance. Thus, performance may improve if mechanisms for unexpected message matching can be disabled.

The `no_any_tag` or `no_any_source` assertions can be specified in order to assert the MPI library that the corresponding wildcard is not used, thus eliminating the need for wildcard matching mechanisms. The compiler analysis is straightforward, requiring only a check for the presence of these predetermined constants in the application.

As discussed in Chapter 2, the usage of wildcards is rather rare, meaning that many applications could potentially benefit from specifying the no wildcard assertions.

4.1.2. `allow_overtaking` Assertion

The semantics of the MPI standard require that messages with the same envelope are received in the same order as they are sent. The `allow_overtaking` assertion relaxes this requirement, making message order irrelevant for message matching. This enables

the MPI library to optimize the communication protocol by eliminating the need to track message order.

The compiler is able to prove in some cases that such relaxation of message matching semantics is appropriate. The basic idea is that the compiler can show that no two messages with the same envelope are ever in the network simultaneously. Therefore, overtaking of messages cannot occur, regardless of whether the MPI library enforces message ordering. This applies to applications that ensure ordering of messages through other MPI synchronization methods, like barriers, rather than relying on message ordering. Analyzing interactions between processes is challenging for the compiler, especially since it does not know the number of processes to begin with. However, for `allow_overtaking` assertion, a process-local analysis is sufficient. The compiler only analyzes the "sender side" of the application. If no two messages that could potentially overtake each other are ever sent, overtaking cannot happen at all. Messages sent from different origin processes to the same target process have no defined ordering between them. Therefore, such messages are not included in our analysis and a process-local analysis is sufficient.

The details of this analysis are explained in Chapter 12. The basic idea is that for each message sent, the compiler tries to determine when the corresponding receive operation is finished on the target process. This is done by analyzing the other MPI synchronization used, such as the `MPI_Barrier`'s present. For a correct and deadlock-free application, the compiler can infer that a message sent before a synchronization, like a barrier, will have been received before that barrier by the target process, as there will be a deadlock otherwise. This implies that any message after the barrier can never overtake a message before the barrier, as the first message has already left the network and arrived at the destination. This analysis is necessary for messages that have a potentially conflicting message envelope, as other messages will be distinguished by envelope instead of message ordering by the application.

4.1.3. `exact_length` Assertion

In MPI it is possible to receive an incoming message with a shorter length than indicated by the receive call. In this case, the message is received into the first part of the buffer, while the rest of the buffer is unchanged. This can lead to additional communication overhead. For example, if the actual message being received does not match to the package length of a lower communication layer, the MPI library may need to take extra precaution that the "leftover" part of the receive buffer is not overwritten, as the MPI standard explicitly requires that. This overhead may be mitigated by using the

`exact_length` assertion, which specifies that all receive operations will receive a message of exactly the indicated length.

In some cases, the compiler is able to identify that this assertion holds for a given application. The compiler is not able to statically match messages at compile time. In particular, it can often not analyze the target rank of any given message, as the number of processes is not known at compile time. However, it can sort all send and receive operations into buckets based on message tags. The compiler can check the buckets, if all send and receive operations have the same size argument. In this case, the length of the messages sent exactly matches the length of the receive operations. The details of this analysis are presented in Chapter 12.

4.1.4. Performance Impact of MPI Assertions

As the MPI standard states that "an implementation is free to ignore all hints" [136], providing these assertions currently has no performance impact, as implementations simply disregard them due to infrequent usage by applications. However, given that the analysis for these assertions, described above, adds only a negligible compile-time overhead ($< 0.3\%$), it is advisable to specify these assertions. This can future-proof codes, allowing them to benefit from more optimized MPI implementations that may respect these assertions in the future.

The potential performance benefit of these assertions has been shown in academia, though. For example, Dang et al. show that omitting the usage of matching wildcards, i.e. specifying the `no_any_tag` and `no_any_source` assertions, can result in up to 3x communication performance improvement in a multi-threaded environment [71].

4.2. Automatic Partitioning of MPI Operations

The 4.0 version of the MPI standard also introduces *partitioned* point-to-point operations as a new type of operations. For a partitioned operation, one message is split into multiple partitions. These operations are designed for use in a multi-threaded environments, where the partitions of a message can be independently handled by different threads. In this case, having only one message with multiple partitions is preferable to having one message per thread, as some message matching overhead can be saved this way. The other possibility of first joining all the threads together and then starting one joint MPI communication operation has the disadvantage that the data transfer cannot start right away, once

```
1 MPI_Psend_init(msg, partitions, count, type, /*...*/ &req);
2 MPI_Start(&req);
3 for(i = 0; i < partitions; ++i){ // may be executed in parallel
4         /* compute and fill partition #i, then mark ready: */
5         MPI_Pready(i, &req); }
6 MPI_Wait(&req);
```

Listing 4.1: Example of a partitioned operation

a thread is ready, if this is beneficial for performance. But the semantics of the MPI standard does not mandate that communication happens for each partition immediately. Therefore, the implementation can wait until multiple partitions are ready in order to optimally use the available network bandwidth.

An example of a partitioned operation's usage is depicted in Listing 4.1. It is similar to the usage of a persistent point-to-point operation, in that first a request is initialized (line 1 of Listing 4.1) that can then be used multiple times for communication operations. In contrast to persistent operations, however, at the point when `MPI_Start` is called, no communication takes place yet. For the actual communication to start, the application needs to mark the partitions as ready (line 5 of Listing 4.1). Once all partitions are marked ready, the completion of the non-blocking data-transfer needs to be ensured, e.g. by using `MPI_Wait`.

For the receive part, one can check if a certain partition is available using `MPI_Parrived`.

4.2.1. Compiler analysis of OpenMP Regions

Most of the performance benefits associated with the usage of partitioned operations can only be realized if the sender side utilizes persistent send operation. If a partitioned send operation is received with non-partitioned receive operation[1], the network bandwidth can be optimally utilized, by sending different partitions at different times, as explained above, which constitutes much of the performance benefit associated with the usage of partitioned operations [104]. If the sending process only uses a single partition however, there is no benefit in receiving different partitions at different times, as the full message is send at once.

[1] Meaning an operation with only a single partition.

Therefore, our compiler based approach is limited to partitioning the sender side. This means that a compiler analysis of only the sender side is sufficient.

As a starting point, we consider the case where all OpenMP threads are joined before sending the message, while the message is computed in a parallel for loop. The other use-case, where multiple messages are sent from all the threads, cannot be covered by our compiler based approach, as the application may need these messages to be distinct for synchronization purposes. Therefore, in this case, an additional analysis and transformation of the receive side is not necessary. As the use-case of multiple messages being sent by different threads requires more effort when integrating OpenMP and MPI, the other case of the message being sent after the join barrier is more prevalent.

The compiler is in some cases then able to detect that a send operation is eligible for partitioning. To this end, the compiler analyzes all send operations and checks the usage of the message buffer. If the message buffer is only written inside the OpenMP parallel region, but not after the join barrier, it concludes that this operation can be partitioned. The idea how the compiler can properly partition the operation, retaining the correctness of the application is discussed in the following section.

4.2.2. Partitioning of Operations

When *automatically* partitioning an operation, it is essential that correctness is preserved. In terms of synchronization with the other process, no precautions needs to be taken, as a partitioned operation can be matched with any receive operation and the message envelope will still be the same.

```
1 // allocate/init msg buffer
2 for(int n = 0; n < num_iter; n++){
3   # pragma omp parallel for
4   for(i = 0; i < size; ++i){
5     /* compute and fill buffer[i] */
6   }
7   MPI_Send(buf,/*...*/);
8 }
```

Listing 4.2: Example code before automatic partitioning.

```
1  MPI_Request req;
2  // allocate/init msg buffer
3  // Instructions to calculate the Loop Access Pattern
4  Init_Partitioned_operation(/* Send Parameters,
5                                Loop access pattern */,&req);
6  for(int n = 0; n < num_iter; n++){
7    MPI_Start(&req);
8    # pragma omp parallel for
9    for(i = 0; i < size; ++i){
10     /* compute and fill buffer[i] */
11     // called after loop chunk (based on OpenMP schedule):
12     ready_partitions_after_loop_iter(i,&req);}
13   MPI_Wait(&req);
14 }
15 MPI_Request_free(&req);
```

Listing 4.3: Example code after automatic partitioning.

As partitioned operations have a very similar usage as persistent operations, the send operation in question is first transformed into a persistent operation, unless it is already a persistent operation. This step is similar to [78]. An example with a standard blocking operation is shown in Listing 4.2. The transformed code is illustrated in Listing 4.3. In order to partition the operation, the number of partitions needs to be calculated. The idea is that those partitions are distributed among the different OpenMP threads and so, we use the number of OpenMP threads of the following parallel region as the number of partitions to use. The persistent operation is then transformed into a partitioned one and the `MPI_Start` call is moved before the parallel region. This is illustrated in line 7 of Listing 4.3. The remaining step is to make sure that the partitions are marked as ready correctly, regardless of the parallel loop schedule used. The compiler calculates the memory access pattern of the message buffer inside the parallel loop, line 3 of Listing 4.3. It uses this information to determine which loop iterations are needed for each partition to be "ready". As the number of loop iterations may not be known at compile time, the compiler inserts the necessary steps for this calculation to happen when the OpenMP parallel region is created. At runtime, the idea is that each thread basically looks up in a table which partitions correspond to its assigned loop iteration, illustrated in line 12 of Listing 4.3. The lookup table itself is initialized when the partition sizes are calculated, which is illustrated in line 4 in Listing 4.3. This approach works for all kinds of loop schedules.

For static schedules, this table lookup could theoretically be compiled just-in-time at the start of the parallel region. As the exact iterations are known, the table lookup can be drastically simplified in cases where the partitions exactly align with the threads, effectively eliminating the need for a table. This was investigated in Laurin Bielich's bachelor thesis, advised by the author of this thesis. Unfortunately, just-in-time compilation is not worth the effort in this context, as the time spent to just-in-time compile an optimized version of the table lookup is in the *serial* part of the application, whereas the potential performance benefit of removing the table lookup is split across the different threads.

Nevertheless, the cost for the necessary table lookup is small ($< 1\%$ of application runtime). Compared to the potential performance benefit from using partitioned communication [61, 104], it is worthwhile to let the compiler automatically partition the operation, even if a manual partitioning of the operations results in a slightly better performance. Grant et al. [104] achieved a performance benefit of $> 25\%$ of communication overhead or $\approx 5\%$ of total application runtime by utilizing a prototype implementation of persistent MPI Operations. As there currently is no widespread MPI implementation that provides such an optimized implementation for persistent operations, however, the usage of our compiler transformation currently results in basically the same performance.

The details are discussed in Chapter 13.

4.2.3. Summary

In this section, we discussed how the compiler can be utilized to automatically transform codes to use more modern MPI features. We note that those compiler based approaches do not require any input or effort from the user, keeping the cost of integrating the new MPI features minimal. Although the compiler may not always make the optimal choices, e.g. in the case of the partition sizes, this serves as valuable a starting point for the user to try out the newer features with minimal effort.

If the user can see a performance benefit, they may be inclined to learn the more advanced features themselves, in order to optimally exploit them. The subsequent modernization of MPI codes can then be helped and guided by further tools such as an integrated development environment (IDE). A prototype of such an IDE plugin that transforms blocking to non-blocking operations was developed in the context of this work, as the master thesis of Sven Donnerhak, advised by the author of this thesis. His code for the IDE plugin is available online: `https://github.com/SvenDo11/mpi-converter`.

A compiler-based approach can also serve as incentive for an MPI library developer to provide a highly optimized implementation of an MPI feature. They can advertise a compiler plugin to be used, in order to get optimal performance with their MPI implementation, allowing them to out-compete competing MPI implementations in certain usage scenarios.

5. Using compiler knowledge to further optimize modern MPI features

In Chapter 4, we discussed how the compiler can be used in order to facilitate the usage of modern MPI features. In this chapter, we want to explore how compiler knowledge can further be utilized in order to optimize these features, to allow for better performance.

Currently, there is only limited research in this regard. To the best of our knowledge, the only study to utilize compiler technology to enable better performance of MPI was conducted by Schneider et al. [53], who utilized just-in-time compilation to optimize un-packing of non-contiguous datatypes. In this chapter, we explore how a-priori compiler knowledge, in contrast to the previously explored just-in-time approach [53], can be utilized to enhance the performance of MPI. Specifically, we show in Sections 5.1 and 5.2 how compiler knowledge can be used in order to reduce the matching overhead for persistent MPI operations.

In addition to optimizing the MPI operations themselves, Section 5.3 explores how compiler knowledge can also be utilized in order to further optimize the code that is using MPI. By adding compiler attributes to the called MPI functions, the compiler is able to perform additional optimizations, e.g. reducing the number of memory accesses to the `rank` variable.

5.1. Optimization of Persistent MPI operations

In this section, we will explain how compiler knowledge can be utilized to lower the overhead of persistent MPI operations, in particular with respect to message matching. First, we explain the matching semantics of the MPI standard, as they apply to persistent operations, in Section 5.1.1. Then we introduce the possible optimizations and their

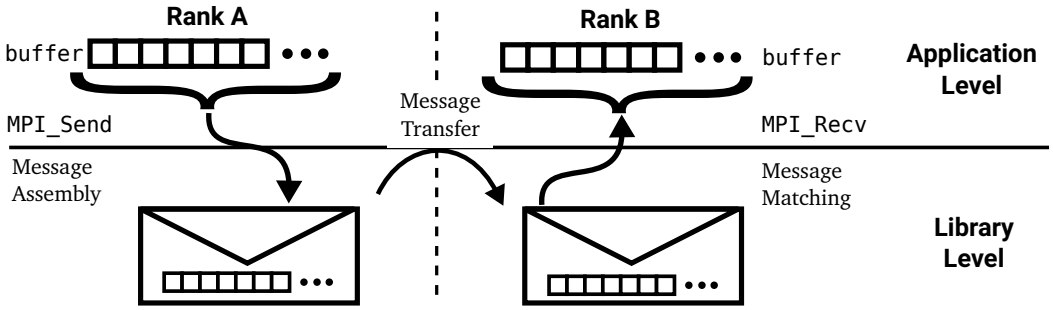


Figure 5.1.: Simplified schematic of MPI data transfer between two communicating ranks. The memory region `buffer` from `Rank A` is copied into a message with its envelope and sent to the receiving process. There, it is matched and disassembled into the memory region of receiving `Rank B`.

performance impact in Section 5.1.2, before showing how the compiler can be utilized to unlock this optimization potential.

5.1.1. MPI Message-matching Semantics

The process of point-to-point communication in MPI is sketched in Fig. 5.1. Messages are transferred with an envelope. This envelope is essential for distinguishing between different messages, ensuring that each message is received by the correct receive operation and in turn placed in the appropriate receive buffer. The MPI standard defines that a message envelope is defined by: (1) The source and destination processes, (2) the MPI communicator used and (3) the user-specified message tag. An incoming message is then only received by a `MPI_Recv` operation that matches all components of the envelope. If multiple messages with the same envelope are used, the MPI standard requires that the order in which the messages are sent is the same order in which they are received (see also Section 4.1.2). This matching semantics is crucial for writing correct MPI applications, as it allows messages to be used for synchronization. As sketched in Fig. 5.1, this matching semantics lead to communication overhead, as the message envelope needs to be handled in addition to the actual data transfer.

This message matching also applies to persistent operations. An example of the usage of persistent MPI operations is shown in Listing 5.1. For a persistent operation, a request handle is created (lines 1-2 of Listing 5.1). At this point, no communication occurs, as the communication starts with a call to `MPI_Start` (lines 4-5 of Listing 5.1), which is

```
1  MPI_Send_init(buffer, count, type, tag, dest, &req_send);
2  MPI_Recv_init(buffer, count, type, tag, src,  &req_recv);
3  for(i = 0; i < iterations; ++i){
4          // compute message
5          MPI_Start(&req_send);
6          MPI_Start(&req_recv);
7          MPI_Wait(&req_send);
8          MPI_Wait(&req_recv);
9  }
10 MPI_Request_free(&req);
```

Listing 5.1: Example of a persistent operation

equivalent to using an `Isend` or `Irecv` operation, respectively. As usual, a non-blocking operation completion needs to be ensured by using `MPI_Wait` or an equivalent MPI usage like `MPI_Test`. As indicated with the for loop (line 3 of Listing 5.1), the idea of persistent operations is that the request created can be re-used multiple times. As the parameters defining the message envelope are only specified at the initialization of the request, the envelope does not change when the request is being reused.

This could mean that, theoretically, the same persistent send/receive operation pair will match for all repetitions of the communication and performing the message matching by the MPI implementation is superfluous in these cases. In practice, however, the MPI library can *not* disable the superfluous message matching for persistent MPI operations. The reason is, that the MPI standard does not require that a persistent send operation is matched with a persistent receive counterpart. In this case, as well as in the case of using matching wildcards, the MPI library must perform the proper matching procedures, as matching the messages by order may be necessary (also refer to Section 4.1).

In the next section, we will present possible optimizations, when matching a persistent send operation with a persistent receive counterpart, while explaining how the compiler can identify these cases in Section 5.1.3.

5.1.2. Matching of Persistent Operations

When we can be certain that a persistent send operation will be matched with exactly one persistent receive counterpart, communication overhead can be saved. After it is established which two persistent operations match, at the first message exchange, further

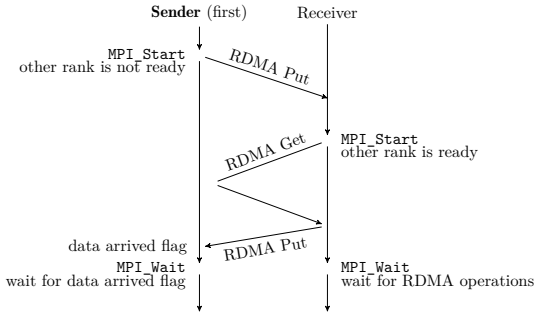


Figure 5.2.: Illustration of the two-sided rendezvous, in case sender arrives first

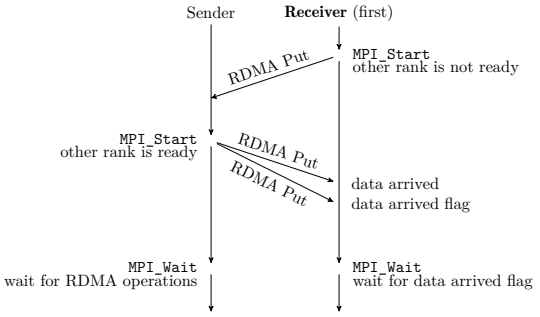


Figure 5.3.: Illustration of the two-sided rendezvous, in case receiver arrives first

message matching is superfluous and can be disabled. This is especially beneficial, as message matching is, especially in the multi-threaded case, a large part of the communication overhead [72]. Other parts of the overhead for message transfer can also be saved, especially for larger messages. Usually, larger messages are sent via a rendezvous protocol employing RMA as the means of actually transferring the message. This typically involves additional overhead, for example for process synchronization, exchange of RMA access tokens and registration of message buffers for RMA transfer. As not only the envelope but also the message buffer's location do not change between different usages of the same persistent request, significant parts of this overhead can also be saved.

We propose an optimized "two-sided" rendezvous protocol in order to transfer the data with as little overhead as possible. One main feature of this protocol is that it is not pre-determined which communication partner actually initiates the RMA data transfer of the message contents. This is illustrated in Figs. 5.2 and 5.3, which show the protocol in case the sender or receiver arrives first.

If a process arrives at the start of the communication, it first checks an agreed-upon memory location to see if the communication partner is ready for the data transfer. In this case, it initiates the RMA data-transfer with the network device and continues execution, as the operation is non-blocking. This means that the network device will handle the actual data transfer in the background. If the communication partner is not yet ready for the data transfer, however, the current process sends a control message indicating its readiness to the other process. This control message is also sent via RMA to the agreed-upon location. This has the advantage that the receiver does not need to interrupt when the control message arrives, as there is no need to handle it right away.

Instead, the data transfer is initialized once a process is ready to do so. This is a clear distinction to other rendezvous protocols, where the receiver needs to handle a control message at some point, in order to perform message matching, which could be avoided if no message matching is required. When a process initiates the actual data transfer, the transmission of the "data arrived"-flag is also scheduled with the network device, so that no further intervention by the processes is necessary. The processes only need to wait for the transfer to be handled by the network device in the background.

Zambre et al. [109] previously found out that more than 90% of the time spent in an `MPI_Isend` is spent in the MPI layer, while only 10% of the time is spent in lower layers that actually communicate with the network device, such as Unified Communication X (UCX). Therefore, this two-sided rendezvous protocol is designed to minimize the time spent in the MPI layer: Only one check for a memory location is performed, before initiating a data-transfer with the network device and returning the control flow from the non-blocking operation. Basically, all the other steps required, including the message matching, only need to be performed at the first occurrence of the communication, when it is established which persistent send operation matches with which persistent receive counterpart.

This "two-sided" protocol has one downside, however: There exists the possibility of crosstalk, where both processes signal the other that they are ready for the actual data transfer and in turn expect the other process to proceed with the actual data transfer. In our experiments, this crosstalk happens very rarely, as the processes naturally "order" themselves to always have a slight interleaving. [110]'s studies reveals that this natural tendency to produce an interleaving happens regularly [110]. This can be modelled to be able to predict the behaviour of the "noise" that creates an interleaving of the different processes [131]. In our case, this means that the occurrence of crosstalk is rather unlikely. Nevertheless, both communication partners can detect the occurrence of crosstalk with our protocol. In this case, the receiving process will proceed with initiating the actual data transfer. The details are described in Chapter 14.

This two-sided rendezvous protocol tries to minimize the time spent inside the send/receive operation. It does not optimize the actual data-transfer that happens in the background and can be overlapped with computation. The goal is to reduce the overhead that can *not* be overlapped with computation, thus increasing the computation-communication overlap. In order to specifically measure the computation-communication overlap of different MPI implementations, Medvedev developed the IMB-ASYNC benchmark as a variant of IMB (Intel MPI Benchmarks) MPI benchmarks [144]. When using the IMB-ASYNC benchmark, the two-sided protocol achieves up to 90% better computation

communication overlap for larger message sizes (1MB), compared to the default UCX implementation in OpenMPI. The details of this evaluation are presented in Chapter 14.

5.1.3. Compiler Analysis of MPI Matching

The optimized protocol described in Section 5.1.2 assumes that one persistent send operation has exactly one persistent receive operation. In this section, we explain how to use compiler analysis, in order to check that the given application adheres to this restriction.

Establishing which send operation matches with which receive operation at compiler time is challenging, however. The compiler usually lacks important information, such as the target process of a message, as the number of processes present at execution time is unknown at compile-time. We can, however, address this challenge by performing a separate compiler analysis for the sender and receiver side. The information gathered at the sender and receiver side can then be combined at runtime.

The compiler analysis establishes the possibility of a "matching conflict" at the sender side. This means that the compiler finds out if there are two different send operations that could send a message to the same destination with the same message envelope. If this is the case, a "matching conflict" is detected, as those two operations may be matched by the same operation on the receiver side. In this case, proper message matching procedures have to be followed, and the exact matching depends on the order of messages (also refer to Section 4.1.2).

The analysis for the receiver side is analogous. The compiler determines if any incoming message matching the analyzed receive operation could be matched by a different receive operation as well.

This combination of the information for the sender and receiver side happens at run-time, when two persistent operations match for the first time. If we can establish that no other operation can send a matching message, and no other receive operation can match the message in question, the two operations in question match, and only those two operations match, meaning they will match again if the persistent operation is executed repeatedly. In this case, an optimized protocol without further tag matching can be used, as explained in Section 5.1.2.

In order to perform this analysis, the compiler does not necessarily need to know the full message envelope. For this analysis to be successful, the compiler only needs to

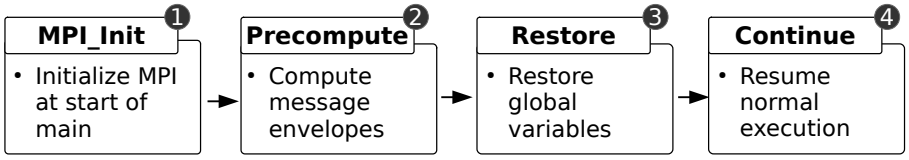


Figure 5.4.: Illustration of the execution of an application altered by our pass. First (1) MPI is initialized as usual. Then (2) the precomputation phase starts, where all message envelopes are computed and registered to the backend library. After the precomputation phase, the original global state is restored (3), before normal execution resumes (4).

prove that the used message envelopes differ somehow, regardless of the exact value of e.g. the message tag used. A static analysis is usually able to tell that `X` and `X+1` will result in different values, regardless of the value of `X`. The full details of this compiler analysis are described in Chapter 14. The current implementation of the static analysis is limited to the scope of one function, meaning that all considered MPI calls must be in the same function. We will discuss how to mitigate this limitation and be able to also cover complex C++ programs using a precomputation approach in Section 5.2.

5.2. Precomputation of Message Envelopes

The limitations of the static analysis explained above in Section 5.1.3 can be overcome by using a precomputation approach. The basic idea is that at the initialization of MPI at the beginning of the application, all MPI message envelopes are computed before they are actually used. This means that all the envelopes are already known to MPI before communication begins. In our case, we will use this information to remove tag matching for persistent MPI operations as described in Section 5.1.2, but other usages to enhance the MPI performance might be possible as well, such as checking if any matching wildcards are used. This idea is outlined in Fig. 5.4. Although this approach, in general, is runtime-based, the compiler also plays a crucial role, as we use it to automatically perform this precomputation step without the need for further interference by the application developer.

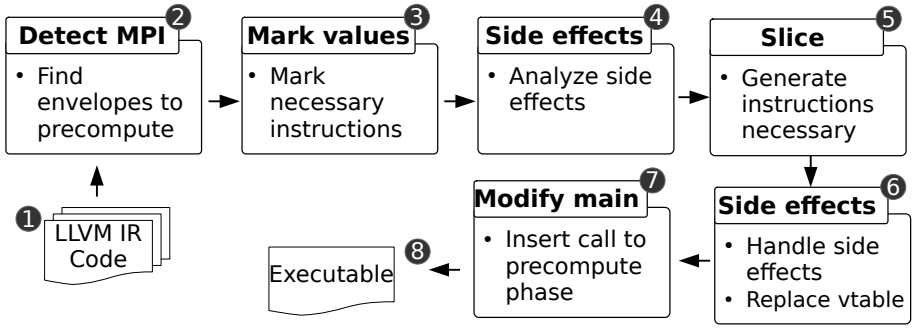


Figure 5.5.: Illustration of the analysis process. Starting from the LLVM IR (1), first the message envelopes that should be precomputed are detected (2). Then, all instructions necessary for calculation and control flow are marked (3) and their side effects analyzed (4). The pass then generates the part of the program necessary for precomputation (5) and adds the handling of side effects (6). Finally (7), a call to the precompute phase is injected into `main`, before the usual LLVM link time optimization (LTO) optimizations pipeline is run (8) in order to produce a binary.

5.2.1. Analysis of Instructions necessary for Precomputation

For the compiler to automatically insert the operations necessary for precomputation, as illustrated in Fig. 5.5 the instructions necessary to compute the used message envelopes need to be identified. This is done using the idea of a taint analysis. The values necessary for the message envelopes, namely the tag and source/destination rank are tainted. Then the compiler recursively taints all values necessary to compute the tainted values. An example is shown in Listing 5.2, where all statements necessary to precompute are highlighted, starting from the envelopes defined in lines 6 and 7. Note that the loop in line 15-17 of Listing 5.2 is not highlighted, as it is not necessary for computing the message envelopes used. The core idea of this approach is that the actual computation of the application is, like in the given example, not necessary to precompute the message envelopes. This fits nicely with the design of persistent operations, as they are initialized only once and subsequently used multiple times.

The details of analyzing the instructions necessary for envelope precomputation across several functions are described in Chapter 15. The general idea is to perform a similar recursive analysis, ensuring that all relevant functions and their callers are included.

```
1  double *DATA;
2  int numprocs;
3
4  void init_communication(int rank, int tag_to_use) {
5  int other = (rank + 1) % numprocs; // next
6  MPI_Send_init(/*buf, size, dtype*/, other, tag_to_use, /*comm, req*/);
7  MPI_Recv_init(/*buf, size, dtype*/, other, tag_to_use, /*comm, req*/);
8  }
9
10 int main(int argc, char *argv[]) {
11 int rank;
12 MPI_Init(&argc, &argv);
13 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14 MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
15 DATA = malloc(sizeof(double) * 100);
16 for (int i = 0; i < 100; i++) {
17 DATA[i] = 42.0; }
18 init_communication(  rank,   argc);
19 // run some calculation
20 }
```

Listing 5.2: Example code with statements "necessary to precompute" highlighted.

Here, we would like to illustrate the particularly challenging issue of handling complex C++ constructs, in particular polymorphic inheritance and exceptions. Templates are also supported by our approach, but as we implemented our analysis on the LLVM IR level, all templates already have been instantiated by the clang front-end at that point.

Polymorphic Inheritance: Polymorphic inheritance is difficult to analyze with static analysis, as it results in indirect calls using a function pointer from an object's virtual table, in order to assure that the correct implementation of a virtual function is called.

LLVM offers a whole program devirtualization pass that replaces indirect calls to virtual functions with direct ones, if this is applicable, mainly as a performance optimization to save the lookup for the function pointer in the object's virtual table. We utilized the analysis part of this pass to find the possible call targets of an indirect call, even if it cannot outright be de-virtualized. In this case, the call is necessary for envelope precomputation, if at least one possible call target is tainted accordingly. The exact details on when a function is necessary for envelope precomputation are described in Chapter 15.

Exceptions: Apart from function calls, exceptions may also alter the control-flow. As illustrated in the example of Listing 5.3, the used message envelopes may depend on

```
struct input_parameters;
try{
  input_parameters = Read_Input_File(); }
catch (const std::system_error& e) {
  // catch I/O errors like file not found
  input_parameters = default_input; }
target_process=depends_on_input(input_parameters);
```

Listing 5.3: Sketch of exception that is "necessary to precompute".

the execution of the exception handling code. If such cases are detected in our analysis, the recursive taint analysis therefore also taints the instructions needed to throw the exception. In the example of Listing 5.3, this is the call to `Read_Input_File()`.

In most cases, however, exceptions like "out of memory" lead to a program abort and are not handled by the application. As C++ semantics require that certain destructors are called in the case of such exceptions, the compiler may automatically add some exception handling code, even if this is not done by the programmer explicitly. Those exceptions are ignored by the precomputation approach: if they occur during the precompute phase, the program will crash. However, a crash of the program that occurs in the precomputation phase is basically the same program behaviour as a crash during the "real" execution of a program. In some circumstances, ignoring of exceptions during the precompute phase may alter the program's behaviour, for example, when an exception is caught by the application programmer, in order to print a message before aborting. We consider those alternations of the program's behaviour similar to the removal of assertions in release codes. This is discussed in detail in Chapter 15, but it is not inherent to the precomputation approach in general, as it is possible to also include those exceptions in the precompute phase.

5.2.2. Side effects in the Precompute Phase

Some operations necessary for envelope precomputation may exhibit side effects that need to be handled in order to ensure that the program can properly continue after the precompute phase.

Side effects that result in an interaction with the execution environment are mostly the result of reading from the execution environment. For example, reading the number of MPI processes. These read operations, however, do not have harmful side effects that need to be handled, in the sense that it is possible to execute these operations again, in order to read the required values during the "real" execution of the program. Therefore,

those read-only operations do not need to be handled specially. Writing to the execution environment is not allowed during the precompute phase, as it would alter the program's behaviour. Write operations to the execution environment, however, are "naturally" excluded, as there is no reason to taint them during the analysis step, as they are not necessary for envelope precomputation.

Write operations to pointers, like global variables, however, are necessary for precomputation. If a pointer load is tainted in the analysis, the operations necessary to compute the result are all relevant store operations to the pointer. Therefore, those store operations are also tainted in the analysis step. The details of this pointer analysis are presented in Chapter 15. This does mean, however, that global variables may have been overwritten during the precompute phase. This is handled after the precompute phase by re-initializing all affected global variables to their initial state.

5.2.3. Applicability of automatic Precomputation

We tested our automatic precomputation approach with a complex C++ application (N-body simulation code MURB[91] with $\approx$ 42.000 lines of code) that utilizes complex C++ features like polymorphic inheritance, templates, and exceptions. The compiler-based approach is able to handle such complex applications. We chose the application such that the optimizations presented in Section 5.1.2 are applicable, as a persistent send operation is matched by exactly one persistent receive counterpart in the application. The precomputation phase is able to correctly determine that this is the case for the application and subsequently, the optimized communication scheme explained in Section 5.1.2 is utilized. We were able to observe up to 15% of speedup for the application. The performance benefit to be obtained may depend on the amount of communication that is part of the application. As the precomputation itself happens at runtime, it does require a small initial runtime overhead. In our experiments, this overhead is rather small, however: Only $< 0.3\%$ of the performance gain is lost to the initial precompute overhead. Therefore, the precompute overhead is well worth it, even if less overall performance benefit can be observed.

5.3. Annotation of Compiler Attributes to MPI Functions

As explained in Sections 5.1.3 and 5.2, compiler knowledge can be utilized to enhance the performance of MPI operations. In this section, we explore how it can additionally

be utilized to enhance the code using MPI, by annotation of compiler attributes to the used MPI functions.

5.3.1. Compiler Attributes

During the compilation process, the compiler derives certain attributes from a function's implementation, such as `nocapture` or `readonly`, in order to allow for code optimization at the call-site. If, e.g., a `readonly` function is called, the compiler then knows that no memory was changed by the function and re-reading variables from memory is not necessary.

Usually, the compiler automatically derives the relevant attributes from the function body during the inter-procedural optimization step. The main limitation of this is, that those attributes are not propagated to other compilation units. This means that if a function from another compilation unit - such as the MPI library - is used, no optimizations at the call-site are possible. One way to mitigate this problem is the usage of LTO [82], where inter-procedural optimizations are performed at the linking stage of the compilation process, where the entire program is available for analysis. As the MPI library is typically linked as a dynamically loaded library, however, LTO is not applicable.

As Moses et al. point out, more than half of the speed-up achieved by using LTO can already be obtained by supplying the compiler with proper function attributes [120]. This means that these function attributes are an integral part of inter-procedural optimization.

Another approach to use compiler attributes across several translation units besides LTO was proposed by Moses et al. The proposed "Header Time Optimization" propagates the attributes inferred by the compiler to other translation units via annotations in the header file [120]. In our work, however, we do not use the compiler to infer the relevant function attributes, rather we infer them manually based on the MPI standard's semantics.

5.3.2. Compiler Attributes applicable to MPI Functions

We implemented an LLVM compiler pass that annotates the used MPI functions with their appropriate attributes. We only utilize attributes that are already well established in LLVM. This means that our approach does not add any significant tool-chain complexity to the compilation process. All optimizations that are enabled by the addition of the

compiler attributes are therefore already part of the LLVM infrastructure and usually being utilized inside of one compilation unit or at LTO.

We manually derived applicable compiler attributes from the semantics prescribed by the MPI standard. This means that our approach is compatible with all standard-compliant implementations. As these attributes only target the call-sites and not the MPI library itself, our approach is also oblivious to the compiler with which the MPI library itself was built.

For a blocking point-to-point operation, the following attributes are appropriate:

```
nofree memory(argmem: readwrite, inaccessiblemem: readwrite)
@MPI_Send(ptr nocapture readonly %buf, %count, %dtype, %dest,
%tag, %comm)
```

According to the MPI standard, the used send buffer is read-only, which is expressed by the `readonly` attribute. The pointer to the sendbuffer is not used after the call returns. This allows the specification of the `nocapture` attribute. `nocapture` means that the pointer is not stored for later usage. Other literature uses the term pointer *escape* for this [11]. When `nocapture` is specified the compiler can be sure, that no other thread or function is going to access the relevant pointer after the send returns, which allows for elimination of unnecessary memory accesses. This attribute may not be inferred by the compiler alone. For the compiler to be able to infer `nocapture`, it needs to *prove* that nothing captures the pointer. As the network might be involved in the communication, the compiler may not be able to prove that all lower network layers do not capture the pointer.
Since the call to `MPI_Send` does not free any memory accessible to the caller, we can annotate the whole function with `nofree` as well.

The `memory(argmem: readwrite, inaccessiblemem: readwrite)` attribute signals that all memory accesses of the send operation happen either via the pointers provided as arguments or to memory that is inaccessible to the caller. Strictly speaking, this is not a valid attribute for this function, or most other MPI functions for that matter. The reason is that it is possible to provide the MPI implementation with an additional user-allocated buffer, which can then be used by the MPI library (buffered-mode). This buffer is potentially accessible from the user's program, which would require to use the attribute that *all* memory is `readwrite`. However, usage of a buffer supplied to the MPI library is undefined behaviour according to the MPI standard's semantics. Therefore, for any compliant MPI application, the user-provided buffer is effectively inaccessible to the

user. This allows usage of the more restrictive memory attribute, potentially enabling more optimizations. Receive operations are annotated similarly, with `writeonly` being used instead of `readonly` for the buffer argument.

Non-blocking point-to-point operations share the same attributes as blocking ones:

```
nofree memory(argmem: readwrite, inaccessiblemem: readwrite)
@MPI_Isend(ptr nocapture readonly %buf, %count, %dtype,
%dest, %tag, %comm, ptr nocapture writeonly %request)
```

Here, the use of `nocapture` might look incorrect at first glance. Strictly speaking, the MPI library may capture the pointer to the send-buffer. But - similar to the above reasoning - the send-buffer pointer is "out of the user-application context" while the non-blocking operation is in progress, as any write usage results in undefined behaviour according to the MPI standard's semantics. A standard compliant user application will not harmfully access the message buffer before it ensures completion with another MPI call.

Our pass also attributes other commonly used MPI functions such as `MPI_Comm_rank` with similar attributes. Those attributes are described in Chapter 16 in detail.

5.3.3. Optimization potential

With the usage of appropriate compiler attributes, additional compiler optimizations are triggered, as the compiler has more information on the semantics of MPI functions. We can observe that the assembly code generated is further optimized with the use of assertions for most of the exascale proxy applications [141].

When compiling the hydrodynamics proxy application LULESH [52], one can clearly see the removal of unnecessary memory loads in the generated assembly code. This is showcased in Listing 5.4, where the left side shows an excerpt of the assembly generated without attributes, while the right side of Listing 5.4 shows the resulting assembly for the same source code location, with the usage of attributes for MPI functions.

When comparing the assembly code in Listing 5.4, we can observe the additional optimizations enabled by annotating attributes to the functions `MPI_Comm_size` and `MPI_Comm_rank`. In line 9 in Listing 5.4, the size of the communicator is loaded into a register by the two assembly codes. Note that it is loaded into different registers. The optimized one (right in Listing 5.4) uses a different register in order to reuse the communicator-size from the register, which can be seen in line 24 of Listing 5.4. In line 13

```
1  mov     rsi, rsp
2  mov     rdi, rbx
3  call    MPI_Comm_size@PLT
4  lea     rsi, [rsp + 4]
5  mov     rdi, rbx
6  call    MPI_Comm_rank@PLT
7  ; ...
8  ; Loads the value of size:
9  mov     eax, dword ptr [rsp]
10 add     eax, 10
11 ; ...
12 ; Loads the value of rank:
13 mov     edx, dword ptr [rsp + 4]
14 lea     rcx, [rsp + 20]
15
16
17 call    ;Mangled function name
18 ; rank is reloaded:
19 mov     esi, dword ptr [rsp + 4]
20 mov     eax, dword ptr [rsp + 40]
21 or      eax, esi
22 ; ...
23 ; size reloaded from memory:
24 mov     esi, dword ptr [rsp]
```

```
lea     rsi, [rsp + 76]
mov     rdi, rbx
call    MPI_Comm_size@PLT
lea     rsi, [rsp + 72]
mov     rdi, rbx
call    MPI_Comm_rank@PLT
; ...
; Uses separate register to load size:
movsxd  rbx, dword ptr [rsp + 76]
lea     eax, [rbx + 10]
; ...
; Uses separate register to load rank:
mov     ebp, dword ptr [rsp + 72]
lea     rcx, [rsp + 12]
; move between registers needed:
mov     edx, ebp
call    ;Mangled function name

; Reuse of the loaded rank
mov     eax, dword ptr [rsp + 32]
or      eax, ebp
; ...
; Reuse of the loaded size
mov     esi, ebx
```

Listing 5.4: Comparison of assembly code generated from `lulesh.cc` with and without attributes. The left side shows the assembly generated without attributes. The right side shows the assembly generated with the usage of applicable function attributes, which leads to the removal of unnecessary memory accesses, as indicated with the comments.

of Listing 5.4, the previously initialized communicator rank is loaded from memory. As above, the unoptimized version (left in Listing 5.4) needs to re-load the same value from memory in Line 19. As both of these values are reused again several times during the course of the entire program, our annotations result in 12 memory accesses being saved by this optimization. The reason the compiler is able to perform these optimizations is the usage of the `nocapture` attribute, annotated to the pointer parameter of both `MPI_Comm_size` and `MPI_Comm_rank`. This attribute signals the compiler that the values of the `size` and `rank` variables stay the same. Without this attribute, the compiler needs to assume that the MPI library may change the variables at any time, e.g. using a

```
1  int mype = 0;
2  MPI_Comm_rank(MPI_COMM_WORLD, &mype);
3  if( mype == 0 )
4          print_input_summary(input);
5  for( int i = 0; i < num_iters; i++) {
6          if( mype == 0 )
7                  printf("keff = %f\n", keff);
8          }
```

Listing 5.5: Excerpt of `main.c` from SimpleMOC [62]

background thread, and therefore needs to fetch the current value (again) before each usage.

A similar optimization can also be observed in the reactor simulation SimpleMOC [62]. This is illustrated in Listing 5.5, where some output, e.g. on the simulations process, is only done by rank 0. With proper attributes to the MPI functions, the conditional in Line 7 in Listing 5.5 can be hoisted out of the loop, as the compiler now has the additional information that the rank of the process does not change. Without the `nocapture` attribute, the compiler needs to assume that the MPI library can overwrite the rank variable at any point and therefore the condition needs to be checked in every loop iteration. Similar compiler optimizations are triggered for the other exascale proxy applications. They are discussed in more detail in Chapter 16.

An example where the above discussed loop hoisting leads to a 16% runtime decrease, due to the optimizations possible when utilizing the attributes, is also discussed in Chapter 16. Unfortunately, in large scale programs, the runtime benefits are rather small. As the compiler optimizations enabled by the usage of the function attributes mostly target the "management" code and not the core compute intensive parts, their performance benefit is limited. But the proposed usage of compiler attributes does not increase the compiler's tool-chain complexity, as only well established LLVM attributes are used. Therefore, the incorporation of such attributes is worth the effort even with limited performance gains.

Although our enhancements currently have only negligible impact on the performance of large scale MPI applications, this work shows that the compiler's capability to enhance the performance of MPI programs is not fully exhausted yet.

6. Conclusions and Outlook

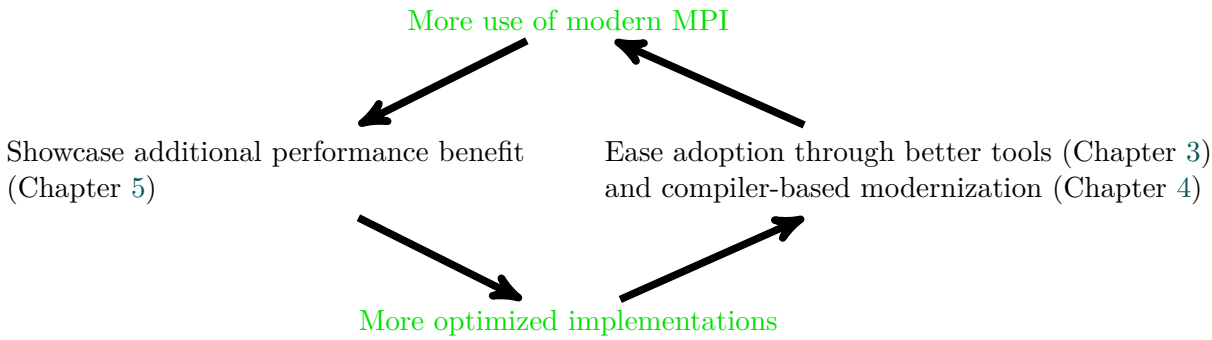


Figure 6.1.: Illustration of this thesis's contributions to address the issue shown in Fig. 1.1

We discussed the current usage pattern of MPI in Chapter 2. We observe that modern MPI features, such as one-sided, persistent, or partitioned operations are not yet widely adopted. This thesis's contributions to facilitate the adoption of modern MPI features are illustrated in Fig. 6.1.

In order to ease the development of correct MPI programs, MPI correctness checking tools have been developed. In order to improve these tools' real world applicability, this thesis introduced correctness benchmark suites, most notably MPI-BugBench (MBB), in Chapter 3. These benchmarks show that the state-of-the-art correctness checking tools are suited for many MPI usage patterns, but still have room for improvement. The unit-test like design of MBB is well suited to directly point out the specific cases for which a tool should be improved, in order to focus development of better tools.

In order to ease the incorporation of modern MPI features, this thesis explored compiler based techniques in Chapter 4. Such compiler-based approaches can modernize some usage patterns without the need for user interference, e.g. for communicator info assertions (Section 4.1). We showed that such a compiler based approach is also suitable for

Hybrid MPI+OpenMP applications, to facilitate the usage of partitioned MPI operations that are explicitly designed for use in a multi-threaded environment (Section 4.2). An automated modernization allows for effortless incorporation of modern MPI into legacy codes. Currently, the modernized code still shows the same performance, as the MPI implementations of the modern features are not yet highly optimized.

In terms of the performance benefit of using modern MPI features, Chapter 5 shows how compiler knowledge can be utilized to achieve better MPI performance. We showcased the compiler's capability to generate optimized code with the annotation of compiler attributes to MPI functions (Section 5.3). To improve the performance of the MPI library itself by saving message matching overhead (Section 5.1), we developed an approach to precompute all message envelopes at the initialization of MPI (Section 5.2). We demonstrated the feasibility of this approach with a medium-sized C++ example application ($\approx$ 140.000 lines of code) and obtained up to 15% better runtime performance as a result.

6.1. Incentives for Adoption

This thesis suggests that MPI library vendors could gain a competitive advantage by adopting a compiler-based approach to enhance their MPI implementation, allowing users to benefit from optimized implementations of modern MPI operations without the need for significant code modifications. Major MPI vendors, such as Intel or Cray, are better positioned to demonstrate the advantages of modern features than academic research. The in-depth understanding of MPI usage patterns presented in Chapter 2 can help to weigh the costs and benefits of such a strategy, which is ultimately a business decision. Our study utilized a set of general applications, but we prepared the analysis process such that a different set of codes can be analyzed. This enables a vendor, for example, to quickly understand client benchmark sets, to decide to what extent these benchmarks would benefit from optimized implementations of modern MPI features.

Another way to encourage the usage of modern MPI features could be the deprecation of older MPI operations in the MPI standard. For example, this is currently discussed for the "Post-Start-Complete-Wait synchronization" of RMA operations [166]. The MPI forum opted for the least invasive approach when adding new communicator info assertions (see Section 4.1), ensuring backward compatibility without requiring changes to existing applications. This means application developers must "opt-in" to an optimized implementation. Alternatively, the MPI standard could mandate that certain assertions

must be specified, forcing users to consider new, potentially optimized approaches. For example, the standard could mandate that the information whether or not matching wildcards are utilized *must* be given. As this thesis demonstrates that the compiler is able to detect such assertions (see Section 4.1), users could rely on the compiler to perform the mandated code modernization, thus minimizing the need for manual changes. However, such a design may be inflexible for future changes, as it hinders the introduction of new assertions in the future, if those assertions cannot be analyzed by the compiler.

6.2. Future Research Directions

In the future, it will be important to assess how the approaches presented in this thesis can be adopted for upcoming versions of MPI. Monitoring changes in MPI usage patterns over time will be of particular interest. The in-depth analysis of MPI usage patterns presented in Chapter 2 is already prepared for this.

This thesis demonstrated that compiler-based modernization approaches can be applied in a completely new fashion in the MPI context, such as communicator-info assertions (see Section 4.1). There are other MPI concepts that have not yet seen widespread usage, such as the newly introduced MPI session model. Exploring how the approaches presented in this thesis can be extended to facilitate the incorporation of malleability and fault-tolerance aspects into MPI applications is an area for future research.

We showed in Section 4.2 that a compiler-based approach can be utilized to modernize hybrid codes, with the example of partitioned operations in MPI+OpenMP codes. In the future, additional research regarding the ways in which further coupling of MPI+OpenMP can benefit from such compiler-based approaches is going to be of great interest. Currently, for example, the MPI forum discusses the idea of MPI continuations [125, 138] to more seamlessly integrate task-based programming models, such as OpenMP tasking.

Besides a better integration with OpenMP, the seamless integration of MPI with other programming models, such as CUDA or SYCL for accelerators, may also benefit from researching such compiler-based approaches. As explained in Section 3.5, tool support that focuses solely on one programming model is not sufficient, as new complexity arises from the combinations of two or more models. Apart from the integration of new programming paradigms, a compiler-based approach could also be utilized to transform a program from one programming model to another, for example from OpenMP to MPI [129].

Another avenue for future research is the application of the discussed approaches to other programming models, such as OpenMP. OpenMP faces a similar issue as MPI regarding the adoption of newer features. For example, the introduction of new clauses for loop constructs such as `fuse`, `reverse` or `interchange` is planned for the next version (6.0) of OpenMP [160]. Our study [155] shows, however, that most applications only use the standard `pragma omp parallel for` directive without any further clauses for fine-tuning. This leads to many inefficient parallel regions where the overhead of thread creation and management likely outweighs the benefits of parallelization. The approaches discussed in this thesis could be useful for enhancing and modernizing OpenMP codes as well.

Further development of advanced tool support, in particular utilization of compiler-based techniques, could serve as a bridge between legacy codebases and new programming paradigms, ensuring that advancements in parallel computing are accessible to a broader range of applications.

Part II.

Understanding MPI usage

7. Investigating the Usage of MPI at Argument-Granularity in HPC Codes

Bibliographic Information

The contents of this chapter has previously been published in:

Alexander Hück, Tim Jammer, Joachim Jenke, and Christian Bischof. "Investigating the Usage of MPI at Argument-Granularity in HPC Codes". In: *Proceedings of the 30th European MPI Users' Group Meeting*. EUROMPI '23. Bristol, UK: ACM, Sept. 2023. DOI: 10.1145/3615318.3615322

The contribution of the author of this thesis is summarized as follows:

Alexander Hück and Tim Jammer came up with the underlying idea and evaluation criteria in mutual discussion. As a co-leading author, Tim Jammer contributed most of the implementation. Alexander Hück drafted most of the manuscript, except for chapter 5 on the MPI parameter analysis as well as many figures and tables, which were mostly contributed by Tim Jammer. Joachim Jenke advised on the MPI Tools perspective, as well as providing invaluable feedback on the manuscript's text and the presentation of the results alongside Christian Bischof. All authors agree with the use of the publication for this thesis.

Abstract

This study focuses on gaining insights into the usage of the Message-Passing Interface (MPI) in a large set of High-Performance Computing (HPC) codes by analyzing MPI

function calls and their argument usage patterns. Previous work has focused on analyzing MPI feature usage by statically matching function calls. However, this approach does not reveal common argument-specific call patterns or cross-interactions between MPI functions. In particular, MPI exposes its internal data structures using handles, and users pass these handles to MPI constructor functions, e.g., to create custom communicators. Tracking the relevant MPI arguments of these constructors and cross-referencing them with other MPI calls in a target code can reveal common user interactions. These insights can be used to optimize, e.g., datatype construction at a library level or to extend MPI correctness debugging tools to verify correct construction of these data structures. To that end, we statically analyze codes to extract MPI function calls and their arguments, cross-reference them with other MPI calls, and provide statistics on common argument patterns and cross-use of MPI functions. We believe that these insights can guide further development within the MPI community to ultimately benefit users.

7.1. Introduction

The Message-Passing Interface (MPI, [136]) is the de-facto standard used for scalable High-Performance Computing (HPC), offering a large number of routines for (a) communication, (b) File I/O, (c) derived datatypes and (d) other functionality to enable efficient distributed computations. The initial MPI standard release defined about 130 routines, whereas, with the continuous addition of features, the current 4.0 standard now defines over 600.

Various approaches have been used in the recent past to gain insights into the usage of MPI features, including (a) user surveys [112, 132], (b) dynamic analysis of (proxy) applications [84, 92], (c) static analysis [105] or (d) a combination of the latter two approaches [139]. The study by Laguna et al. [105] stands out as the most comprehensive one, as it statically analyzed 110 open-source HPC codes. The study provides an overview of categorized MPI features, including their required MPI version and, the minimum MPI standard required based on the detected MPI usage. While approximately 80% of the codes only required features up to the second MPI standard release, overall, many of the available MPI features were used. The study also showed that MPI is often combined with other (multi-threading) paradigms (i.e., MPI+X), OpenMP [10] being used in almost 50% of the codes.

The findings of such studies are valuable for the HPC community, as they can help the MPI standard body shape future revisions for improved interoperability of, e.g., different

paradigms such as OpenMP, and also help MPI library developers prioritize their (e.g., optimization) efforts towards the most commonly used features.

Similarly, the MPI correctness community benefits from such studies, as their debuggers and tools [42, 57, 59, 153] need to support commonly used features in HPC codes to be useful to MPI users. Recently, MPI correctness benchmarks have been developed [134, 135, 143] to help guide these efforts. They aim to provide a standardized test suite for correctness tools by providing codes ranging from small-scale unit tests to mini-applications that contain erroneous use of MPI. Hence, MPI users can assess the relevant feature support of existing tools for their application. Likewise, these benchmarks can be used by tool developers to improve implementation quality and guide new feature support. Correctness benchmarks should therefore reflect MPI usage in HPC applications and, to that end, the study of Laguna et al. can provide guidance.

While the Laguna et al.'s study provides a useful overview, it only analyzed these HPC codes at the granularity of function call sites by statically matching MPI function names. However, further insight can be gained by also analyzing the parameters of these calls. For instance, MPI offers an interface for constructing derived datatypes, exposed as opaque handles to the user, which represent potentially complex user data structures [136, Section 5]. The user can, for example, recursively construct complex memory overlays to define a vector of a structure type. Without matching function call arguments, we can not deduce the interplay of these datatype constructors, see Listing 7.1. Knowledge of this interplay is beneficial for improving datatype correctness checking [95] but also for library developers to focus further optimizations of common datatype construction [45].

To better understand the use of MPI handles and, in general, user behavior in calling MPI functions, we present a survey based on a static analysis of call sites *and* arguments, partially based on the set of HPC codes analyzed in the study of Laguna et al.

At a high-level, our tool chain (a) matches MPI function calls, then (b) extracts arguments (variables or constants) and categorizes these as *buffer handle*, *datatype* (opaque handle), etc. and finally (c) cross-references these opaque handles w.r.t. use in other function calls. The latter generates connections between, e.g., the handle of a type constructor and, say, an MPI send operation call.

We believe that there are several reasons why knowledge of how users interact on this granularity with MPI functions is beneficial:

1. *Optimizations*: Library developers and tool developers can optimize based on common argument usage patterns [45, 142].

```
1  MPI_Datatype struct_type, vector_type; // Opaque type handles
2  // Construct struct type:
3  MPI_Type_create_struct(num_members, block_length,
4                         offsets, member_types, &struct_type);
5  // Construct vector of struct type:
6  MPI_Type_vector(count, block_length_v, stride,
7                  struct_type, &vector_type);
8  // Commit handle, so MPI library knows about vector type & send data:
9  MPI_Type_commit(&vector_type);
10 MPI_Send(buffer, 1, vector_type, ...);
```

Listing 7.1: Example of constructing a vector of structs. Without considering arguments of MPI functions, we cannot deduce that (1) a vector of structures is constructed, (2) only the vector type is committed; hence, the struct is never used directly for MPI (communication) calls and (3) the send operation makes use of the vector datatype.

2. *Correctness Tools*: Tool developers can identify interaction patterns that may be error-prone and need to be supported, such as the nesting of datatypes.

3. *Correctness Benchmarks*: These codes must represent real-world usage, including erroneous argument usage patterns.

4. *Education*: Understanding how developers use MPI is beneficial for teaching strategies to reflect common process interactions and discuss potential error-prone usage patterns.

In summary, we make the following contributions.

- We analyze the MPI usage in the codes used by Laguna et al., considering the newly available MPI 4.0 release. We concentrate on three key insights, (a) overall usage of MPI feature, (b) minimum required MPI version and (c) which, if any, hybrid model (MPI+X) is used.

- We analyze how users interact with the MPI library at the granularity of function call arguments.

- We provide our tool chain and our pre-computed MPI function call and argument classification database for other researchers to use, available at https://github.com/tudasc/mpi-arg-usage.

The paper is organized as follows: In Section 7.2, we provide relevant background information on MPI in the context of our study. Then, in Section 7.3, we describe the implementation details of our static analysis tool and its main limitations. In Section 7.4, we revisit the findings of the previous study [105] to observe any major changes in MPI usage of these HPC codes since then. Subsequently, Section 7.5 gives a more detailed analysis of MPI call sites, focusing on the usage of MPI user handles and variable versus fixed constants. In Section 7.6, we briefly discuss identified error-prone coding patterns and lessons learned for correctness tooling. Finally, we conclude the paper in Section 7.7.

7.2. Background

MPI is the de-facto standard for distributed computations in HPC. The MPI standard defines communication routines and other related functionalities with a low-level API. For instance, users must manually define the length and datatype of a communication buffer, as well as message-related information, such as the destination within a group of processes. Predefined datatypes, like the floating-point type `MPI_DOUBLE`, are provided by MPI. Additionally, predefined communicators encompassing all processes in an MPI application are available. Users can also create customized datatypes and communicators using specific constructor functions.

In Section 7.2.1, we present our MPI feature classification for analyzing HPC codes. This classification is based on Laguna et al.'s study, updated to reflect the current MPI standard. We discuss details of so-called opaque object handles, which reference the MPI library's internal representation of, e.g., the aforementioned communicators or datatypes. These are relevant for our analysis as the user interacts with them in their codes.

7.2.1. MPI feature classification

The MPI 4.0 standard defines more than 600 routines, about 150 more than the previous release. In particular, the addition of new *large count* functions `MPI_{...}_c` of many existing routines is the reason for this increase [136, Section B.1.2]. Other contributions are a new feature for partitioned communication and persistent collectives. Taking the

classification of Laguna et al. [105], adding partitioned communication, MPI 4.0 then provides:

- **Point-to-Point Communication** (*PtP*): Defines routines for message exchange between two processes.
- **Collective Communication** (*Collective*): Defines operations involving all processes within a communicator for data movement, synchronization, or reduction operations.
- **Persistent Communication** (*Persistent*): Features persistent request objects that are used for the optimization of repeated communications, where the communication arguments are the same.
- **Partitioned PtP Communication** (*PartPtP*): Allows a sender to partition data within a single communication transfer via, e.g., threads, and the receiver to reconstruct it. New with MPI 4.
- **One-Sided Communication** (*One-sided*): Routines for handling access to the memory of a target process from another one without its involvement.
- **Derived Datatypes** (*Datatypes*): Feature to create custom memory overlays to build complex user-defined data structures, which can be used for file I/O or communication operations.
- **Group and Communicator Management** (*Comm Group*): Defines groups of processes with ranks, communicators that encompass a group, and how to construct these. MPI comes with built-in communicator handles: `MPI_COMM_WORLD`, which includes all the MPI processes, and `MPI_COMM_SELF`, which contains only the calling process.
- **Attribute Caching** (*Attrib*): Feature for attaching user-defined information to, e.g., communicators and datatypes.
- **Process Topology** (*Topology*): Defines routines to map MPI processes to a geometric shape, such as a Cartesian topology. Also includes topology-based neighborhood communication routines.
- **File I/O** (*File*): Provides routines for parallel read/write operations to a file from multiple processes.
- **Error Handling** (*Error*): Routines to attach error handlers to, e.g., communicators for systematic abort in erroneous states.

- **Info Object** (*Info*): Functionality to attach an unordered set of key-value pairs to pass additional information to other MPI APIs.
- **Process Management** (*Proc. Mgmt*): Specifies how a running MPI program can create new MPI processes and communicate with them.
- **Tools Interface** (*Tools*): Defines the profiling interface and functions to access internal MPI library information.

Like Laguna et al., we do not categorize miscellaneous functions like `MPI_Wtime`. The new set of large count functions is classified like their original functions.

7.2.2. MPI opaque objects and handles

MPI does not expose internal data structures for objects such as datatypes or communicators to the user. They are implementation-defined. For portability, these *opaque* objects are exposed to the user via a *handle*, used as a reference to the internal data structure.

Named constant handles such as `MPI_COMM_WORLD`, which represents all processes, are predefined by the library. For user-initialized handles such as the `MPI_Datatype` handle for a vector, it is first initialized with an API call (e.g., `MPI_Type_vector`) and later used to refer to the vectors internal representation for communication, see Listing 7.1. In general, the user creates and modifies these handles using the MPI library API.

Handles are typically created, among others, for (a) communicators to facilitate communication of process groups, (b) derived datatypes for communication or I/O, (c) files for read/write operations, and (d) requests to track communication operation status.

In our analysis, we search for these constructor functions and, in particular, the handles passed to them. The latter can be cross-referenced, as shown in Listing 7.1, to reason about user interaction with the MPI library. Likewise, we search for the aforementioned predefined named handles to contrast their usage with user-created handles.

7.3. Static analysis

Our tool chain, like Laguna et al.'s implementation, works by processing the source code textually. Thus, we do not depend on any particular build system, hardware or software

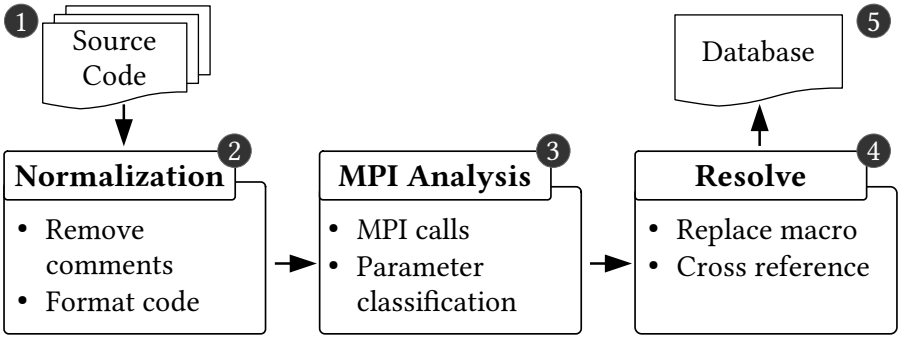


Figure 7.1.: Our analysis tool chain consists of the following steps. (1) Downloaded HPC codes are analyzed per source file (Fortran, C, C++). (2) The code is normalized by removing code comments and formatting the code. (3) Matched MPI calls are classified w.r.t. our MPI feature categories. For each call, we also classify the call parameters, e.g., `BUFFER`. (4) The data set is post-processed by resolving macro `define` statements before cross-identifying named handles. (5) The database contains all MPI calls with cross-resolved symbols.

requirements. Based on the downloaded codes, per source file we apply several steps to enable a parameter-based MPI usage analysis, see Figure 7.1. Our static analysis tool is described in Section 7.3.1. The inherent limitations are discussed in Section 7.3.2. We briefly discuss the MPI codes selected for our analysis in Section 7.3.3.

7.3.1. Tool chain

To analyze MPI call sites and their arguments, we need to handle the various ways codes are formatted and the various ways of passing arguments. To that end, we implemented a multi-stage static analysis tool executed on each source file of a target code. The core of our analysis collects MPI-related call sites and their arguments, including appropriate feature and parameter classifications. We also collect information about MPI+X use.

The key challenge is the analysis of arguments, as they can be encoded as, e.g., (a) complex expressions, (b) (*template*) function calls to retrieve, say, the datatype, or (c) preprocessor definitions, used like constants, including compile-time dependent value selection. We use a best-effort process that classifies and cross-references arguments.

MPI Analysis

We use predefined lookup tables for name- and argument-based classification of MPI functions. The argument table encodes the categories of parameters for all MPI functions. To that end, we first match MPI calls by their prefix and assign one of the aforementioned MPI feature categories, e.g., `MPI_Send` is categorized as *P2P*. Subsequently, we split up the arguments and assign a position-dependent category to the resulting list entries. For instance, the first and second parameter of a `MPI_Send` operation are categorized as `BUFFER` and `NUM_ELEMENTS`, respectively. This results in a database of all categorized MPI call sites including source location.

Resolve symbols

Classifying and cross-referencing MPI call site arguments is critical to reason about MPI usage in a target code. To that end, for each argument, we try to cross-reference the identifier with the other collected call sites. For instance, for the `vector_type` constructor in Listing 7.1 (line 6–7), we would find cross references in the call sites for type commit (line 9) and the send operation (line 10) based on argument name matching.

We classify each MPI call parameter in our database as (a) *variables*, with cross-referencing as described above, (b) other *variables*, where cross-referencing was not successful (e.g., the variable name is a function parameter), (c) *MPI constants*, such as pre-defined datatypes, (d) *literals*, (e) *expressions*, such as arithmetic expression for the rank, (f) *function calls* or (g) *preprocessor defines*. We try to resolve the latter with best-effort heuristics if the macro definition name is unique, i.e., there is no multiple macro definition with the same name.

7.3.2. Limitations

At a high level, we can only observe MPI usage trends, but optimizing based on specific usage patterns may not be worthwhile if, e.g., the MPI routines are only called once during runtime and do not significantly impact performance.

Static analysis that is only based on source texts lacks features such as control-flow sensitive argument data-flow tracking which requires more sophisticated and complex tooling. We believe our accuracy is acceptable as we match based on MPI call site names. However, we can not guarantee that we cover every MPI usage with our approach. In

particular, MPI usage that is wrapped in macros is not detected with our approach, as these can not be properly handled without build system information. The precision of the call site *argument analysis* and the cross-referencing of user handles depend on the particular programming style of a code.

- Compile-time dependent macro definitions cannot be resolved. An example would be a code that can switch between single and double precision at compile time.
- Variable names that are defined outside of the scope of a user handle constructor (and are not a predefined MPI identifier) are not in the scope of our analysis. We only cross-reference handles based on their identifier name. If our analysis tool detects the construction of a vector type with a specific name `vector_type`, and this name is used in a communication operation, we assume a cross-reference to that constructed vector type. Hence, a re-use of `vector_name` in the MPI context for multiple differing datatypes (outside of constructors) would lead to erroneous analysis. Such a style is in our opinion (a) bad coding practice and, also, (b) unlikely. However, if the identifier name is re-used in multiple MPI datatype constructor functions, we assign the category *Inconclusive* (but a derived datatype). On the other hand, if an unknown identifier in the communication is detected, we classify this as a datatype identifier in the category *Other* (without possible cross-reference).

Generally, these cases therefore decrease the precision of our analysis but do not compromise its validity or relevance.

7.3.3. Target MPI code selection

We analyze 96 HPC codes that use a mix of C, C++ and Fortran, with various MPI+X models where X may be OpenMP, CUDA, OpenCL, OpenACC, or a combination thereof. Our sample comprises the 110 selected codes by Laguna et al., except for (a) ten codes that required some form of registration to access the source, (b) one code that switched to an external C++ library that wraps MPI, and (c) two codes that have since removed MPI use.

7.4. MPI usage analysis

While our study is focused on MPI function call site argument analysis, we repeat the analysis of Laguna et al. w.r.t. (a) overall MPI feature use, (b) required minimum MPI

version and (c) hybrid MPI+X model use, to see whether significant changes occurred since.

To that end, we only search for MPI call site names and statements that show the use of MPI+X hybrid model use, without considering any call site arguments as described in Section 7.3. We analyze 96 HPC codes, as described in Section 7.3.3. The MPI feature categories and their abbreviations are given in Section 7.2.1. The functions to query the communicator size (`MPI_Comm_size`) and the process rank (`MPI_Comm_rank`) are not counted towards the *Comm-Group* category as they are commonly called. Other MPI functions not assignable to any of the categories, e.g., `MPI_Init`, are omitted in our graphics. Analysis results are independent of the number of function calls in any feature category in a code. Likewise, this applies to hybrid use of, e.g., OpenMP if a single directive is detected. Henceforth, we selectively omit the `MPI` prefix (of function names) for brevity.

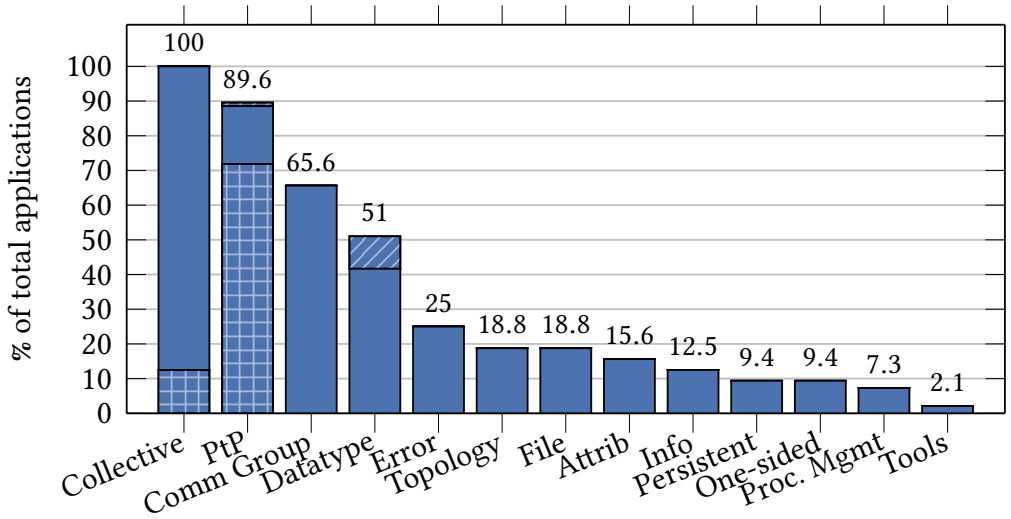


Figure 7.2.: MPI feature usage. The squared areas represent contributions of non-blocking functions. For PtP and Datatypes, the cross-hatched areas are contributions defined by the standard that we do not consider as *core* functionality for our analysis. For PtP, this includes request handling, which is also used for collectives. For Datatypes, this includes functionality that does not construct or duplicate a type.

7.4.1. MPI feature usage

Figure 7.2 shows the different features we detected with the percentage of codes using them. We categorized MPI functions as they are defined in the respective sections in the MPI standard. In particular, request handling routines are defined in the *PtP* section of the MPI standard but are also used for *Collectives*. To that end, the hatched areas are contributions to categories that are not *core* features for our analysis, like the request handling. For *Datatypes* this includes functionality that does not construct a (new) type, like querying the size of a predefined MPI datatype.

We discuss noteworthy insights for selected categories:

- *Collective*: All codes use collective communication. `Allreduce` and `Barrier` are used most often, detected in 80 and 82 codes, respectively. Non-blocking variants are significantly less used. `IAllreduce` and `Ialltoall`, the most used, are detected in five codes each.

- *PtP*: The standard non-blocking variants `Irecv` and `Isend` are used most often. In particular, the non-blocking `Irecv` is used in 66 codes, followed by `Isend` used in 60 codes. The standard blocking variants are used in 60 and 59 codes, respectively. More specialized routines like (non-blocking) buffered or synchronous communication variants are used in less than 11% of codes.

- *Comm Group*: For constructors, `Comm_split` and `Comm_dup` are used most often in 48 and 37 codes, respectively. The use of `Comm_f2c` and `Comm_c2f` is detected in 15 and 11 codes, respectively. These routines are used in mixed language codes to convert communicator representations from one to another language to ensure seamless cross-code interaction of routines. In contrast, the group analogs are only used in one code.

- *Datatypes*: We focus on type constructors and ignore, e.g., the `commit` or `free` functions. The top three types are `contiguous`, `struct`, and `vector`, detected in 26, 23, and 10 codes, respectively. We give a more detailed analysis of their usage in Section 7.5.

- *Error*: Error handlers for communicators (`set_errhandler`) are set in 15 codes. The analogs for file I/O are used in two codes, and the one-sided communication handler is never used. Half of all call sites that set error handlers, set `MPI_ERRORS_RETURN`, while ≈18% use `MPI_ERRORS_ARE_FATAL`. Therefore, only the remaining ≈32% (12 call sites) register some form of user-defined handler.

- *Topology*: Cartesian layout functionality is predominantly used compared to graph-layout ones. Only one code uses neighbor collectives, and only the all-to-all standard or vector variants.
- *Tools*: The tools interface `MPI_T_{...}` is never used. We counted two codes using `MPI_PControl`, which is defined in the tools section of the MPI standard [136, Section 15.2]. The routine is used to set profiling levels (of external profilers).

Notably, compared to Laguna et al.'s study, more than one code now makes use of the process management feature of MPI, which allows (new) processes to join existing MPI communication environments at runtime. Other results are similar to Laguna et al.

Table 7.1.: MPI minimum version requirement.

	MPI 1.0	MPI 2.0	MPI 3.0	MPI 4.0
# applications	35	35	26	0

7.4.2. MPI version usage

In Table 7.1, the minimum required version of the codes is shown.

We analyze the distribution of MPI feature categories for each set of codes that require a specific standard. The need for MPI 3.0 primarily arises from the use of non-blocking collectives and one-sided communication calls, which together constitute two-thirds of all the detected calls for this version. Our analysis of MPI 2.0 codes revealed that the two most frequently used categories were I/O (≈24%) and one-sided communication (≈18%), with all other categories being below 10% in frequency. Notably, none of the codes has adopted any features of the recent MPI 4.0 revision. We suspect that the recency of the revision and legacy codes working around the integer count argument limitation contributes to its lack of use.

7.4.3. Hybrid model

We analyze our set of HPC codes for using any hybrid MPI+X programming model. In addition, we categorize OpenMP use w.r.t. more advanced parallelism features, namely task-based parallelism and GPU target offloading.

Figure 7.3 shows the use of hybrid MPI+X programming models in our set of HPC codes. In particular, for ≈27% of codes, we detected no multi-threaded paradigm usage. For hybrid programming models, OpenMP is the most common: (a) about ≈39% of codes use it exclusively and, in total, (b) about ≈68% use OpenMP in some form of paradigm mixture.

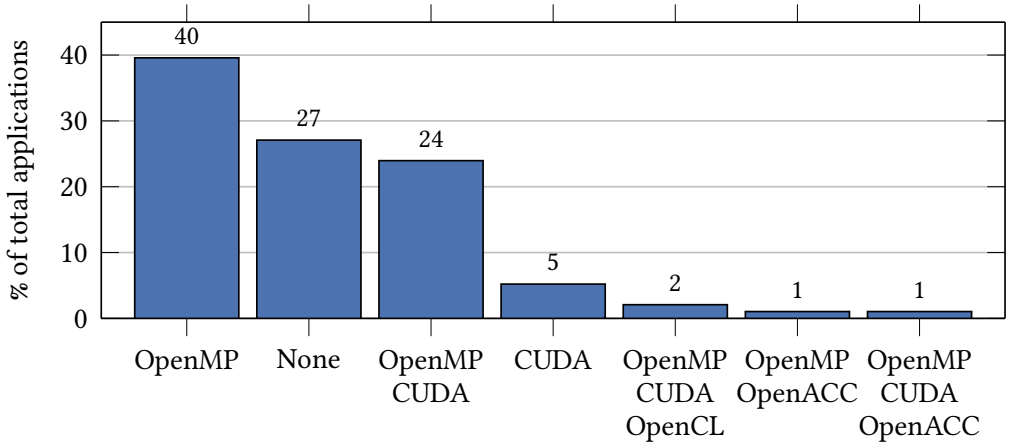


Figure 7.3.: Hybrid MPI+X use. The None category includes codes where we found no additional paradigm use.

Table 7.2 shows the use of OpenMP task and target offload features in our set of OpenMP codes, i.e., ≈68% of all codes. While the task feature, which enables task-based parallelism, was introduced in OpenMP revision 3.0, the more recent addition of target offload in revision 4.0, which enables offloading workloads to a GPU, was detected more than twice as often. Typically, however, loop-based parallelism is used most commonly.

Table 7.2.: Task and target offload use in codes using OpenMP (in some combination).

	Target Offload	Tasks & Taskloop
# OpenMP applications	9	4

7.5. MPI parameter analysis

We analyze MPI parameters, particularly the cross-use of MPI handles in function call sites. Our analysis is based on statically searching the source code, as discussed in

Section 7.3.

The rest of this section is structured as follows. In Section 7.5.1, we examine derived datatype constructors and their use w.r.t. other MPI features such as communication and file I/O. In Section 7.5.2, we present analysis w.r.t. reduction operations, highlighting user-defined operations and the general distribution of operation types. We discuss usage of communicators in Section 7.5.3, focusing on communicator constructors and communication patterns. Finally, in Section 7.5.4, we present a communication value analysis that examines the usage of predefined constants and their roles in ranks and tags for point-to-point and collective communication.

7.5.1. Datatypes

Derived datatype

Figure 7.4 shows the detected derived datatype constructors and their total number of call sites for all codes. We identified the categories *PtP*, *Collective* and *File* as major users of derived datatypes. The `contiguous` type is most commonly found and describes replicating a user-specified MPI type without gaps or padding in between the types. Next are `struct` types, which encode a combination of (non-contiguous) MPI type elements.

Derived datatype cross-use

To show nested datatype construction, we analyzed each type constructor call w.r.t. their input datatype handle, see Figure 7.5.

Our findings are categorized according to the type of input datatype, which can either be a predefined MPI type, e.g., `MPI_INT`, or a derived datatype. In addition, the category *Inconclusive* includes handles that our static analysis has identified as derived datatypes, but these are created by multiple datatype constructors, making them non-unique. The *Other* category describes variables that our analysis tool cannot cross-reference with other MPI calls or constants. Figure 7.5 shows the percentage of call sites with the described input datatype regarding the total number of calls for the given type creation function. We highlight significant construction patterns, omitting the `MPI_Type_create` prefix:

- With *predefined* datatypes: We detect a strong association w.r.t. `contiguous`, indicating that this constructor is frequently used with predefined MPI datatypes, with byte and char (≈40%) being the most common, followed by double (≈15%).

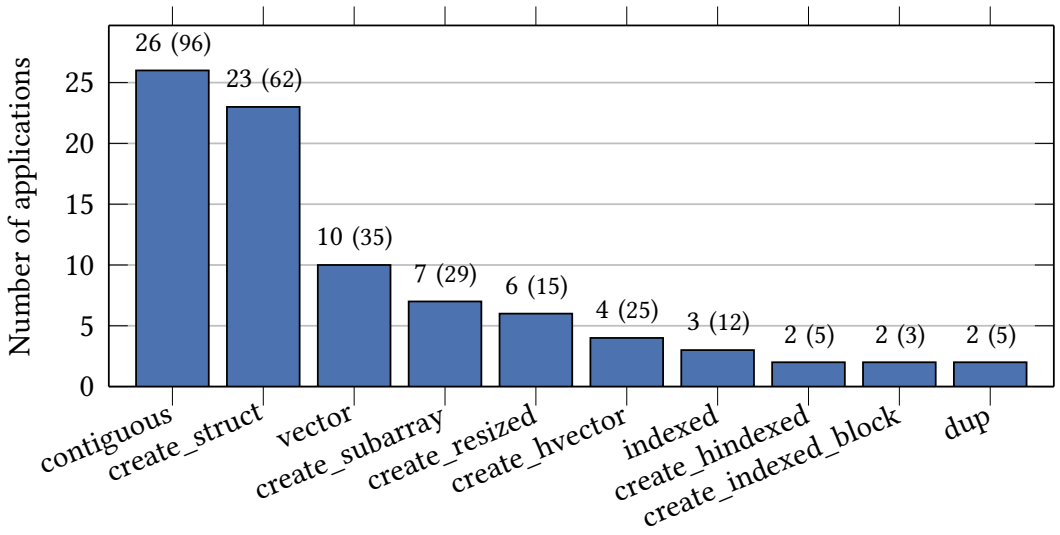


Figure 7.4.: Use of constructor functions for derived datatypes. For each constructor, we list the number of applications and, in parentheses, the total number of function call sites. For brevity, we removed the `MPI_Type` prefix.

The `vector` and `subarray` constructors show usage of primarily the double datatype, at ≈40% and ≈28%, respectively.

While `indexed` and `hindexed` also have a strong correlation, these types are used significantly less compared to the others. `indexed` is primarily constructed from double (≈50%) and `hindexed` from byte (≈40%).

- With *derived* datatypes: This category describes nested derived datatype construction. The `contiguous` type is most frequently used in `indexed_block` (≈67%), followed by `hindexed` (≈40%). The `vector` type has the highest usage in `hvector` (≈60%). Notably, the `hvector` and `indexed` constructed types are primarily used with themselves.

- With *other* handles: For these handles, our tool cannot detect any association. This applies, in particular, to the **`struct`** type as they are constructed from an array of types (and offsets and sizes). Here, our tool cannot resolve these array entries.

Overall, some constructors like `contiguous` are commonly used with predefined MPI datatypes, while others like `hvector` are not. The usage of each constructor likely depends on the specific requirements of each application.

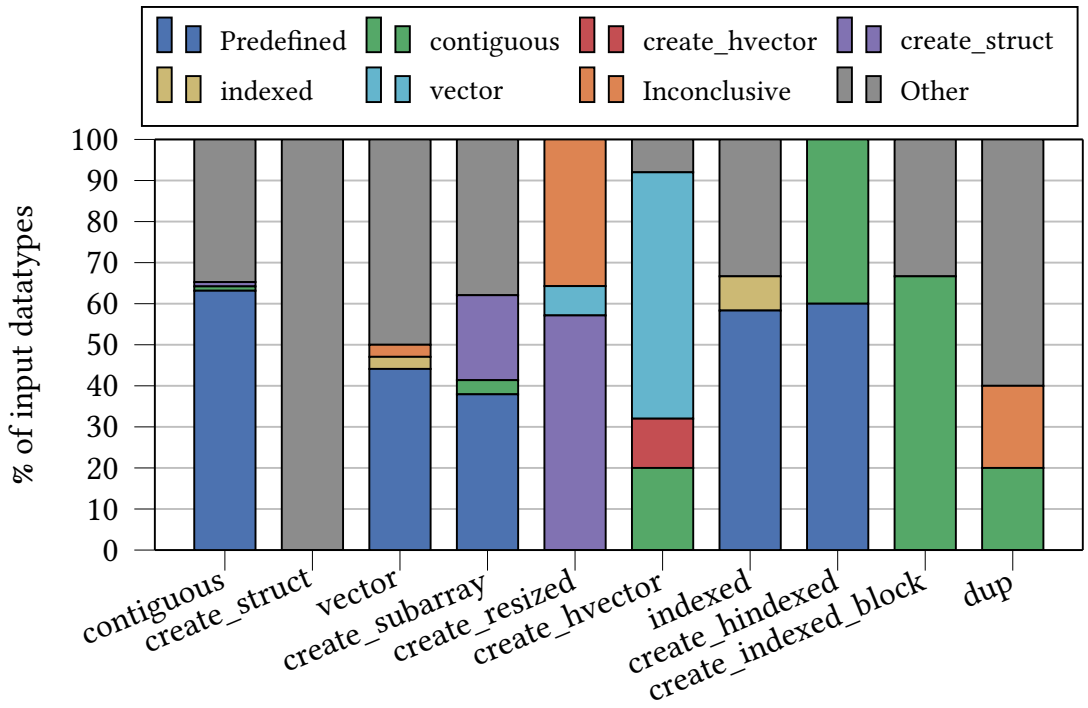
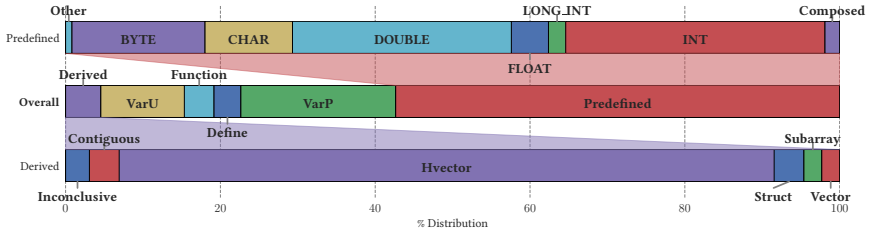


Figure 7.5.: Handles passed to constructors of derived datatypes. For brevity, we removed the `MPI_Type` prefix.

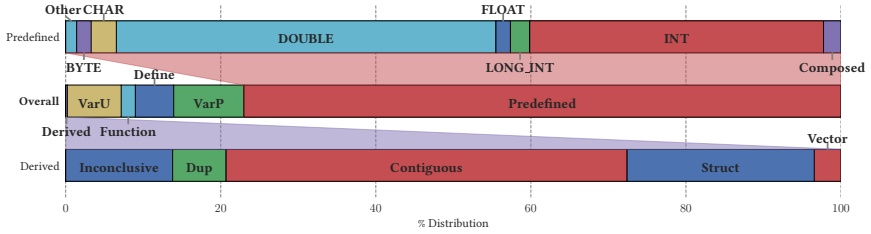
Datatype usage in categories

Figure 7.6 shows the overall usage of datatypes for the categories *PtP*, *Collective* and *File*. The figure visualizes the percentage of call sites using the shown datatype. For our analysis, we do not discern between C and Fortran datatypes and combine, e.g., signed and unsigned int types to the single category `INT`. Likewise, this applies to other datatypes.

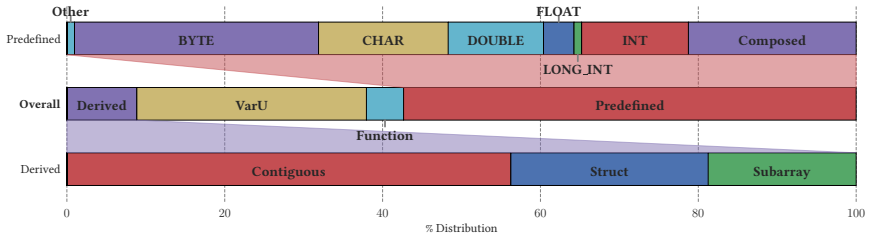
- With *PtP*, predefined datatypes like double and int are most common, followed by byte and char. The OpenFOAM solver, for instance, uses a wrapper around MPI to serialize data structures and communicate using the byte datatype. Derived datatype use is less frequent, with `hvector` being the most common.
- With *Collective*, we mainly see use of predefined datatypes, particularly double and int, with minimal use of derived datatypes like `contiguous` and `struct`.



(a) *PtP* datatype usage distribution.



(b) *Collective* datatype usage distribution.



(c) *File* datatype usage distribution.

Figure 7.6.: Categories and their datatype usage distribution. For each subfigure, (1) the middle bar shows the overall datatype distribution, (2) the top bar shows predefined datatypes only, and (3) the lower bar shows derived datatype handles only. *VarU* and *VarP* are variables we cannot cross-reference, while the former also indicates derived datatype constructor usage in codes. Datatypes from *Function* calls and preprocessor *Defines* are listed separately. *Composed* describe types like `MPI_COMPLEX`.

- With *File*, we detect predominant use of predefined datatypes, particularly, the byte and char types, which account for ≈50%. Among derived datatypes, `contiguous` is most common (≈57%), followed by **`struct`** and `subarray`.

Overall, predefined datatypes are predominantly used across all categories. *File* operations

show a trend towards byte and char types, while *Collective* and *PtP* mainly use double and int types.

7.5.2. Reduction operations

In Figure 7.7, we show the usage distribution of operators w.r.t. `Allreduce`. *VarU* and *VarP* are variables we cannot cross-reference, while the former also indicates operator constructor usage in codes. In general, `Allreduce` (≈5,600 calls) and `Reduce` (≈835 calls) are the most used reduction routines. Other detected variants are called less than 30 times each. Predominant use can be detected for `SUM` operations (over 50%), followed by `MAX` then `MIN` and `MIN/MAXLOC`. This usage pattern also applies similarly to `Reduce`.

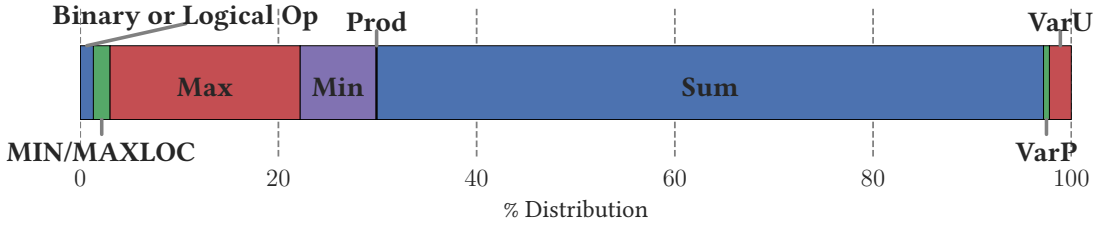


Figure 7.7.: Allreduce operator usage distribution.

Datatypes and operations

Figure 7.8 shows the four most used operations and their datatype distribution. The *Other* category includes datatypes our tool cannot resolve. For both `SUM`, `MAX` and `MIN` operations we detect a strong association with the predefined double and int datatypes. Only for `MIN/MAXLOC`, we detect use of derived datatypes.

User-defined reduction operations

We detected 83 call sites of `MPI_Op_create`. Of those operations, most are commutative ≈47% compared to only ≈20% non-commutative operations. The rest use variables our tool cannot resolve further (≈33%).

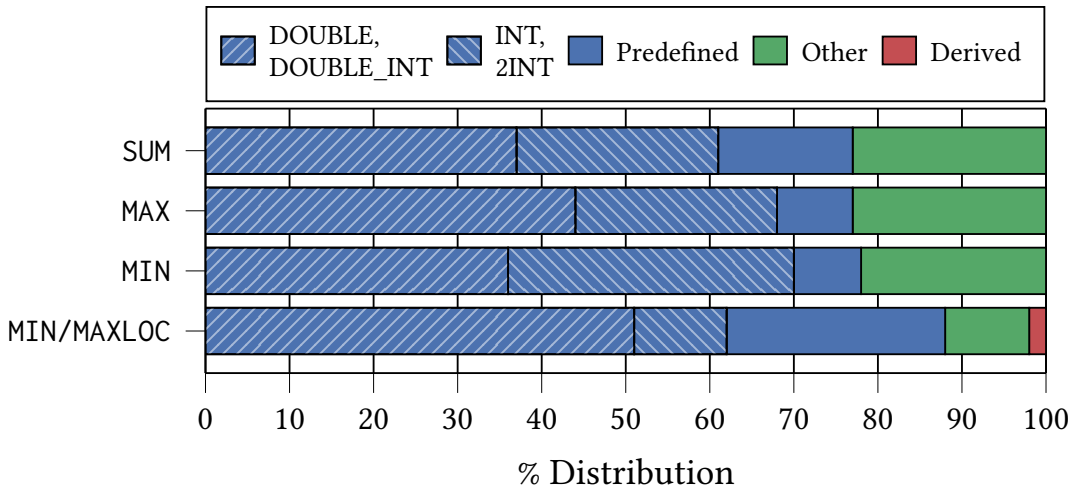


Figure 7.8.: Reduction routines and their datatype distribution. Crosshatched areas represent the most used predefined datatypes. For `MIN/MAXLOC`, in particular, it is `DOUBLE_INT` and `2INT`.

7.5.3. Communicators

In analyzing communicator usage, we omit analysis of the group creation functions for brevity.

Constructed Communicator

Figure 7.9 shows the communicator constructor function usage. `Comm_split` is the most used routine. Comparatively, more complex communicators find little use, e.g., `Intercomm` -related ones.

Communicator cross-use

Figure 7.10 shows communicator construction routines and their input handle. The *Other* category consists of variables that can not be cross-referenced. *Inconclusive* includes handles that our static analysis has identified as a communicator, but these are created by multiple communicator constructors, making them non-unique. In particular,

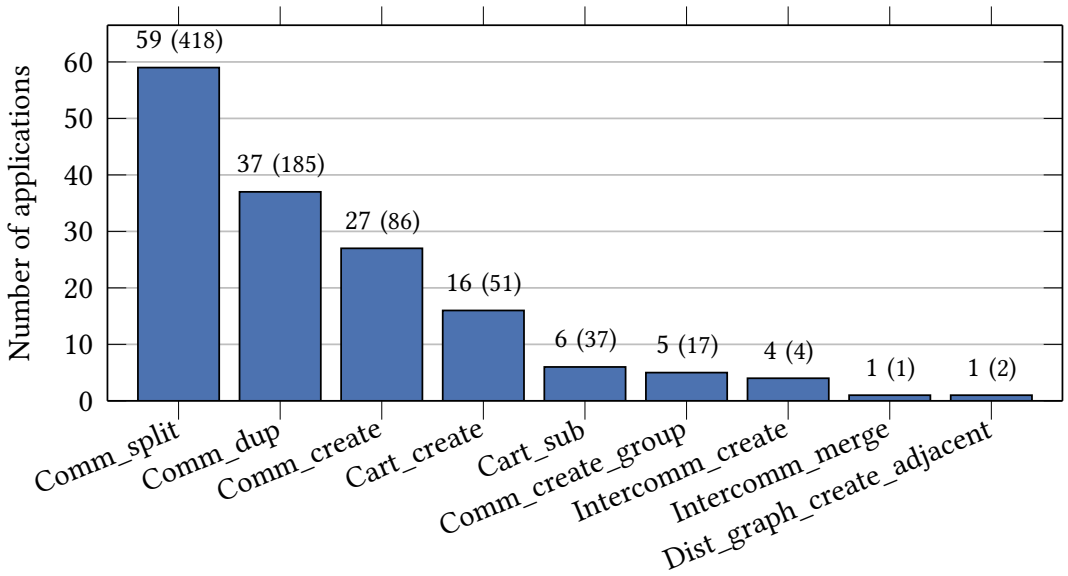


Figure 7.9.: Communicator constructor functions. For each constructor, we list the number of applications and, in parentheses, the total number of function call sites. For brevity, we removed the `MPI` prefix.

`COMM_WORLD` has a strong association with communicator construction. In contrast, nested communicator construction is detected less often.

Communicator usage

Three main categories of usage frequency exist, *Collective*, *PtP* and *Comm Group*. The latter category is omitted for brevity as we are not interested in analyzing query-like functions, e.g., `Comm_rank`. Figure 7.11 shows the use of communicators for *Collective* and *PtP* according to frequency of use. *VarU* and *VarP* are variables we cannot cross-reference, while the former also indicates communicator constructor function usage in codes. *Inconclusive* includes handles that our static analysis has identified as a constructed communicator, but these are created by multiple datatype constructors, making them non-unique. The predefined `COMM_WORLD` and the constructor function `Comm_split` are predominantly used in both categories.

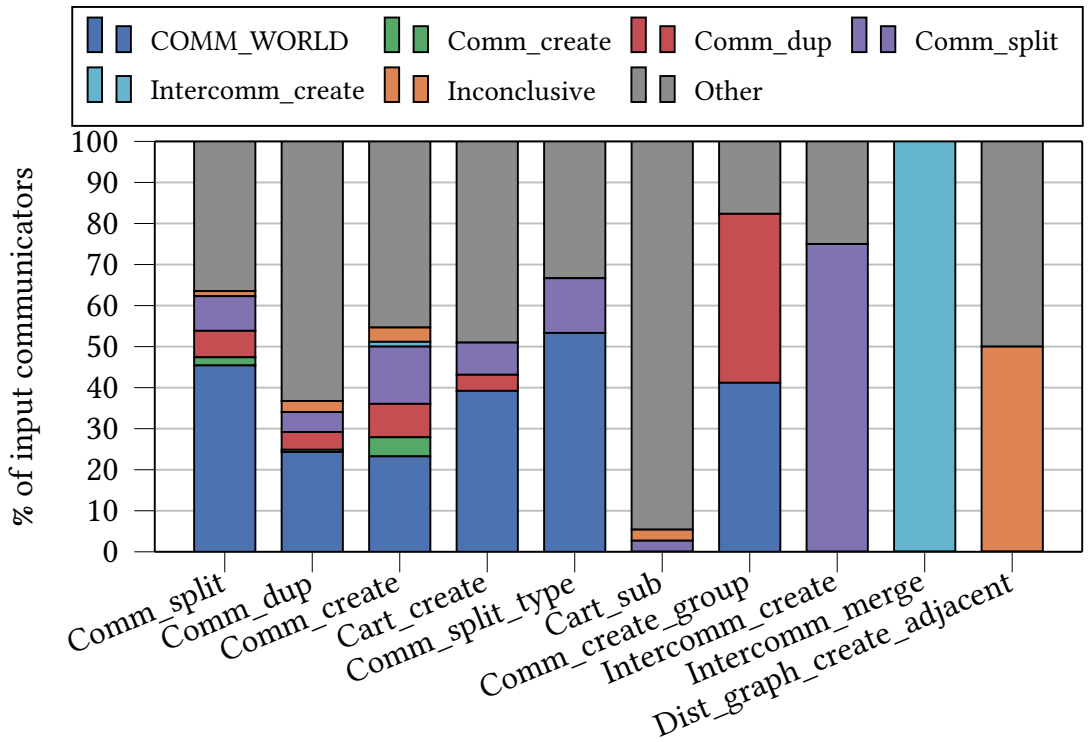


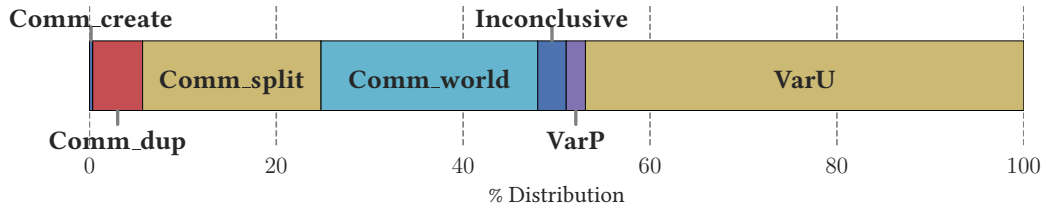
Figure 7.10.: Distribution of handles passed to constructors of communicators. For brevity, we removed the `MPI` prefix.

7.5.4. Communication value analysis

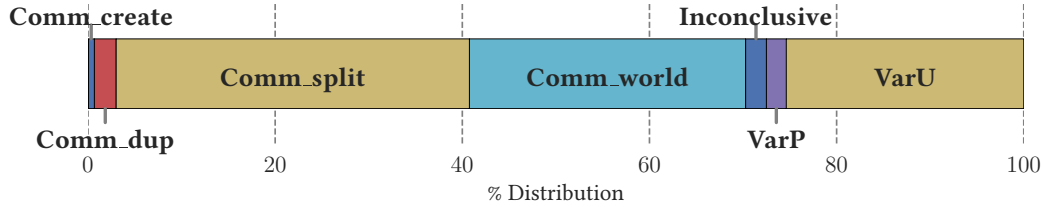
We analyze values relevant to communication, namely rank, message tag and (buffer) count variables.

Usage of rank

We look at all call sites. For *PtP*, we see overall use of the `MPI_ANY_SOURCE` wildcard in ≈3% of call sites in 23 codes. One code stands out as it uses the wildcard in ≈18% of calls, other codes have negligible use. The literal 0 is detected in ≈7% of call sites, other literals are detected less than one percent. In contrast, for *Collective*, ≈80% use the rank literal parameter 0. Other literal use is similarly negligible. While our tool does not



(a) *PtP*



(b) *Collective*

Figure 7.11.: Distribution of communicator use.

detect use of `MPI_PROC_NULL`, some codes (≈9%) assign this value to a variable. Hence, we suspect these codes may use it as the rank argument.

Usage of message tags

Few call sites use the `MPI_ANY_TAG` wildcard (≈1%) in 11 codes. Call sites use a tag literal in ≈41% of cases, with 0 being the predominant one at ≈40% of all literal usage.

Usage of count

For *PtP*, ≈30% use a literal as the count argument. Here, `1` is the most used value at ≈24% of all call sites. If a derived datatype is used, the literal 1 is used in ≈90% of respective call sites. In contrast, for *Collective*, the majority of call sites use the literal 1 (≈59%). Other literals are used at ≈8%, while the rest use some variable.

For *PtP* communications of floating-point types, constant count values are less prevalent (≈10%) than when communicating integer types (≈57%). This trend can also be observed for *Collective* to a lesser extent: ≈69% for floating point vs. ≈80% for integer types.

7.6. Reflecting on MPI correctness

This section discusses our MPI usage analysis results in the context of correctness. To that end, we briefly discuss identified coding patterns that we deem error-prone. We also analyze the recently developed MPI correctness benchmarks [134, 135, 143] w.r.t. their test coverage compared to the MPI usage in our analyzed set of codes. In particular, the MPI correctness benchmarks provide a test suite ranging from small-scale unit tests up to mini applications that contain use of MPI, either correct or erroneous, to assess the quality of MPI correctness tools like MUST [42]. Hence, these benchmarks are particularly useful if they reflect real-world usage of MPI.

7.6.1. MPI-related code smells

Our analysis revealed certain coding patterns that indicate *code smells*, i.e., code that may not follow best practices. For MPI, in particular, this includes resource leaks by not freeing previously constructed objects such as derived datatypes. Such insights provide value for teaching best-practices, especially in educational settings where teaching proper (MPI) resource management, such as de-allocating previously allocated objects, is crucial.

Table 7.3 summarizes our findings. Most issues are related to potential resource leaks, where we detect an allocation but the code contains no corresponding de-allocation call. In addition, two codes construct derived datatypes that require byte offsets without using the `MPI_Get_address` to calculate these offsets. This indicates portability issues, as other hardware platforms may generate different type layouts in memory.

Finally, we detected a coding pattern we consider problematic. A code sets the datatype for a send operation using a conditional:

```
MPI_Send(..., sizeof(size_t) == 4 ? MPI_INT : MPI_DOUBLE, ...);
```

We believe, this style can lead to issues as int and double types are fundamentally different.

Table 7.3.: MPI applications with code smells. Pair denotes codes where both creation and corresponding free functions are called. Single denotes codes where only the creation function is called, indicating a resource leak.

Function pairs needed together		Pair	Single
`Type_Commit`	free	33	10
Comm creation	free	46	13
`Op_create`	free	17	1
Persistent Operation	free	8	1
Win creation	free	5	1
`Type_create_h{...}`	`MPI_Get_address`	4	2
One-sided operation	synchronization	3	0

7.6.2. Lesson learned for correctness tooling

We applied our tool chain to the correctness benchmarks MPI-CorrBench [135] and the MPI Bugs Initiative (MBI, [134])[1] to assess their usage of MPI based on the provided test cases. For our analysis, we assume a MPI function to be covered, if it is called in the test cases of a benchmark suite. This, however, does not make a strong claim regarding completeness of the benchmarks as we count a function as covered if it exists irregardless of how thoroughly it is tested. A complex set of MPI calls may have multiple possibilities of erroneous use. To cover these, only one test case for a particular function may be insufficient. In particular, this also applies for the hybrid MPI+X model. CorrBench recently introduced tests for MPI+OpenMP [143]. Such combination of paradigms leads to an extended potential error set w.r.t. the *same* set of MPI function calls interacting in a multi-threaded environment. This recent effort is therefore unlikely to show up in our comparison, which regards the MPI function set covered.

Table 7.4 shows the comparison of both correctness benchmarks. As both benchmarks contain faulty and correct test-cases to assess correctness tools, we split the table in two parts. The upper half shows the test cases containing faulty MPI codes and the lower half shows correct ones. We note that CorrBench has more coverage for the correct test cases as they use a test suite of a MPI library implementation for their correct test set. In contrast, MBI has more coverage for the faulty test cases as they use a generator program to generate their test cases. To that end, a specific test will not only be performed with, e.g., the standard send operation but also all other modes of sending messages in MPI. CorrBench does not include unit-tests for the `Allreduce` operation, the most common

[1] Before our analysis of MBI, we generated their test cases, and removed the source code of the included MPI correctness tools.

Table 7.4.: Comparison of MPI-Corrbench's and the MPI Bugs Initiative's test case coverage w.r.t. the MPI functions used in our analyzed set of codes. `RC` is function usage in analyzed set and *covered* by correctness benchmark. `RnC` is usage in analyzed set but *not covered* by correctness benchmark. `nRC` is coverage by correctness benchmark but no usage in analyzed set. `topH` denotes the coverage in percent of the top half of functions, that are most commonly used in the set of codes. Higher numbers for `RC` and `topH` are better.

		CorrBench				MBI			
		RC	RnC	nRC	topH (%)	RC	RnC	nRC	topH (%)
Erroneous test cases	Overall	39	192	7	3	61	172	10	10
	PtP	9	22	0	60	11	20	0	53
	One-sided	7	16	0	55	9	14	0	72
	File I/O	0	31	0	0	0	31	0	0
	Collective	7	19	0	46	22	4	1	84
	Comm Group	3	21	0	25	7	22	0	42
	Datatype	6	15	0	60	3	18	0	30
	Topology	0	16	0	0	2	14	0	25
	Persistent	0	7	0	0	3	4	0	66
	Proc. Mgmt	0	7	0	0	0	7	0	0
Correct test cases	Overall	139	92	56	30	61	172	10	10
	PtP	27	4	0	93	11	20	0	53
	One-sided	21	2	10	100	9	14	0	72
	File I/O	0	31	0	0	0	31	0	0
	Collective	26	0	10	100	22	4	1	84
	Comm Group	15	9	1	67	7	22	0	42
	Datatype	20	1	4	90	3	18	0	30
	Topology	3	13	3	13	2	14	0	25
	Persistent	4	3	0	67	3	4	0	66
	Proc. Mgmt	0	7	0	0	0	7	0	0
		RC	RnC	nRC	topH (%)	RC	RnC	nRC	topH (%)
		CorrBench				MBI			

collective operation. However, CorrBench includes mutations to introduce errors to MPI mini-apps, where cases with the `Allreduce` operation are covered. We did not include these mini-apps in our assessment to maintain comparability.

For both benchmarks, the erroneous test cases contain few functions that are not used in our analyzed set of applications, hence showing overall relevancy of the test set. Both benchmarks lack tests for MPI File I/O although ≈19% of the analyzed applications use this feature, see Section 7.4.1. Likewise, tests for error handling are missing in both

benchmarks. This feature has relevance for the correctness benchmarks as the MPI correctness tool MUST lacks analysis support of this particular feature. To that end, if implemented in the correctness benchmark, MUST and other tools can validate their future implementation accordingly.

7.7. Conclusion

Our study presents an analysis of MPI usage in a large set of HPC codes by statically analyzing MPI function calls *and* their argument usage patterns. Previous work by Laguna et al. focused on MPI function calls without considering arguments. To the best of our knowledge, analysis of MPI codes by cross-referencing arguments (i.e., MPI handles) of function calls has not been reported.

Our analysis of MPI handle usage not only confirms the previous findings of Laguna et al., but also provides new insights. For instance, we discovered that (a) `MPI_Type_contiguous` is the most commonly used function for creating derived datatypes, (b) they are also used in nested contexts, (c) `MPI_SUM` is the most used reduction operation (≈60%) with the majority of reduction operations being reduced to rank 0 (≈80%), (d) a significant amount of PtP communication (≈40%) use constants values as message tags, (e) the `MPI_ANY_SOURCE` and `MPI_ANY_TAG` wildcards are rarely used in codes, and (f) communication operations involving derived datatypes are predominantly performed with a count argument of 1 (≈90%).

We also verified that all test cases featuring erroneous MPI usage in the recently developed MPI Correctness benchmarks are representative of real-world MPI usage. However, some features such as File I/O were noticeably absent in these benchmarks.

While some of our findings, such as reductions primarily targeting rank 0 or the predominant use of `MPI_SUM` as a reduction operation, may be anticipated, their confirmation can contribute to informed optimization for common MPI library usage. Consequently, our study can serve as a valuable resource for future research in the area of MPI library optimization, as well as in the area of MPI correctness checking. Our MPI usage collector tool chain is available at https://github.com/tudasc/mpi-arg-usage.

Part III.

Aiding Developers to write correct MPI Applications

8. MPI-CorrBench: Towards an MPI Correctness Benchmark Suite

Bibliographic Information

The contents of this chapter has previously been published in:

Jan-Patrick Lehr, Tim Jammer, and Christian Bischof. "MPI-CorrBench: Towards an MPI Correctness Benchmark Suite". In: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. virtual: ACM, June 2021. DOI: 10.1145/3431379.3460652

The contribution of the author of this thesis is summarized as follows:

As a co-leading Author, Tim Jammer contributed most level0 testcases (unit tests) as well as running the evaluation. Jan-Patrick Lehr contributed the level2 testcases (mutation of the mini-apps) as well as performing the performance tests on them. Tim Jammer and Jan-Patrick Lehr equally contributed to the categorization of MPI errors through mutual discussion. In terms of the Manuscript, Jan-Patrick Lehr mostly contributed the Background and Related work chapters, while Tim Jammer focussed on the evaluation and discussion chapters. The others chapters, notably chapter 3 concerning error categorization and correctness benchmark design, were developed through collaborative idea exchange. Christian Bischof provided invaluable feedback on the manuscript's text and the presentation of the results. All authors agree with the use of the publication for this thesis.

Abstract

The Message Passing Interface (MPI) is the de-facto standard for distributed memory computing in high-performance computing (HPC). To aid developers write correct MPI programs, different tools have been proposed, e.g., Intel Trace Analyzer and Collector (ITAC), MUST, Parcoach and MPI-Checker. Unfortunately, the effectiveness of these tools is hard to compare, as they have not been evaluated on a common set of applications. More importantly, well-known and widespread benchmarks, which tend to be well-tested and error free, were used for their evaluation. To enable a structured comparison and improve the coverage and reliability of available MPI correctness tools, we propose MPI-CorrBench as a common test harness. MPI-CorrBench enables a structured comparison of the different tools available w.r.t. various types of errors. In our evaluation, we use MPI-CorrBench to provide a well-defined set of error-cases to MUST, ITAC, Parcoach and MPI-Checker. In particular, we find that ITAC and MUST complement each other in many cases. In general, MUST works better for detecting type errors while ITAC is better in detecting errors in non-blocking operations. Although the most-used functions of MPI are well supported, MPI-CorrBench shows that for one sided communication, the error detection capability of all evaluated tools needs improvement. Moreover, our experiments reveal a MPI standard violation in the MPICH test suite as well as several cases of discouraged use of MPI functionality.

8.1. Introduction

The Message Passing Interface (MPI) [66] is the de-facto standard for distributed computations in high-performance computing (HPC). Its application programming interface (API) and the semantics of the respective functions is defined by the standard. Different implementations exist, of which some are open source, e.g., OpenMPI [19], while others are closed source, e.g., Cray MPI. Programmers rely on the specified functions to parallelize their application.

The MPI standard is relatively low-level, as users have to handle many details explicitly. For example, a user needs to specify array lengths and data types for the underlying buffer to be sent, as presented in Listing 8.1. This is particularly error-prone in large software packages that, additionally, need to be configurable for various data types as, e.g., in mixed-precision computing.

The challenges of developing parallel programs with low-level APIs make tool-supported correctness checking desirable, if not necessary. In particular, identifying the existence of errors can be challenging, such as errors that come from wrong memory layout assumptions. These types or errors, among others, were identified in [21] as some of the most important ones to application developers.

Both static, e.g., Parcoach [57] or MPI-Checker [59], as well as dynamic, e.g., MUST [42] or Intel Trace Analyzer and Collector (ITAC) [153], MPI correctness-checking tools have been proposed. The tools differ in approach, focus and capability, hence, their evaluation uses different selected sets of application. Such applications can be special test cases, as in [116], or a selection of well-known MPI programs, as in [59]. This diversity, however, makes a reasonable comparison of their capabilities hard, if not impossible.

```
1 int b[3] = {1, 2, 3};
2 // Assume target and tag hold valid values
3 MPI_Send(b, 4, MPI_INT, target, tag, MPI_COMM_WORLD);
```

Listing 8.1: Erroneous `MPI_Send` that reads beyond the array bounds of `b`, i.e., four instead of three `MPI_INT` elements.

For programming aspects and paradigms outside of MPI, such correctness benchmark suites have been proposed and proven useful. DataRaceBench [85] is a collection of OpenMP-parallel programs which enabled a comparison of different data-race detection tools [107], or the development of new data-race detection tools, e.g., ROMP [93]. It also motivated work on other, similar, benchmark suites, e.g., DataRaceOnAccelerators [127].

The idea of tools that assist programmers write correct programs of high code-quality is not unique to the field of HPC, hence, benchmark suites for their evaluation have been proposed. Examples are benchmarks to evaluate static-analysis tools for web security [97], as well as benchmarks for design pattern [48] or clone detection tools [58]. The Rigorous Examination of Reactive Systems (RERS) challenge [168] provides a forum to compare solvers for various program verification tasks. It is comprised of sequential linear temporal logic (LTL) [1] and reachability solvers, and, more recently [96], parallel LTL and computation tree logic (CTL) [2] solvers.

The MPI library, however, has received most attention for performance evaluation. Benchmark suites for MPI, e.g., the Intel MPI benchmarks [163], have been proposed and used to measure and compare performance [40]. A variety of performance benchmarks exist that are designed to exhibit the different performance characteristics of the various

implementations of the MPI standard. However, to the best of the authors' knowledge, no structured benchmark suite to evaluate existing MPI correctness tools has been proposed.

In this paper we address this situation by proposing the MPI correctness benchmark suite *MPI-CorrBench*. MPI-CorrBench can be used to evaluate MPI correctness tools for both their capability to detect various types of errors as well as their induced performance penalty. For MPI-CorrBench's development, we analyze the MPI standard's API for potential erroneous usage. From this analysis, we derive error categories that we use subsequently to classify the various individual benchmark programs. As of now, the benchmark suite consists of 308 small-scale tests that individually provoke one specific error. In addition, a selected set of mini-apps is added to evaluate a tool's ability to identify errors in more complex situations, and to evaluate time and memory overhead, respectively.

MPI-CorrBench covers error cases for the most commonly used MPI functions [105], i.e., it covers point-to-point communication (both blocking and non-blocking), collective communication and derived data types. In addition, MPI-CorrBench implements test cases for the less frequently used one-sided communication.

In our evaluation, we use MPI-CorrBench to evaluate the four available tools ITAC, MPI-Checker, MUST, and Parcoach. This evaluation is, to the best of the authors' knowledge, the first structured comparison of these MPI correctness tools.

Hence, the main contributions of this paper are

- an analysis of potential errors in the usage of the MPI API and a subsequent error category model,
- a benchmark suite containing small-scale programs that provoke errors from all categories, together with mini-apps that allow the analysis of runtime and memory overhead, and,
- an evaluation of the four MPI correctness tools MUST, ITAC, MPI-Checker and Parcoach.

The paper is structured as follows: We present background information and related work in Section 8.2. Thereafter, we develop the error category model and introduce MPI-CorrBench in Section 8.3. We evaluate the four state-of-the-art MPI-correctness tools in Section 8.4, and discuss our findings in detail in Section 8.5. Lastly, we conclude our work and outline future research in Section 8.6.

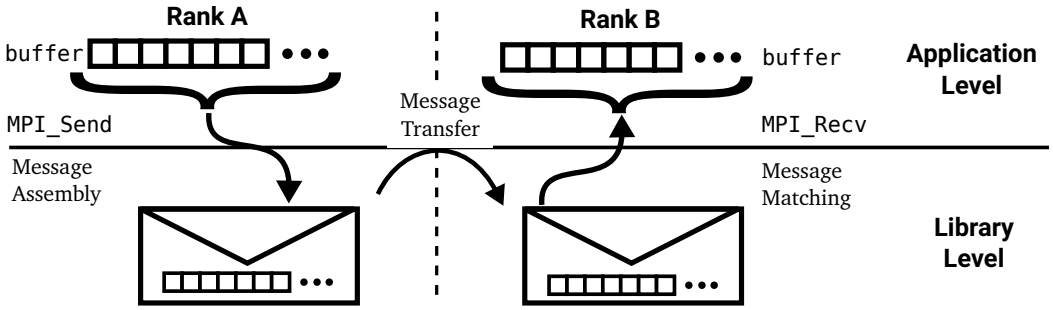


Figure 8.1.: Simplified schematic of MPI data transfer between two communicating ranks. The memory region `buffer` from `Rank A` is copied into a message with its envelope and sent to the receiving process. There, it is matched and disassembled into the memory region of receiving `Rank B`.

8.2. Background and Related Work

First, we explain in some more detail relevant parts of the MPI standard. Second, we introduce general concepts for program analysis and outline different existing correctness approaches for MPI. Lastly, we briefly explain the current state of practice w.r.t. the evaluation of MPI correctness tools.

8.2.1. Message Passing Interface

The Message Passing Interface (MPI), is a standardized API that was introduced to consolidate the, at that time, existing different libraries for distributed memory computing [5, 6], and specifies how messages between participating processes should be exchanged. Sending and receiving messages is performed by participating processes, e.g., point-to-point communication involving two processes or collective communication involving potentially more processes. The MPI operations specify various information, e.g., the message's data type or its tag. One part of the message transfer, see Figure 8.1 for a schematic, is the message matching [66, §3.2.3] which allows selective message reception, e.g., based on the tag value.

For the correctness of an MPI program it is crucial that the specified arguments passed to the MPI functions match both the intention of the programmer and the state of the program. As an example, specifying the wrong MPI data type for the memory buffer

used can result in program crashes or silent errors. The latter potentially leads to data corruption, which, in the worst case, will lead to undiscovered wrong computation results.

The MPI standard offers features for error handling. While this is useful in some cases, e.g., for file input/output the MPI library can issue an `MPI_ERR_NO_SUCH_FILE` error, its applicability for writing correct MPI programs is quite limited. The MPI standard states: "The set of errors that are handled by MPI is implementation-dependent. [...] Specifically, text that states that errors *will* be handled should be read as *may be* handled." [66] In particular, an MPI implementation may not be able to detect errors, or the error may be catastrophic, preventing the application to continue in a consistent state [66]. Therefore, one cannot solely rely on an MPI implementation to find misuses of the MPI API.

To discover potential misuse of the MPI API, various MPI correctness tools have been proposed. We give an overview of approaches and tools proposed in the following section, and group them by their general analysis approach. Thereafter, we briefly outline the current state of practice when evaluating newly proposed approaches.

8.2.2. MPI Correctness Approaches

We introduce various (tool-supported) approaches to correctness checking of MPI applications. Throughout, we use the terms approach and tool interchangeably, as all approaches are tool supported, if not, an actually implemented tool. In general, four different categories of tools for the correctness analysis of MPI programs can be distinguished. The classification depends on the general analysis approach (static or at runtime), if the analysis is performed stateless or stateful, and whether human interaction is required to a certain extent. For the latter, we consider fully-automatic and partly-automatic in the following classification.

Stateless Static Analysis

As stateless static analysis we consider techniques that analyze the source code, or some other representation, e.g., the LLVM IR[1], of the program, without executing it. Tools that employ stateless static analysis can discover local errors, e.g., the violation of the type signature of an API function prototype. Hence, the most common stateless static analysis is the contextual analysis a compiler performs during parsing of the input.

[1]The intermediate representation used in the LLVM compiler infrastructure framework. https://llvm.org/docs/LangRef.html

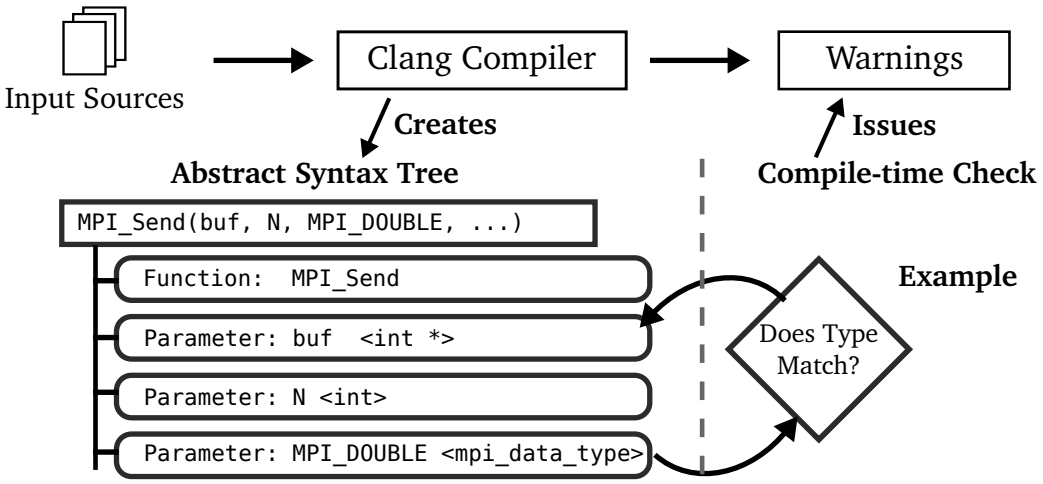


Figure 8.2.: MPI-Checker parses the application's sources into an abstract syntax tree (AST) and performs static checks on this representation to generate warnings.

MPI-Checker [59] is a stateless static analysis tool of which some checks are now available within clang-tidy.[2] It performs its analyses on the program's abstract syntax tree built by the Clang compiler. Hence, it can detect type violations, e.g., passing a `long int` argument where an `int` is expected. Since the tool has knowledge about the MPI semantics, it can also detect certain mismatching argument combinations, see Figure 8.2. Consider a call to the `MPI_Send` routine that is passed an `int*` as the send buffer to be used. However, the user specifies the MPI data type `MPI_DOUBLE` as the type of the buffer elements. Given that MPI-Checker knows about the semantic connection between these arguments, it can determine the type of the buffer and the meaning of `MPI_DOUBLE`, and issue an error.

An extension to Parcoach [57] for correctness checking of hybrid MPI+X applications that relies on stateless static analysis is presented in [76]. It builds the control-flow graph (CFG) of the target application as the basis for its analyses, see Figure 8.3. Using the CFG, the tool determines if MPI functions are called from concurrent regions to check for its conformity with the MPI threading requirements. Moreover, it analyzes the CFG for calls to MPI collective communication routines along different paths. Should the tool detect missing calls, it issues a warning for a potential deadlock and inserts instrumentation to validate its occurrence at runtime.

[2] clang-tidy, available as part of Clang/LLVM: https://clang.llvm.org/extra/clang-tidy

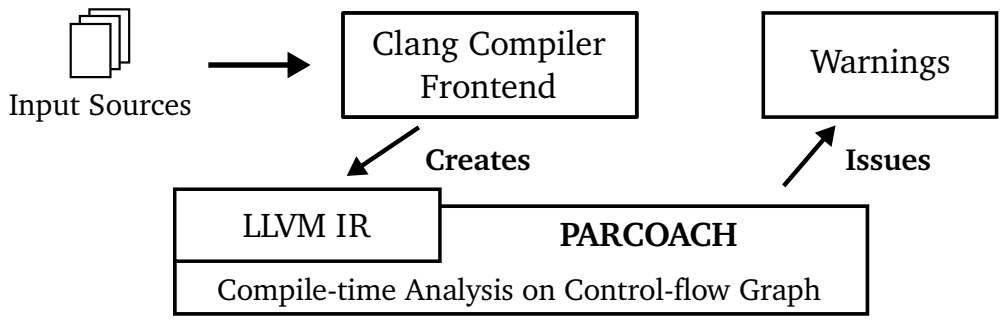


Figure 8.3.: Parcoach operates on LLVM IR and performs control-flow graph analyses to determine if collective operations can be called from diverging program paths.

The main limitation of stateless static analysis is its inability to differentiate the different executions of program paths depending on the state of the program. Hence, errors that depend on the data flow cannot be detected reliably for specific situations, but only generically. Consider a program in which the size of the buffer is determined dynamically through different program paths. Stateless static analysis is unable to determine whether a path through the program exists that determines the buffer size erroneously.

Stateful Static Analysis

Introducing state into static analysis can overcome the limitations outlined in the previous section. While the target program is still not executed, it allows the exploration of multiple paths through the program. The variety ranges from model-checking approaches to symbolic execution. In typical model-checking approaches, properties of the target program are formulated as, e.g., temporal logic formulas, stating allowed states of the program. The model-checking software then tests if the different paths through the application fulfill the logic formulas.

Early work to apply model checking to the correctness properties of an MPI application is presented in [20]. The authors use an example implementation for an MPI-parallel (heat) diffusion equation solver and model its respective communication scheme to be verified by SPIN [8]. A more general modeling of MPI execution semantics was presented in [23], and demonstrated on a simple Jacobi iteration target program. The work was continued and extended for certain MPI non-blocking communication routines in [31].

In [60], the authors propose to use symbolic execution to verify MPI-parallel programs. Their approach builds on top of the Klee [32] symbolic execution engine, and extends it for the selection and scheduling of virtual MPI processes.

An example of a model-checking approach to the correctness analysis of MPI applications is SimGridMC presented in [88]. It uses the SimGrid [55] simulator to explore the different states of the target application and compare them, e.g., with a particular temporal logic formula. SimGridMC re-uses the possibility of SimGrid to intercept MPI communication and steer the target application, making it possible to execute the different possible paths through the program.

Model checking in combination with symbolic execution is used in the verification backend of CIVL [67, 87]. The framework provides a general abstract concurrency model that the verifier is based on. Any particular concurrency model, such as MPI, is translated into this abstract representation. This allows the authors to prove different properties in the target application, such as deadlock detection or invalid pointer dereferences. Their approach, however, requires the target program to be written in the specific CIVL-C dialect, hence, it does not allow to execute arbitrary MPI programs.

Runtime Analysis

Runtime analysis can be performed online or offline. The latter case requires the application to be executed, although the actual analysis is performed after execution.

An example of such an offline-analysis approach is presented in [79]. It proposes a propositional encoding scheme that uses partial ordering to prove the absence of deadlock states. Therefore, the tool executes the MPI application and collects a trace that it subsequently translates into the proposed encoding to check the encoding's satisfiability. This allows the authors to proof the correctness of the target application w.r.t. deadlock freedom.

Online analysis tools like MUST [42], MPI-CHECK [17] or Umpire [13] analyze the MPI usage at runtime to detect if errors happen. They provide special libraries that are loaded when executing the target application. These libraries wrap the regular MPI function calls and intercept them at runtime, see Figure 8.4. Hence, the tools first perform the error checking and then continue with invoking the original MPI implementation for the actual communication.

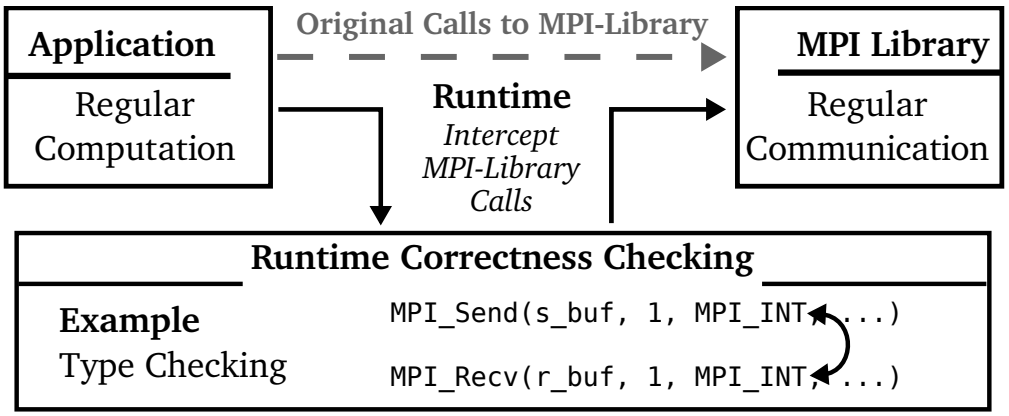


Figure 8.4.: Dynamic tools typically intercept MPI function calls at runtime for correctness checking, e.g., if the declared MPI types between send and receive operation match.

Debugger

Apart from (mainly automated) tools that directly focus on the detection of (in)correct MPI usage, debuggers like TotalView [124] or DDT [111] can help in pointing out errors. However, their main use-case is to find the root cause of an error once its existence is identified. As such, debuggers are less helpful in automatically detecting the presence of an MPI usage error.

Still, in large applications, a wrong usage of the MPI library may result in complex non-deterministic synchronization errors. As it is typically cumbersome to debug non-deterministic behavior, approaches as presented in [75] exist, which enforce the consistent occurrence of synchronization errors. Using this, the error manifests itself deterministically and developers can reliably use a debugger in order to determine the root cause and address it.

While debuggers are valuable, in this paper, we exclude them from consideration, as we focus on tools that automatically point out MPI usage errors. Given that some of the approaches presented earlier in this section are not publicly available or require extensive manual specification and re-implementation, we consider the tools MUST, ITAC, Parcoach, and MPI-Checker in our evaluation.

8.2.3. Current Practice of Correctness Tool Evaluation

Some papers showcase their tool by applying it to well-known benchmark or production codes, e.g., the NAS Parallel Benchmark suite [18], or selected SPEC MPI [37] and other mini-apps. As expected, these well known production codes are mostly correct, as they have been used and tested for extended periods and by different people. As a consequence, however, the insights in terms of the tool's ability to detect errors in the given target codes are limited. While sometimes a new tool finds previously unknown bugs in such a code [13, 59], this type of evaluation is better suited to determine the performance impact of the proposed tool. Nevertheless, the comparability of the various tools' performance impact results is limited as there is no standard set of benchmarks and configuration that were used. It would, thus, be valuable to have a common set of benchmark codes that are run with equal configuration to determine the different performance impact of the available tools.

To demonstrate the successful application of the proposed approach, most papers, e.g., the presentation of Umpire [13], use a set of manually created example applications. The quality of such an evaluation varies from one example only, e.g., the application of MUST in a domain-specific setting [114], up to an own set of benchmark codes [116]. Although testing a tool against a large number of manually created examples is suited to show the feasibility of a proposed approach, it lacks the comparability against other approaches, as they were demonstrated on other manually created examples. This impediment is amplified as the target codes used in either respective evaluation may focus on different aspects of the MPI standard. We are convinced that it is the comparability aspect of testing that is crucially important if the basic design of two approaches differs. Hence, a common set of test cases that tools can be evaluated against is desirable.

From a user perspective an evaluation of the false positive rate is highly desirable. False positives, i.e., false reports of an error, hinder the practical application of an automated approach, as developers need to manually validate every tool-issued error or warning. With a high false positive rate, this becomes highly frustrating and developers lose trust in the helpfulness of the tool. Therefore a tool with a high false positive rate may disqualify itself for practical use in more complex scenarios. As a consequence, a set of evaluation test cases should always contain non-erroneous test cases that try to trick tools into potentially reporting false positives.

8.3. Correctness Benchmark

In this section, we analyze at what level errors in the usage of MPI can occur, and use the results to derive error categories. These error categories are the basis of the final small-scale correctness benchmark applications of MPI-CorrBench.

One can approach the analysis for potential errors from multiple angles. The first angle is an erroneous usage of the provided API in the sense that wrong values are passed as parameters. This approach is similar to how a programmer or a static code analysis tool analyzes the code and looks for errors. The second angle is to consider the effect of a misused call to a library function. In the case of MPI, such an effect may be a process crash or data corruption due to wrong usage of a receive operation. The last angle resembles the point of view of a dynamic analysis tool that intercepts the calls to those library functions, focusing on the effect of a wrongful interplay of the different MPI functions e.g. by provoking a deadlock. In the latter case, it is important to note that some of the originally available information, e.g., type information about memory buffers in the application, are lost due to the MPI interface.

These general considerations w.r.t. errors are contrasted by technology aspects that increase analysis complexity for certain analysis approaches. The most obvious technological aspect is the programming language used, e.g., C or Fortran. Other aspects include the structuring of programs in multiple modules or translation units, which may increase the implementation complexity for static analysis tools. As these technical details influence the usefulness of a tool for production environments significantly, we include such challenges into this initial version of the benchmark suite.

Similarly, the rich feature set of the MPI standard makes the development of an MPI correctness benchmark suite challenging, when all possible features and their potential combinations should be considered. A similar complicating factor is the combination of MPI with other programming models, i.e., MPI+X. One such example is to use OpenMP for threading within the application. We acknowledge that these scenarios are important, yet see them as orthogonal to the core of MPI-CorrBench. Hence, in this initial version of the benchmark suite, we focus on the use of MPI only, and, in particular, we focus on the most commonly used MPI features pointed out in [105]. As already mentioned, the addition of less commonly used features is of high interest.

Figure 8.5 shows the layered conceptual architecture for MPI-CorrBench. In this paper, we provide explanations, implementations, and an evaluation of levels zero and two. Level zero contains small-scale benchmarks that are similar to unit tests, i.e., each benchmark

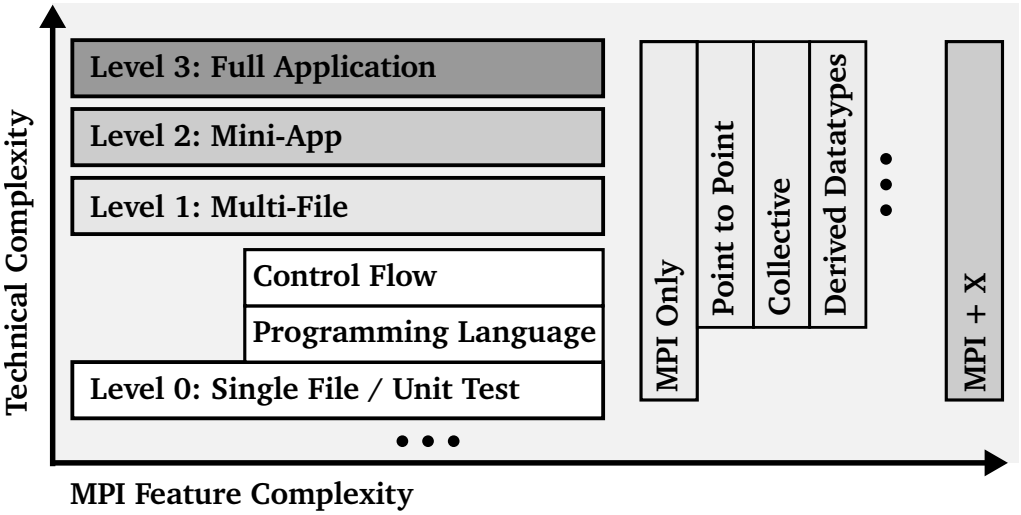


Figure 8.5.: Schematic of the benchmark suite hierarchy. The complexity on the x axis account for complexity of MPI features, while the y axis accounts for technical complexity.

provokes exactly one particular error. Level two contains three existing mini-apps that are used without modification as well as different mutations that provoke different MPI errors. In this initial version of MPI-CorrBench, level one is left out intentionally, as we think that the mini-apps currently used within MPI-CorrBench are mostly simple enough to provide insight into a tool's support for multi-file tests. Nevertheless, MPI-CorrBench considers simple, multi-file tests that are similar to unit tests as an important steppingstone to be included in the future.

In the subsequent sections, our approach to identifying misuses is more closely related to the first of the three angles mentioned. Hence, we analyze and categorize the different errors from an API-use perspective rather than from an effects, or a dynamic tool's perspective, unless necessary, e.g., a potential program deadlock.

Generally, some of the errors outlined here may seem trivial in the isolated and small code snippets. However, in a large and complex code base with control flow reaching across multiple translation units or numerous compile-time configuration possibilities, such errors can be hard to identify.

```
1 int *buffer = NULL;
2 /* n denotes the number of elements to send from within the buffer */
3 MPI_Send(buffer, n, MPI_INT, target_rank, tag, MPI_COMM_WORLD);
```

Listing 8.2: Erroneous call to `MPI_Send`. The `NULL` value is passed as the data buffer, which may result in a segmentation fault when the memory is accessed.

8.3.1. MPI Error Cases and Categories

We analyze the MPI interface to derive potential misuses of its provided functionality, categorize the identified types of errors and use the categories for the creation and organization of MPI-CorrBench. At the high level, we consider *erroneous arguments*, *erroneous program flow*, and *mismatching arguments across communication calls* as the key error types. We present examples for these errors together with a detailed explanation in the next sections.

Erroneous Arguments

Erroneous arguments are errors local to one specific function call and are not particular to MPI. Such errors can occur at any (low-level) interface, e.g., when using checkpoint/restart libraries [117]. Typical errors may include passing uninitialized memory, or confusing two parameters and passing them in the wrong order. In some cases, the compiler can emit a warning, since the argument types do not match. In other cases, much more elaborate analysis is necessary to identify the potential error.

Listing 8.2 shows a simple error, in which a `NULL` pointer is passed to an MPI function. Should the length argument `n` be zero, then this is a benign error, which does not manifest itself during the execution of the program, and is explicitly allowed by MPI. However, in the case that the number of elements to send, denoted by `n`, is larger than zero, the program crashes with a segmentation fault.

Another example for erroneous arguments is given in Listing 8.3. In this case, the `buffer` is a valid memory region, holding `n` elements of type `int`. However, the `MPI_Recv` call specifies a buffer size of `x` elements of type `int`. Should the matching `MPI_Send` indeed transfer `x` elements, and `x` is greater than `n`, the receive operation will write beyond the allocated memory of `buffer`. Depending on the particular layout in memory, this can result in corrupted data that was laid out in memory after the allocation of `buffer` while

```
1 /* A matching MPI_Send operation sending x elements somewhere in the code. */
2 int *buffer = (int*)malloc(n*sizeof(int));
3 /* x denotes the maximum number of elements to receive into the buffer. */
4 MPI_Recv(buffer, x, MPI_INT, source_rank, tag, MPI_COMM_WORLD);
```

Listing 8.3: Erroneous call to `MPI_Recv`. The `buffer` can hold `n` elements of type `int`, but the user specifies to receive up to `x` elements of type `int`. Should `x` be larger than `n`, the program writes beyond the limit of the allocation of `buffer`.

```
1 /* Assume in and out to be allocated correctly */
2 MPI_Reduce(in, out, 1, MPI_INT, MPI_REPLACE, 0, MPI_COMM_WORLD);
```

Listing 8.4: Erroneous call to `MPI_Reduce`. The passed MPI operation `MPI_REPLACE` is not valid for this MPI collective function.

the application continues, or a segmentation fault. Especially the application continuing with corrupted data is dangerous and undesirable.

We consider the former cases as errors in which a single argument passed to the library function is invalid. In addition to these case, an illegal argument combination can be specified, as shown in Listing 8.4. It is important to note that illegal argument combinations are still local to a particular function call. The example shows that the user wants to perform an `MPI_Reduce` operation. However, the provided `MPI_REPLACE` is an invalid operator for an MPI reduction, hence, this forms an illegal argument combination. The MPI standard explicitly lists all allowed operations along with the allowed data types for its collective communications [66, §5.9.2].

It is apparent that many more such erroneous argument cases exist for the MPI API. Hence, we are not giving examples for all of them for brevity, and refer the reader to the benchmark suite available at https://github.com/tudasc/mpi-corrbench.

Mismatching Arguments

This category extends the *Erroneous Arguments* category to error cases in which the arguments between multiple **matching** MPI functions do not match. An example of such a case is given in Listing 8.5. Therein, the declared data types of a matching pair of `MPI_Send` and `MPI_Recv` do not match.

Please note that this category of errors is not limited to matching send and receive operations. Listing 8.6 gives an example of a much more intricate mismatching arguments

```
1 int *ib = (int*)malloc(n*sizeof(int));
2 double *db = (double*)malloc(n*sizeof(double));
3 if (rank == source_rank) {
4   MPI_Send(ib, n, MPI_INT, target_rank, tag, MPI_COMM_WORLD);
5 } else if (rank == target_rank) {
6   MPI_Recv(db, n, MPI_DOUBLE, source_rank, tag, MPI_COMM_WORLD);
7 }
```

Listing 8.5: Mismatching arguments of declared MPI data type between matching `MPI_Send` and `MPI_Recv`.

```
1 MPI_Win w;
2 int *b = (int*)malloc(NB); // NB = N*sizeof(int)
3 MPI_Win_create(b, NB, MPI_INFO_NULL, MPI_COMM_WORLD, &w);
4 MPI_Win_fence(0, w);
5 if (rank == src_rank) {
6   int local_buf[N] = {0};
7   /* access start at position 5 so it will be out of the target processes win */
8   MPI_Put(&local_buf, N, MPI_INT, tgt_rank, 5, N, MPI_INT, w);
9 }
10 MPI_Win_fence(0, w);
```

Listing 8.6: Mismatching arguments for `MPI_Win_create` and `MPI_Put` that lead to an out of bounds RDMA access.

error for a remote direct memory access (RDMA) operation. While the size of the `window` and the `MPI_Put` operation match, the `MPI_Put` operation is erroneous. Its access starts at position 5 instead of position 0, which leads to an illegal access outside of the target process' window.

Erroneous Program Flow

As another category of errors that can occur within MPI applications, we identify the case that the program flow leads to erroneous situations. This category includes cases in which a process of a communicator never reaches a collective communication, or a process calls the required MPI functions in the wrong order. We also consider fully absent calls to required MPI functions as a special case of a wrong call order. Depending on which functions are called in the wrong order, the observable results range from silent errors to deadlocks or program crashes.

A fairly simple error that disrespects the requirement of all ranks within a communicator

```
1 /* Assume rank, loc, gs are reasonably initialized */
2 if (rank != root) {
3   MPI_Reduce(&loc, &gs, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
4 }
```

Listing 8.7: Control flow excludes the `root` process from the collective `MPI_Reduce`, which leads to a deadlock.

```
1 if (rank == root) {
2   MPI_Request req;
3   MPI_Status stat;
4   MPI_Isend(buf, n, MPI_INT, tRank, tag, MPI_COMM_WORLD, &req);
5   for (int i = 0; i < n; ++i) { buf[i] += i; }
6   MPI_Wait(&req, &stat);
7 }
```

Listing 8.8: Erroneous placement of call to `MPI_Wait` resulting in the process potentially modifying the content of `buffer` while it is being sent.

to participate in the same collective operation is shown in Listing 8.7. Listing 8.8, however, shows a potentially silent error, in which a process invokes a non-blocking `MPI_Isend` operation. Since the call immediately returns to the caller, program execution continues. The misplaced call to `MPI_Wait` can result in a silent error and corrupt the application data, as the write to the memory of `buffer` changes the data while it is being transferred.

8.3.2. Benchmark Suite

For the MPI-CorrBench implementation, it is important to consider different MPI features as the benchmark suite serves two purposes. First, it allows a structured comparison of correctness tools w.r.t. their handling of the most commonly used [105] MPI features. Second, it allows to test MPI correctness tools with advanced features, and help in their development by identifying blind spots. Hence, MPI-CorrBench consists of the test cases and available driver code to ease the evaluation of additional MPI correctness tools.

The MPI-CorrBench driver is implemented in a modular way to make the integration of new tools as convenient as possible. For the level zero test cases, it automatically (1) creates an execution directory per testcase, and copies the testcase's source file to the directory while renaming it to `testcase.c`, (2) applies the MPI correctness tool by invoking a user-provided execution-script therein, and, (3) collects the result into a single json file for visualization, using a user-provided `parse_output` function. The

Table 8.1.: Overview of level zero test cases. Erroneous Arguments (Err-Arg), Mismatching Arguments (Mis-Arg), and Erroneous Program Flow (Err-PF) are designed to provoke the specific error. Correct denotes the number of test cases for this category taken from the MPICH2 test suite.

Category	Err-Arg	Mis-Arg	Err-PF	Correct
Collectives	72	8	9	72
Point to Point	83	17	17	40
RMA	33	2	22	73
User types	34	7	4	18

execution script contains the tool-specific instructions, e.g., special compilation flags, on how to apply the MPI correctness tool. The `parse_output` function is given the testcaes's execution directory. It then indicates this testcase's result by reading the tools output and returning predefined values for an identified error, an issued warning, a successful execution without warning or error, or a failed execution. Finally, the results are output into a json file, which is the basis for the graph visualization. Thus, to include a new tool for level zero testing, only two scripts are required to be implemented. Application of tools to the level two testcases requires additions to the mini-app Makefiles.

The MPI-CorrBench testcases are organized according to the layered schematic depicted in Figure 8.5. Its basic *level zero* builds the foundation and is comparable to unit tests, i.e., test programs that are of very low complexity and provoke exactly one error. Nevertheless, test cases at level zero can still make use of the full feature set of the respective host language they are implemented in. The organization of level zero into several small unit tests allows for a more nuanced comparison of the different tools, as one may compare each specific error individually, in contrast for example counting the number of detected errors in a larger scale test case. Table 8.1 shows the number of test cases in level zero. All erroneous test cases are newly developed to provoke a specific error.

Starting from level one, the test cases can be split into multiple input files. We have added this particular step, as it commonly poses a challenge to static analysis tools. Level two consists of mini-apps, i.e., small programs that have some properties of a production application, yet do not exhibit their full complexity. Finally, level three is reserved for full-scale and production applications.

In its current form, MPI-CorrBench consists of 510 test cases in level zero and uses the mini-apps LULESH [52], AMG [86] and Kripke [65] as a basic level two coverage.

Most of the level zero test cases cover MPI functionality pointed out to be the most frequently used in open-source software [105]. Moreover, we include tests for one-sided communication to provide an initial assessment of the different MPI correctness tools w.r.t. their capabilities of these more elaborate features of MPI. In addition to the cases with false MPI usage, the level zero tests include relevant test cases from the MPICH2 [14] test suite, which we used to present the MPI correctness tools with presumably correct test cases. We selected parts of the MPICH2 test suite as their implementation in many small test inputs fits our unit-test scheme well. This means that we incorporated the implementation of the relevant functions of its test environment directly into the header file, so that only one object file per test case is needed. Having a significant number of correct cases allows to estimate false positive rates a tool may produce.

While AMG and LULESH were used in MPI correctness related research papers before [57, 95, 114], Kripke was selected as another C++ benchmark that poses performance challenges on tools that use instrumentation. For each of the included mini-apps, MPI-CorrBench contains manually created *mutations* that introduce a particular error into the application, which are explained in the next paragraphs. The mutations concern both point-to-point communication and collective communication. While the errors are different for the respective mini-app, we aim to provoke similar errors in the applications, in order to establish a structured pattern. All test cases and mutations are created w.r.t. MPI in its current version 3.1. This required the creation of small patches to the AMG mini-app, which used MPI derived data type symbols that were removed in MPI 3.0.

AMG2013 Mutations In the AMG mini-app, we created a mutation for a count mismatch error in a `MPI_Bcast` operation during initialization of the grid, in which two instead of one `int` elements are broadcast. This mutation results in a silent error on our system. Also, we created a mutation for an erroneous root in a call to `MPI_Bcast`, which results in a program crash. Furthermore, we created two mutations that concern the data type. One mutation changes the data type from `long int` to `int`, which results in a silent error, and the other changes the data type from `long int` to `char`, which results in a program crash on our system.

For point-to-point communication, we created mutations that (1) result in an exceeded receive buffer, which leads to a program crash on our system, (2) confuse destination rank and tag, which leads to a deadlock in the particular case and on our system, and, (3) specify the wrong type of elements, i.e., `MPI_FLOAT` instead of `MPI_DOUBLE`, which leads to a program crash on our system.

Kripke Mutations The mutations in Kripke are similar to those in AMG. For the collective communication in Kripke, we created mutations that (1) passes a too large number of elements to a `MPI_Allreduce` function, (2) two cases of a wrong buffer element type, i.e., `MPI_INT` and `MPI_SHORT` instead of `MPI_LONG`.

The point-to-point error mutations implement (1) tag errors, in which erroneously the target ranks are sent as tag values, instead of the computed tag values, (2) an illegal rank error, in which the wrong data member of Kripke's communication class is passed to the underlying MPI function, and (3) a data type mismatch between `MPI_Isend` and `MPI_Irecv`, in which `MPI_DOUBLE` is sent, but `MPI_FLOAT` is received. While the tag errors result in a deadlock, both illegal rank and type errors result in an application crash on our system.

LULESH Mutations In LULESH, we introduce mutations that affect an `MPI_Allreduce` with a count error, i.e., two instead of one data element are reduced, together with mutations that affect the point-to-point communication. Here, two mutations result in sent or receive buffer being exceeded by the number of data elements specified. This leads to program crashes on our system.

8.4. Evaluation

In our evaluation, we apply the MPI correctness tools MUST, ITAC, Parcoach and MPI-Checker to our benchmark suite. All measurements are performed on the Lichtenberg II HPC system. Each node is equipped with two Intel Cascade Lake AP 9242 processors and 384 GB of main memory. MUST is used in its current release version 1.6.0 with the GNU compiler in version 8.3 and OpenMPI in version 4.0.1. For ITAC, we use the Intel compiler and IntelMPI in version 2020.4. MPI-Checker and Parcoach[3] are using Clang/LLVM 10.

For MUST, we run one additional process that is reserved for MUST, while the benchmarks themselves are run with two MPI processes. The low number of processes is sufficient to provoke the respective errors, and no performance studies are performed with the level zero benchmarks. We execute all benchmarks at least five times to account for non-deterministic errors. In these cases, the error can occur non-deterministically, and we consider the tool to successfully detect an error, if it detects it at least once. However, in our evaluation ITAC and MUST detected the errors, if at all, then in the majority

[3]Used most recent `git` commit `7aaae65f` from https://github.com/parcoach/parcoach

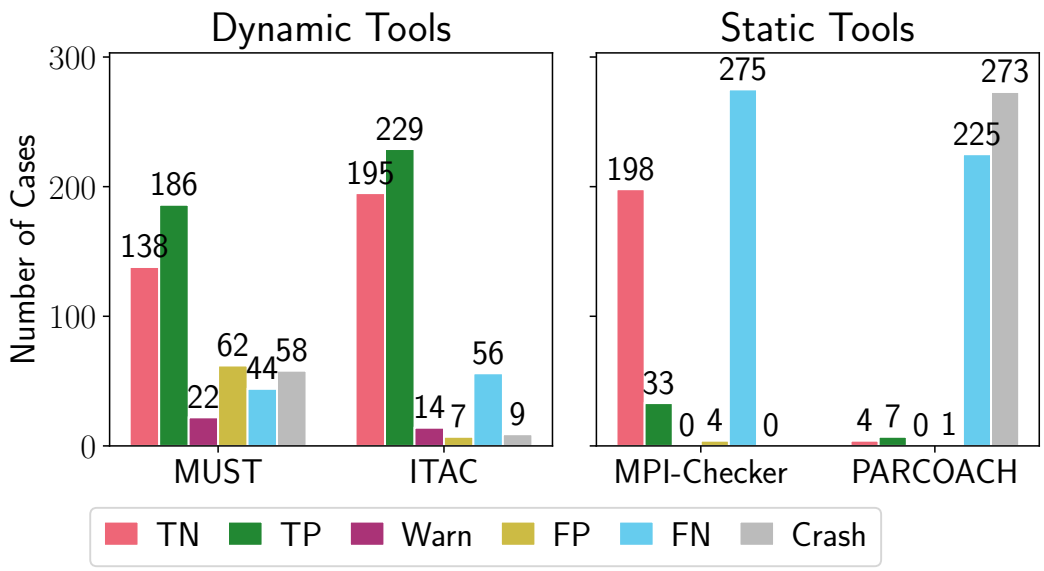


Figure 8.6.: Overall error detection results for MUST, ITAC, MPI-Checker and Parcoach. The total number of tests is 510.

of executions. This means that non-deterministic occurrence of error state was not problematic.

After the level zero test cases, we apply all tools to the level two test cases, i.e., the mini-apps and the respective mutations. MPI-CorrBench defines the setup of the mini-apps to use 8 processes and a problem configuration that ran for at least two minutes on our test system. While we are aware that modern system consists of hundreds and thousands of processors, initial experimentation is valid with smaller process counts.

In the subsequent sections, we first discuss each tool's capabilities to correctly detect the given errors, while the second part briefly evaluates the dynamic tools' performance.

8.4.1. Correctness

Figure 8.6 shows the error detection results for each of the analyzed tools. We compare the dynamic-analysis tools MUST and ITAC in the left chart and the static-analysis tools MPI-Checker and Parcoach in the right chart. The Y-axis denotes the number of level-zero benchmarks classified into the different classes:

Table 8.2.: Error detection results for the dynamic-analysis tools: true negative (TN), true positive (TP), false positive (FP), and false negative (FN). Crashed means the tool produced no interpretable output. The total number of tests is 510.

	MUST				ITAC			
	TN	TP	FP	FN	TN	TP	FP	FN
Point to Point	28	89	12	6	39	96	1	6
Collective	45	70	25	5	68	79	4	8
User Types	14	27	4	14	16	27	2	16
RMA	51	0	21	19	72	25	0	26
Total	138	186	62	44	195	229	7	56
	Crashed in 58 Cases				Crashed in 9 Cases			
	Warned in 22 Cases				Warned in 14 Cases			

- **True Negative (TN)**: The tool issues no error and no error is expected.
- **True Positive (TP)**: The tool detects the expected error.
- **Warning issued**: The tool issues a warning about suspicious behaviour when an error is present in the code.[4]
- **False Positive (FP)**: The tool discovers an error although no error is expected.
- **False Negative (FN)**: The tool does not detect the expected error.
- **Tool Crashed**: The execution of the tool fails for some reason and no (helpful) output is produced.

As mentioned in Section 8.3, we organized the benchmarks according to the different MPI features covered. The results are listed in detail according to this categorization in Table 8.2 for the dynamic tools and Table 8.3 for the static tools respectively[5]. We will take a closer look how the tools compare for each of these categories throughout the following subsections. However, we defer a closer discussion, especially of false positives, to Section 8.5.

[4] MPI-Checker and Parcoach do not differentiate between error and warning.

[5] For Table 8.2 and 8.3, if a tool only raised a warning about suspicious behaviour, this does not count as detecting an error.

Table 8.3.: Error detection results for the static-analysis tools: true negative (TN), true positive (TP), false positive (FP), and false negative (FN). Crashed means the tool produced no interpretable output. The total number of tests is 510.

	MPI-Checker				Parcoach			
	TN	TP	FP	FN	TN	TP	FP	FN
Point to Point	38	17	2	98	0	0	0	113
Collective	72	16	0	73	0	7	0	80
User Types	16	0	2	45	0	0	1	30
RMA	72	0	0	57	0	0	0	0
Total	198	33	4	275	4	7	1	225
	Crashed in 0 Cases				Crashed in 273 Cases			

Point-to-Point Communication

Figure 8.7 shows the different tool's error detection results for point-to-point communication. We see that ITAC and MUST detected the majority of errors injected, with only a limited number of false positives detected.

The tools ITAC and MUST are mostly complementary w.r.t. the errors detected. MUST missed some errors regarding non-blocking communication, e.g., a corrupted buffer due to a misplaced `MPI_Wait`. ITAC, in contrast, missed some "silent" type or count errors/mismatches between send and receive operations, as, e.g., shown in Listing 8.9. The call to `MPI_Recv` specifies that the buffer offers space for at least $2 * N$ elements, while it has only the size of N. This is an error according to the MPI standard. However, as only N elements will be received, this is unlikely to cause any out of bound access.

In a few cases both ITAC and MUST's execution failed and produced no output. In all these cases, the MPI library implementation aborted as a forbidden MPI Function was called before `MPI_Init`, and exited with a rather helpful error message. Additionally, in some cases which involve misuse of `MPI_Request` objects, ITAC aborted with a segmentation fault and no further information.

MPI-Checker's static approach also detected some errors, such as missing calls to `MPI_Wait`, or various type mismatches. As Parcoach is developed solely for collective operations, we did not expect it to catch any errors for point-to-point communication.

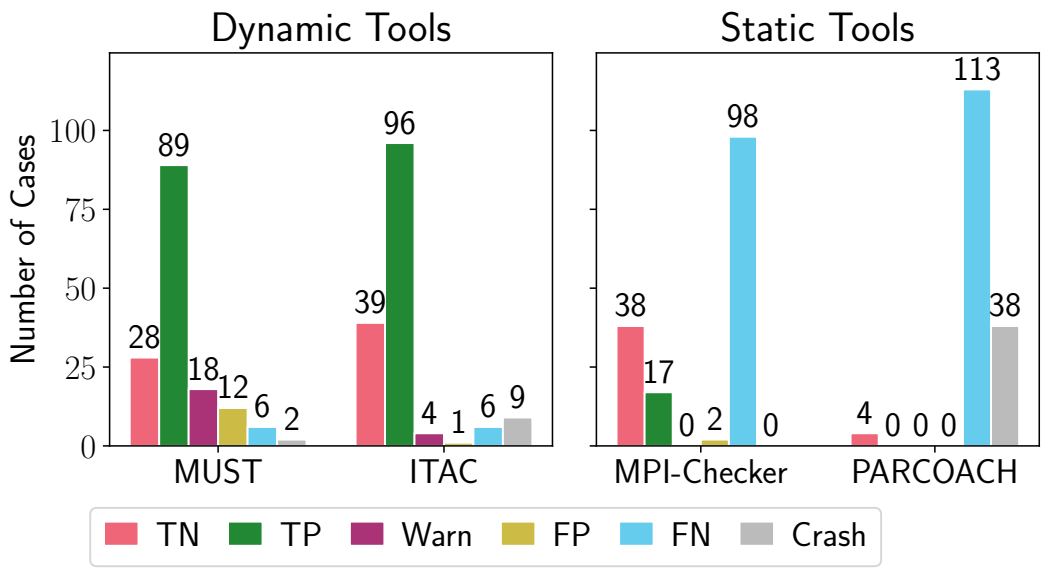


Figure 8.7.: Error detection results for point-to-point communication

Collective Communication

For the collective operations, the comparison of the tools is shown in Figure 8.8.

One can see that MUST was in some occasions not able to produce useful output. This occurred in some cases when (1) the MPI implementation catched the error and aborted, or (2) the error led to an application crash, for instance, due to a segmentation fault as the result of a `NULL` pointer being provided as receive buffer. Finally, both tools missed errors that concern buffer-type mismatches, with MUST detecting more of such errors.

```
1 int b[N] = {0};
2 MPI_Status stat;
3 if (rank == 0) {
4   MPI_Send(b, N, MPI_INT, 1, 9, MPI_COMM_WORLD);
5 } else if (rank == 1) {
6   MPI_Recv(b, 2*N, MPI_INT, 0, 9, MPI_COMM_WORLD, &stat);
7 }
```

Listing 8.9: "Silent" buffer-usage error, as only N integers will be received, an out of bounds buffer access is unlikely

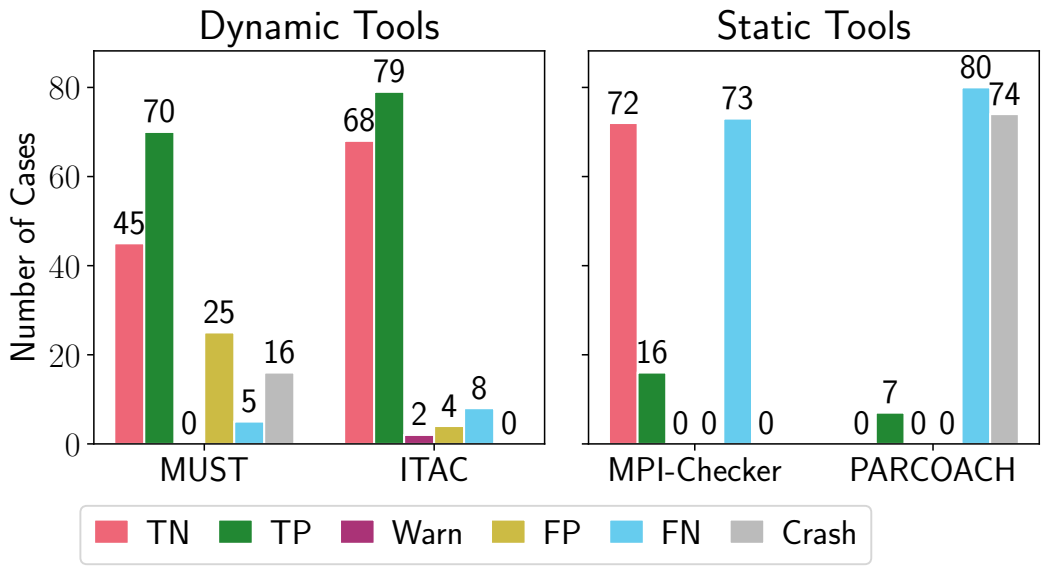


Figure 8.8.: Error detection results for collective operations

Parcoach indeed detected errors that lead to a deadlock, which is what the tool is developed for. However, it did not detect these errors in all occasions. Examples are mismatching root processes in a reduce operation or different communicators being supplied. Additionally, many more argument errors were not caught, but these are not part of Parcoach's design. MPI-Checker was only able to detect two of the deadlock test cases. However, it detected various type mismatch test cases correctly. Unfortunately, Parcoach was unable to analyze a variety of test cases, as support for some MPI functions used therein is missing, e.g., `MPI_Initialized`.

User Defined Types

For the category of user-defined types the tools are compared in Figure 8.9. We see that ITAC and MUST are quite comparable in the detection of type errors. Both tools also miss mostly the same types of errors, i.e., a struct type is created that does not match the underlying C-struct layout. In the user-defined type test cases, MUST issued more warnings about suspicious behavior compared to ITAC.

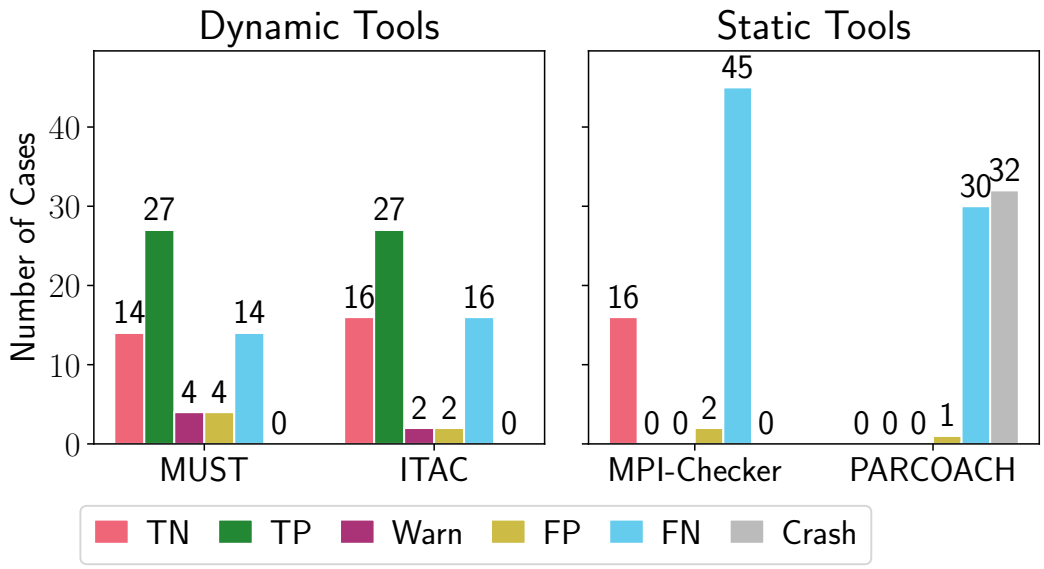


Figure 8.9.: Error detection results for operations with user-defined types

One Sided Communication

As no support for one-sided communication is implemented in Parcoach or MPI-Checker, they were not able to detect any errors of this category, see Figure 8.10.

MUST also has no support for one-sided communication, hence, it did not output useful information when the application crashed due to an error. Despite its lack of support of these operations, it reported some errors, i.e., false positives, as it cannot detect the proper usage of the one-sided operations. For example, MUST reported the error of an unknown request being used in an `MPI_Wait` call, and cannot detect that the request was used in an RMA operation.

ITAC detected some errors for one-sided communication, e.g., invalid arguments are used or wrong synchronization leading to a deadlock. However, its error detection capability is not as good as with other tested MPI features.

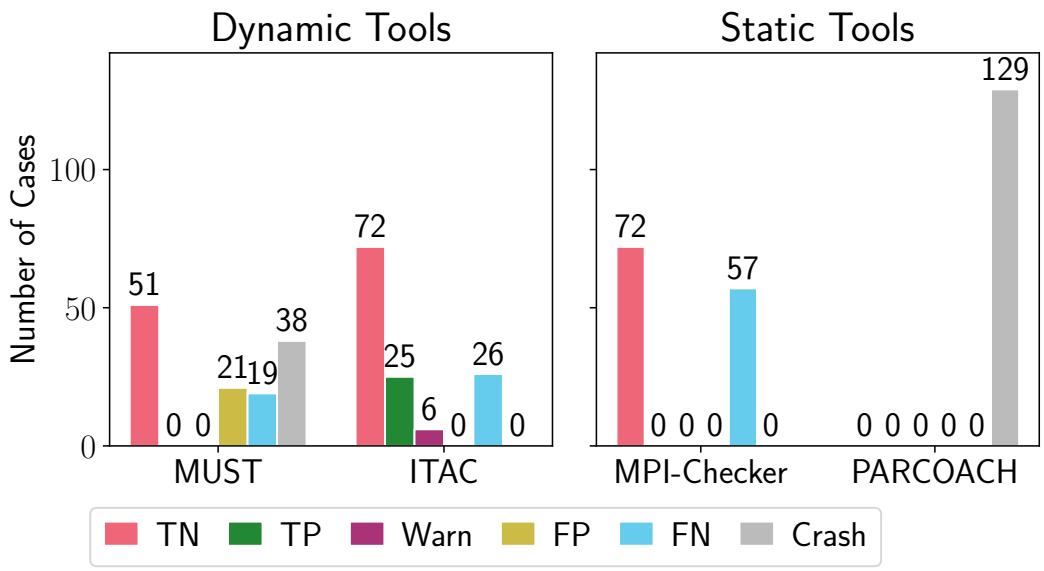


Figure 8.10.: Error detection results for one-sided communication

8.4.2. Runtime Performance Impact

Performance is also an important factor when evaluating different tools, especially for tools that detect errors at the applications runtime. A comparison of the runtime overhead is shown in Figure 8.11. We see that ITAC has less performance overhead. We also note that MUST's additional process causes a constant memory overhead of roughly 2MB on our system. ITAC has no relevant additional memory overhead for the benchmark configurations that we used. Especially with the micro benchmarks it is important to note that in our experiments MUST always needs a considerable initialization time compared to the micro-benchmark's runtime.

8.5. Discussion

We see that MUST and ITAC perform quite well overall. However, we expected a lower rate of false positives, and the MPICH test suite to be standard compliant.

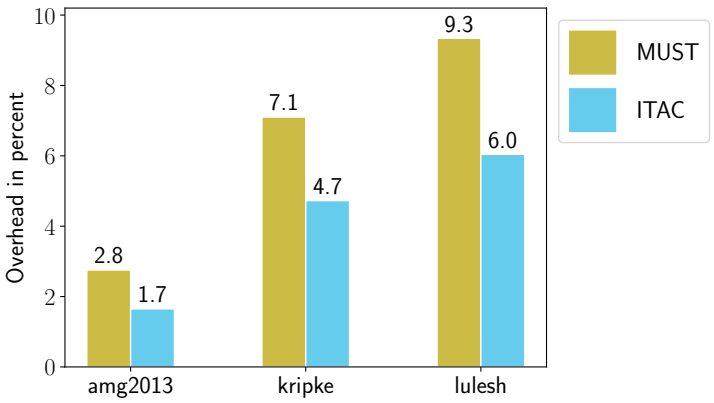


Figure 8.11.: Overhead of MPI correctness checking. With baseline runtime for (AMG / Kripke / LULESH) for MUST (199s/ 121s / 166s) and ITAC (189s / 158s / 143s).

```
1 int rbuf[10];
2 if (rank == 0) {
3   MPI_Request r[size];
4   for (int i = 0; i < size; i++) {
5     MPI_Irecv(rbuf, 10, MPI_INT, i, tag, MPI_COMM_WORLD, &r[i]);
6   }
7   MPI_Waitall(size, r, MPI_STATUSES_IGNORE);
8 }
```

Listing 8.10: Overlapping receive buffer may result in data corruption.

One example where both ITAC and MUST detect an error is shown in Listing 8.10.[6] A `DATA_TRANSMISSION_CORRUPTED` is detected by ITAC while MUST points out that "This communication overlaps with the other communication at position [...]". The MPICH test suite frequently uses the construct with a single receive buffer for the messages from all other ranks. As the MPI standard states that one "should not access any part of the receive buffer", we think that ITAC and MUST are correct marking this as an error. For all other occurrences of this construct, ITAC only gives a warning and no error.[7] MUST always considers such overlapping buffers an error. This partly explains the higher false positive rate compared to ITAC. Strictly adhering to the standard, i.e., "should

[6]The original code from the MPICH test suite slightly changed for brevity.

[7]The test cases use uninitialized message buffers for which sending zeros is quite probable, hence, no data corruption occurs.

not" in contrast to "must not", Intel's approach is correct. Nevertheless, in our opinion, overlapping buffers are definitely a bad practice and considering it as an outright error is valid.

Additionally both tools fail in a few special, technically legal, edge cases, e.g., using a collective operation with a send count of 0. We think that reporting such cases as an error is a helpful practice, as these special cases will rarely be of meaningful use in real applications. In contrast to the flaw in the MPICH test suite that we pointed out earlier, this is valid, albeit uncommon, MPI. Hence, its implementation's correctness needs to be tested.

MUST and ITAC detected a MPI standard violation in a MPICH test case when -1 was used as send count in a collective operation.[8]

Apart from that, MUST also flags some errors when there is no violation of the standard. MUST detects several deadlocks when sending in the buffered mode (`MPI_Bsend`) although no deadlock can happen, as enough buffer space was supplied. As there is the possibility of a deadlock, if an insufficient buffer space is supplied in these cases, a warning instead of an error might be the clearer solution. It seems that MUST sometimes confuses inter- and intra-communicators. A call to `MPI_Comm_remote_size` may lead to error messages similar to "Argument 1 (comm) is an intercommunicator and was used where an intracommunicator was expected". MPI-CorrBench also revealed an error in ITAC as in one case it finishes with the message "ERROR: Unknown error. Please file a bug report."

Overall, we think that the false positives of ITAC and most false positives of MUST are actually justified. MUST follows a stricter approach by flagging more suspicious behavior as error instead of only issuing a warning.

Table 8.4 shows the commonly used evaluation metrics of precision, recall and accuracy. As MUST and ITAC differentiate between an error and a warning, we give the metrics for the cases that treat a warning as error (W+) or no-error (W-). We note that the interpretation changes the tools' scores, making a comparison of these metrics challenging, because the standard does not state at which point a bad practice should result in a warning.

When using the tools on the mini-apps LULESH, AMG and Kripke, we did not find the error detection capabilities of ITAC and MUST to be severely limited by the scope of this

[8] Although the comment reads that it "is a very weak sanity test that all non-blocking collectives specified by MPI-3 are present", it is still a standard violation

Table 8.4.: Precision, recall and accuracy of the tools: Warning treated as error (W+) and no-error (W-). MPI-Checker and Parcoach do not differentiate between warning and error.

	Precision		Recall		Accuracy	
	W+	W-	W+	W-	W+	W-
MUST	0.719	0.719	0.686	0.655	0.655	0.642
ITAC	0.781	0.781	0.696	0.673	0.722	0.717
MPI-Checker	0.891	–	0.107	–	0.452	–
Parcoach	0.875	–	0.030	–	0.046	–

programs. As expected a tool could not suddenly detect an error it missed in the level zero test cases. For example, MUST still misses cases when the buffer of a non blocking operation is modified before it has completed, while ITAC detects those cases. Both tools cannot detect application bugs. For example, if one wants to send a particular part of an array to another process, wrongfully increasing the number of elements to send and receive will likely result in an application bug, but no MPI standard violation. Therefore coming up with meaningful mutations to the mini-apps that consistently violate the MPI standard can be quite challenging.

Applying the static tools to the larger codebase of the mini-apps was more challenging than using the dynamic tools, as changes to the build systems were necessary. We personally find the output of MPI-Checker for complex control flows hard to comprehend, as a notice is given for each branch that might lead to the error. Additionally, MPI-Checker's analysis is bound to one translation unit at a time making, the detection of an error that arises in the interplay of the different parts impossible. Parcoach's implementation suffers the severe limitation that its analysis aborts if an unknown function is present, which apparently also applies to some LLVM internal functions (e.g. `llvm.lifetime.start.p0i8`). Therefore, in its current state, its usage for larger code bases is quite limited.

We also took a look at the error messages in order to check if they are helpful by actually pointing out the cause of the error. Although this is a highly subjective judgement, we find that ITAC was in some rare cases not able to produce a helpful error message compared to MUST. Section 8.4 mention some rare cases where ITAC aborts with a segmentation fault, without any hint on the location of this segmentation fault. Apart form that, in some cases involving an invalid message length, ITAC pointed out a deadlock, with one process waiting to receive the invalid message and the other process waiting in

`MPI_Finalize`, while MUST directly pointed to the wrongful sending operation. Therefore, in our opinion, MUST's error messages more frequently directly point to an errors cause. But ITAC's error messages also point out an errors cause quite directly in all but a handful of cases.

We see that all tools are missing support for one-sided communication. While some support for user-defined types is present, many errors are not detected. In contrast to the point-to-point and collective operations, MUST and ITAC do not complement each other and both miss several errors especially in the false creation of struct types. For this purpose TypeART [95] has been suggested as an extension to MUST. As it is currently not part of the release version of MUST, we did not cover it in this study.

8.6. Conclusion

MPI application development is challenging due to the complexity of distributed systems development and MPI's low-level API. To aid developers in using MPI correctly, different MPI correctness checking tools have been proposed. Dynamic tools, such as MUST and the Intel Trace Analyzer and Collector (ITAC), operate at the target application's runtime, while static tools, such as Parcoach and MPI-Checker, operate at compile time. So far, these tools were not compared on a common set of test cases or applications. This makes a fair comparison of their capabilities impossible.

To overcome this impediment, we propose MPI-CorrBench – a MPI correctness benchmark suite dedicated to evaluate the capabilities of MPI correctness tools. MPI-CorrBench contains both erroneous cases as well as correct cases, of which the latter are taken from the MPICH test suite. Hence, the benchmark suite consists of different layers of complexity to enable the evaluation of tools w.r.t. (1) particular error cases on unit-test-like programs, (2) their ability to handle certain technical challenges, e.g., multi-file test cases, and, (3) application-level error reporting. MPI-CorrBench is available as open source at `https://github.com/tudasc/mpi-corrbench` and we welcome contributors to extend its coverage.

Using MPI-CorrBench, we were able to evaluate the aforementioned tools on a common set of benchmarks and test cases. Our key findings are that the currently available dynamic tools MUST and ITAC are better prepared for production use than the static tools Parcoach and MPI-Checker. Moreover, we identified a violation of the MPI standard in MPICH's test suite as well as several cases which the MPI standard discourages.

In more detail, we found that MUST and ITAC complement each other in many cases. MUST works better for the detection of type-related errors while ITAC works better for the detection of errors in non-blocking operations. Parcoach was specifically developed to detect deadlock errors in collective communications. While it does detect most errors, it falls short to detect some more intricate errors, e.g., a deadlock when different communicators are used. MPI-Checker reports only very little false positives, while also finding some of the errors correctly. However, the larger number of false negatives, i.e., the tool fails to detect the error, makes us believe that the true negative rate is partly a by-product.

We find that the tools work well for the most used MPI features, while support for other MPI features, such as the creation of struct types or one sided communication, is limited.

8.7. Future Work

One of the most important future directions for MPI-CorrBench is to include Fortran test cases, as well as to improve and extend the coverage of more elaborate MPI features. In particular, extending the benchmark suite to include MPI+X test cases is important. Moreover, the inclusion of mutation testing practices, to generate more mutants, and the investigation of other tools is of interest.

With new versions of the MPI standard, MPI-CorrBench should be extended to account for newly added features. One potential addition for the upcoming MPI 4.0 standard is the introduction of test cases to evaluate whether analysis tools can correctly detect if a target application can employ relaxed message ordering hints.

9. Towards a Hybrid MPI Correctness Benchmark Suite

Bibliographic Information

The contents of this chapter has previously been published in:

Tim Jammer, Alexander Hück, Jan-Patrick Lehr, Joachim Protze, Simon Schwitanski, and Christian Bischof. "Towards a Hybrid MPI Correctness Benchmark Suite". In: *EuroMPI/USA'22: 29th European MPI Users' Group Meeting*. Chattanooga, TN: ACM, Sept. 2022. DOI: 10.1145/3555819.3555853

The contribution of the author of this thesis is summarized as follows:

As the co-leading author, Tim Jammer contributed a majority of the testcases, alongside the evaluation technique and the implementation of the evaluation process. Alexander Hück reviewed - and contributed additional - testcases. Simon Schwitanski contributed by providing his experience regarding data race detection for one-sided MPI operations as well as running the evaluation. Joachim Protze contributed insights on the correctness Tools perspective. In Terms of the Manuscript, Tim Jammer focused his contribution on the Evaluation and discussion chapters. Joachim Protze contributed Chapter 3 about the analysis tools considerations, while the other Chapters were mainly contributed by Alexander Hück. Jan-Patrick Lehr and Christian Bischof provided invaluable feedback on the manuscript's text and the presentation of the results. All authors agree with the use of the publication for this thesis. After publication, Joachim Protze changed his name to Joachim Jenke.

Abstract

High-performance computing codes often combine the Message-Passing Interface (MPI) with a shared-memory programming model, e.g., OpenMP, for efficient computations. These so-called hybrid models may issue MPI calls concurrently from different threads at the highest level of MPI thread support. The correct use of either MPI or OpenMP can be complex and error-prone. The hybrid model increases this complexity even further. While correctness analysis tools exist for both programming paradigms, for hybrid models, a new set of potential errors exist, whose detection requires combining knowledge of MPI and OpenMP primitives. Unfortunately, correctness tools do not fully support the hybrid model yet, and their current capabilities are also hard to assess. In previous work, to enable structured comparisons of correctness tools and improve their coverage, we proposed the MPI-CorrBench test suite for MPI. Likewise, others proposed the DataRaceBench test suite for OpenMP. However, for the particular error classes of the hybrid model, no such test suite exists. Hence, we propose a hybrid MPI-OpenMP test suite to (1) facilitate the correctness tool development in this area and, subsequently, (2) further encourage the use of the hybrid model at the highest level of MPI thread support. To that end, we discuss issues with this hybrid model and the knowledge correctness tools need to combine w.r.t. MPI and OpenMP to detect these. In our evaluation of two state-of-the-art correctness tools, we see that for most cases of concurrent and conflicting MPI operations, these tools can cope with the added complexity of OpenMP. However, more intricate errors, where user code interferes with MPI, e.g., a data race on a buffer, still evade tool analysis.

9.1. Introduction

The Message-Passing Interface (MPI, [136]) is the de-facto standard for distributed memory computing in high-performance computing (HPC). HPC applications can have more than hundreds of thousands of lines of code, and use complex MPI communication patterns to achieve the most efficient distributed computations. At the same time, the MPI application programming interface (API) is low-level and type-unsafe. Data is transferred as a type-less pointer to a buffer. The user has to specify (1) the length and datatype manually and, also, (2) process-communication-related information (e.g., destination and message tagging). Such an interface is error-prone.

To illustrate the underlying complexity of correctly using MPI, we note that several previously dormant bugs in popular MPI mini-applications widely used for, e.g., benchmarking in HPC, have been uncovered only after several years of existence [13, 59]. Unfortunately, MPI users can expect little error checking by the MPI library to find such bugs, neither for, e.g., (1) datatype safety [136, Section 4.1.12], nor, more generally, for (2) consistent, reliable error handling [136, Section 8.3]:

> "The set of errors that are handled by MPI is implementation-dependent. [...] Specifically, text that states that errors *will* be handled should be read as *may* be handled."

As a consequence of this complexity, numerous MPI correctness checker tools have been developed, e.g., [42, 57, 59, 153]. In particular, they target errors due to misuse of the MPI library, such as (1) deadlocks resulting from MPI communication calls, (2) erroneous datatype usage [95], or (3) missing MPI resource de-allocations.

However, the coverage and effectiveness of these MPI correctness checkers are not straightforward to compare for users as they have not been applied to a common, structured test suite. Hence, we developed *MPI-CorrBench* [135] and others developed the *MPI Bugs Initiative* [134] as an MPI correctness checking test harness allowing for such a comparison. MPI-CorrBench and the MPI Bugs Initiative both use small-scale tests and mini-applications that contain erroneous use of the MPI library, including illegal value ranges of MPI arguments, deadlocks and data races of non-blocking communication. Such a standardized benchmark allows the users to identify the best performing tool for their use-case. Equally important, it enables MPI correctness tool developers to improve their implementations based on a structured test-suite that covers many aspects of likely MPI misuse.

For so-called *hybrid* models, though, no such test suite exists. Hybrid models typically combine MPI for process-level distributed computations with shared-memory, multi-threaded programming models. A recent study [105] of about 110 MPI applications showed that the multi-threaded hybrid model is widely used.

While shared-memory programming promises more efficient node-level resource utilization, it further increases the existing complexity of MPI applications as it has its own class of errors. These include data races on shared memory or deadlocks due to misuse of thread-level locks for mutual exclusion [24, 56]. To that end, DataRaceBench [85] exists for similar reasons as MPI-CorrBench.

The MPI standard defines several fundamental requirements for multi-threaded MPI applications, e.g., correctly initializing the MPI library for multi-threading. In addition, it is the user's responsibility to avoid posting *conflicting* MPI (communication) calls from different threads, e.g., [136, Section 11.6.1]:

> "It is the user's responsibility to prevent races when threads within the same application post conflicting communication calls."

From the perspective of a user, an MPI correctness checker should detect errors w.r.t. the aforementioned aspects. A user likely also expects overlapping operations involving, e.g., a concurrent user-level write operation with a reading MPI communication call to be detected. This is not a *conflicting* MPI call — instead, it is a data race stemming from an application's thread in combination with MPI.

Correctness tool developers, hence, need to tackle a combinatorial error set consisting of *typical* shared-memory and *typical* MPI errors, respectively. Unfortunately, tools do not fully support error checking for hybrid models with high levels of MPI thread support yet. This can mainly be attributed to almost no hybrid MPI application fully exploiting these concurrency levels. In most applications, multi-threaded execution is only used inside a compute kernel between MPI communication calls. As such, tool developers have yet to identify an urgent need to tackle the aforementioned complexities to support it. Without capable correctness checkers, however, developing correct hybrid MPI applications is very complex. Thus, MPI users need correctness tools to support (fully) multi-threaded MPI applications before starting to develop them.

Consequently, to help developers extend the capability of their tools and subsequently encourage use of the hybrid model at the highest level of concurrency, we make the following contributions:

- A structured discussion of thread-safety, including threading-related errors that pertain to the hybrid model and *conflicting* MPI calls.
- Extending our MPI-CorrBench test suite targeting the discussed issues with the multi-threaded MPI hybrid models, including an evaluation of well-known MPI correctness tools to assess their current level of support for hybrid models.

The paper is structured as follows. In Section 9.2, we briefly discuss relevant background information w.r.t. MPI and OpenMP and also highlight typical issues with either MPI or OpenMP. In Section 9.3, the technical requirements for tools to detect correctness issues related to this work are presented. Subsequently, in Section 9.4, we present our classification of issues of the hybrid MPI-OpenMP model, which form the basis of our

MPI-CorrBench test-suite extension. In Section 9.5, we evaluate two state-of-the-art correctness tools, namely MUST [42] and Intel Trace Analyzer and Collector (ITAC, [153]), concerning their analysis capability w.r.t. hybrid codes using our test-suite. Section 9.6 discusses the helpfulness of the tools' error messages and, also, how the MPI standard contains ambiguity in its wording w.r.t. hybrid codes. Finally, we conclude in Section 9.7.

9.2. Background

This section gives a brief overview of hybrid MPI applications. *Hybrid* refers to "MPI + X", where *X* typically stands for a shared-memory programming model (e.g., OpenMP [10, 122]) or hardware-based accelerators [47]. OpenMP is most commonly used in the hybrid model [105], and the MPI standard defines how an MPI library has to interact with threads. Therefore, we limit the overall discussion to the MPI-OpenMP model based on the thread- or task-based semantics. For more information about this hybrid model, see [38].

9.2.1. Hybrid MPI model

The MPI standard defines a standardized API for distributed memory computing. It defines, e.g., how a set of processes communicates data with point-to-point and collective operations — both can be either blocking or non-blocking — as well as one-sided operations. It also defines how the MPI library interacts with multiple threads. While the standard is agnostic to any particular shared-memory framework, it assumes POSIX-like threads [7] for its definitions. In addition, for more recent MPI features, OpenMP is used in their examples, e.g., see [136, Section 4.3.1].

Multi-threading

The MPI standard [136, Section 11.6] defines how "MPI and Threads" ought to interact for any valid MPI library implementation. For an extended discussion about MPI library thread-safety implementation concerns, see [28].

In particular, thread-compliance is given only (1) if all MPI routines are thread-safe, i.e., the outcome of concurrent calls to these MPI routines is as if the calls were executed serialized in some order and, in addition, (2) if blocking MPI calls only block the calling

thread, whereas other threads can still make progress. Finally, MPI ranks are still process-based even when using multiple threads.

The standard defines four levels of thread-safety (henceforth, also *threading level*), which must be selected by the user during the initialization with `MPI_Init_thread` of the MPI library using one of the following flags see [136, Section 11.2.1]:

`MPI_THREAD_SINGLE` Each process has only one executing thread, like a standard single-threaded MPI program using `MPI_Init`.

`MPI_THREAD_FUNNELED` Only the thread which initialized MPI executes MPI calls.

`MPI_THREAD_SERIALIZED` Any thread can make MPI calls, but only one thread at a time.

`MPI_THREAD_MULTIPLE` All threads can make MPI calls concurrently.

Hereafter, we abbreviate these threading levels, e.g., *thread-multiple*. The level of thread support is implementation-dependent, and the aforementioned initialization call returns the actual level. A thread-compliant MPI library, however, will be able to return the highest level. While MPI routines themselves are thread-safe (for a thread compliant MPI library), it is the user's responsibility to prevent race conditions of conflicting communication calls.

9.2.2. MPI errors

In our previous work [135], MPI issues and their classification have been presented, namely *erroneous* or *mismatching arguments* and *erroneous program flow*. In [134], compared to [135], more kinds of MPI-related errors have been defined. We give a brief summary.

Erroneous arguments

This category of errors pertains to a *single* MPI call. It refers to, e.g., (1) passing invalid arguments, such as, a message size smaller than 0, (2) wrongly ordering MPI arguments, or (3) reading beyond the length of the given buffer argument with a send operation.

Mismatching arguments

In contrast to the previous error class, *multiple matching* MPI calls are considered in this category. This includes, e.g., mismatches of MPI datatypes across a send and receive operation, or rank mismatches in collective operations.

Erroneous program flow

An instance of an erroneous program flow is a deadlock. Deadlocks typically occur with wrongly ordering point-to-point send and receive data exchanges between ranks, or a missing collective call on some rank for a communicator.

Concurrency

Given a single-threaded MPI program, data races can still happen, e.g., in the context of non-blocking MPI calls when the buffer is accessed before the operation finished.

Message races

Using wildcard matching of message tags or ranks can lead to non-deterministic message matching and, hence, even deadlocks.

Relevancy to our contribution

With multi-threaded MPI, the added level of concurrency can lead to orderings of MPI calls that are erroneous even though their arguments are correct. Hence, in our discussion, for issues relevant to the hybrid model, we primarily consider *erroneous program flow* for our test-suite extension.

9.2.3. OpenMP

OpenMP uses the so-called fork-join model for, e.g., parallel loop execution or *task*-based parallelism. Code between the parallel regions is executed by a single (master) thread, parallel regions are typically executed by a team of threads. *Tasks* are defined by a directive, and, typically, one thread spawns these tasks, which are then executed by the team of threads in some order. The order can be explicitly controlled with appropriate task-dependency declarations.

The developer specifies `pragma`-based directives for the compiler to automatically generate (possible nested) parallel regions in a code. The compiler (1) parses these directives, (2) outlines the parallel region and (3) adds OpenMP API function calls to invoke the parallel region and work- and memory-sharing functions according to the fork-join model. It is the user's responsibility to ensure mutual exclusion of shared-memory regions. To that end, OpenMP defines several directives (e.g., `omp critical`) and, also, a runtime library API for manual locking, see [122, Section 3.9].

9.2.4. OpenMP errors

In previous work [24, 56, 70], OpenMP issues and classifications thereof have been presented. OpenMP errors can stem from, e.g., syntactical problems like misspelling a compiler directive or semantic defects that may manifest as, e.g., a deadlock during execution. We briefly summarize issues based on the classification in [56].

Uninitialized memory

In [56], the authors denoted *unallocated* memory as a conceptional issue. In contrast, we believe that *uninitialized* memory is a more general concern as using uninitialized values may lead to undefined behavior. In particular, the data sharing semantics of OpenMP allow making variables private to each thread inside a parallel region (the variable is, thus, local to the thread's stack). Any prior initialization outside the parallel region is ignored, and it is the user's responsibility to initialize the value inside the parallel region, [122, Section 2.21.3 C/C++]:

> "The new list item is initialized, or has an undefined initial value, as if it had been locally declared without an initializer."

Assumption about parallelism

Assuming a certain degree of parallelism may cause issues, e.g., expecting the number of threads is always greater than two for a parallel region. Breaking this assumption can, e.g., lead to errors described in Section 9.2.4 and 9.2.4.

Race condition

Race conditions describe scenarios where the result of an operation depends on the order of execution in a multi-threaded program. Typically, it can be defined as two or more threads accessing a shared resource, where at least one of these threads executes write access. No assumptions can be made about the result of a read access by some thread, as it may happen before or after a write operation of a different thread. Hence, this scenario results in a data race if the shared resource is not protected by synchronization routines such as an OpenMP *lock*.

Deadlock

While a data race may lead to silent bugs during execution, a deadlock is a program state that can not make *progress* anymore. A race condition may lead to a deadlock. Typically, a wrong order of locking for mutual exclusion can lead to deadlocks of participating threads.

Relevancy to our contribution

We omit other issues, as they are mostly focused on (1) OpenMP standard violations, (2) syntactic problems and (3) performance issues. We believe those issues to be less relevant for our test suite, and they should be handled by OpenMP-specific checkers.

Hence, in our discussion, we are concerned with issues that are most applicable to the hybrid MPI-OpenMP context. Issues we consider relevant are (1) conceptual issues, (2) data races and (3) deadlocks. Conceptual issues are not direct violations of the OpenMP standard but denote OpenMP use that may cause unwanted program behavior. In addition, data races and especially deadlocks are relevant to both MPI process-based and OpenMP shared-memory parallelism, respectively.

9.3. General Analysis Tool Considerations

To analyze and detect correctness issues presented in this work, a tool needs to observe different classes of application behavior. We distinguish low-level hardware instructions, like loads and stores, from high-level parallel semantics defined by MPI and OpenMP. Tools that observe only a subset of this information can still detect some issues but not all. Knowing about the general tool capabilities, an application developer using the tool might derive a better understanding which classes of issues a specific tool can technically detect. To that end, in Section 9.4, we relate our error classification with the required tool capabilities to detect them.

9.3.1. Observing low-level instructions

To capture low-level instructions, in particular memory accesses, analysis tools use compile-time instrumentation [46, 95] or binary instrumentation [93, 153, 162] based on a corresponding framework [22, 30, 161]. Independent of the instrumentation technique, we can also distinguish the coverage of low-level instructions observed for analysis. At the instruction level, tools can observe plain memory accesses, but they could, for example, also observe the ordering semantics of atomic memory accesses.

9.3.2. Observing MPI procedure call semantics

The commonly accepted method to observe MPI procedure calls is to implement the MPI procedure and call into the PMPI interface as suggested by the MPI standard. MUST and ITAC both rely on PMPI in their analyses. Observing the MPI call is not sufficient to understand the memory access and synchronization semantics of the call, as additional arguments describe the memory layout of the message to be transferred. Some tools limit their analysis to predefined datatypes which also means that they only consider the communication of compact memory regions. In order to consider the memory access semantics for derived datatypes, a tool can either observe the construction of derived datatypes or use specific MPI routines to decode the structure of these. Träff [77] combines the latter approach with using MPI attributes to cache the collected type information for constant-time reuse.

9.3.3. Observing OpenMP semantics

Some correctness tools [69, 93] adopt the OMPT interface introduced with OpenMP 5.0 for their analysis. A tool using the OMPT interface registers callback functions with the OpenMP runtime, which are called whenever certain OpenMP events occur during the execution. Protze et al. [89] showed that OMPT provides sufficient information to reconstruct OpenMP synchronization semantics.

An aspect that is not covered by OMPT is the memory access and privatization aspect of OpenMP. An OpenMP implementation might pass a `firstprivate` variable by reference to the function implementing the task body, or call into a runtime function to load the value for the task. In either case, a tool would need to observe an assignment and track the initialization state of the variable.

9.3.4. Combining observations

For deadlock detection in hybrid applications we can distinguish execution with *thread-multiple* from execution with lower threading levels. While a thread is blocked in an MPI call, no other thread in this process can call into MPI to release another blocking MPI call. Thus, the lower threading levels can be treated like single-threaded execution. Only for *thread-multiple*, thread synchronization must be considered. It is not sufficient to find all threads in an OpenMP barrier, as tasks might be scheduled in the barrier. To understand whether all threads are blocked, a tool needs to observe the threading events, and also build a model for possibly ready tasks.

To identify data races resulting from different threads performing MPI communication calls on the same or overlapping memory buffers, observing MPI API calls is sufficient. Only for execution with *thread-multiple*, the analysis has to consider thread synchronization to avoid false alerts.

9.4. Hybrid MPI-OpenMP Errors

In this section, we discuss identified errors related to MPI-OpenMP hybrid codes. In particular, error categories like *deadlocks* are relevant to either pure MPI or pure OpenMP-based applications. However, combining these programming models forms a combinatorial set of errors that need to be handled by correctness checking tools.

```
1 MPI_Init(...);
2 #pragma omp parallel
3 {
4   /* ... do work ... */
5 }
```

Listing 9.1: `MPI_Init` requires a single-threaded process. Using it in conjunction with OpenMP is not correct. At a minimum, e.g., if the parallel section (line 2) does not call MPI, `MPI_Init_thread` with *thread-funneled* is required.

At the high-level, we consider (1) threading-related *initialization* and *completion* of MPI, (2) *ordering* of MPI operations, (3) *uninitialized memory* and, finally, (4) *data races* as the key error types. We give a detailed discussion of these errors together with code examples. For each error, at the end, we refer to the required tool capabilities to detect them as discussed in Section 9.3. We focus our discussion on the threading level *thread-multiple*, where each thread may call MPI functions concurrently.

9.4.1. Initialization and completion

The MPI standard requires setup and corresponding cleanup of each process for a correct MPI program. For a multi-threaded MPI program, additional requirements are specified, see [136, Section 11.2.1].

Initialization

For a multi-threaded MPI program, and more specifically, a multi-threaded process, the user is required to call `MPI_Init_thread`, indicating the required level of thread support. We consider programs where users (1) fail to call `MPI_Init_thread`, (2) set an incorrect minimum required level, or (3) do not check the level of thread-support returned by MPI, to be problematic.

In Listing 9.1 an erroneous MPI initialization of a multi-threaded process is shown. Our test suite contains all erroneous combinations of requested threading-level versus actual required level.

```
1 #pragma omp parallel
2 {
3 /* ... do work ... */
4   MPI_Finalize();
5 }
```

Listing 9.2: MPI finalization error: `MPI_Finalize` is called from a parallel region by each thread.

Completion

A correct MPI program calls `MPI_Finalize` on each process once after all its MPI (communication) completed to clean up. In a multi-threaded context, `MPI_Finalize` should be called by the same thread that initialized MPI. Likewise, all threads should have their MPI communication completed at this point. In Listing 9.2 finalization is called per thread, which is erroneous.

Tools perspective

When errors in this category manifest during execution, they can be observed at MPI level, see Section 9.3.2. A precise analysis that verifies serialized ordering of MPI calls (e.g., *thread-serialized*, collective communication) needs to observe thread synchronization and therefore OpenMP semantics, see Section 9.3.3.

9.4.2. Ordering: Conflicting MPI calls

The order of MPI calls for a single thread is well-defined. However, between multiple threads in an MPI process, there are no strict ordering guarantees. The standard specifies that for concurrent MPI operations, the "outcome will be as if the calls executed in some order, even if their execution is interleaved" [136, Section 11.6.1]. The user is responsible for avoiding orderings that lead to erroneous program behavior.

Examples of tests that are erroneous under specific (non-deterministic) ordering are: (1) freeing some opaque MPI resource, while (or before) another thread uses it, or (2) two threads blocking on the same request object for non-blocking communication calls. These issues can be hard to detect as such an erroneous order may rarely happen, or only on

```
1 #pragma omp task
2 { MPI_Send(..., other_comm_world , ...); }
3 #pragma omp task
4 { MPI_Comm_free( &other_comm_world ); }
```

Listing 9.3: If freeing of the communicator happens before the send operation, the program is erroneous.

specific hardware platforms. Consider the deallocation of a communicator, see Listing 9.3. This code is (only) erroneous, if the free happens before the send [136, Section 7.4]:

> "Any pending operations that use this communicator will complete normally; the object is deallocated only if there are no other active references to it."

While we may receive an error message from the MPI library, this is not guaranteed, as stated previously. Conflicting calls should therefore be handled by MPI correctness checkers that are aware of these particular MPI semantics.

Collectives

The matching order of collective calls on a communicator is dependent on the order of invocation. For multi-threaded MPI, using the same communicator, "it is up to the user to make sure the calls are correctly ordered, using interthread synchronization" [136, Section 11.6.2]. Likewise, concurrent use of the same communicator must be avoided [136, Section 6.14]:

> "[...] it is the user's responsibility to ensure that the same communicator is not used concurrently by two different collective communication initialization calls at the same MPI process."

As an example of violating the aforementioned restriction, consider Listing 9.4. Here, a process with at least two threads may use the same communicator concurrently, violating the restriction.

Matching (Probe)

Receiving messages of unknown size requires three steps, namely (1) a probe for the data size with subsequent (2) allocation of a buffer and, finally, (3) the MPI receive operation.

```
1 #pragma omp task
2 { MPI_Bcast(..., 0, MPI_COMM_WORLD); }
3 #pragma omp task
4 { MPI_Bcast(..., 0, MPI_COMM_WORLD); }
```

Listing 9.4: With at least two threads per process, these collective calls may happen concurrently on the same communicator — the user is responsible to avoid these scenarios.

```
1 #pragma omp single
2 {
3   #pragma omp task
4   { MPI_Recv(recv_buffer, ...); }
5   #pragma omp task
6   { MPI_Send(send_buffer, ...); }
7 }
```

Listing 9.5: Assume two processes exchange data with a send and receive operation using OpenMP tasks. This likely works correctly for processes with at least two threads. Nevertheless, it is non-conforming to the OpenMP standard, as it may deadlock regardless of the number of threads used.

`MPI_Probe` can be used for determining the data size. However, this probe operation is not thread-safe [43]. In a multi-threaded context, the aforementioned steps may happen concurrently, where one thread receives a message that was already probed by a different thread. Thread-safe *matched* probes (`MPI_Mprobe`) exist avoiding this issue, otherwise "it is up to the user to enforce this condition using suitable mutual exclusion logic" [136, Section 11.6.2].

Deadlocks

MPI applications with a level lower than thread-multiple have the property that a deadlock is detectable as a pure MPI or a pure OpenMP deadlock, respectively [63]. For instance, deadlocks in single-threaded MPI applications primarily occur due to specific ordering of blocking operations, see [135]. The added degree of concurrency in multi-threaded MPI codes makes such erroneous states much more complicated, see Section 9.3. In particular, assumptions about the degree of parallelism or behavior of the OpenMP implementation may cause issues, e.g., in HPC-related performance measurements w.r.t. thread-related scaling tests. To illustrate the complexity of developing deadlock-free hybrid codes (for

all degrees of parallelism), consider Listing 9.5, which is a task-based hybrid code that may deadlock. This is erroneous according to the OpenMP standard, as a "program must behave correctly and consistently with all conceivable scheduling sequences [...]" [122, Section 2.12.6]. Two issues make the code in Listing 9.5 non-conforming to the OpenMP standard:

1. The code may deadlock as the thread of the single-directive "may immediately execute the task [...]" [122, Section 2.12.6]. In this case, the thread will not create the corresponding receive task, even if more threads are available in the parallel region.

2. In contrast, given that both tasks are successfully put into a pool of tasks, they can be freely executed by the threads of the parallel region. If only one thread is available, a deadlock is likely if both ranks execute the same send or receive task in their respective parallel region.

Tools perspective

For threading levels below *thread-multiple*, errors in this category can be observed at MPI level, see Section 9.3.2. Otherwise, a precise (correctness) analysis additionally needs to observe OpenMP semantics, see Section 9.3.3.

9.4.3. Memory

As discussed in Section 9.2.4, the `private` directive for variables does not create private value copies for each thread, instead they are put on the thread's local stack as if "it had been locally declared without an initializer", see [122, Section 2.21.3]. Hence, communicating and using such a value without explicit initialization inside the parallel region may lead to undefined behavior in C/C++, see Listing 9.6. Therefore, we include such cases in Hybrid-MPI-Bench.

Tools perspective

As discussed in Section 9.3.3, information about privatization is not provided with OMPT. Therefore, errors in this category need information from Section 9.3.1.

```
1 int x = 1;
2 #pragma omp parallel private(x)
3 {
4   MPI_Send( &x , 1, MPI_INT, ...);
5 }
```

Listing 9.6: The integer `x` is uninitialized inside the parallel region due to the use of the `private` clause.

```
1 #pragma omp parallel
2 {
3   buffer [omp_get_thread_num()] = -1;
4   #pragma omp master
5   { MPI_Send( buffer , ...); }
6 }
```

Listing 9.7: Race condition: In l. 3, a buffer is written to. In l. 5, although only the master thread executes the send call, there is no synchronization. Hence, the send may happen while other threads still write to the buffer.

9.4.4. Data races

Data races are a common issue with multi-threaded codes, as discussed in Section 9.2.4 w.r.t. OpenMP issues. Data races for single-threaded MPI applications can happen in the context of non-blocking communication, see [134, 135].

In multi-threaded MPI codes, data races may happen due to, e.g., concurrent usage of some opaque MPI object, as discussed in the previous paragraphs. The MPI standard refers to such operations as *conflicting* MPI calls that the user should avoid see [136, Section 11.6.1]:

> "It is the user's responsibility to prevent races when threads within the same application post conflicting communication calls. [...]"

At the same time, though, the standard does not explicitly define data races that originate from the user's code as erroneous. From the perspective of a user, however, these data races should be detected by a correctness checker. To illustrate this issue, a data race w.r.t. an MPI buffer is shown in Listing 9.7. The race condition exists due to missing thread synchronization in the user code. Simply observing MPI calls for correctness checking will fail to detect the memory accesses that lead to the data race.

Tools perspective

For data races between buffer accesses in MPI function calls, information from the MPI level, see Section 9.3.2, and the OpenMP semantics is required, see Section 9.3.3. Such data races can only occur on level *thread-multiple*. For data races involving memory accesses from application code, additional information from the low-level instructions of the application is needed, see Section 9.3.1. These data races can occur with all threading levels.

9.5. Evaluation

We use our test suite, based on insights presented in Section 9.4, to assess the current level of support by MPI correctness tools for checking hybrid MPI-OpenMP codes. To that end, we evaluate MUST and ITAC, which, to our knowledge, are the only tools that support hybrid MPI-OpenMP codes w.r.t. their error detection capabilities. A discussion of the helpfulness of the generated error messages for finding the defect is given in Section 9.6.

9.5.1. Setup

Our MPI-CorrBench extension consists of 103 new test cases of the different error categories described in Section 9.4, of which 44 are correct test cases to check for any false positive error reports.

Non-determinism

The occurrence of any erroneous behavior of our test cases depends on many factors besides the tests' source code. For instance, MPI calls may be ordered differently for multiple test executions due to thread scheduling. In order to provoke the erroneous behavior, we designed our evaluation to execute each test multiple times with different parameters. We (1) vary the buffer sizes from bytes to megabytes, in order to likely trigger different internal procedures of the MPI implementations, (2) range the available OpenMP threads from 1 to 8, in order to have more or less potential conflicts between threads, and (3) introduce an artificial load imbalance between the threads, letting a thread sleep up to 10µs, in order to either favor a *likely* correct execution path or, in contrast, one where

correct ordering is *unlikely*. As we do not use any thread-synchronization routines for the introduction of the load imbalances, these operations are still "logically concurrent" from the point of view of the MPI standard. For every possible parameter combination, we executed the test case 5 times.[1]

Observable errors

To observe errors in the context of non-deterministic execution behavior, we included checks in our test cases to determine if an error manifested during execution. To that end, in our Hybrid-MPI-Bench suite, we define three error categories that we check. The observable errors are as follows.

1. *Environment error*, where the error is always present for a given execution environment, e.g., setting the wrong threading level.
2. *Corrupted buffer error*, e.g., when a data race produces an unexpected buffer-value.
3. *Ordering error*, where the error may not exclusively lead to any buffer corruption, e.g., overlapping of conflicting collective operations.

For environment errors, we observe the execution environment, e.g., the number of threads used, to see if we encounter an environment error. For corrupted buffer errors, we compare the resulting buffers to their expected content. For ordering errors, we employ a logical clock that is atomically updated by each thread before and after a relevant MPI function call, respectively. Hence, for a thread observing an *odd* clock value upon entry to an MPI function, an ordering error is detected as the MPI operations overlap. Generally, the correct ordering of MPI operations is known a priori, thus, we can compare the clock value to the expected value for any particular MPI call to determine the presence of an error.

Limitations Detecting *ordering errors* may fail if the operating system stalls a relevant thread between atomic clock updates and the MPI operation. However, we note that the introduction of a logical clock does not introduce any relevant thread synchronization, i.e., keeping MPI operations "logically concurrent".

Note: Even if an error is *not observable* during execution of any of the 59 erroneous test cases, these codes are still erroneous and should, therefore, be reported by a correctness tool.

[1]The exact specification of the parameter space is available in our repository.

Correctness tool setup

We use the MPI correctness tools ITAC, included with the Intel oneAPI toolkit version 2021.5, and MUST preview version 1.8. The latter includes an LLVM ThreadSanitizer (TSan, [169]) integration for analysis of thread-related errors in hybrid codes. We use the MPI implementation of Intel included in the oneAPI toolkit version 2021.5. To compile our tests, we use the Clang compiler 10.0 (required by MUST).

We note that a different MPI implementation may react differently to the nondeterminism included in the test cases, and may therefore influence a tool's capability to observe the errors at runtime.

Result visualization

As we executed each case multiple times, we visualize each test case as a separate bar in the plots. The main category is defined as discussed in Section 9.5.1. For each case, in addition, the tool output is categorized as follows.

1. *Found*, where a test concludes *with* an error message.
2. *Missed*, where a test concludes *without* an error message.
3. *Crash*, where a test is *inconclusive* as no output indicating existence or absence of an error is produced.

The latter may happen, e.g., if the MPI library aborts due to an error. The categories are color encoded for each bar. The y-axis of the plots are the test case number of only the erroneous tests. We do not plot test cases for checking false positives. See our repository for the respective test case code: `https://github.com/tudasc/mpi-corrbench`.

9.5.2. Wrong threading level

Figure 9.1 shows the error detection capabilities of MUST and ITAC, when a wrong MPI threading level is used. In this case, executions where an error was not observable also include the executions where only one thread was used, as errors regarding the usage of a wrong threading level do not apply in a single threaded case. Nonetheless, we think it is a good idea, if a tool also warns for these cases, as the number of threads can be

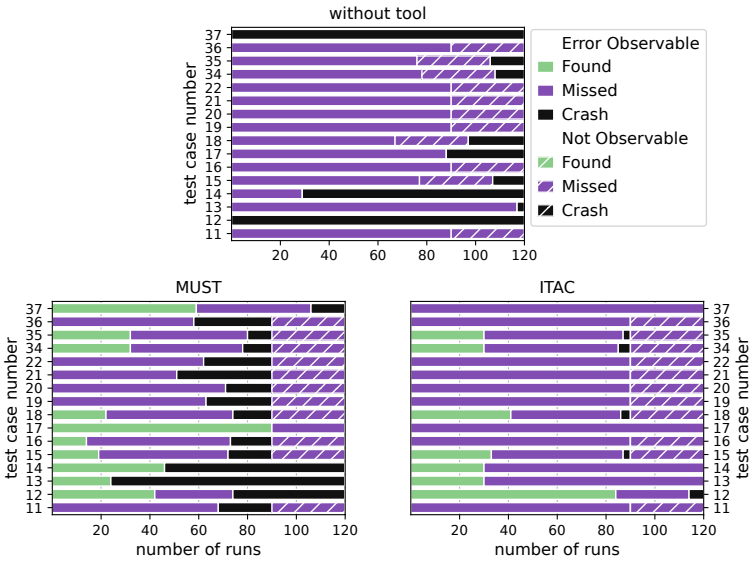


Figure 9.1.: Error detection results for MUST and ITAC for cases with wrong usage of MPI threading level. Each bar represents one test case; the x-axis denotes the number of test case executions (with varying runtime settings). The upper bars show the execution without the tool, while the lower bars show the tool-enabled execution. The textured bars denote executions, where no error (violation of the MPI threading level) was observable.

influenced by an environment variable and does not solely depend on an application's source code.

In general, MUST catches more errors than ITAC. Sometimes the runtime-interference of using ITAC even prevents an erroneous test case from crashing (e.g., case 37).

Both tools where not able to point out the usage of a wrong MPI threading level, when the test case requires *thread-serialized* or lower (cases 11, 19–22).

9.5.3. Uninitialized memory

We implemented tests, where an OpenMP `private` variable is communicated. This may lead to undefined behavior, as this variable is uninitialized according to the OpenMP standard, see Section 9.4.3.

We see that both tools do not detect such errors. Hence, we did not include these test cases in our visualization (cases 1–10). Sending uninitialized memory is not an error according to the MPI standard, hence, there is no expectation of any MPI correctness tool to find these issues. However, the receiving process may expect such incoming message to be initialized. The subsequent usage of this uninitialized memory (e.g., in a condition) results in undefined behavior with C/C++.

9.5.4. Conflicting communication calls

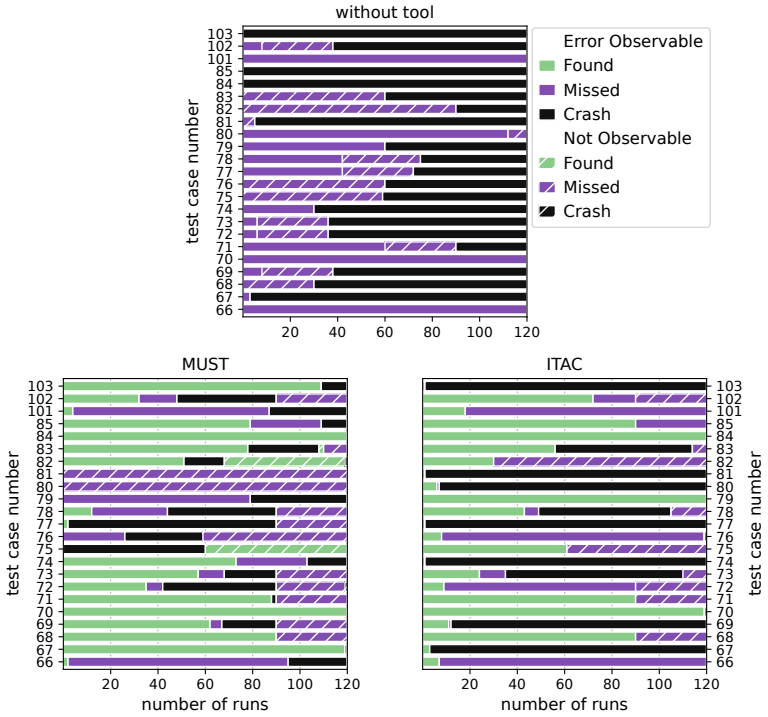


Figure 9.2.: Error detection results for cases where the wrong ordering of calls leads to conflicts. Each bar represents one test case; the x-axis denotes the number of test case executions (with varying runtime settings). The upper bars show the execution without the tool, while the lower bars show the tool-enabled execution. The textured bars denote executions, where no wrong call ordering was observable during execution.

In this category, both tools perform reasonably well, see Figure 9.2. Although the error may not always be observable during execution, all our test programs are erroneous according to the MPI standard.

MUST misses cases where a `MPI_Probe` is used instead of the correct `MPI_MProbe` (cases 80 and 81). Unfortunately the runtime influence of MUST prevents the error from being observable in these cases.

In some cases (75 and 82), we see that MUST is able to detect that the combination of MPI operations may cause a deadlock, even if we did not observe it during execution. For ITAC it is not possible to detect any errors that are not observable. This is a major drawback, a developer may experience a correct ordering during testing with the error only manifesting in production, making it cumbersome to find the error. But the support for detecting errors that do not manifest themselves during execution is also limited in MUST.

Overall, the tools perform adequately given the difficulty detecting conflicting MPI communication calls in a nondeterministic multi-threaded environment. But as we pointed out, there is still room for improvement.

9.5.5. Data races

Data races between MPI and the user-code are not covered by the MPI standard. Nonetheless, the occurrence of a data race is undefined behavior in C/C++. Therefore, such errors should be detected, especially, when a data race leads to any observable buffer corruption (but the presence of a data race, not manifesting itself in an observable buffer corruption still is undefined behavior). Again, we used varying degrees of load imbalance, thread counts and buffer sizes for different executions, in order to provoke different non-deterministic behaviors at runtime.

The results are depicted in Figure 9.3. Test case runs without observable buffer corruption are textured. To determine the tool's influence on the nondeterministic runtime behavior, we also show execution of our tests without the tools (upper part of Figure 9.3).

ITAC is not able to detect a possible data race if a buffer corruption does not manifest itself. In particular, data races when the message buffer of a collective operation is corrupted are rarely detected by ITAC (e.g., cases 48, 49, 39 are mostly missed). With TSan, MUST is able to detect most data races, even if they do not result in an observable buffer corruption (e.g., cases 65, 50, 47). We note, that the remaining executions of those

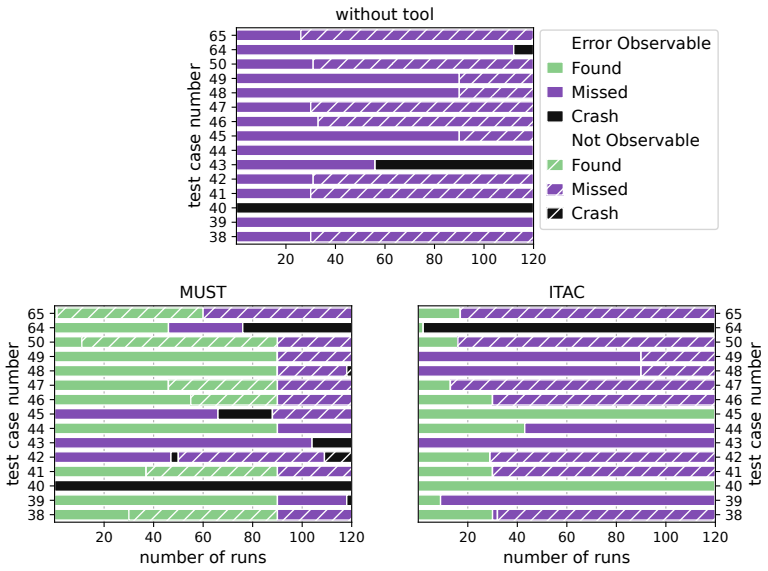


Figure 9.3.: Error detection results for data races. Each bar represents one test case; the x-axis denotes the number of test case executions (with varying runtime settings). The upper bars show the execution without the tool, while the lower bars show the tool-enabled execution. The textured bars depict the executions when a data race does not lead to buffer corruption.

cases were done with only one thread and, therefore, without the possibility for a data race. Cases where MUST fails entirely to detect an error typically involve the use of non-blocking operations immediately followed by a corresponding wait (cases 45, 42). The concurrent usage of an `MPI_Info` object by `MPI_Info_set` is missed by both tools (case 43).

9.5.6. False error reports

Neither tool produced any false positive error reports for our 44 correct test cases.

9.6. Discussion

In this section, we give a subjective and hence opinionated interpretation on the quality of error messages produced by both ITAC and MUST, and note how the MPI standard language may be ambiguous in certain cases.

Quality of error messages — We define helpful as (1) providing more information than a normal program crash (e.g., backtrace of function calls) and, also, (2) a pointer to at least one faulty or conflicting MPI operation involved with the error.

Threading level: ITAC detected the presence of an error in 7 of 16 cases (see Figure 9.1). But in our view, only two of the error messages where actually helpful to find the cause of the error, often ITAC only points out that a `SIGTERM` was raised with no additional information hinting to a wrong usage of MPI threading levels. MUST on the other hand was able to detect the presence of 10 errors (in at least some executions). Of the error messages given, all but two actually hint a wrong usage of MPI threading levels.

In just three out of the total 32 errors detected for all our error categories (threading level, data race and wrong call ordering), the given error message by MUST was only about an unknown threading level, and no additional helpful information was given. In other cases, these messages about an unknown threading level often fill the report without cause, thus, distracting from the otherwise helpful message of the actual error.

Conflicting MPI calls: ITAC typically gives helpful error messages for conflicting MPI operations, pointing to at least one of the faulty operations. In six out of the 23 cases, the error message only points out that a (exit) signal was raised with no additional helpful information. With MUST, no output file is created in test cases where the MPI library crashes. We marked these tests as crashed in Figure 9.2.

Data races: Both ITAC and MUST give helpful error messages for all data races they can detect. For test cases, where a data race exists in the code but does not occur during execution, both tools should inform the user about their existence. Such transient errors are extremely hard to track down, and we therefore see the need to point out the possibility of such data races. Currently, ITAC lacks the capability to detect these transient errors. In contrast, MUST detects many of these.

Memory: Use of uninitialized variables due to the OpenMP directive `private` is not detected by either tool. As the use of such variable may lead to undefined behavior, detection capabilities should include OpenMP's data sharing directives.

Ambiguity of the MPI standard — Finally, we also note an additional obstacle for developing hybrid MPI correctness tools in the current MPI standard itself. In some instances, the MPI standard may be ambiguous in its wording. For instance, in the official issue tracker of the MPI forum, an active discussion about ordering of MPI calls is held w.r.t. the question [167]:

> "If two threads ensure a particular order of MPI operations using thread synchronization is the MPI library allowed to still treat these operations as 'logically concurrent', i.e., unordered?"

In our view, if an application ensures a *happens before* relationship of two MPI calls via thread synchronization, they can no longer be considered *logically concurrent* by the MPI library. Therefore, if MPI calls are considered ordered w.r.t. the OpenMP semantic, they ordered w.r.t. the MPI semantic as well. Hence, our test cases to check for false positives are correct only under this interpretation of the current MPI standard [136]. However, since this issue is already in discussion by the MPI Forum, we expect it to be resolved in a future clarification.

9.7. Conclusion

Developing correct MPI applications is challenging due to the low-level interface and complex communication patterns for the most efficient distributed computations. Combining MPI with OpenMP for shared-memory computations promises better node utilization and, thus, performance. However, the additional complexity of developing hybrid MPI-OpenMP programs is high, and there are many possibilities of usage errors, as previously discussed in this work. We believe that this is one likely reason why only a few codes make use of the highest level of thread support in MPI. We also believe that hybrid models with a high level of shared-memory parallelism will only find further adoption with sufficient support from MPI correctness tools. Here, the proposed Hybrid-MPI-Bench test-suite helps these tool developers. Our test cases each provoke a single violation, thus, allowing for a focused development effort.

Our evaluation has shown that there is still room for improvement for MPI correctness tools w.r.t. analysis of the hybrid model. Especially when more intricate errors, such as (1) sending an uninitialized variable due to the OpenMP *private* directive, or (2) data races on buffers due to user-code interfering with MPI calls, are considered. We have seen that many of these errors may not always be observable as faulty behavior, i.e., they appear only during some executions of a hybrid code, highlighting the overall complexity

of hybrid codes. In these cases, where errors are transient, capable tool support is especially valuable to find the cause and subsequently fix the error.

In summary, a *pure* MPI or a *pure* OpenMP correctness checker is not enough to find all the presented errors of the hybrid MPI-OpenMP model as shown in this work. Hence, further development of capable tools is needed to further encourage the adoption of this hybrid model at the highest level of thread support. To that end, Hybrid-MPI-Bench is able to help correctness tool developers.

10. Investigating the Real-World Applicability of MPI Correctness Benchmarks

Bibliographic Information

The contents of this chapter has previously been published in:

Alexander Hück, Tim Jammer, Joachim Jenke, and Christian Bischof. "Investigating the Real-World Applicability of MPI Correctness Benchmarks". In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W 2023. Denver, CO: ACM, Nov. 2023. DOI: 10.1145/3624062.3624091

The contribution of the author of this thesis is summarized as follows:

Alexander Hück and Tim Jammer came up with the underlying Idea and evaluation criteria in mutual discussion. As the co-leading author, Tim Jammer contributed most of the Implementation for the analysis and scoring workflow, while Alexander Hück implemented the PMPI-Wrapper to trace the run-time usage of MPI arguments. To ensure the reproducibility of this work, both leading authors collaborated to orchestrate all necessary steps into an automated continuous integration pipeline using a GitHub CI. In terms of the manuscript, Tim Jammer focused on the evaluation and discussion chapters, while Alexander Hück concentrated on the Introduction and analysis chapters. Joachim Jenke advised on the MPI Tools perspective, as well as providing invaluable feedback on the manuscript's text and the presentation of the results alongside Christian Bischof. All authors agree with the use of the publication for this thesis.

10.1. Introduction

MPI is the de-facto standard for distributed computations in HPC. The interface, however, is low-level and error-prone as data is transferred as a type-less pointer to a buffer, and the user has to specify, e.g., the datatype and data length manually. MPI libraries typically provide little error checking, and the MPI standard does not guarantee consistent or reliable error handling [136, Section 2.8].

Consequently, MPI correctness tools have been developed [13, 42, 57, 59, 153] to aid the development of bug-free MPI codes. For instance, several dormant bugs in popular mini-applications used for benchmarking in HPC have only been uncovered after years of usage [13, 59]. The correctness tools need to support commonly used MPI features and check for related correctness issues to be useful for MPI users.

To aid these efforts, two MPI correctness benchmark suites have recently been developed, namely the *MPI-CorrBench* [135, 143] and the *MPI Bugs Initiative* [134]. Both represent a *standardized test harness* that contains small-scale tests with correct and erroneous use of the MPI library, ranging from illegal argument values to deadlocks and data races. Hence, the correctness benchmarks help (a) correctness tool developers with prioritizing development effort to improve and extend their tools checking capabilities, and (b) code developers with assessing correctness tools for their particular needs.

To assess the real-world applicability of MPI correctness tools, the correctness benchmarks should encompass representative MPI (mis-) usage scenarios encountered in HPC codes. Such coverage enables correctness tool developers to refine and optimize error checking according to practical MPI usage, ultimately benefiting MPI code developers. Hence, our research question is:

> *How closely do the correctness benchmarks mirror the MPI usage of real-world HPC codes regarding MPI test cases?*

We utilize our previously collected data set to answer this question, containing MPI usage at function- and argument-granularity of 96 HPC codes [147]. This data set contains a classification for each detected MPI call *and* their parameters, e.g., the third argument of a `MPI_Recv` operation is extracted and categorized as `DATATYPE`. We compare this data set with the MPI (argument) usage in the correctness benchmarks. The benchmarks' proximity to real-world MPI usage is determined based on factors such as the presence of specific MPI functions like `MPI_Allreduce`, along with considering the frequency of their arguments, such as `MPI_SUM` being the most commonly found reduction operation in the

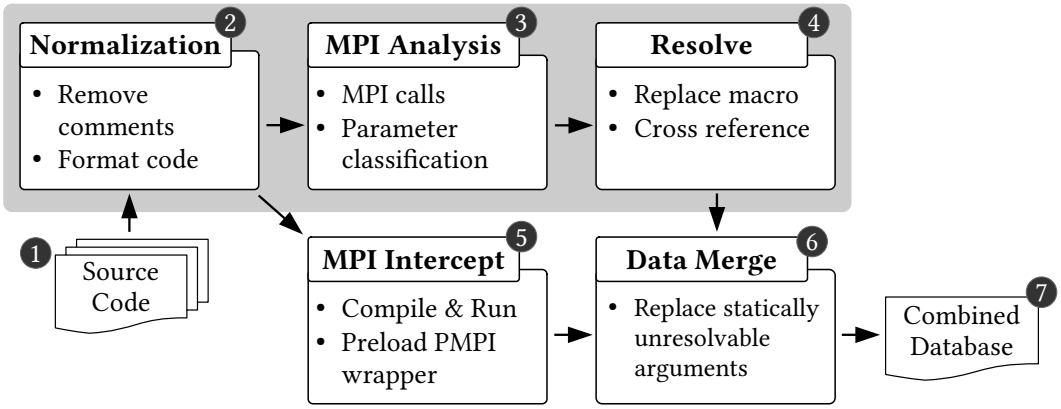


Figure 10.1.: Our static analysis tool chain [147] (grey) is extended with an optional runtime analysis. The static analysis creates a database with all MPI calls, their arguments, and the cross-usage of MPI handles. The dynamic analysis creates runtime traces of MPI function calls and their arguments. These data sets are merged, replacing statically undecidable values.

data set. Our proposed scoring serves to assess the current state of representativeness and provides a guideline for continuously improving the coverage of representative MPI usage in these correctness benchmarks.

10.2. MPI Feature Usage Analysis

Figure 10.1 shows our toolchain that classifies MPI usage at argument granularity based on a set of input codes. It automatically classifies MPI calls and their arguments based on a structure closely resembling the MPI standard [136], see [147] for details.

The MPI usage data set of 96 HPC codes (henceforth, called *HPC data set*) was generated through static source text analysis, ensuring independence from any specific software, hardware or build system requirements. However, static analysis may encounter cases where MPI arguments cannot be resolved, particularly when compile-time dependent macro definitions are used to set, e.g., the MPI datatype. Hence, the parameter set contains the classification wildcard *Other Parameter* for some MPI calls.

To address this limitation for the correctness benchmarks, we added a runtime analysis component capable of tracing all relevant MPI calls and their runtime argument values

using the MPI profiling interface (PMPI). By merging the static database with the dynamically generated one, we obtain a precise and comprehensive data set of MPI usage patterns for the correctness benchmarks.

Scoring MPI Usage Pattern

For each MPI feature category, we calculate a percentage value representing the coverage of the correctness benchmarks regarding the MPI usage patterns found in the HPC data set. Hence, a score of 100% indicates a complete coverage of the MPI usage patterns.

Scoring Contribution We compare if an MPI call is present and match their relevant MPI arguments to obtain the scoring. Only if both match, they are counted towards the final score. If the HPC data set contains the aforementioned *Other Parameter* category, it is treated as a wildcard where the correctness benchmark's particular argument always matches. In the following, we list the relevant parameters that we consider to be the core MPI usage pattern and how we match the data sets.

- *Datatype* and *Reduction Operator*: *Datatype* includes built-in and derived MPI datatypes. There must be an exact match.
- *Count*: The count must match. In particular, we consider the combination of count and datatype to be integral for our scoring.
- *Rank*: The rank must match only for the MPI *collective* category. We believe the rank has particular importance in this category and less so in, e.g., *PtP* communication.
- *Tag*: We match the `ANY_TAG`, other values are treated as wildcards. Evaluating discrete tag values (e.g., 1 or 8) has no particular merit for our coverage metrics.

The scoring contribution is weighted according to usage frequency. Assume the HPC data set contains nine `Allreduce` and one `Reduce` call with various argument variations. To score 100%, the correctness benchmark must include both the `Allreduce` and `Reduce` calls with all parameter variations. If the `Reduce` is fully matched, but out of nine `Allreduce` calls with, e.g., eight `MPI_DOUBLE` and one `MPI_FLOAT` datatype arguments, only the `MPI_FLOAT` case is covered by the correctness benchmark, the score for *Reductions* is 20%.

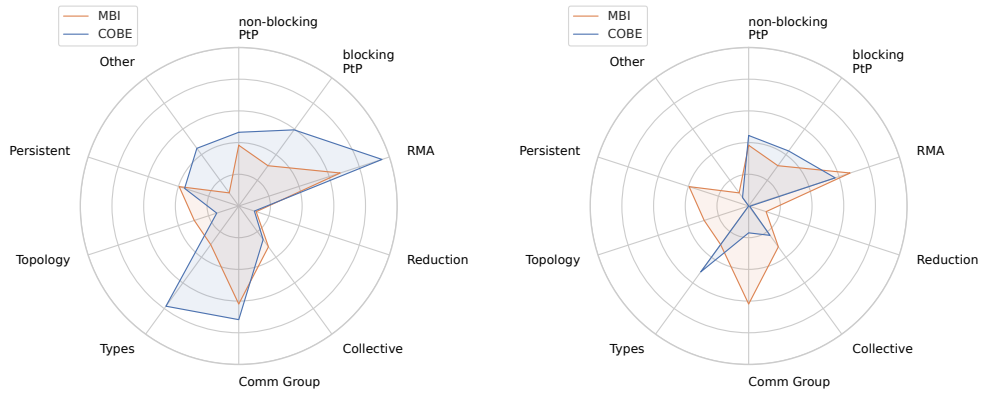
Other MPI usage patterns If there are calls without the aforementioned parameter categories, such as `MPI_Wait`, we match only the MPI call. We consider these other parameters not to be core MPI usage patterns for our analysis.

10.3. Evaluation

We evaluate MPI-CorrBench v1.2.1 (COBE) and MPI Bugs Initiative v1.0.0 (MBI) w.r.t. their coverage of MPI use against the statically collected HPC data set. COBE contains 510 (202 correct and 308 erroneous) test case files, while MBI has 1691 (697 correct and 994 erroneous) test case files. The test timeouts (e.g., for deadlocks) are set to 30 s and 120 s for COBE and MBI, respectively. COBE also contains hybrid MPI+OpenMP tests and mini-applications that we do not consider in this evaluation to keep the results comparable. We show the total score and, subsequently, individual scores of the most essential MPI categories. Finally, the weighting of the coverage is presented. It is based on the occurrence frequency in the HPC data set. For all evaluations, we do not consider the MPI functions `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Init`, `MPI_Finalize` and `MPI_Init_thread`. We believe these are core to MPI applications and hence add no value to coverage metrics.

Table 10.1.: Total MPI coverage score w.r.t. the HPC data set.

	Total (%)	Correct (%)	Erroneous (%)
COBE	39.26	36.53	23.13
MBI	28.88	28.88	28.88



(a) Correct MPI test cases.

(b) Erroneous MPI test cases.

Figure 10.2.: Scoring per MPI category for each MPI correctness benchmark. A higher score is better.

Total score Table 10.1 shows the coverage scoring w.r.t. the HPC data set. In particular, we show the score for (1) total test cases (2) correct MPI test cases only and (3) erroneous MPI test cases only. COBE has a higher coverage w.r.t. correct test cases, as it re-uses unit test codes from an MPI library implementation [15]. In contrast, MBI has better coverage for erroneous MPI use, which can be attributed to its automatic test generation scheme from a set of templates.

Categorized score Figure 10.2a and Figure 10.2b show the scoring per MPI category, taken from [147], for each correctness benchmark's correct and erroneous MPI test case set, respectively.

Surprisingly, COBE (and MBI) have little coverage of real-world reduction usage patterns. This indicates that the MPI library test suite (used by COBE) itself also does not extensively test for these real-world patterns. For erroneous MPI use, in particular, COBE has no relevant coverage of reduction or persistent operations, whereas MBI has some coverage in each category.

Weighting of MPI feature score Figure 10.3 shows the distribution of MPI feature usage in the HPC data set. These values determine the contribution to the overall score, shown in Table 10.1. *Collective* and *Reduction* routines are predominantly used in the HPC codes, indicating that the correctness benchmarks should focus on these predominant categories. Comparatively, the categories one-sided communication (RMA) and topology-related routines find little use in the HPC data set. Hence, any such test cases in the correctness benchmarks contribute little to the overall coverage score.

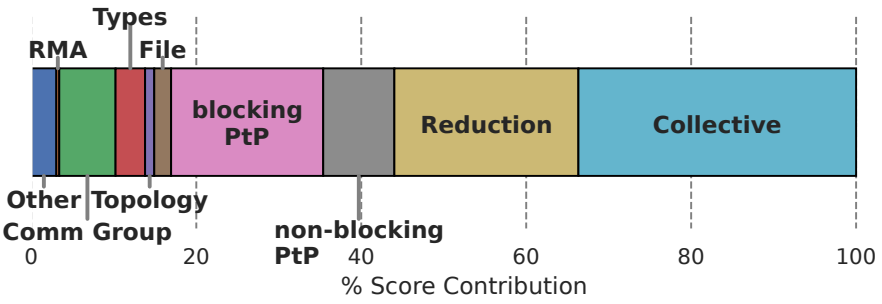


Figure 10.3.: Weighting of MPI categories. Categories are taken from [147].

10.4. Discussion

Table 10.1 assesses whether existing MPI correctness benchmarks mirror the MPI usage of real-world HPC codes. While neither benchmark appears to mirror real-world usage patterns fully, we believe our toolchain can guide improvement to coverage by continuously scoring the correctness benchmarks. To help guide these efforts, we discuss some further aspects of our analysis.

High score does not mean thorough correctness testing We believe a score of 100% is not the gauge for the quality of a correctness benchmark. In order to achieve such a high score, many tests would need repetition with various argument combinations. However, the semantics of the introduced MPI defect is essential, i.e., is a "*off-by-one*" buffer error important to be tested with all built-in datatypes? Likewise, *deadlocks* need not cycle through every parameter combination (the call might never happen). Instead, the scoring is a guide for improvements in particular directions.

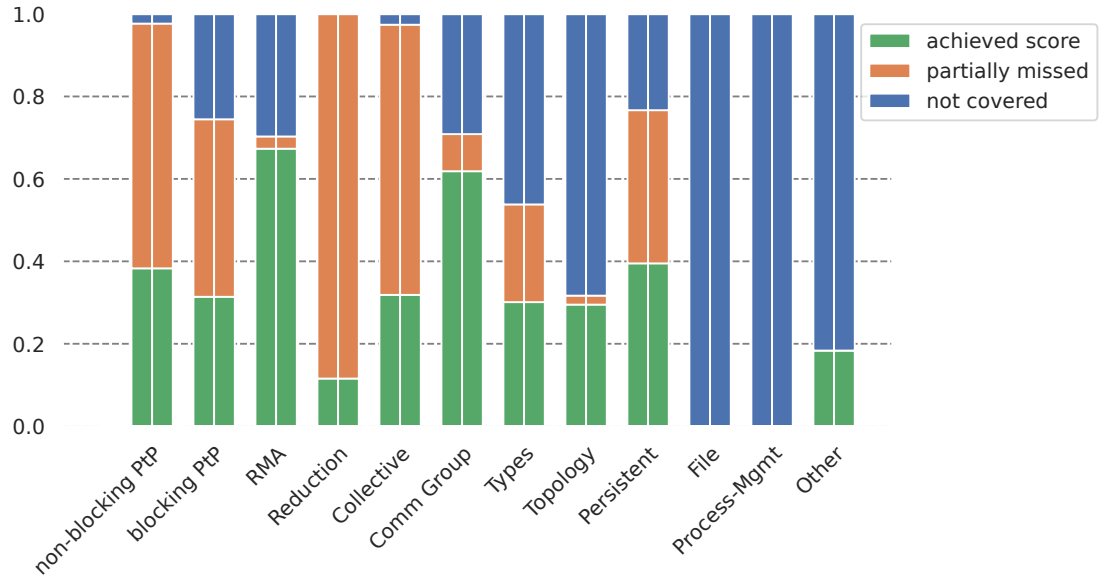
Which MPI calls are not covered? Figure 10.4 shows (potential) scoring contributions for each benchmark, divided into two bars for erroneous (left bar) and correct (right bar) MPI use per category. MBI has mirrored behavior for correct and erroneous test cases, which can be attributed to automatic test generation. We see that they cover all reductions and most collectives but miss many of the argument variations present in the HPC data set. COBE has better coverage for the correct test cases due to the MPI library test suite integration. In particular, erroneous use of reductions has little coverage and, crucially, the most used reduction `Allreduce` is missing. In contrast, the correct reductions cover the calls but miss many argument variations for a higher score.

Which parameter combinations are not covered? In MBI we detect little use of the `MPI_DOUBLE` datatype. It is only used for the construction of the derived datatype `contiguous`. In contrast, this datatype is used often in the HPC data set, which causes the score of MBI to be lowered. COBE has comparatively more `MPI_DOUBLE` usage but still lacks sufficient coverage. While conceptually similar, mismatches between data types like integer and float versus integer and double may require distinct handling by correctness tools. For instance, a tool solely checking datatype size will encounter failures in these scenarios. Both benchmarks lack sufficient tests with custom communicators. In the HPC data set, we observe the usage of, e.g., `Comm_split` for collective communication that is underrepresented in the benchmarks. Constructed communicators may introduce mismatches not present in codes with `MPI_COMM_WORLD`-only usage. Hence, these benchmarks should include such tests to assess correctness tool handling.

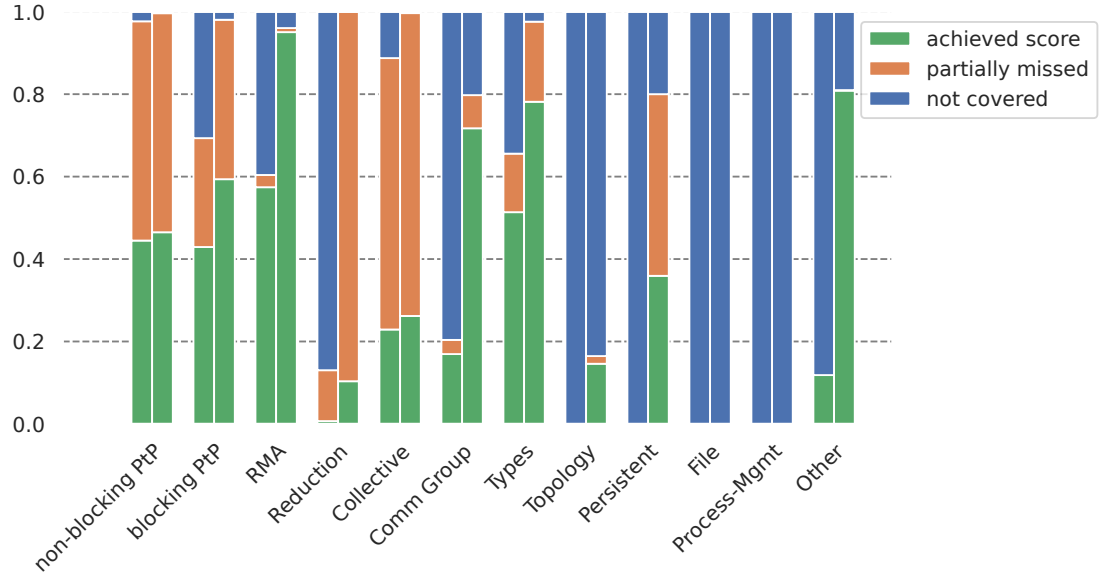
Which categories are not in the HPC data set? For MBI, we observe only the collectives `MPI_Iexscan` and `MPI_Iscan` that are absent in the HPC data set. COBE includes more calls absent from the HPC data set, such as several non-blocking collective and one-sided routines, which mostly originate from the integrated MPI library test suite.

10.5. Conclusion

Our study compares the test coverage of the MPI-CorrBench, and the MPI Bugs Initiative correctness benchmarks w.r.t. MPI usage patterns in a set of 96 HPC codes. We assess the presence of MPI calls *and* their parameters in both benchmarks. Scoring is based on the weight (frequency of appearance) in the HPC data set, reflecting the importance of specific MPI usage patterns for developers. Correctness benchmarks should encompass these essential MPI calls to ensure coverage of correct (and erroneous) real-world MPI usage. We identified that COBE lacks crucial reduction operations, and MBI lacks tests with the `MPI_DOUBLE` datatype. In conclusion, our scoring guides for improving the coverage of MPI usage patterns in HPC applications.



(a) MBI



(b) COBE

Figure 10.4.: Scoring by categories and their *total* potential score contribution. (a) *Achieved score* is the actual covered MPI usage pattern, (b) *partially missed* represent any missing argument variations of MPI calls that contributed to the score and (c) *not covered* are missing MPI calls.

11. MPI-BugBench: A Framework for Assessing MPI Correctness Tools

Bibliographic Information

The contents of this chapter has previously been published in:

Tim Jammer, Emmanuelle Saillard, Simon Schwitanski, Joachim Jenke, Radjasouria Vinayagame, Alexander Hück, and Christian Bischof. "MPI-BugBench: A Framework for Assessing MPI Correctness Tools". In: *Proceedings of the 31th European MPI Users' Group Meeting*. EUROMPI 2024. Perth, Australia, Sept. 2024. DOI: 10.1007/978-3-031-73370-3_8

The contribution of the author of this thesis is summarized as follows:

The original works of MPI-Corrbench, MPI Bugs Initiative and RMA Racebench where developed independently. As one of the lead authors, Tim Jammer enhanced the test generation infrastructure to ensure that the generated test cases achieved high coverage of real-world usage patterns. Tim Jammer integrated all test cases for point-to-point operations, while Emmanuelle Saillard incorporated all test cases for collective operations, and Simon Schwitanski focused on RMA test cases. Radjasouria Vinayagame advised on the RMA testcases. Emmanuelle Saillard and Simon Schwitanski adopted the evaluation scripts and run the evaluation of the Correctness tools. Joachim Jenke advised on the correctness tool developers perspective. In terms of the manuscript, Alexander Hück drafted Sections 1, 2 and 6, while Tim Jammer focussed on 3, 4.1 and 5, with section 4.2 mainly being contributed by Emmanuelle Saillard. All authors and especially Christian Bischof provided invaluable feedback on the manuscript's text and the presentation of the results. All authors agree with the use of the publication for this thesis.

Abstract

MPI's low-level interface is prone to errors, leading to bugs that can remain dormant for years. MPI correctness tools can aid in writing correct code but lack a standardized benchmark for comparison. This makes it difficult for users to choose the best tool and difficult for developers to gauge their tools' effectiveness. MPI correctness benchmarks, MPI-CorrBench, the MPI Bugs Initiative, and RMARaceBench have emerged to address this problem. However, comparability is hindered by having separate benchmarks, and none fully reflects real-world MPI usage patterns. Hence, we present MPI-BugBench, a unified MPI correctness benchmark replacing previous efforts. It addresses the shortcomings of its predecessors by providing a single, standardized test harness for assessing tools and incorporates a broader range of real-world MPI usage scenarios. MPI-BugBench is available at https://git-ce.rwth-aachen.de/hpc-public/mpi-bugbench.

11.1. Introduction

The Message Passing Interface (MPI, [148]) enables distributed computations in high-performance computing (HPC). MPI, however, requires users to manually specify details like datatypes or message tags, which is error-prone. This complexity has led to dormant bugs only uncovered after several years [42, 59].

To address this issue, several MPI correctness tools have been developed using static [59], dynamic [42, 153], or a combined analysis [57] to detect issues such as process deadlocks or invalid argument values. While numerous correctness tools exist, it is difficult to compare their performance in terms of error detection capabilities objectively, implying the need for a standardized benchmark. A general evaluation framework is necessary for users to select the most suitable tool for their particular MPI usage. It allows tool developers to measure their tools' capabilities against others effectively.

As a consequence, two MPI correctness benchmarks were introduced: MPI-CorrBench (COBE, [135]) and the MPI Bugs Initiative (MBI, [134]). They offer a collection of tests, ranging from small unit tests to mini-applications and covering correct and erroneous usage of the MPI library. These tests enable the assessment of the precision of correctness tools by comparing their output against expected results. In addition, RMARaceBench (RRB, [149]) was created more recently. It is a smaller test set focused on data races of MPI one-sided communication (RMA) and other RMA programming models only.

However, separate benchmarks have created complications for both users and developers. A unified benchmark is necessary for direct comparison and evaluation. Additionally, benchmarks need to capture the diversity of real-world MPI usage, which otherwise limits their effectiveness in guiding tool development to address users' practical challenges.

To investigate the real-world relevance of the existing benchmarks, we conducted a study [146] comparing the MPI calls and their arguments present in COBE and MBI with a dataset of real-world MPI usage [147]. While neither fully represents real-world MPI usage, each benchmark has unique strengths (e.g., MBI's coverage of persistent operations and COBE's of datatypes) that complement each other.

Hence, we developed a unified correctness benchmark, MPI-BugBench, to improve tool comparability and real-world relevance. It combines the strengths of COBE, MBI, and RRB, thereby improving coverage of real-world MPI usage patterns. In summary, we make the following contributions:

- MPI-BugBench, a unified framework for consistent tool evaluation and comparison, which combines the strengths of COBE, MBI, and RRB.
- A domain-specific MPI test generator that enables the creation of tests with varying levels of error complexity, from basic misuse to granular real-world scenarios or exhaustive error combinations.
- A comprehensive evaluation of three state-of-the-art MPI correctness tools, MUST [42], ITAC [153], and PARCOACH [57], using MPI-BugBench.

The rest of the paper is structured as follows. In Section 11.2, we present existing MPI correctness benchmarks and discuss our definition of real-world MPI usage. Section 11.3 discusses our joint effort MPI-BugBench. In particular, we discuss our MPI error classification and automatic MPI test generator, and reflect on our approach's real-world MPI coverage. In Section 11.4, we evaluate three state-of-the-art MPI correctness tools using MPI-BugBench. Section 11.5 discusses limiting test code generation to real-world scenarios (as opposed to exhaustive generation) and the complications of classifying errors based on the output of correctness tools. Finally, we conclude and outline future work in Section 11.6.

11.2. Previous Work

In previous work [146], we evaluated COBE and MBI's real-world applicability by applying a feature-scoring method based on our previous MPI study [146] that resulted in a dataset of 96 HPC codes (henceforth called HPC dataset). Our scoring method assesses how closely the MPI usage patterns in these correctness benchmarks reflect those found in the HPC dataset. Specifically, COBE covers approximately 40% of the MPI usage observed in the HPC dataset, while MBI covers around 30%.

The relatively low scores of these benchmarks can be attributed to their creation predating our MPI study. Previous studies, e.g., [105], primarily focused on MPI function calls without considering parameter usage patterns. These patterns provide a more detailed understanding of how users interact with MPI, such as determining the most common datatypes in collective operations.

In Section 11.2.1, we provide a brief overview of the correctness benchmarks COBE, MBI, and RRB. We elaborate on our scoring algorithm in Section 11.2.2. It serves as the guideline for designing MPI-BugBench's test code generator described in Section 11.3.

11.2.1. Correctness Benchmarks

MPI-CorrBench COBE [135] contains 510 small-scale C language tests, including 202 correct and 308 incorrect codes. In particular, the correct test cases have been extracted from an MPI library implementation test set. The incorrect codes, on the other hand, have been hand-written. Each incorrect test contains a brief description of the error. COBE also includes well-known (C/C++) HPC mini-apps where the authors manually introduced specific errors. In COBE, erroneous arguments, erroneous program flow, and mismatching arguments across communication calls are the key error types.

The MPI Bugs Initiative MBI [134] contains 1668 C language codes, including 682 correct and 986 incorrect codes. To generate these test cases, MBI uses a template engine. Templates for the various MPI categories contain placeholder tokens that get replaced by (in-)correct usage of MPI calls to generate the final test case. Each test defines a header that describes the intent of each code and specifies how to execute and evaluate it. In MBI, the errors are categorized by the scope in which they can arise: (a) single call (invalid parameter), (b) single process (resource leak, request lifecycle, and local concurrency), and (c) multi processes (parameter matching, message race, call ordering, and global concurrency).

RMARaceBench RRB [149] focuses on different data race test cases for the remote memory access (RMA) models MPI RMA, OpenSHMEM [123], and GASPI [81]. It consists of 107 different small-scale C codes targeting MPI RMA, of which 43 are correct and 64 are incorrect. Like MBI, the test cases are semi-automatically generated based on code templates, covering different combinations of RMA communication and synchronization methods. The errors are categorized into (a) local concurrency issues, i.e., a user-specified local buffer of an RMA operation is accessed before completion, (b) global concurrency issues, i.e., concurrent conflicting RMA accesses from different processes are not correctly synchronized, (c) incorrect atomicity, i.e., the atomicity semantics of MPI RMA are violated, and (d) hybrid races in combination with threading via OpenMP.

11.2.2. MPI Feature Usage Analysis

In a previous study [147], we developed a static code analysis toolchain that (a) extracts MPI calls and their arguments, (b) applies an MPI feature classification (closely following the MPI standard's categorization), and (c) cross-references MPI argument handles across detected function calls, see Listing 11.1.

```
1 MPI_Datatype struct_type; // Opaque type handle
2 // Construct struct type:
3 MPI_Type_create_struct(num_members, block_length,
4 offsets, member_types, &struct_type);
5 MPI_Type_commit(&struct_type); // Make MPI aware of struct type
6 MPI_Send(buffer, 1, struct_type, ...);
```

Listing 11.1: Example of constructing a struct type. We need to consider arguments of MPI functions to deduce that the send operation uses the struct datatype.

This resulted in a dataset of MPI usage at an argument granularity for 96 real-world MPI codes. We use this dataset to assess whether the existing MPI correctness benchmarks and MPI-BugBench reflects real-world MPI usage. In the following, we describe our scoring workflow for this assessment developed in our previous study [146].

Scoring MPI Usage

For each MPI feature category, e.g., point-to-point (PtP) operations, we calculate a percentage representing the proportion of (real-world) MPI usage patterns that are covered by the correctness benchmark. To that end, for this MPI usage pattern scoring, we consider: (1) The MPI call, (2) the *Datatype* and *Reduction Operator*, (3) the *Count* (4) and if wildcards, such as `ANY_TAG` are used; (5) for collective operations, the root *Rank* is also important to the usage pattern. To be counted towards usage scoring, for each individual MPI call in the real-world dataset, the aforementioned aspects must be present in the MPI correctness benchmark.

Additionally, the score for each usage pattern is based on its frequency in our real-world dataset, as shown in Fig. 11.1. As the dataset contains operations with varying arguments, a 100% score requires all operations for each MPI feature category to be present with all arguments in the correctness benchmark. Consider a dataset with six scatter operations and four broadcast operations. If all six scatter operations fully match, but only one out of four broadcast operations match (e.g., one uses `MPI_FLOAT` and the others use `MPI_DOUBLE`), the score of the benchmark would be 70% for collectives. The full details of this scoring workflow are explained in [146]. The reasoning behind testing the same errors with several distinct MPI usage patterns is further detailed in Section 11.3.2.

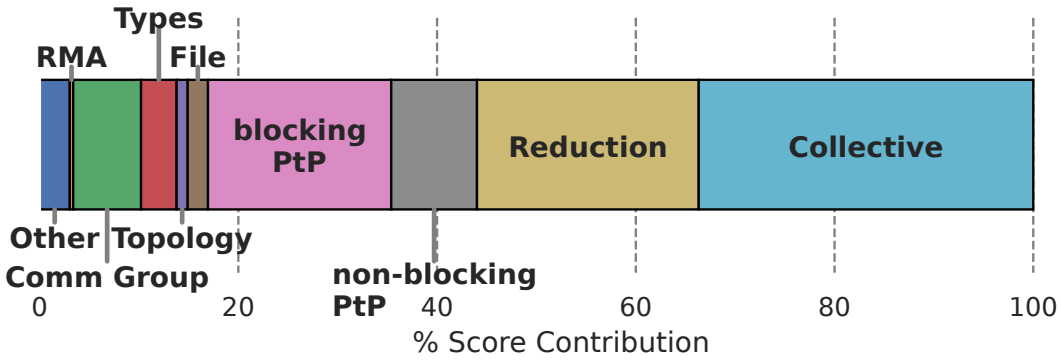


Figure 11.1.: Weighting of MPI categories, adapted from [146]. Collectives and reductions are predominantly used, followed by PtP operations. This indicates that correctness benchmarks should focus on these categories first to improve their scoring.

11.3. Design of MPI-BugBench

In the design of MPI-BugBench (MBB), we adopted and expanded upon the test code generation idea of the MPI Bugs Initiative while integrating the test cases from MPI-CorrBench (COBE) and RMARaceBench (RRB). MBB contains different unit tests with exactly one MPI usage error per test case. This allows for a more fine-grained evaluation of a correctness tool, as one can individually compare the tool's performance for specific errors. The tools are evaluated in a container-based infrastructure to ensure portability (see Section 11.4.2). In this section, we first discuss the different types of errors we consider in Section 11.3.1, while Section 11.3.2 discusses different instantiations of errors that can be generated. The explanation of how the test cases are generated follows in Section 11.3.3.

11.3.1. Error Types Covered

Different kinds of programming errors in MPI may lead to different (non-deterministic) failures at runtime. Both COBE and MBI provide similar classifications of such MPI programming errors. MPI-BugBench mainly adapts the classification introduced by MBI. For detailed code examples of the different error classes, we refer to COBE [135] and MBI [134].

In general, MPI errors can be categorized into three different categories:

1. **Single call errors**: These errors are only related to local MPI functions and can be detected by only analyzing the parameters of a given MPI function.
2. **Process-local errors**: These errors often consist of an inconsistency between the local context of a process and the parameters of a given MPI call in that process. Thus, the detection of these errors requires analysis of local process information.
3. **Multi-processes errors**: These errors result from the interplay of multiple application processes, such as a deadlock.

Single call errors

1. Invalid Parameters: This category contains invalid parameters in an MPI call, such as a negative value for a rank.

Process-local errors

2. Resource Leak: Any improper destruction of MPI resources (e.g., datatype, request, communicators) leads to a resource leak.

3. Initialization of MPI: Wrong initialization or finalization of MPI can lead to errors, e.g., if messages are sent before MPI is initialized.

4. Request Lifecycle: Request lifecycle errors occur if, e.g., a wait to complete a nonblocking operation is missing.

5. Local Concurrency: A local concurrency error occurs when a process accesses a memory region asynchronously read or written by MPI. This type of error is produced with nonblocking and one-sided communication. An example would be using a message buffer before the nonblocking operation completes.

6. Epoch Lifecycle: An epoch lifecycle error occurs when MPI RMA operations are wrongly synchronized by, e.g., mixing different RMA synchronization modes (fences and locks) or performing an RMA operation outside an access epoch.

Multi-processes errors

7. Message Race: Wildcard receive calls can lead to non-deterministic message matching with potential senders, possibly leading to deadlocks.

8. Parameter Matching: Parameter matching corresponds to MPI calls matched with incompatible arguments. An example is a collective operation, where the processes do not agree on a root, which can result in a deadlock.

9. Call Ordering: Wrong ordering of MPI calls can lead to a call mismatch, e.g., when all processes call a receive operation before any process calls a send operation, resulting in a deadlock.

10. Global Concurrency: Global concurrency errors occur if two or more processes access the same memory region (at least one access is a write). An example would be two concurrent `MPI_Put` operations accessing the same memory region at the target process.

11.3.2. Error Instantiations Covered

Apart from testing for different *kinds* of errors explained in Section 11.3.1, MBB also tests for different instances of the *same kind* of error. For example, a datatype mismatch between a `double` and an `integer` may result in a message size mismatch. It may be more easily spotted by a correctness checking tool than a different mismatch, resulting in the same message length (e.g., between `integer` and `unsigned integer`). Therefore, MBB can be used to assess the coverage of a tool regarding the different possible MPI errors supported and can additionally be used to assess the implementation quality regarding the reliability of finding different instances of the same type of error.

Testing for all the possible instances of an error (e.g., wrongfully matching each datatype with all other types) leads to a high combinatorial complexity. MBB can generate over four million test cases when all combinations are considered exhaustively.

Several studies like [105, 147] have shown that most applications only use a limited subset of MPI functionality. Hence, testing a tool with all possible MPI usage patterns is probably unnecessary. Furthermore, the study by Hück et al. points out that derived MPI datatypes are more commonly used in point-to-point operations than in collective operations [147].

In order to limit the number of cases generated and to facilitate a more efficient evaluation of correctness tools, MPI-BugBench contains different *coverage levels* to generate different sets of test cases gradually:

1. **Basic cases**: The basic cases include only one instance of an error (e.g., one datatype mismatch).
2. **Sufficient coverage**: This level contains multiple instances of the "same" kind of error, that should be sufficient to evaluate if specific MPI errors are supported by a tool by containing multiple examples, covering all possible values for the mismatching parameters involved. In terms of datatype mismatches, for example, for each MPI datatype, at least one mismatch is included. In order to only include usage patterns that can be observed in the real world, we refine this level into 2.1 "Sufficient coverage of real-world patterns" and 2.2 "Sufficient coverage of all possible MPI usage pattern".
3. **Full Testcaseset**: The full set contains all possible instantiations of an error. In the case of datatype mismatches, all possible mismatches between all possible MPI datatypes are included. Again, this level can be refined into 3.1 "Full coverage

Table 11.1.: Number of test cases per feature.

	Level 1	Level 2.1	Level 2.2	Level 3.1	Level 3.2
P2P	49	870	24,116	2,789	3,004,968
COLL	40	774	23,937	3,246	1,121,865
RMA	39	415	415	1,898	1898
Total	128	2,059	48,539	7,933	4,128,731

of real-world usage patterns" and 3.2 "Full coverage of all possible MPI usage patterns" to include only usage patterns found in the real world or all theoretically possible ones.

The number of cases generated for the five levels for point-to-point, collective, and RMA operations is shown in Table 11.1.

In order to determine which usage patterns occur in the real world, MBB uses the data set collected by Hück et al. [147], excluding the Fortran cases, as Fortran is currently not part of MBB. MBB can also be used to generate test cases tailored to a more limited set of use cases by replacing the complete real-world data set with a different one, e.g., for one specific application.

11.3.3. Test Case Generation

MBB's test generation builds upon the infrastructure of the MPI Bugs Initiative and enhances it in several ways. The original test case generation infrastructure relied on text replacement. We improved it to generate a variety of test cases automatically based on test generator scripts. Each test generator is a Python class that implements a generator function. The infrastructure will execute all applicable generators to produce the test case set. The user can target a specific MPI version or feature, with the effect of cases not fitting the user's criteria being discarded.

In particular, Listing 11.2 shows the generation of an invalid buffer error. In line 4, a default point-to-point template is instantiated. Then, the `generate` function needs to find the call where the buffer argument should be replaced with the invalid one. Line 5 of Listing 11.2 iterates over the send and receive call of the default template, while line 7 selects the send call. The `MPICall` object allows setting an invalid buffer argument (line 8). Line 9 marks the erroneous MPI call for later evaluation. The only thing left for the

```
1  def generate(self, generate_level, real_world_score_table):
2    for func in mpi_send_funcs :
3      for buf_to_use in [ "NULL", 'MPI_BOTTOM', 'MPI_IN_PLACE']:
4        tm = get_send_recv_template(func, "mpi_irecv") # get a TemplateManager
5        for call in tm.get_instruction(identifier="MPICALL", return_list=True):
6          # send is executed by rank 1 in default template
7          if call.get_rank_executing() == 1:
8            call.set_arg("buf", buf_to_use) # set buffer to invalid value
9            call.set_has_error() # mark where the error is
10         # set an appropriate description for the error:
11         tm.set_description("InvalidParam-Buffer-" + func, # short description
12                            "Invalid Buffer: "+buf_to_use) # long description
13         yield tm
```

Listing 11.2: Illustration of the test-case generation function for invalid buffer errors.

`generate` function is to set an appropriate error description in line 11 of Listing 11.2, before yielding the instantiated `TemplateManager` to the generation infrastructure. The generation infrastructure will then generate the resulting erroneous code (illustrated in Listing 11.3) into a file for later compilation and execution with a correctness tool. The Python yield construct turns the function into a generator. Iteratively calling the function returns different errors. In the example of Listing 11.2, multiple different invalid buffer argument errors are created for all flavors of MPI point-to-point send functions. The arguments to the `generate` function limit the generated instantiations of an error, as it may lead to redundancy to test for all possible circumstances where an error can occur, as explained in Section 11.3.2.

11.4. Evaluation of Correctness Tools

This section first discusses the real-world coverage of the generated test cases in Section 11.4.1. We then evaluate three state-of-the-art MPI correctness tools using the generated test cases in Section 11.4.2.

11.4.1. Real-World Applicability

MPI-BugBench's approach vastly improves its real-world applicability when considering the coverage level 2.1, as described in Section 11.3.2. The real-world applicability, as

```c
// Description: Invalid Buffer: NULL // header is shortened
int main(int argc, char **argv) {
  // init MPI and setup rank variable
  int *buf = (int *)calloc(10, sizeof(int));
  if (rank == 0) {
    MPI_Irecv(buf, 10, MPI_INT, 1, 0, MPI_COMM_WORLD, &mpi_request_0);
    MPI_Wait(&mpi_request_0, MPI_STATUS_IGNORE); }
  if (rank == 1) {
    /*MBBERROR_BEGIN*/ MPI_Send(NULL, 10, MPI_INT, 0, 0,
    MPI_COMM_WORLD); /*MBBERROR_END*/ }
  // free and finalize
}
```

Listing 11.3: Excerpt of one error code produced by the generator shown in Listing 11.2.

explained in Section 11.2.2, is illustrated in Fig. 11.2. High coverage of the most relevant real-world usage pattern is obtained in most categories, with an overall coverage score of more than 75%. MBB has surpassed the original works of MPI-CorrBench and MPI Bugs Initiative in all categories. The current lack of coverage for the *Types* category is explained in a surprisingly high number of conversion functions like `Type_f2c`, which our benchmark currently does not cover as Fortran is not supported. For the `other` category, the lack of coverage is mostly due to MPI I/O which is currently not covered by MPI-BugBench. Nevertheless, the most important aspects of real-world MPI usage are covered, which also can be seen in Table 11.2, where the overall coverage scores are compared between the three benchmarks.

We note that using a higher generation level of MBB does not further increase the coverage score. The reason is that the coverage score only counts if a usage pattern is included at least once. As explained in Section 11.3.2, the idea of coverage level 2.1 is that each usage combination is part of at least one instance of every applicable error. However, not all erroneous combinations are tested to allow for a more efficient tool evaluation.

11.4.2. Evaluation of Correctness Tools

We use MPI-BugBench to evaluate three active MPI correctness tools relying on different techniques to detect errors: ITAC (v2021.3), MUST (v1.10.0), and PARCOACH (v2.4.0). Intel Trace Analyzer and Collector (ITAC) [153] profiles and analyzes MPI

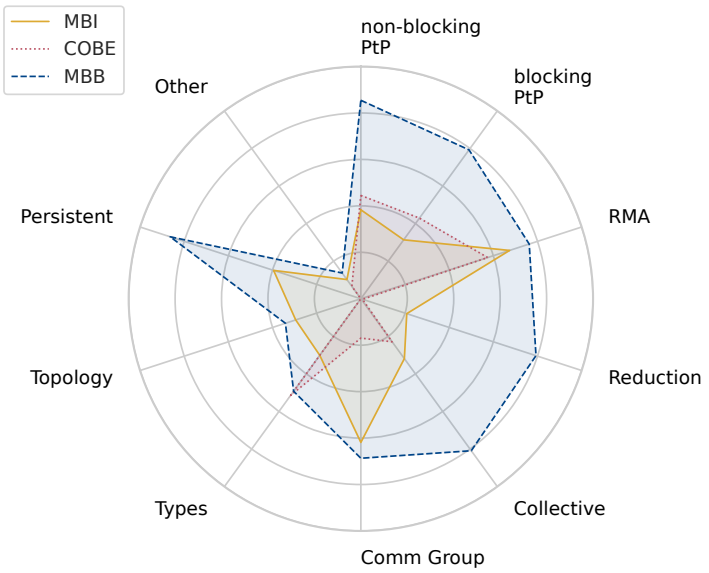


Figure 11.2.: Comparison of real-world applicability of MPI Correctness Benchmarks. MBI (orange) denotes the MPI Bugs Initiative. COBE (blue) denotes MPI-CorrBench. MBB (green) denotes MPI-BugBench with coverage level 2.1. We used the methodology proposed by Hück et al. [146] for evaluation, without including the Fortran codes in the real-world dataset (see Section 11.2.2).

programs to check their correctness. It intercepts MPI calls and generates trace files that can be analyzed to understand the program's behaviors. MUST [42] detects different kinds of MPI errors such as deadlocks, type mismatches, or invalid arguments during the execution. For extended type correctness checks of user-specified buffers, it relies on TypeART [95] and LLVM 14. PARCOACH [57, 151] detects collective and one-sided operation misuse with a static/dynamic approach. It emits warnings for potential errors found at compile-time and verifies these potential errors during the execution of programs. PARCOACH's static analysis is based on LLVM 15. In this section, we only used the static analysis of PARCOACH.

We use coverage level 2.1 of MBB that generates roughly 2000 test codes. The experiments used the MBB infrastructure in a Docker image based on Debian 12, which contained all tool dependencies. The MUST container uses MPICH 4.0.2, the PARCOACH container uses Open MPI 4.1.4, and the ITAC container uses Intel MPI 2021.12.

Table 11.2.: Total real-world coverage score. The scoring is described in Section 11.2.2.

	Erroneous (%)	Correct (%)	Total (%)
MBB coverage level 2.1	74.94	51.15	74.97
MBB coverage level 1	28.45	19.85	28.74
COBE	25.61	47.85	51.68
MBI	28.32	28.32	28.32

Since some tests may crash or hang up in a deadlock, we specified a timeout of 120 seconds for each test execution. To speed up the overall runtime of the benchmark, the MBB infrastructure supports parallel test execution. We performed the evaluation on a cluster node with 96 cores, using a pool of 16 runners so that 16 tests could run in parallel. This leaves enough spare cores to start additional tool processes, as required by MUST. On the described setup, the execution of the tests requires 2 minutes for PARCOACH, 27 minutes for ITAC, and 58 minutes for MUST. PARCOACH is the fastest tool, as its static analysis does not execute the tests.

Result Categorization

The resulting tool output of each test is classified into exactly one of the following categories:

- True Positive (TP): Error reported on an erroneous test case.
- True Negative (TN): No error reported on a correct test case.
- False Positive (FP): Error reported on a correct test case.
- False Negative (FN): No error reported on an erroneous test case.
- Compilation Error (CE): The test case could not be compiled with the tool.
- Runtime Error (RE): The tool execution on a correct test case crashed or ran into a timeout (here: 120s)

A compilation error (CE) may occur when the tool does not support all MPI calls in the test, e.g., due to an outdated MPI library compatibility. An execution is classified as runtime error (RE) when the execution on a *correct* test case crashes or runs into a timeout without any error report. If the tool falsely reports an error on a correct test

case and the execution crashes or runs into a timeout, this still counts as FP. Thus, an issued error report always takes precedence for FP in the classification over a runtime error (RE). Further, an execution on an erroneous test cases is never classified as runtime error, because the test may crash or timeout itself, independently of the tool.

Tool Results

Table 11.3 shows the results of each tool for all tests on coverage level 2.1. The first part of the table shows the aforementioned classification of test executions for the different tools. The rest of the table gives the following derived metrics, as also defined by MBI:

- Coverage $Cov = 1 - \frac{CE}{Total\ tests}$, Conclusiveness $Cc = 1 - \frac{CE+RE}{Total\ tests}$
- Specificity $S = \frac{TN}{TN+FP}$, Recall $R = \frac{TP}{TP+FN}$, Precision $P = \frac{TP}{TP+FP}$
- F1 Score $F1 = \frac{2 \cdot P \cdot R}{P+R}$, Overall Accuracy $OA = \frac{TP+TN}{Total\ tests}$

Coverage and conclusiveness demonstrate the robustness of a tool, i.e., the ability to compile and draw a diagnostic on codes. Specificity, recall, precision, and F1 score are standard metrics used to evaluate tools. Specificity measures the ability to avoid identifying errors in *correct* codes, while recall measures the ability to find existing errors. Precision is the confidence in TN results, and F1 score is the overall bug-finding quality. Finally, overall accuracy gives the proportion of correct diagnostics for all tests when considering compilation and runtime errors. The last row of the table gives the results of an ideal tool.

PARCOACH and MUST compile all test cases and, therefore, have a coverage of 1. ITAC comes with Intel MPI that currently does not support MPI 4.0 features such as partitioned communication. Due to undefined MPI functions, this leads to compilation errors (CE) for those test cases. With MUST, 9 executions on correct tests resulted in runtime errors (RE) that are related to the internal type verification. It does not consider less frequently used data types such as `MPI_DOUBLE_INT` or `MPI_2INT` correctly. ITAC did not crash or timeout on any test. Since PARCOACH analyzes the codes only statically, it also cannot have any runtime errors.

ITAC has the best classification results of the tools, with an overall accuracy of 0.85 and an F1 score of 0.92. MUST achieves a similarly high precision of 0.99 compared to ITAC but detects fewer issues overall leading to a recall of 0.68. Still, the F1 score of MUST is 0.81. PARCOACH is focused on a small subset of errors, in particular collective

Table 11.3.: Tool evaluation against MPI-BugBench on coverage level 2.1. The best results are in bold.

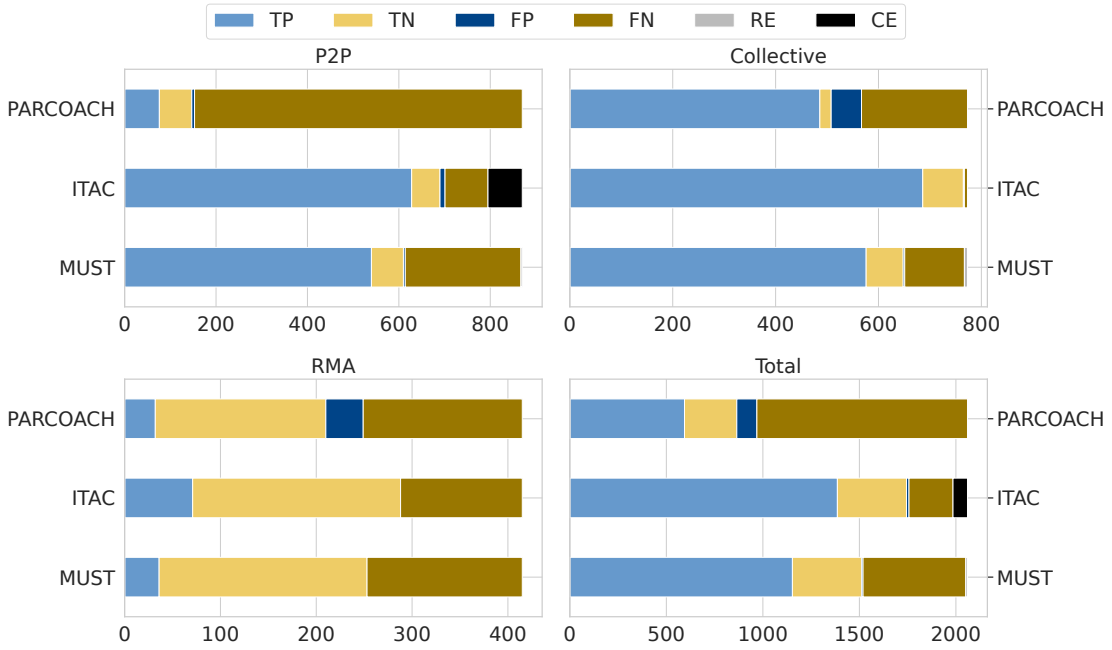
Tool	Errors		Results				Robustness		Usefulness				**OA**
	CE	**RE**	**TP**	**TN**	**FP**	**FN**	**Cov**	**Cc**	**S**	**R**	**P**	**F1**	
ITAC	75	0	**1386**	**358**	13	**228**	0.96	0.964	0.95	**0.82**	**0.99**	**0.92**	**0.85**
MUST	**0**	9	1153	359	**7**	532	**1**	0.996	**0.96**	0.68	**0.99**	0.81	0.73
PARCOACH	**0**	**0**	594	271	104	1091	**1**	**1**	0.72	0.35	0.85	0.50	0.42
Ideal tool	*0*	*0*	*1685*	*375*	*0*	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*

CE: Compilation Error, RE: Runtime Error, TP: True Positive, TN: True Negative, FP: False Positive, FN: False Negative, Cov: Coverage, Cc: Conclusiveness, S: Specificity, R: Recall, P: Precision, F1: F1 Score, OA: Overall Accuracy

operations. Thus, it returns many false negatives and has low scores for most of the metrics, resulting in an F1 score of 0.50.

Fig. 11.3 illustrates the tool results individually for the different MPI features. MUST and ITAC perform similarly for the P2P and Collective tests. Both tools have a larger number of false negatives (FN) because they fail to detect issues in erroneous test cases containing less frequently used MPI calls or data types. Since PARCOACH focuses on error detection in collective operations, it performs well on those tests, but falls short on the P2P tests. For the RMA test cases, MUST only detects invalid parameter errors. ITAC detects invalid parameter and additionally epoch lifecycle errors. Both tools do not detect local concurrency or global concurrency errors in RMA. Only PARCOACH detects some of the RMA local concurrency issues but also detects such issues in correct test cases, leading to some false positives (FP). An extension of MUST [145] to check for local and global concurrency errors in RMA programs has not been integrated into the current release and thus has not been tested. Similarly, PARCOACH implements a dynamic analysis [150, 151] for RMA that significantly improves the detection quality on local and global concurrency errors in RMA, but has also not been tested in our infrastructure. For both MUST and PARCOACH, we plan to integrate and incorporate those extended RMA analyses in future work.

We compared the results shown in Table 11.3 on coverage level 2.1 with those on coverage level 1. The derived metrics show similar results on both levels. Nevertheless utilizing level 2.1 does reveal some additional shortcomings of the tools, that cannot be uncovered by using only coverage level 1. An example are the runtime errors (RE) of MUST when



CE: Compilation Error RE: Runtime Error, TP: True Positive, TN: True Negative, FP: False Positive, FN: False Negative

Figure 11.3.: Tool evaluation using MPI-BugBench including a breakdown for each MPI feature category.

utilizing less frequently used datatypes, as these runtime errors are not present, when testing MUST only with coverage level 1. As these cases are rare however, they do not significantly impact the overall scoring of the tools. This means that the tools do support a broad range of real-world usage patterns, with some errors in some rather rare cases.

Summarizing the results, the three tested state-of-the-art tools ITAC, MUST, and PARCOACH show a good coverage on real-world usage patterns. For some less frequently used MPI features, the tools sometimes do not detect errors correctly or crash. With MBB, we provide a test set that should motivate tool developers to improve their tool's classification quality by also considering corner cases that are still relevant in real-world MPI programs.

11.5. Discussion

Compared to the previous benchmarks COBE, MBI, and the focused benchmark RRB, the test cases of MPI-BugBench are guided by a dataset of real-world MPI usage. Although, MPI-BugBench can generate all possible erroneous MPI usage combinations, we only generate a limited subset for our evaluation. As discussed in previous work [146], a high coverage score does not indicate rigorous testing. The coverage score only counts MPI usage patterns; it does not account for the possible erroneous usage of any particular pattern. Hence, our scoring is merely a guide for generating test cases with MPI usage patterns of real-world relevancy. Furthermore, MPI-BugBench combines all error types included in COBE and MBI to encompass as many MPI error combinations as possible. To that end, the number of erroneous combinations must be balanced against the time required to execute those tests. In our opinion, MPI-BugBench is well suited to find this balance with the different test generation levels included.

Another critical point is evaluating the tool's feedback: Are the tool's error messages helpful in pointing out the root cause of an error? In this work, the evaluation discussed in Section 11.4.2 only checks whether the tool reported an error on a test case or not. It does not verify the usefulness of the error report. From our perspective, there are two aspects that contribute to the usefulness of a tool. First, the correct error class should be reported by the tool. For example, if a tool reports a data race on a test case that contains a deadlock, then this error report is not useful at all. Further, if the error report does not include the affected source code lines, users have to locate the root cause of the error on their own which is not applicable to larger codes. Moreover, since the tools' output is not standardized, it may be difficult to argue whether the expected error was actually discovered. Nevertheless, MBB facilitates in-depth analysis of the origin of the error by marker comments pointing to the location of the erroneous MPI usage in the generated test cases. This work focuses on how real-world usage data can guide the test cases, but we plan to add a tool's feedback analysis like in [36, 85].

11.6. Conclusion

In this work, we introduce MPI-BugBench, a unified benchmark for assessing MPI correctness tools. It consolidates previous efforts, offering a standardized test harness that mirrors real-world MPI usage in HPC codes. Utilizing a test code generator, MPI-BugBench creates tests with varying levels of error complexity, covering basic

misuses to exhaustive error combinations. This allows for detection of bugs in the tools implementation for some more rarely used cases.

We evaluate three state-of-the-art MPI correctness tools using 2,060 generated codes. These test codes cover 75% of MPI usage patterns identified in 96 HPC codes, doubling the coverage compared to previous MPI correctness benchmarks. The test code generator produces test codes, which can be correctly compiled and run by the static and dynamic correctness tools without (unexpected) issues. The dynamic tools ITAC and MUST have a relatively high degree of real-world applicability with an overall accuracy of about 85% and 73%, respectively. PARCOACH is limited in focus and only performs well on collective operations and some RMA features. However, it has a clear advantage regarding run time, as it only utilizes static analysis without the need for the actual execution of the tests.

For future work, we plan to extend MPI-BugBench with a more detailed tools report analysis to analyze the helpfulness of the error messages given by the tools, in particular the reported error type and source code lines. Although 80% of real-world usage patterns are already covered by MPI-BugBench, we further want to expand the scope of our correctness benchmark suite by incorporating currently not covered MPI features such as MPI I/O. Another way to increase the scope of MPI-BugBench is to include Fortran test cases. Additionally, we want to add the hybrid OpenMP+MPI errors from COBE alongside other MPI+X programming models. We also want to incorporate more complex error cases, such as nondeterministic instances of errors, that depend on the input parameters.

In summary, MPI-BugBench serves as a comprehensive and unified benchmark reflecting actual MPI usage in HPC environments. The infrastructure is available at https://git-ce.rwth-aachen.de/hpc-public/mpi-bugbench.

Acknowledgments

This work was supported by the Hessian Ministry for Higher Education, Research and the Arts through the Hessian Competence Center for High-Performance Computing, and by the Federal Ministry of Education and Research (BMBF) and the states of Hesse and North Rhine-Westphalia as part of the NHR program.

Disclosure of Interests

The authors have no competing interests to declare that are relevant to the content of this article.

Part IV.

Aiding developers to use Modern MPI features

12. Automatic Detection of MPI Assertions

Bibliographic Information

The contents of this chapter has previously been published in:

Tim Jammer, Christian Iwainsky, and Christian Bischof. "Automatic Detection of MPI Assertions". In: *High Performance Computing. ISC High Performance 2020. Lecture Notes in Computer Science*. virtual: Springer International Publishing, 2020, pp. 34–42. DOI: 10.1007/978-3-030-59851-8_3

The contribution of the author of this thesis is summarized as follows:

As the leading Author, Tim Jammer contributed the Idea, Implementation and evaluation. Christian Iwainsky and Christian Bischof provided invaluable feedback on the manuscript's text and the presentation of the results. All authors agree with the use of the publication for this thesis.

Abstract

The 2019 MPI standard draft specification includes the addition of defined communicator info hints. These hints are assertions that an application makes to an MPI implementation, so that a more optimized implementation is possible. The 2019 draft specifications defines four assertions: `mpi_assert_no_any_tag`, `mpi_assert_no_any_source`, `mpi_assert_exact_length` and `mpi_assert_allow_overtaking`. In this paper we will explore the capability of a Clang/LLVM based static analysis to check whether these assertions hold for a given program. With this tool, existing codebases can benefit from this new addition to the MPI standard without the need for costly human intervention.

12.1. Introduction

The upcoming version 4.0 of the MPI standard includes many changes. In particular, "the largest changes are the addition of persistent collectives, application info assertions, and improvements to the definitions of error handling" [108].

With the addition of defined application info assertions, it is possible to hint that MPI is used in a certain way enabling the possibility of a more optimized implementation; the `allow_overtaking` assertion is of particular interest. Using this assertion, an application can notify an MPI implementation that its correctness does not depend on the strict ordering of messages as described in section 3.5 of the standard [66]. If an application asserts that the overtaking of messages is allowed, or cannot happen due to the use of different tags and synchronization, an MPI implementation may entirely skip one of the costly phases of message matching ensuring the order of messages is preserved [100]. As Dang et al. point out [71], matching the messages in the correct order is especially costly in the case of multithreaded MPI.

The `no_any_tag` and `no_any_source` assertions quite straightforwardly tell an implementation that the `MPI_ANY_TAG` and `MPI_ANY_SOURCE` constants are not used in receiving operations. This would enable specific optimizations in the MPI library. The `exact_length` assertion tells that "the length [of] messages received by the process are equal to the lengths of the corresponding receive buffers" [108]. This does not mean the actual size of the buffer given to the receive operation contains exactly the space indicated by the arguments for count and datatype. This is already implied by the standard anyway and tools like MUST [42, 94] already check if the programmer made this specific error. Rather the `exact_length` assertion tells an implementation that the size of the send messages match the size of the receive operations buffers. The MPI standard allows the sender to issue a send operation with less than the expected elements by the matching receive operation (e.g., five integers being sent and matched by a receive operation for 20 integers). In this example, only the first 5 locations of the 20 integer long buffer will be filled. The program can then use the `MPI_Status` and `MPI_Get_count` to assess how many elements have actually been received. As the standard writes: "In the case of a message shorter than the receive buffer, MPI is quite strict in that it allows no modification of the other locations. A more lenient statement would allow for some optimizations" [108]. Therefore, the newly introduced assertion will allow for such optimizations.

As far as the MPI standard is concerned, it is up to the programmer to decide whether or not such an assertion to the MPI implementation can be used in a given program. In

this paper, we present a compiler based tool that automatically detects whether one of these assertions can be made safely. Using this tool, an existing code base can benefit from the newly defined assertions, without a costly human analysis of the code base to check if it is safe to specify a given assertion.

Our work is not directly concerned with optimizing MPI applications. Therefore, we do not consider numerous works regarding the optimization of MPI applications (e.g., by overlapping computation and communication [34]) as related. As our approach is more related to other work on the usage of static analysis for MPI programs (like [17, 57, 59, 64]). This work is mostly focused on detecting programming errors made by an application engineer and therefore facilitate a correct usage of MPI. In contrast, our work does not focus on correct MPI usage by the programmer[1], but support developers in the possibility to utilize a more optimized MPI implementation by specifying additional optional assertions.

Apart from that there is lots of work to optimize MPI Applications, for example by overlapping computation and communication (like [34]). But in our work, we do not change the application. Rather we check if a certain condition hold for a given application. An applications for which these assertions hold may then utilize a more optimized implementation.

In the next section, we will explain our tool's static analysis approach for deciding if the newly defined assertions hold. In Section 12.3, we discuss the feasibility and shortcomings of our approach, summarize in Section 12.4 and provide an outlook on possible future work.

12.2. Analysis approach

To analyze whether or not an application can be augmented with an assertion to the MPI library, we developed an LLVM compiler pass to statically analyze the program at compile time. Currently we restrict ourselves to C and C++ applications using the Clang compiler. As the analysis operates on the LLVM intermediate representation, our approach should be transferable to other LLVM input languages. This explicitly includes Fortran, once a stable version of flang is available in later LLVM versions.

Our analysis assumes that the input program is fully standard compliant and deadlock-free. Otherwise, the assumptions made during the analysis might not hold.

[1] We assume the use of MPI in an application to be correct

Section 12.2.1 covers the analysis of the `allow_overtaking` assertion, while Section 12.2.2 covers the `exact_length` assertion.

We skip a detailed description of the analysis for the `no_any_tag` and `no_any_source` assertions. It is quite trivial to check for constant values during compilation. Hence, if the analysis does not detect any `MPI_ANY_SOURCE` and `MPI_ANY_TAG` used in MPI operations, the corresponding assertion can be used.

Our Analysis is interprocedural, but currently limited to the scope of one object file[2]. Therefore, if a function from another object file is called that is neither part of the standard library nor MPI itself, our pass will state that it is not safe to use any of the assertions.

12.2.1. `allow_overtaking` assertion

To detect if the `allow_overtaking` assertion can be specified, our pass analyzes all occasions where a message is sent and checks if it could conflict with other sent messages in regards to these messages not overtaking each other.

Conflicting send operations

A send operation is considered conflict-free, if there is no other send operation (`a`) with the same communicator and (`b`) message tag (`c`) to the same target. Furthermore, it is considered conflict free, if a send operation with matching communicator, tag and target is (`d`) not on the same codepath or (`e`) separated by a synchronization of the processes.

Conditions `a`,`b` and `c` can in many cases be checked statically (e.g., the used message tag is different). In order to get the best results for this analysis, it is important to use an optimization level that includes constant propagation. Hence, the analysis pass we developed should be executed after all other LLVM optimization passes. Our tool will only consider the message tag, target process or the used communicator to be different, if a difference can be statically proven. For this purpose it is not necessary, that the exact value can be computed statically in a particular case. In many cases a data-flow- and dependency-analysis can show that there will be a difference in the value at any time during the program's execution. For example, it is sometimes possible to prove that the message tag of a send operation is different in each loop iteration and this operation

[2]Refer to Section 12.4 for ideas on how to overcome this shortcoming.

can therefore not conflict with itself. This analysis, which is the base for several well known compiler optimizations like loop transformations, is done by the LLVM tool-chain through the `ScalarEvolution` analysis pass.

If a unique message envelope cannot be detected statically, two send operations will be marked as conflicting, although in the actual execution of the program they might still be conflict free. We will discuss an example of this case in Section 12.3.

For condition `d`, our pass will check if there exist any code path (i.e. a path in the control-flow graph) between the two operations.

Synchronization

Finally, our LLVM pass tries to prove that on *all* possible codepaths between two send operations there is a process synchronization (condition `e`). In the case of a `barrier`, i.e. when a sending process exits the barrier, the analysis can assume that the target has successfully posted a matching receive operation; otherwise, the program is erroneous, due to the possibility of a deadlock. Therefore, the analysis can assume due to our requirement of deadlock-freeness that a matching receive has completed on the target process. As the other process also reached the synchronization point, all other open receive operation have been completed up to that point, unless it is implemented as a non-blocking operation, which we discuss in the next paragraph. By assuming the receive operation has completed at the synchronization point, the analysis can safely assure that no overtaking is possible between a message sent before the synchronization and after the synchronization.

The MPI standard mandates that we may not draw any conclusion about the execution state of another processes from other collective operations apart from the completion of the barrier. (see section 5.13 of the MPI standard [66]) However, the use of an `MPI_Allreduce` implies a barrier-like synchronization. This implicit barrier is not implied by the MPI standard in any way, rather it results of the required data flow for the `allreduce` operation. A program cannot have the result of an `allreduce` operation available, unless all processes have entered it.

Of course, a synchronization as explained above only prevents a conflict, if it uses the same communicator as the conflicting messages. Otherwise the analysis cannot draw any conclusion on the receiving processes status.

When considering `Rsend` or `Ssend`, we know by definition that the corresponding receive has started.[3] Therefore, it is not possible that the next message send will overtake and wrongfully match the receive operation currently active.

Non-blocking operations

For `Isend` and `Bsend`, a synchronization only prevents a conflict if the operation completed locally using either `MPI_Wait` or `MPI_Buffer_detach`, respectively[4]. This holds, as our analysis only covers programs that are deadlock-free, according to the standard. If the receiving process has not posted the matching receive when all processes synchronize, a deadlock might occur.

For non-blocking collectives, at the point when the collective completes, as indicated by the use of `MPI_Wait`, the analysis can consider the processes synchronized, implying that the matching receive has to have been posted by that point.

If an application uses `MPI_Irecv`, we additionally have to check the assumption that a receive operation has finished when a synchronization takes place. To this end, we analyze if the scope where a non-blocking receive operation may be active does not cross a synchronization point. This means that we have to check if an `Irecv` operation is issued before a synchronization and if the matching `Wait` is issued after the synchronization. To take these cases into account, our tool also analyzes whether the receive operations are conflict free in the same manner as explained earlier for the send operations (conditions `a-e`).

There is no need for special considerations when applying our tool to a multithreaded program. If two threads send a message, the standard does not mandate that there is a defined order between these messages (see section 3.5 of the MPI standard [66]). Hence, if the assertion allows for messages to overtake each other, this does not have an effect on a correct program, as the order of these two operations was not defined in the first place. The standard only guarantees that messages sent by one thread will be received in that order by the other process. Therefore the considerations taken for the single-threaded program are also sufficient for a multi-threaded program.

[3] The particular differences between `Rsend` and `Ssend` are not important for this consideration. Important is that when either of those operations successfully finishes, we can conclude that the matching receive has started on the target process

[4] Currently we have not implemented coverage for the `Ibsend` operation, but the implementation would in principle be the same with the difference that both an `MPI_Wait` and a following `MPI_Buffer_detach` is required for the operation to be considered locally complete.

12.2.2. `exact_length` assertion

In our opinion it is a good programming practice to distinguish messages with different length by the use of different message tags, in order to avoid programming mistakes that may accidentally match a longer message to a shorter receive buffer. Hence, we group the communication operations by the used message tag for our analysis. All messages with a non-constant message tag will be grouped in the same group. If a receive specifies `MPI_ANY_TAG` it matches to all message tags, meaning that it is considered part of every group.

The `exact_length` assertion can safely be specified, if the length of all sending and all receiving operation is identical within each group. Otherwise the assertion still could hold, but this is not assertable by our approach using only static analysis. We implemented this analysis as a proof of concept in our LLVM pass, knowing that its use is currently limited and that a more refined approach is desirable. Such an approach could leverage synchronization points and code structure to further segregate the message-size groups into fine grained groupings. Messages split by a synchronization point, are not required to have the same size when using the same message tag.

12.3. Evaluation

We have tested our developed Clang pass using 48 different small MPI programs[5] designed for this purpose. As the detection for `no_any_tag` and `no_any_source` is trivial constant-matching extensive tests are not necessary - the compiler with its pre-implemented analyzes does the work here. For the `exact_length` assertion, we validated using our test suite and did not observe any issues. As this analysis is conservative, we might miss opportunities where this assertion might be used, but the correctness of the implementation was shown to be ok. As the analysis of the `allow_overtaking` assertion required more program logic and is more complex, we will focus the evaluation on this aspect.

The test cases were written in order to test the design aspects of our analysis. This means we designed them from scratch as white box tests for our implementation. We also included several different implementations of a mini-app alike 2D-stencil code calculating heat dispersion. This more elaborate example originated from teaching of MPI programming. In order to meaningfully evaluate our implementation with other mini-apps, one has to

[5] Available together with the source code of the analysis tool.

manually do extensive and rigorous checking if the assertions can be specified for the respective code. As this can be error-prone and comes with a huge effort, we stick to this well known teaching example. Nevertheless, this example will illustrate the feasibility and shortcomings of our approach.

For roughly 60 percent of the test cases, it is not safe to specify the `allow_overtaking` assertion. Of these cases, our tool detected all cases where it was not safe to use the assertion.

Listing 12.1: Exemplary test case with the `Ibarrier`, where our pass could not prove, that no overtaking of the messages is possible.

```
switch (rank) {
  case 0:
    MPI_Recv(/*from rank 1*/);
    MPI_Ibarrier(MPI_COMM_WORLD, &bar_req);
    MPI_Wait(&bar_req, MPI_STATUS_IGNORE);
    MPI_Recv(/*from rank 1*/);
    break;
  case 1:
    MPI_Ibarrier(MPI_COMM_WORLD, &bar_req);
    MPI_Send(/*to rank 0*/);
    MPI_Wait(&bar_req, MPI_STATUS_IGNORE);
    MPI_Send(/*to rank 0*/);
    break;
}
```

There were, however, some similar cases involving an overlap of communication and synchronization, where our approach could not detect that there is no possibility for the involved messages to overtake. One example is shown in Listing 12.1. The messages sent by rank 1 can never overtake each other in this case, but our approach cannot show at compile time that the send operation in line 10 can never overtake the message sent in line 12. This is due to the use of blocking send/receive in the time between an asynchronous barrier and its corresponding wait. With logical inference, we can conclude that the receive of rank 0 must have completed, for the barrier to complete. However, our current implementation cannot draw such conclusions. It must, according to our current design, consider the send of rank 1 to be part of both the pre-barrier and post-barrier communication region. And, as it conflicts in the post-barrier region, it must consider a potential overtaking and advice against using the `allow_overtaking` assertion. Further

research may enable to draw such conclusions, but currently we accept the conservative limitation of our current approach.

Another case, where our tool could not detect that the `allow_overtaking` assertion could be specified, occurs when each process communicates with its predecessor and successor in a ring-like fashion. This means, that for this communication scheme rank 0 is the successor of rank $n-1$. In this case, the shortcoming of our static approach cannot prove at compile time, that the predecessor and successor are different. After all, the program could be called with only two MPI ranks. This is not a shortcoming of our implementation, as with only two ranks the predecessor and successor are the same, meaning that it is indeed possible that the message sent to the next rank will overtake the message to the previous rank in this case. Using different message tags for forward and backward communication mitigates this problem.

In the more complex example of a 2D-stencil code, our tool detects that all assertions could be specified, if different message tags are used for the different iterations. If the same message tag is used for all iterations, our tool rightfully concludes that the `allow_overtaking` assertion could indeed not be specified, because the order of messages must match the order of iterations as a message from one iteration should not match to a receive in the next iteration.[6] This is a classic case of a loop driven dependency [41].

Besides the functional aspects, we tested the overhead our analysis contributes to the compilation time with our elaborate mini-app alike example with about 400 lines of code using Clang's `-ftime-report` option. As Clang reports that our analysis pass requires only 0.3% (0.0014 seconds) of the total execution time for all LLVM passes executed on optimization level `-O2`, we consider the overhead of our pass to be negligible when other optimizations are applied.

12.4. Conclusion

MPI 4.0 offers (in its current draft [108]) new assertions enabling aggressive optimizations by the MPI library: the `allow_overtaking` and `no_any_tag` as well as `no_any_source` and `exact_length` assertions. However, in order to specify those users must analyze their code to ascertain if the requirements for those are met. We present an LLVM compiler-pass that can statically check program properties and provide feedback to

[6] Splitting the iteration with a synchronization is also possible but only introduces unnecessary synchronization overhead in this case.

the user for those assertions. While our analysis can provide perfect feedback for `no_any_tag` and `no_any_source`, feedback for `allow_overtaking` and `exact_length` depends on the properties of the program. With `allow_overtaking` and `exact_length` our analysis does not detect all cases where it is possible, but when it infers that the use of those assertions is possible, this is correct. Especially the analysis for overtaking messages is complex and relies on detailed understanding of the semantics of MPI send, receive and collective operations. Evaluating the resulting tool, available at https://github.com/tudasc/mach, on a custom set of test-codes showed no observable deficiencies. In summary, we provided a first step to lower the burden of manual analysis and therefore facilitate the usage of those assertions.

Using a Clang optimization pass, we will extend the analysis pass to automatically insert the assertions into the code. Therefore, our tool can be viewed as a compiler optimization that in some cases can optimize the use of the MPI library.

In the future we plan to address the main drawback of our current implementation, i.e. its limitation to a single object file. Besides the usage of a whole program representation, an annotation whether an assertion holds for a function will facilitate such an analysis with multiple object files. Additionally, one can also limit the scope of the assertion to one object file, by duplicating the used MPI communicator[7] and only specify the assertions to the duplicated communicator, while replacing all uses of the original communicator with its duplicate. In the future, we also want to further refine our analysis (e.g., to also incorporate the usage of `MPI_Probe`).

An empirical analysis of the performance gained in real applications is planned, once MPI implementations have incorporated the usage of the proposed assertions and exploited the expected performance benefits. When we know about the actual performance gain, it is possible to determine if there is a trade-off in performance, if one changes the application in order to allow the specification of an assertion (e.g., use different message tags), or if such a change would lead to less overall performance. With these data we plan to extend our tool to give the programmer guidance on how to change an application so that the assertions hold, in order to facilitate the expected performance gains.

Acknowlegements: This work was supported by the Hessian Ministry for Higher Education, Research and the Arts through the Hessian Competence Center for High-Performance Computing. We want to thank the anonymous reviewer for their suggestion to address the current limitation of our tool by duplicating the MPI communicator.

[7] Using `MPI_Comm_dup_with_info`

13. Automatic Partitioning of MPI Operations in MPI+OpenMP Applications

Bibliographic Information

The contents of this chapter has previously been published in:

Tim Jammer and Christian Bischof. "Automatic Partitioning of MPI Operations in MPI+OpenMP Applications". In: *High Performance Computing. ISC High Performance 2021. Lecture Notes in Computer Science*. virtual: Springer International Publishing, 2021, pp. 191–198. DOI: 10.1007/978-3-030-90539-2_12

The contribution of the author of this thesis is summarized as follows:

As the leading Author, Tim Jammer contributed the Idea, Implementation and evaluation. Christian Bischof provided invaluable feedback on the manuscript's text and the presentation of the results. Both authors agree with the use of the publication for this thesis.

Abstract

The upcoming MPI standard includes a new chapter about partitioned point to point communication operations. These partitioned operations allow multiple actors of one MPI process (e.g. multiple threads) to contribute data to one communication operation. These operations are designed to mitigate current problems in multithreaded MPI programs,

with some work suggesting a substantial performance benefit (up to 26%) when using these operations compared to their existing non-blocking counterparts.

In this work, we explore the possibility for the compiler to automatically partition sending operations across multiple OpenMP threads. For this purpose, we developed an LLVM compiler pass that partitions MPI sending operations across the different iterations of OpenMP for loops. We demonstrate the feasibility of this approach by applying it to 2D stencil codes, observing very little overhead while the correctness of the codes is sustained. Therefore, this approach facilitates the usage of these new additions to the MPI standard for existing codes.

Our code is available on github: `https://github.com/tudasc/CommPart`

13.1. Introduction

The new version 4.0 of the MPI standard includes many changes. In particular, "the largest changes are the addition of persistent collectives, application info assertions, and improvements to the definitions of error handling" [136]. Additionally, the new version includes a whole new chapter on Partitioned Point-to-Point Communication (chapter 4). Partitioned operations allow "for multiple contributions of data to be made, potentially, from multiple actors (e.g., threads or tasks) in an MPI process to a single communication operation" [136]. Therefore, the partitioned operations are specifically designed for efficient multi-threaded usage, an area where the usage of traditional non-blocking operations currently lacks performance [100].

The usage of partitioned operations is quite similar to the usage of persistent operations, as shown in Listing 13.1: First, a request object has to be initialized by using the `MPI_Psend_Init` or `MPI_Precv_init` functions (line 1). This initializes the request object, but does not start the communication. Starting the communication is done with the `MPI_Start` routine (line 2). In contrast to a persistent operation, the communication does not start once the `MPI_Start` operation is used. Rather, each partition separately has to be marked ready with `MPI_Pready` at which point the message transfer might start (line 5).[1] Similar to persistent and non-blocking operations, a partitioned operation has to be completed by using `MPI_Wait` or equivalent (line 6).

[1] meaning that the modification of *this partition* the sending operation is forbidden until the sending operation has completed locally.

Listing 13.1: Illustration of the usage of partitioned communication

```
MPI_Psend_init(msg, partitions, count, type, /*...*/ &req);
MPI_Start(&req);
for(i = 0; i < partitions; ++i){ // may be executed in parallel
  /* compute and fill partition #i, then mark ready: */
  MPI_Pready(i, &req); }
MPI_Wait(&req);
```

Partitioned operations were also known as “finepoints” before their adoption to the MPI standard [104], and their use resulted in up to 26% improvement in communication time in the case of a 2D-stencil mini-App [61, 104].

Currently, adoption of the already existing persistent operations, on which the partitioned operations are based, is quite low [105]. This strongly suggests that the incorporation of the even more complex partitioned operations into existing codebases will present a challenge. We see a sort of “chicken and egg problem”: If no one uses partitioned operations, there is no incentive for implementors to highly optimize these operations in order to harness the full potential of the performance benefits. On the other hand, if there is no real performance gain, programmers will have no incentive to use the partitioned operations.

Therefore, we explore the capabilities of the compiler to help with the adoption and usage of partitioned operations.

Related work on the topic of using the compiler to optimize MPI usage covers the automatic replacement of blocking operations with their non-blocking counterparts [27, 34, 74, 78, 126], as well as the compiler-based detection of the application info assertions [116], which are also included, in the new version of the standard. Some of this work [78] also explored the compiler’s capability to replace these operations with persistent operations instead.

In this paper, we extend the existing approaches of replacing blocking operations with their non-blocking counterparts by replacing them with partitioned operations instead. Our replacement tries to partition those operations among different OpenMP threads in OpenMP for loop constructs.

First, we discuss our approach and the resulting tool in Section 13.2. We then demonstrate the feasibility in Section 13.3 and conclude in Section 13.4.

Listing 13.2: Example code before the partitioning.

```
// allocate/init msg buffer
for(int n = 0; n < num_iter; n++){
  # pragma omp parallel for
   for(i = 0; i < size; ++i){
     /* compute and fill buffer[i] */
   }
 MPI_Send(buf,/*...*/);
}
```

13.2. Analysis approach

We implemented an LLVM compiler pass that replaces blocking `MPI_Send` operations with a partitioned equivalent if possible. The current version of the pass works on a function level, meaning that it currently is only able to partition sending operations issued in the same function where the parallel region is.

The compiler pass first tries to extend the non-blocking communication window similar to the approaches presented in [78, 126]. If our pass finds that the message buffer is modified within an OpenMP for construct, it tries to partition the operation. This happens in 4 steps: First, we use the LLVM scalar evolution analysis to calculate the memory access pattern if possible. We do this by analyzing the memory addresses of all store operations inside the OpenMP loop and check if their evolution throughout the different loop iterations is computable and follows a linear pattern. Note that it is sufficient if a linear pattern (based on the loop index) can be computed symbolically, as not all values (e.g. the loop boundaries), might be known at compile time. If no linear pattern (linear in the loop index) can be derived, a partitioning does not seem useful, as the semantics of the MPI partitioned operations require that each partition has the same size, meaning that only a linear partitioning of the message is possible.

If a (symbolic) memory access pattern was computed, our pass transforms the blocking operation into a partitioned one. An example is given in Listings 13.2 and 13.3, which illustrates the transformation done by our pass.

[2]Note that this only illustrates the transformation, as the transformation happen on the LLVM IR, no source code is being output

Listing 13.3: Example code after the partitioning.[2]

```
MPI_Request req;
// allocate/init msg buffer
// Instructions to calculate the Loop Access Pattern
Init_Partitioned_operation(/*Send Parameters, Loop access pattern*/
                           ,&req);
for(int n = 0; n < num_iter; n++){
  MPI_Start(&req);
  # pragma omp parallel for
  for(i = 0; i < size; ++i){
    /* compute and fill buffer[i] */
    ready_partitions_after_loop_iter(i,&req);}
  MPI_Wait(&req);
}
MPI_Request_free(&req);
```

First, our transformation inserts the allocation of a new `MPI_Request`-alike object (line 1) alongside with the corresponding call to `MPI_Request_free` (line 13) into the function that contains the parallel region. With the information on the memory access pattern and the loop bounds, the pass will insert the initialization of the partitioned operation (line 4). Instead of directly inserting a call to `MPI_Psend_init`, we call a self written library function, that also takes the memory access pattern and loop chunk size as arguments. This function then calculates a viable partition size at runtime and calls `MPI_Psend_init` accordingly. This means that the operations to actually compute the symbolic memory access pattern at runtime will also be inserted into the code.

Once the operation is partitioned, a call to `MPI_Start` is inserted before the parallel region (line 6) and the original send operation is replaced with the corresponding `MPI_Wait` call (line 11). Lastly, we need to insert calls to `MPI_Pready` (line 10). These calls are also wrapped by a library function that is called after every chunk of the loop and determines the partitions that can be marked ready based on the current loop chunk. Listing 13.3 does not illustrate the fact that the library function is called after every chunk of loop iterations instead of after every iteration of the loop, with the chunksize given by OpenMP.

We note that overlapping access patterns are possible. Therefore, our `MPI_Request` alike object also keeps track of the number of threads that have finished working on a particular

partition of the message, with the call to `MPI_Pready` only being issued once all threads finished working on a particular message partition. The amount of overlap is computed at the initialization of the partitioned operation, as the memory access pattern is already known at this time, keeping the overhead for each loop chunk as low as possible.

One limitation of the current implementation is that the operation has to be partitioned alongside the boundaries of the MPI datatype used. For example, an operation that sends n datamembers of type `MPI_DOUBLE` can be partitioned, while an operation sending *one* contiguous type[3] containing n doubles cannot be partitioned. In the future, it might be desirable to also allow for the partition of such types.

Currently, the chunksize used to calculate the partition-sizes is the chunksize used by the OpenMP for loop. This means that if the user did not specify a chunksize in the OpenMP for pragma directive, 1 is used as the default (for the used LLVM implementation), which likely will cause a huge overhead if `MPI_Pready` has to be called after every iteration. We plan to investigate this tradeoff in the future and adjust our tool to also take it into account.

At this point, we only tested the implementation with a static loop schedule, but it can easily be extended to also cover dynamic ones.

For simplicity, we referred to the MPI functions as named in the standard throughout this section, but we implemented a wrapper that is used for correctness checking (e.g. `MPIX_Start` instead of `MPI_Start`), meaning that the pass will insert a call to those functions.

13.3. Demonstration of feasibility

To demonstrate the feasibility of our approach we consider performance and correctness. We use a 2-dimensional heat dispersion code, based on a 5 point stencil and with block-wise domain decomposition, as an example to demonstrate our approach, as this sort of code was shown to benefit from the usage of partitioned operations [104].

In order to explore if the partitioning done by our tool is correct, we developed a correctness checking wrapper utilizing valgrinds Memcheck tool [128]. Our wrapper functions wrap around the existent persistent communication operations, providing the API defined by the MPI standard for partitioned operations.

[3] For example a type created by `MPI_Type_contiguous`.

The communication itself will only start once the wait call is issued[4], and all partitions are marked ready. This means that our correctness checking wrapper behaves like a blocking communication operation and no performance benefit can be expected. When a `Pready` call is issued, no data transfer takes place. Instead, our wrapper marks the corresponding part of the message buffer as inaccessible using `VALGRIND_MAKE_MEM_NOACCESS`. This ensures that every access to this particular part of the message buffer will trigger an error in valgrinds analysis, as modification of the message part that is marked as ready is forbidden by the MPI standard. This might, however, lead to some false positive error reports from valgrind, as reading the data is allowed[5]. Therefore, we developed a script that filters out all false positives from valgrinds output[6]. We do this filtering by specifying a valgrind block handle for each send and receive operation, which gives us the ability to distinguish the send and receive operations to filter out the false positives. This also allows us to distinguish the errors introduced by the message partitioning from other errors that may be present in an application.

We tested our tool with a handful of different versions of our examplary 2D stencil code, which are semantically the same but use a different set of MPI operations to achieve the halo exchange. The used stencil code is also included in our github repository. In all tested cases, no error was introduced by our message partitioning tool.

In terms of compile performance, we note that our tool uses a reasonable amount of the compilation time: 4.5% of the total 1.3 seconds for 175 lines of code on optimization Level O2, as reported by clangs `-ftimereport` option.

For the application performance, we first note that our correctness checking wrapper described above will result in some overhead in and of itself. Nonetheless, we see that even this naive implementation results in very low overhead of 0.8% when executing it with 16 processes using 24 threads each on a cluster[7], when not running under valgrinds supervision.

Some part of the overhead measured results from our library functions that determine the partitions to ready after each chunk of the loop and calculate appropriate partition sizes, while another part likely results from our clearly not optimal implementation of the communication. The distinction which part is actually caused by the “administrative” part

[4] This is a valid implementation according to the MPI standard.

[5] For a receive operation, reading and writing is forbidden, though.

[6] False positives in the MPI implementation or the application itself are not filtered.

[7] On the Lichtenberg cluster equipped with Intel Xeon Platinum 9242 CPUs, the execution of unaltered version compiled with clang 11.1 took 614 seconds on average, while the execution of the automatically partitioned version took 619 seconds.

rather than the inefficient wrapper is difficult. In the current version of the partitioning, each request is partitioned independently. In future versions, we aim to further lower the required "administrative" part by considering groups of requests together, avoiding unnecessary recalculation of, for example, the loop memory access pattern. Nevertheless, if an optimized implementation is able to achieve close to the suggested 26% performance benefit [104], being able to harness a net gain of 25% will still present a benefit, especially when considering that our approach avoids the need for a human intervention to make use of the partitioned operations.

13.4. Conclusion

In this work, we demonstrated the compiler's capability to automatically partition MPI operations in some MPI+OpenMP applications by presenting an LLVM compiler pass that is able to partition `MPI_Send` operations. As our approach results in very low overhead, this allows to harness much of the expected performance benefit without the need for a developer to manually adapt the code. We plan to extend our tool to explore when partitioning makes sense and when the necessary overhead becomes too large. We also plan to refine our implementation with the analysis of different OpenMP loop scheduling and the partition of messages with a contiguous datatype.

In the future, the further evaluation of the performance potential of partitioned operations for other communication patterns besides stencil codes is important. Nonetheless, the presented approach may evolve to be a valuable additional optimization step in a domain specific stencil compiler as well as provide incentive for highly optimized implementations of these operations.

Our code is available on github: https://github.com/tudasc/CommPart

Acknowlegements: We especially want to thank Dr. Christian Iwainsky (TU Darmstadt) for fruitful discussion. This work was supported by the Hessian Ministry for Higher Education, Research and the Arts through the Hessian Competence Center for High-Performance Computing. Measurements for this work were conducted on the Lichtenberg high performance computer of the TU Darmstadt. Some of the code analyzing the OpenMP parallel regions originated from CATO [129] (https://github.com/JSquar/cato)

Part V.

Using compiler knowledge to further optimize modern MPI features

14. Compiler-enabled optimization of persistent MPI Operations

Bibliographic Information

The contents of this chapter has previously been published in:

Tim Jammer and Christian Bischof. "Compiler-enabled optimization of persistent MPI Operations". In: *2022 IEEE/ACM International Workshop on Exascale MPI (ExaMPI)*. Dallas, TX: IEEE, Nov. 2022. DOI: 10.1109/exampi56604.2022.00006

The contribution of the author of this thesis is summarized as follows:

As the leading Author, Tim Jammer contributed the Idea, Implementation and evaluation. Christian Bischof provided invaluable feedback on the manuscript's text and the presentation of the results. Both authors agree with the use of the publication for this thesis.

Abstract

MPI is widely used for programming large HPC clusters. MPI also includes persistent operations, which specify recurring communication patterns. The idea is that the usage of those operations can result in a performance benefit compared to the standard non-blocking communication. But in current MPI implementations, this performance benefit is not really observable. We determine the message envelope matching as one of the causes of overhead. Unfortunately, this matching can only hardly be overlapped with computation. In this work, we explore how compiler knowledge can be used to extract

more performance benefit from the usage of persistent operations. We find that the compiler can do some of the required matching work for persistent MPI operations. As persistent MPI requests can be used multiple times, the compiler can, in some cases, prove that message matching is only needed for the first occurrence and can be entirely skipped for subsequent instances.

In this paper, we present the required compiler analysis, as well as an implementation of a communication scheme that skips the message envelope matching and directly transfers the data via RDMA instead. This allows us to substantially reduce the communication overhead that cannot be overlapped with computation. Using the Intel IMB-ASYNC Benchmark, we can see a communication overhead reduction of up to 95 percent for larger message sizes.

14.1. Introduction

The Message-Passing Interface (MPI, [136]) is the de-facto standard for distributed memory computing in high-performance computing (HPC). With MPI, one can overlap communication and computation in order to hide the overhead necessary for communication. Unfortunately, not all communication overhead can be hidden behind computation. In particular, overlapping the work necessary to perform message envelope matching with computations is quite challenging, although proposals have been made to offload this matching to specialized hardware [25, 102]. The requirements for tag matching hardware for different HPC workloads have been studied [106], but hardware-based approaches will always have less flexibility regarding possible future changes in communication patterns. Therefore, message envelope matching is a particularly costly part of MPI communication [100]. In this work, rather than further optimizing tag matching, we want to focus on avoiding it when possible, using compiler knowledge.

The compiler can aid with the efficient usage of MPI. For example, a compiler can automatically extend the computation-communication overlap [34, 74, 121]. In previous work, we showed that the compiler can use static analysis to find out if the assertions introduced in MPI 4.0, hold for a given application [116]. Based on this approach, in this work, we want to explore if the compiler could use static analysis to assist with message matching. In particular, we want to statically find opportunities where we can avoid the usually required matching procedure altogether.

MPI also includes persistent operations that can be used multiple times. The usage of a persistent MPI operation is illustrated in Listing 14.1: First, one needs to initialize

```
1  MPI_Request request;
2  //Initialize the persistent request, this does not start any communication:
3  MPI_Send_init(buf, count, dtype, dest, tag, comm, &request);
4  for (i=1; i<N; i++) {
5   // start the communication, (basically the same as):
6   // MPI_Isend(buf, count, dtype, dest, tag, comm, &request)
7   MPI_Start(&request);
8   do_some_computation()
9   // wait for completion:
10  MPI_Wait(&request, MPI_STATUS_IGNORE);
11 }
12 // free the persistent request
13 MPI_Request_free (&request);
```

Listing 14.1: Usage of a persistent MPI sending operation

a persistent request (line 3), then one can use the request for an arbitrary number of non-blocking communication operations. Communication is initialized with `MPI_Start` (line 7 in Listing 14.1) and can be completed like any non-blocking operation with the usual function calls like `MPI_Wait` or `MPI_Test` (line 10). Finally, one needs to clean up all resources used by the persistent request (line 13 in Listing 14.1). To allow programming flexibility, a message sent by a persistent operation may be received with any receive operation and vice versa. But if a persistent send operation will always match with a persistent receive operation, one may skip the message matching once it has been established which two persistent operations match. This is possible as the message envelope of a persistent operation never changes (as can be seen in Listing 14.1). Therefore, we want to explore the compiler's capability to decide if a persistent operation can only be matched by another persistent operation. In this case, message matching is only really needed for the first instance of communication and can be skipped for all following ones.

In this paper, we make the following contributions:

- Present the compiler analysis necessary to avoid most of the message matching for persistent MPI operations
- Present a rendezvous protocol to send persistent MPI messages without the need for message matching
- Evaluate the possible performance benefit of this approach

This paper is structured as follows. First, we will present the necessary compiler analysis in Section 14.2, before discussing possible communication protocols without the need for extensive message envelope matching in Section 14.3. We evaluate the performance gained by this approach in Section 14.4. We briefly discuss the limitations of our current implementation and how our approach relates to another approach for optimizing persistent MPI operations in Section 14.5, before concluding in Section 14.6.

14.2. Compiler Analysis

In this section, we describe the compiler analysis we implemented as an LLVM compiler pass in order to find out if we can eliminate some of the necessary message matching. We first explain the idea of our analysis and what information is necessary in order to avoid unnecessary matching operations in Section 14.2.1. Subsequently, we explain how the compiler is able to gather the required information at compile time in Section 14.2.2.

14.2.1. Analysis Overview

If we want to skip proper message matching for persistent operations, we have to make sure that a persistent send/receive pair that matches once will also match for every subsequent communication operation of the two involved persistent requests. This way, we only need to do the matching for the first occurrence of the persistent communication and can skip it for the other occurrences of the involved persistent communication operation.

For each persistent sending operation, we need to decide if there is a possible "matching conflict" with any other MPI communication that may take place during the lifetime of the persistent request. A "matching conflict" for a send operation means that there is another outgoing message that possibly can have a matching message envelope. In this case, we know that a matching persistent receive operation on the other side can match both: the persistent operation as well as the other conflicting message sent. Therefore, in this case, we can *not* skip the proper message matching to ensure a correct execution. If there is no possibility of a "matching conflict", we know that *if* the receiver matches with a persistent receive operation (with no wildcards), the persistent receive can only match the message sent by this sending operation, as we know that no other matching messages will be sent. In this case, we have found a possibility to skip some of the message matching (as long as no wildcards are used on the receiver side).

But at this stage, the receiver may still match with a mix of persistent and standard communication operations. Therefore, we also "mirror" this analysis for the receive operation and check if there exist any "matching conflict" on the receiver side. This time the analysis also needs to take wildcards into account, to check for possible "matching conflicts". If a persistent receive operation without a "matching conflict" is found, it means that when a matching message arrives, it can only be matched by the persistent operation in question.

Combining both "directions" of the analysis gives us all the knowledge necessary to state, that if a persistent send/receive pair is matched once, it will always match. In this case, we can skip the matching for the subsequent occurrences of this persistent operations. The combination of both parts will happen at runtime, when the first message is matched. Therefore, the compiler only "annotates" that this operation is eligible for skipping further matching, *if* the matching operation on the other rank is also eligible for it. This means, that the compiler can find out the necessary information by a local analysis only, without the need to match send/receive operations at compile time.

This is illustrated in Listings 14.2 and 14.3. For the receiver side in Listing 14.2, the persistent operation in line 2 does not have any matching conflict, as both the persistent operation in line 3 as well as the blocking operation in line 6 use a different message tag. In this case, it does not matter that the operation in line 6 does use a wildcard for message matching. In this example, *any* incoming message that has a matching envelope with the first operation in line 2 can only be matched by this operation. But the second persistent operation (line 3 in Listing 14.2) does conflict with the blocking operation in line 6 of Listing 14.2, because an incoming message with the envelope required to match the operation in line 3 can also be matched with the operation in line 6. The barrier in line 7 prevents this sort of matching conflict from the programmers perspective, as a message received before the barrier can never match a receive operation after the barrier. But our compiler analysis does *not* take into account any information about the sender side. The sender side in Listing 14.3 by itself does not contain any matching conflict. But we see that the combination of both does *not* have the desired property, "If two persistent operations match once, they will match always", as the send in line 2 of Listing 14.3 will match both receive operations in lines 3 and 6 of Listing 14.2. This means that in this case message matching can *not* be skipped.[1] The first operations in line 2 of each listing do have the desired property, and in this case further message matching can be skipped

[1] In the example, it may be the case, that the wildcard operation matches a message from another rank, which leads either to a deadlock or the first sent message matching to the second receive after the barrier, depending on whether the message is buffered.

```
1   MPI_Request request1, request2;
2   MPI_Recv_init(buf, count, dtype, source, tag, comm, &request1); //has no matching
      conflicts
3   MPI_Recv_init(buf, count, dtype, source, tag+1, comm, &request2); //can conflict with
      MPI_Recv (line 6)
4   MPI_Start(&request1);
5   MPI_Wait(&request1, MPI_STATUS_IGNORE);
6   MPI_Recv(buf, count, dtype, MPI_ANY_SOURCE, tag+1, comm, status);
7   MPI_Barrier(MPI_COMM_WORLD);
8   MPI_Start(&request2);
9   MPI_Wait(&request2, MPI_STATUS_IGNORE);
10  MPI_Request_free (...); // free both requests
```

Listing 14.2: Illustration of matching conflicts on the receiver side

```
1   MPI_Request request1, request2;
2   MPI_Send_init(buf, count, dtype, dest, tag, comm, &request1); //has no matching
      conflicts
3   MPI_Send_init(buf, count, dtype, dest, tag+1, comm, &request2); //no matching
      conflicts
4   MPI_Start(&request1);
5   MPI_Wait(&request1, MPI_STATUS_IGNORE);
6   MPI_Start(&request2);
7   MPI_Wait(&request2, MPI_STATUS_IGNORE);
8   MPI_Barrier(MPI_COMM_WORLD);
9   MPI_Start(&request2);
10  MPI_Wait(&request2, MPI_STATUS_IGNORE);
11  MPI_Request_free (...); // free both requests
```

Listing 14.3: A possible sender side for Listing 14.2, which does provoke a matching conflict on the receiver besides the barrier synchronization, although no local matching conflicts exist

for those operations.

14.2.2. Technical details

For each occurrence of `MPI_Send_Init`, the compiler can often determine the corresponding `MPI_Request_Free` by tracking the usage of the MPI Request pointer. In order to find if a "matching conflict" exists, the compiler will then explore all possible codepaths between `Init` and `Request_Free` in order to find all other MPI operations that may result in a conflict.[2] In case of a persistent send operation, possible conflicting operations are all other sending operations, including the usage of other persistent requests. The compiler then tries to prove that no conflict exists, for all possibly conflicting operations.

Often the compiler can statically prove that two message envelopes are different i.e. use a different tag, target-rank or communicator. We note that the compiler does not need to know the complete message envelope at compile time in order to prove two envelopes different. For example, the compiler can prove that message tag `x` and `x+1` are different, as long as `x` does not change between these two communication operations. In this case, the value of `x` does not need to be known at compile time. We use existing features of the LLVM tool-chain, such as Scalar evolution analysis, to check if two values can be considered different. As wildcards are predefined constants, the compiler can also check at compile time if any wildcards are used. This checking for a potential matching conflict works very similar to the approach presented in [116] to detect if messages can overtake each other.

If the compiler has proven that no message conflicts exist, it will replace all MPI calls that use the given request (such as `MPI_Send_Init`, `MPI_Start`, ...) with equivalent functions that we implemented (e.g. a call to `MPI_Start` will be replaced with a call to `MPIOPT_Start`). These functions employ the same API as the original ones. The difference is that at the first message, our modified functions will check if the matching counterpart is also a modified request. If it is matched with a modified counterpart, our functions will skip the matching for subsequent operations, otherwise they will call the standard MPI communication. We will discuss the details of our implementation in Section 14.3.

[2] If the compiler is unable to find the corresponding free, it uses the end of the execution e.g., `MPI_Finalize` as the stopping point for the conflict analysis.

14.3. Communication Protocol

In this section, we explain different possible communication schemes that can be utilized when no tag-matching is required. Our focus is on communication schemes that allow for as much overlap of communication with computation as possible. Therefore, we first introduce the existing methods for progressing asynchronous MPI operations.

14.3.1. Asynchronous MPI progress

In order to handle the communication, the MPI Library needs to use the CPU for various actions, such as responding to control messages, posting network operations or performing message envelope matching. These operations are commonly referred to as "progressing the communication" or simply "progress" [98].

When overlapping communication with computation, the CPU usually is busy with performing the calculation and therefore not calling into the MPI Library. This means that it is necessary to achieve communication progress without the control flow going to the MPI library.

Approaches to tackle this problem can, in general, be categorized as either thread based (such as [68]), process based (such as [90]) or interrupt-based (such as [26]). Another approach is to modify the application code, to regularly pass the control flow to the MPI Library e.g. by calling `MPI_Test`, so that communication can progress. Besides the overhead added by calls to the MPI Library this technique has the major drawback, that the application code needs to be adjusted. As [144] pointed out, the efficiency of such an approach depends on the interval where MPI is called, as too many calls into MPI leads to unnecessary overhead, while not enough calls leads to insufficient communication progress, so that the communication hasn't finished in time. This means that it is very cumbersome to use such an approach efficiently, as one has to manually select an appropriate amount of calls into the MPI library for each specific application manually.

Interrupt-based approaches suffer from a lack of scalability and a high overhead associated with context switches and are therefore not common today [137].

Process and thread based approaches usually suffer from the necessity to reserve some CPU resources, for example for a dedicated progress thread, as these resources are no longer available to the applications threads [33]. Additionally, similar to the rate of calls to the MPI library, the number of additional CPU resources depends on the particular communication scheme and therefore also needs to be fine-tuned by the user [137]. This

is especially problematic when the application has different phases with different resource requirements. [137] tries to mitigate the need for specifying the amount of spare resources in advance, by using work stealing, where one process of a node can also progress the communication of all other processes.

Nonetheless, all these approaches need CPU resources in order to perform message matching, meaning that these CPU resources cannot be used by the application. But in our case, the compiler has determined that this message matching is not necessary and could be avoided entirely. Clearly, some progress is still needed in our case, as one needs to respond to control messages for a rendezvous protocol. We do think however, that this progress costs considerably less CPU resources, as less work needs to be done. We will discuss this in the next sections.

14.3.2. UCX Active Messages

MPI implementations such as Open MPI or MPICH often build on top of low level layers such as UCX (Unified Communication X)[3]. As UCX is a common building block for many MPI Implementations, it is natural to consider active messages, when no tag matching is required. One advantage of these active messages is lower overhead, due to the absence of tag matching [113].

Active messages are "matched" with an ID. When an active message arrives, UCX calls a user-provided callback function, based on the ID of the arriving message. The user needs to register a callback for each ID that it wants to receive. Therefore, when the first message of a persistent MPI operation is exchanged (and matched), the sender and receiver can also perform a handshake and negotiate a message ID to use for all later communication, as no tag matching is needed for the further communication operations for this persistent request. The receiver in turn registers the corresponding callback, so that from this point on, UCX active messages are used instead of the tag-based ones.

Active messages can be sent either eagerly, or using a rendezvous protocol. Usually an eager protocol is used for smaller message sizes, whereas larger messages are sent with a rendezvous. The rendezvous protocol for UCX active message is shown in Figure 14.1.

One thing that we note for this rendezvous protocol is that the receiver needs to progress communication in the case that the rendezvous has not started when the receiver wants to begin receiving the data. One way to mitigate this problem is to call `MPI_Test` during

[3] https://openucx.org/

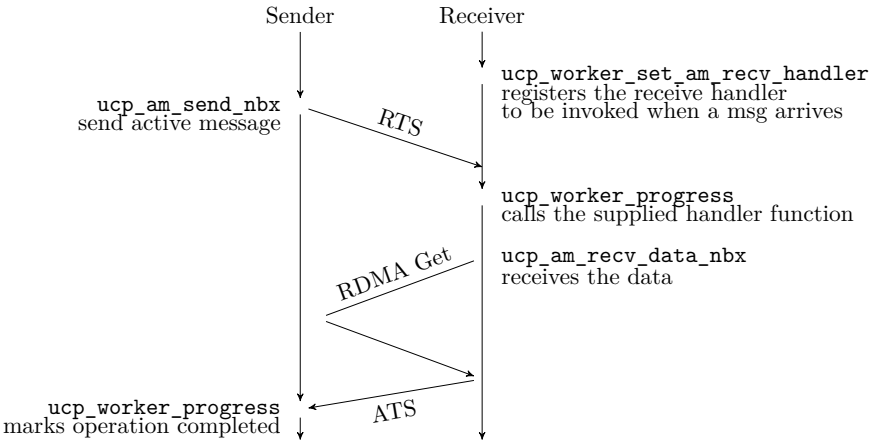


Figure 14.1.: Illustration of the rendezvous protocol used for UCX active messages. Based on [113]

the computation and start the data transfer once the message has arrived. On the other hand, inserting regular calls into the computation can add additional overhead. As we want to avoid such overhead, we propose to use a two-sided rendezvous protocol instead, as described in the next subsection.

We note, that this is not an "artificial" problem, as the sender may first need to calculate the necessary data before communicating it, meaning that the situation where the sender enters the communication later than the receiver will happen frequently in real-world applications.

14.3.3. Two-sided Rendezvous Protocol

In contrast to a UCX active message, the sender does know, where the receiver needs the data as the message buffer cannot change for a persistent operation. Therefore, it is also possible for the sender to initiate the RDMA operation for the data transfer.

In order to synchronize the communication, we propose a lightweight two-sided rendezvous protocol. We illustrated this protocol in Figures 14.2, 14.3 and 14.4, each with a different possible order of processes.

At the first execution of a persistent operation, in addition to the original MPI matching, both communication partners also perform a handshake, where all the relevant information

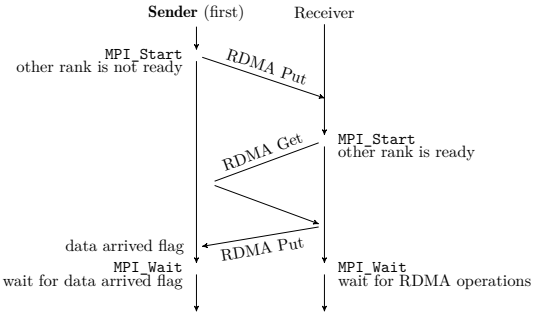


Figure 14.2.: Illustration of the two-sided rendezvous, in case sender arrives first

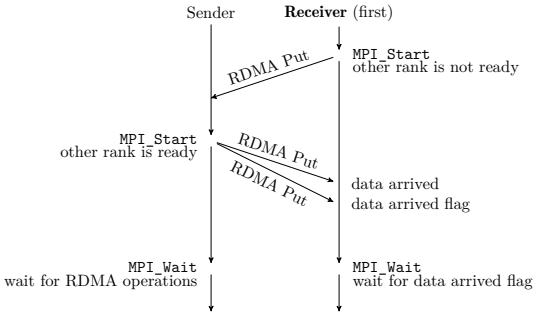


Figure 14.3.: Illustration of the two-sided rendezvous, in case receiver arrives first

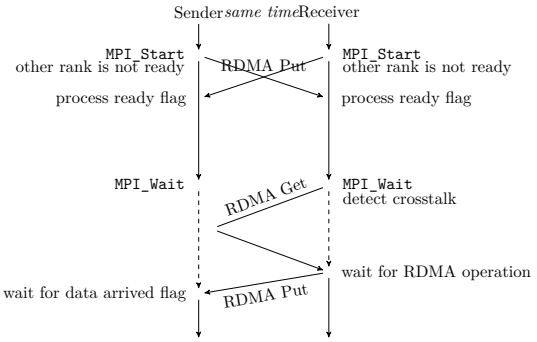


Figure 14.4.: Illustration of the two-sided rendezvous, in case crosstalk occurs

for RDMA transfer is exchanged. In addition to the data buffer, both partners also expose a flag for synchronization of RDMA operations.

After this handshake, the following communication operations are very lightweight:

When one communication partner arrives at `MPI_Start`, it basically only needs to start one RDMA operation. Either one writes to the flag of the other communication partner that one is ready for the data transfer, or, if the local flag indicated that the other partner is ready for the data transfer, one starts the transfer of the data. When the data transfer is initialized, a second RDMA operation, signaling the communication partner the completion of the operation, is also scheduled. This is illustrated in Figures 14.2 and 14.3, depending on which process arrives first. This protocol does not require any participant to

further progress communication, if the RDMA data transfer can be completely offloaded to the network hardware. When the corresponding `MPI_Wait` call is encountered, one needs to wait until the RDMA Operation for the data transfer has completed, or the other communication partner has signaled completion via the synchronization flag. We call this a two-sided rendezvous protocol, as both communication partners can initiate the data transfer with RDMA. This is a key difference to other rendezvous protocols used for communication, as usually only the receiver will initiate the data transfer after it has been told where the data is located on the sender. In our case with persistent MPI operations, where the compiler has determined that message matching is not necessary, it is possible to use such a protocol, as both communication partners can perform a handshake in advance, exchanging all the necessary information. We see that such a protocol lacks flexibility, compared to a one-sided rendezvous. For example, it is not possible to receive data from different communication partners without performing a distinct handshake with each one in advance and also the same message buffers needs to be used for each communication operation. Therefore, such a two-sided rendezvous protocol is not widespread, but in our case this lack of flexibility is no impediment.

There are different variations of "one-sided" rendezvous protocols outlined in [44]. They all have in common that the processes agree beforehand which one initiates the data transfer, by selecting the protocol to use. We can utilize a "two-sided" protocol, as we do not need to agree beforehand which process initiated the RDMA data transfer. One reason why this kind of rendezvous may not be applicable to other MPI operations so easily is the possibility of crosstalk, when both processes arrive roughly at the same time, both expecting the other to initiate the data transfer. This case is shown in Figure 14.4. For non-persistent MPI Operations it may be hard to tell if an incoming control message is the result of crosstalk, or if it is originating from the initiation of a different operation. This is one of the disadvantages encountered by [39], where a "two-sided" protocol, where both the sender and receiver can initiate the data transfer, has been proposed previously called a speculative rendezvous protocol. In our implementation the receiver will detect crosstalk and initiate the data transfer in this case, as shown in Figure 14.4. The detection of crosstalk and distinguishing it from the initiation of another operation is easily possible in our case, as each pair of operations is matched in advance and has a dedicated flag for synchronization. In our implementation, we use this flag to count the number of operations performed on each request, to make sure that both participants try to execute the same operation. This means that the approach presented in [39], does need additional control messages to ensure that matching is done correctly even if miss-predictions in the speculative protocol occur. This approach by [39] does additionally create the problem, that miss-predictions in the speculative protocol will

slow down matching for all other messages, as they will cause “unexpected matchings”, which have been shown as particularly harmful for good matching performance [72].

When crosstalk happens often, the advantage of our protocol, that you do not need to call `MPI_Test` to achieve progress, is lost. A general downside of all rendezvous protocols is that they are not ideal when the sender and receiver arrive at the same time [44]. [44] therefore proposes to use a protocol where one of the communication partners will wait for the control message to arrive in those cases. We will discuss how this affects the overall performance in Section 14.4.

We do note an additional benefit over other proposed rendezvous protocols is the smaller size of the control messages. Even less than one byte will suffice for our purpose, as the control messages in our case no *not* need to contain the information necessary for the RDMA data transfer, such as the message size and buffer location, as this information already has been exchanged during the initial handshake. In our implementation, these control messages are also realized with RDMA operations. TAs an additional benefit this means, that the incoming control messages do not take any resources (e.g. buffer space) from the receiving end until they are handled.

Implementation Details of the Two-sided Rendezvous Protocol

As described in Section 14.2, we defined our functions as replacement for the original functions for persistent operations. This means that our implementation adheres to almost[4] all requirements of the MPI standard, as long as *if* two of these persistent operations matches once, they match always. Matching these operations with other MPI operations as well as the original implementations of the persistent operations is still allowed, in this case the existing implementation will be used as a fallback option.

To make the replacement as seamless as possible, we added a proof-of-concept implementation of those functions into our build of Open MPI. This way, the compiler can replace the functions without the need to adjust linker settings, but one can also still replace calls by hand if desired. In order to make this extension as modular as possible, we also introduced functions for init and finalize, that must be called after the normal `MPI_Init` or before `MPI_Finalize` respectively. But this could easily be included in the original procedure for initialization and finalization.

[4] see Section 14.5 for a discussion on where our proof of concept implementation currently does not strictly comply to the standard.

At the first execution of a modified persistence operation, our implementation initiates a handshake to find out if the other process also matches with a modified persistent operation. The handshake message is achieved by using a different communicator[5] but otherwise the same message envelope, so that it matches the correct operation on the receiver side. Therefore, our current implementation only works with the `MPI_COMM_WORLD` communicator, but can be adapted to support other communicators as well. As there cannot be any matching conflict on the receiver, it cannot happen that this handshake is accidentally matched by the wrong persistent operation. The receiver then either

1. responds to the handshake, or
2. receives the payload without responding to the handshake, indicating that it has matched with a standard receive implementation

If no successful handshake at the first usage of the persistent operation was performed, we use the existing implementation as a fallback option for all further operations on this persistent request.

If the handshake was successful though, we know that we can skip matching for all subsequent usages of the given operation. Therefore, the handshake will exchange the necessary information (e.g. location of the send/receive buffer on the target rank), so that subsequent usages of the operation can result in a direct RDMA data transfer without the need for prior message matching. For subsequent operations, we use our proposed rendezvous protocol, as explained in Section 14.3.3. We note that this handshake can introduce a "hiccup" at the first usage of the operation. Performing such a handshake will disturb the normal flow of communication, as it introduces additional communication overhead. But as the design goal of a persistent operation is that it is used multiple times, we think that this initial overhead is reasonable and justified.

Once the handshake was established, all subsequent usages of the operation will skip the message matching. This allows us to keep the overhead to a minimum, minimizing the time spend in the MPI layer and directly calling into UCX. When a send operation is started, either

1. the information that the data is ready is signaled, or
2. if the receiver has already signaled that it is ready: the RDMA data transfer is initialized.

[5] only one duplication of the used communicator is necessary for all of the handshakes

The receive operation behaves analogous. Therefore, our implementation only checks a local flag and then immediately calls UCX to initiate an RDMA data transfer. Either the actual data transfer is started, or one directly writes to the flag of the other process indicating that this other process should start the data transfer.[6] This keeps the time spent in the MPI layer close to none. This is a significant improvement considering that, for example, MPICH spends up to 92 percent of the latency of an `MPI_Isend` call inside of MPICH and only 8 percent in the lower UCX level [109].

14.4. Evaluation

In order to evaluate the performance gained by our approach, we use the IMB-ASYNC benchmark [119, 144], an extension of Intel's MPI benchmarks[7] specifically designed to measure the computation-communication overlap. As all persistent operations are non-blocking, we see this benchmark as suitable for our evaluation. In our opinion, the overhead that could *not* be overlapped by computation is the most important to reduce, as the other part of communication overhead can be hidden behind the computation. We extended the original IMB-ASYNC benchmark to also include a test case for persistent operations. Basically, we duplicated the existing case for non-blocking point to point communication and replaced the communication with a persistent one[8]. Our extension of Intel's IMB-ASYNC benchmark is based on version 0.0.5[9]. The basic idea of the IMB-ASYNC benchmark is to measure the computation time as well as the total time, to derive the communication overhead as the difference of these two.

We ran our experiment on the Lichtenberg cluster at TU Darmstadt. The nodes are running Red Hat Enterprise Linux 8.5 and are equipped with 2 Intel Xeon Platinum 9242 processors. The nodes are connected with InfiniBand HDR100 in a fat tree topology. The cluster uses the Slurm scheduler version 20.02.5. We used clang 11.1 as the compiler.

The capabilities of our compiler analysis are demonstrated by successfully using it on the IMB-ASYNC benchmark code. For the compilation of the IMB-ASYNC benchmark, we did not notice a large impact on compilation time when using our analysis. In the case

[6] The flag has been agreed upon at the handshake as well.

[7] https://github.com/intel/mpi-benchmarks

[8] The extended IMB-ASYNC benchmark is also included in our github repository: https://github.com/tudasc/MPI-CompMatch.

[9] https://github.com/a-v-medvedev/mpi-benchmarks/tree/b3ec89acf23c7b804b860294564d495ba9d2d4fc

of the IMB-ASYNC benchmark code, our analysis added less than 0.2% overhead to the ≈ 6 seconds it took us to compile the benchmark.

For our experiments, we used the UCX component of Open MPI (4.1.1) with OpenUCX version 1.12.0. We used four different implementations of the communication, all based on the same Open MPI version, as described in Section 14.3:

Normal: The unaltered implementation as the baseline,

Eager: UCX active messages (without tag matching) with an eager protocol (see Section 14.3.2),

Rendezvous 1: UCX active messages (without tag matching) with a rendezvous protocol (see Section 14.3.2),

Rendezvous 2: The proposed two-sided rendezvous protocol described in Section 14.3.3.

In all cases, we do *not* include extra measures to progress the communication, such as an additional CPU set aside for a progress thread or regular calls to `MPI_Test` (also refer to Section 14.3.1).

We repeat each individual measurement 25 times. Inside each measurement, we measure at least 64 communication operations. To measure the cost of network communication, we split the involved processes among different compute nodes. In all measurements, we use a computation-workload that in theory is sufficient to completely hide all communication overhead. In order to measure a real world scenario, we do not take special measures for exact placement of the processes in the network topology and used slurm's assignment. We note, that the latency of the network should be hidden behind the computation by all of the approaches anyway, therefore sampling a random distribution of process placements is sufficient to judge the general applicability of our proposed approach. There are various ways, how system noise may affect the time necessary to perform the *computation*, such as the operating system interrupting processes for some reason, or difference in CPU speed, for example, due to thermal constraints. In order to filter out this system noise, we only take into account the overhead values for the processes with the slowest *computation* time. For any process, where the *computation* time is below 90% of the *computation* time of the slowest process, we do not consider it as overhead, as the "overhead" measured by the benchmark does also include the time spend waiting for the other slower processes to finish their computation. This case is illustrated in Figure 14.5, where one can see that the time spend inside MPI does not only include the message transfer overhead but also the time necessary to wait for the other process. In these cases, we observed the overhead

measured on all of the faster processes to be basically the same for all approaches, as it is dominated by the waiting time. For the slower rank, the messages are sent in time and therefore can be overlapped with computation, meaning that only the relevant overhead is measured.

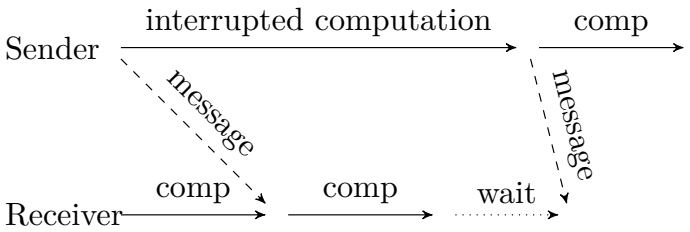


Figure 14.5.: Illustration of the case that computation takes longer on one process.

Figure 14.6 shows the communication overhead for varying message buffer sizes after a warm-up period of 8 communication operations for the communication of two processes. Each measurement includes 64 communication operations. The Y axis depicts the communication overhead that can ***not*** be overlapped by computation. The X axis shows different message buffer sizes. In order to show the distribution of measurements, we show a violin plot for each set of measurements and also depicted all measured overhead values as individual points. As one can see in the violin plots, we have not included all outliers for a better overview. For each measurement, no more than two data points are outside of the plotted area.

Forcing UCX to use an eager protocol does lead to a deadlock due to the lack of resources in our implementation, which is the reason why no data is plotted for the larger message sizes. We do not see a significant improvement, when using eager UCX active messages instead of tag-matching ones for smaller message sizes. We think that this is attributable to the fact, that copying the data from the network buffer to the destination is more expensive than tag matching in this case.

For the usage of the rendezvous protocol of UCX active message (depicted as Rendezvous 1 in Figure 14.6), we see that it does not bring a performance benefit. The reason is that in our implementation, the receiver does not initiate the data-transfer until `MPI_Wait` is called for most communication operations, as the control-flow does not return to MPI during the calculation. There are strategies to circumnavigate that, for example a thread that will be scheduled when a message arrives, even if there isn't a spare CPU core around for it. As we see, the normal implementation does successfully employ those strategies, in

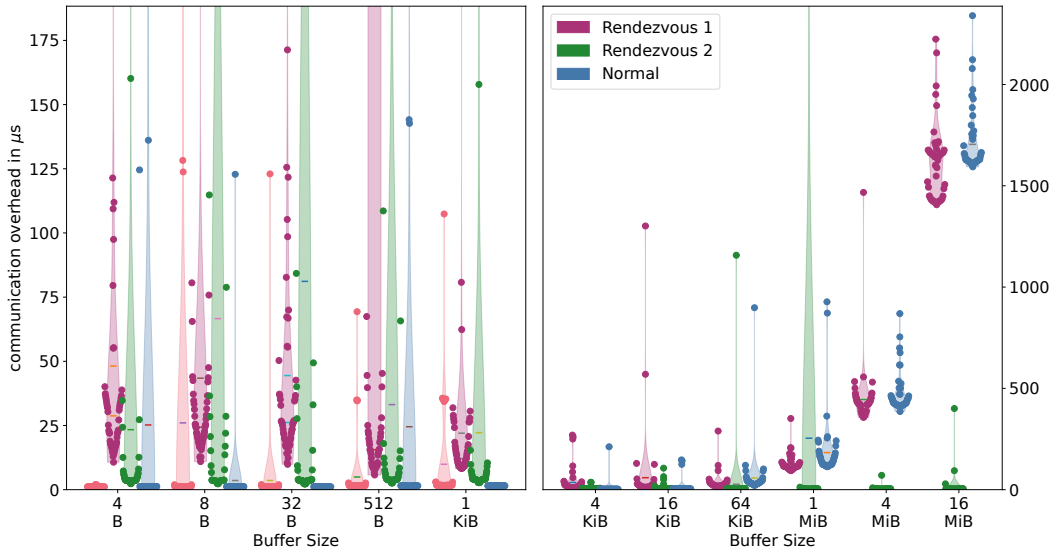


Figure 14.6.: Communication overhead for varying message buffer sizes for two processes. The Y axis depicts the communication overhead that can ***not*** be overlapped by computation, while the X axis shows different message buffer sizes. Normal means the existing implementation. Eager and Rendezvous 1 show the usage of UCX active messages with the eager and rendezvous protocol respectively, as explained in Section 14.3.2. Rendezvous 2 depicts our two-sided rendezvous protocol described in Section 14.3.3.

order to achieve a good performance. But the normal implementation falls short, when comparing it to our proposed two-sided rendezvous protocol (explained in Section 14.3.3 and depicted as Rendezvous 2 in Figure 14.6). For the median communication overhead, we see a reduction of communication overhead of 95% for message sizes of at least 1 MiB. This is attributable to the fact that the time spent in the MPI layer is kept to a minimum and no tag matching takes place. In the default implementation, the actual data transfer can *not* start until the message matching has completed and the implementation therefore knows where the correct receive buffer is located, a downside we can also circumvent by eliminating the need for message matching.

When measuring the overhead incurred during the first 8 communication operations, we see that especially for smaller buffer-sizes, our implementation performs worse with a higher variability, due to the ”hiccup” introduced by the handshake in our implementation,

as described in Section 14.3. The normal implementation does also perform worse during this initial warmup period, although not to the of our proposed communication protocol. But this effect is not present once this initial warm-up phase is over. As the idea behind persistent communications is that they can be used multiple times, we do think that having this additional overhead during the warm-up period is justifiable. But this of course depends on the number of communication operations performed afterwards to amortize this initial overhead for the handshake. In our test, the cost of this initial hiccup amortizes after 40 communication operations[10]. But when one factors in, that the normal implementation also performs better after a warm-up phase, our implementation outperforms the original one after as little as two to three communication operations.

When repeating the experiment for two processes with the processes confined to one compute node, instead of spread among different ones, we observed similar results. The only difference we observe are a couple of more outliers for the standard implementation, but the relative difference between the different protocols stays roughly the same.

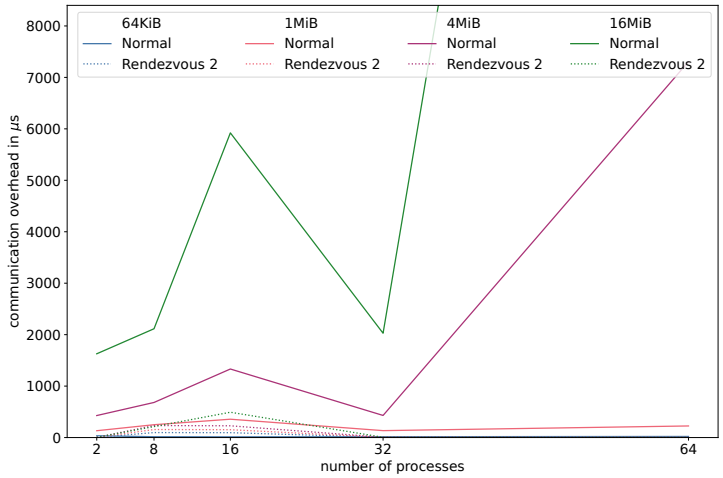


Figure 14.7.: Communication overhead for varying number of processes. The Y axis depicts the communication overhead that can ***not*** be overlapped by computation, while the X axis shows the amount of processes. Normal means the existing implementation. Rendezvous 2 depicts our two-sided rendezvous protocol described in Section 14.3.3.

Figure 14.7 shows the (median) communication overhead for various number of processes

[10] measured with the 16KiB message size

for larger message sizes. We do note that, although the processes only communicate pair-wise in this test, the standard implementation does suffer a performance penalty, when more processes are around. As OpenMPI maintains multiple matching queues, one for each other processes, the cost of message matching grows with more involved processes [100]. We do see that the absence of tag matching does eliminate this performance penalty (refer to the dotted lines), presenting an opportunity for improved scalability.

14.5. Discussion

In this section we discuss technical details where our current implementation falls short of the MPI standard's requirements, as well as the possibility to enhance our approach by combining it with other approaches for optimization of persistent communication.

Depending on how strict one reads the MPI Standard, our current implementation does not fully adhere to all of its requirements. The MPI standard mandates that the "point-to-point persistent initialization calls involve no communication" (section 3.9 of the standard [136]). But in our implementation in order to speed up the handshake, the initialization of a persistent operation will also initialize a non-blocking send/receive operation for the handshake. One can mitigate this "flaw" of our implementation by only performing the handshake when the persistent request is used for the first time. Nevertheless, as we use a non-blocking operation to start the handshake, we think that this still follows the meaning of the standard insofar as the initialization of a persistent operation is a local operation and will return independent of the other processes' state.

Another point, where our current proof-of-concept implementation may not meet the expectation of a quality MPI implementation, is the possible leak of resources. If a handshake message does not get a response, the associated `MPI_Isend` is never properly finished. This problem can be mitigated by using a better handshake implementation which does clean up the resources used for an unsuccessful handshake. For this initial experiment, we have not tested our implementation for multi-threaded usage, but according to MPI, the user is responsible for proper synchronization of the usage of persistent requests anyway.

Regarding the effect of possible crosstalk in our two-sided rendezvous protocol, we observed that after the initial warm-up phase, the processes naturally "ordered" themselves, so that crosstalk does not occur in our experiments anymore. We note that this effect of the processes "ordering" themselves may be influenced by other MPI operations present, especially collective operations. Maybe our approach can benefit from work that models

idle waves and studies the influence collective operations have on them, such as [130]. As the presence of a particular collective can be detected by the compiler, it may be beneficial to include such a model and only use the two-sided rendezvous protocol when crosstalk is unlikely to happen after the warm-up phase.

We note that our implementation of the proposed two-sided rendezvous protocol, as described in Section 14.3.3, may not be fully optimized yet. It may be worth to investigate how it can further benefit from proposed optimizations of rendezvous protocols, such as [54], to remove one of the necessary control messages. This means that we also have not yet taken into account if our approach can also benefit from other approaches that were suggested to enhance the communication performance specifically for persistent MPI operations. We mention [50] in particular, where communication is scheduled via multiple RDMA engines in order to achieve a better performance. Coincidentally, this approach also has the "downside" that it assumes that persistent send operations always match with persistent receive operations. The authors stated that "these restrictions can be considered to be very natural and not a big problem in most MPI programs"[50]. But with our proposed compiler analysis one can make sure that persistent operations can only be matched with persistent operations and one needs not solely to rely on the programmer to adhere to this restriction that is not mandated by the MPI standard. Therefore, this other approach may also benefit from our proposed compiler analysis.

Currently, the scope of our compiler analysis is limited, in the way, that the lifetime of a persistent request, from initialization until the call to `MPI_Request_free`, has to be confined to one function. In the future, we want to extend the compiler analysis to also be able to detect matching conflicts between different functions. One idea based on MetaGC [118] is to annotate a call-graph of the program with all the possible matching conflicts and then check the graph for paths between these.

We propose that the information that a persistent operation will always be matched with one persistent counterpart be added as a new MPI assertion, so that the user can still benefit from the performance gain even if the compiler analysis fails to detect that this holds. We have seen that this restriction of MPI's behavior opens up the possibility for several different optimization strategies. Therefore, having this assertion may encourage other researchers as well as MPI implementers to seek out further optimization potential in this area.

14.6. Conclusion

In this work, we demonstrated the compiler's ability to find opportunities where MPI message matching can be avoided for persistent operations. As we explained in Section 14.2, this can be done as an entirely local analysis without the need to match messages statically. Additionally, the concrete values of the message envelopes do not need to be determined at compiler time, as the compiler can often symbolically determine if message envelopes differ. With our two-sided rendezvous protocol explained in Section 14.3, we find that this can achieve a communication overhead reduction of up to 95 percent for larger message sizes, for the part of the overhead that can not be hidden behind computation (see Section 14.4).

As presented in Section 14.5, we want to further enhance the scope of our compiler analysis and optimize our implementation of the communication protocol in the future.

The code for the compiler analysis, as well as the implementation of the communication, is available at our github: `https://github.com/tudasc/MPI-CompMatch`

15. Precomputation of MPI Message Envelopes

Bibliographic Information

The contents of this chapter has previously been published in:

Tim Jammer, Tim Heldmann, Michael Blesel, Michael Kuhn, and Christian Bischof. "Compiler-Based Precalculation of MPI Message Envelopes". In: *High Performance Computing. ISC High Performance 2024. Lecture Notes in Computer Science*. Hamburg, Germany: Springer International Publishing, May 2024. DOI: 10.1007/978-3-031-73716-9_1
The contribution of the author of this thesis is summarized as follows:

As the leading Author, Tim Jammer contributed the Idea, Implementation and evaluation Technique. Michael Blesel executed the evaluation measurements, as well as providing insight into call graph analysis on the clang-frontend level. Tim Heldmann provided insight into call graph analysis on the llvm-IR level. The Manuscript was mainly written by Tim Jammer, while all authors and especially Michael Kuhn and Christian Bischof provided invaluable feedback on the manuscript's text and the presentation of the results.

Abstract

In this paper, we propose a novel approach to optimize MPI communication overhead by precomputing the used message envelopes at the initialization of MPI. This allows for selective disabling of MPI message matching when no conflicts are present, which leads to an enhanced communication performance, while still ensuring correctness.

We developed an LLVM compiler pass that automatically inserts the precomputations of message envelopes into an application and is capable of handling advanced C++ language features like polymorphism and exceptions. We demonstrate the effectiveness of our approach using a complex C++ N-body simulation code, achieving performance gains despite the runtime-overhead of precomputation. In our experiments, the runtime overhead of envelope precomputation is negligible compared to the potential performance gain ($\leq 0.25\%$ of performance gain are lost to precompute overhead).

Our code is available at `https://github.com/tudasc/MPI-precompute`

15.1. Introduction

MPI stands as the cornerstone of distributed memory computing, serving as the de facto standard in facilitating large-scale distributed memory applications. Handling messages inside the MPI library is associated with a certain communication overhead. One major technique to reduce this overhead is the overlapping of communication with computation by using non-blocking MPI operations. For non-blocking operations, the network hardware can handle the message transfer, while the application continues its calculations. But even in the case of non-blocking operations, MPI still needs to perform matching based on the message envelopes. A message envelope, as defined by the MPI standard [136], consists of the MPI communicator used for communication, the target and destination processes (rank) as well as a user-defined integer (message tag).

In some cases, the message matching is superfluous, though. An example is the usage of persistent MPI operations. In a persistent MPI operation, one first defines a message envelope and then sends repeated messages with the **same** envelope. So if a persistent send and receive operation match once, they will also match in every other instance, making all operations spent to determine that those two operations will match again superfluous. However, the MPI standard does not mandate that a persistent send operation is matched by a persistent receive operation. Additionally, several different persistent operations may use the same envelope. In order to ensure a correct program execution, message matching is still required in those cases. Therefore, an MPI implementation is not allowed to just disable message matching for persistent MPI operations.

Here, we see a potential to further reduce the overhead of MPI communication by eliminating the superfluous message matching in cases where this is applicable. This is especially beneficial to the performance of message passing, as the operations needed to perform the matching can not be overlapped with computation without special hardware.

Schonbein et al. identified the cost of message matching, especially in multi-threaded environments, as one component that adds the most overhead to MPI communication [99]. But the communication overhead does not only consist of envelope matching. Zambre et al. points out that for a non-blocking send operation, up to 90% of the overhead is spent within the MPI implementation whereas only less than 10% is spent in the lower levels of the software stack actually dealing with the network hardware [109]. Before actually engaging with the network hardware the MPI library may perform various other steps such as checking if the data is contiguous, computing which communication interface and protocol to use, etc. Similar to matching, these operations are also mostly superfluous if the same (persistent) MPI operation is executed repeatedly. Our approach also targets these superfluous operations.

Several techniques to reduce the cost of message matching have been proposed. Some are software-based, like Flajslik et al. who propose to "utilize a hash map that is extended with message ordering annotations" [73]. Their work complements ours, focusing on irregular communication patterns with matching wildcards, while ours concentrates on regular communication patterns.

Most studies explore hardware-based approaches, where specialized hardware is used to accelerate the message envelope matching instead of the CPU performing the required operations [25, 29, 102, 103, 106]. One drawback is their potential inability to adapt to future changes in MPI usage. For example, the current constraints of MPI message matching do not translate well to GPU architectures [83]. We argue that in the case of redundant matching operations, it is better to utilize such hardware to perform tasks relevant for the application, like in-network computing or reduction operations for example.

In terms of completely eliminating superfluous message matching operations, Jammer et al. demonstrated that in certain cases, the compiler can identify redundant matching [142]. The idea is that the compiler can in some cases prove that all envelopes sent out are different and therefore, no "matching conflict" can occur. In order to guarantee a correct execution one needs to prove that no matching conflict can occur **before** message matching is disabled. Therefore, speculatively disabling message matching is not a viable option.

The compiler-based approach by Jammer et al. is currently limited to MPI calls that occur within the same function [142]. Additionally, a purely static analysis may not always conclusively decide message envelopes to be different, even if they are so during execution.

Therefore, we propose to precompute all used MPI message envelopes before they are used to check for the presence of matching conflicts at runtime. The compiler does however also play a crucial role in our approach as it is used to automatically generate the instructions necessary to precompute the used MPI message envelopes. Consequently, no manual work by the programmer is needed in order to exploit the performance gain achieved by the optimization developed by Jammer et al., where message matching is disabled [142].

This paper is structured as follows: First, we introduce how we utilize the compiler to compute all message envelopes in Section 15.2. In Section 15.3, we discuss in which cases our approach does not work optimally, while providing a case study on a complex application, the MUrB N-body simulation code, in Section 15.4. We summarize and conclude with an overview over possible avenues of future work in Section 15.5.

15.2. Analysis Approach

The general idea of our approach is to precompute all used message envelopes before the program starts right after `MPI_Init`. Then the MPI library can check all used envelopes for matching conflicts and disable message matching if applicable. The compiler will automatically determine how to compute the message envelope used and insert the necessary instruction after the call to `MPI_Init`. This is done by generating a copy of the original program's instructions containing only ones necessary for calculating the message tag and source (or destination) process. This is sufficient, as a difference in a subset of the envelope, as defined by the MPI standard, is already enough to tell different messages apart.

We do not consider the communicator as integral to the envelope as the tag and involved processes, as a significant portion of MPI communication operations directly use `MPI_COMM_WORLD` [147]. Nonetheless, the idea of precomputation can also be applied to the communicator.

We implemented a compiler transformation pass operating on the LLVM intermediate representation (IR). The analysis and transformation workflow is illustrated in Fig. 15.1. The compiler transformation will search for all MPI send/receive operations in LLVM IR and mark the rank and tag parameters as "necessary to precompute". Then, the values used to compute the values marked as "necessary to precompute" are also marked recursively, until we arrive at a value known to the compiler (e.g. a constant value) or a value derived from the execution environment (such as a command line argument or

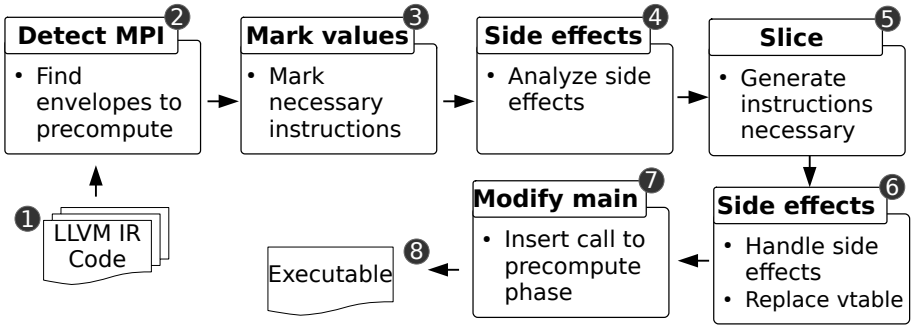


Figure 15.1.: Illustration of the analysis process. Starting from the LLVM IR (1), first the message envelopes that should be precomputed are detected (2). Then, all instructions necessary for calculation and control flow are marked (3) and their side effects analyzed (4). The pass then generates the part of the program necessary for precomputation (5) and adds the handling of side effects (6). Finally (7), a call to the precompute phase is injected into `main`, before the usual LLVM LTO optimizations pipeline is run (8) in order to produce a binary.

the number of MPI processes present). An example is shown in Listing 15.1, where the statements "necessary to precompute" are highlighted. In this example, first step, lines 5 will be marked as "necessary to precompute", as the result of this calculation is used as the target rank argument.

When marking values as "necessary to precompute" the analysis also marks the instructions necessary for the control flow to reach the precomputed value as "necessary to precompute". This way, the generation of multiple envelopes inside a loop is also correctly handled. This also happens recursively until we reach the program entry. In the example in Listing 15.1, the call to `init_communication` is marked (line 18 in Listing 15.1), as some instructions in this function are considered "necessary to precompute". The for loop (lines 17–18 in Listing 15.1) is not "necessary to precompute", however, as its execution has no effect on the envelopes.

When the compiler analysis does not find any remaining value that is "necessary to precompute", it copies only those marked instructions. In this regard, this is an application of program slicing techniques [3], as we extract the part of the program that is necessary for the calculation of the message envelopes used.

The MPI calls used to define send/receive envelopes are then replaced with a callback

```
1  double *DATA;
2  int numprocs;
3
4  void init_communication(int rank, int tag_to_use) {
5  int other = (rank + 1) % numprocs; // next
6  MPI_Send_init(/*buf, size, dtype*/, other, tag_to_use, /*comm, req*/);
7  MPI_Recv_init(/*buf, size, dtype*/, other, tag_to_use, /*comm, req*/);
8  }
9
10 int main(int argc, char *argv[]) {
11 int rank;
12 MPI_Init(&argc, &argv);
13 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14 MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
15 DATA = malloc(sizeof(double) * 100);
16 for (int i = 0; i < 100; i++) {
17 DATA[i] = 42.0; }
18 init_communication(  rank,   argc);
19 // run some calculation
20 }
```

Listing 15.1: Example code with statements "necessary to recompute" highlighted.

to the precompute backend library, that keeps track of all the values precomputed. The information, which envelope belongs to which MPI operation, is based on order of execution, as the order of calls to the precompute backend will match the order of MPI calls in the "real" execution of the application. After the precalculation of all the message envelopes is finished, the compiler inserts instructions, to make sure the program is in the same[1] global state as it was in the beginning of the precomputation. Then, the normal execution of the program resumes. This execution process is illustrated in Fig. 15.2.

Our analysis does work even if the envelopes are spread across multiple translation units, as we run the compiler transformation pass at the Link Time Optimization (LTO) phase. If a function needs to be called that is not part of the information that is available at the LTO phase, our analysis will abort with an error, unless the user explicitly marked this function as "side-effect free if called during precompute phase" with an environment variable.

The following sections details how to handle complex control flows and side effects of the instructions necessary to precompute.

[1]The state of the precompute backend library is excluded in this consideration.

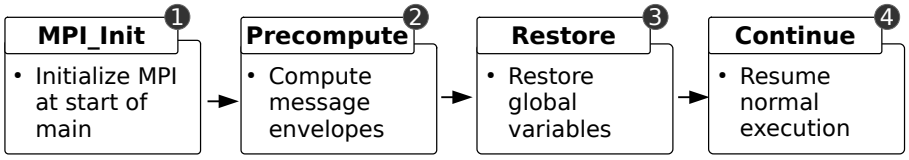


Figure 15.2.: Illustration of the execution of an application altered by our pass. First (1) MPI is initialized like normal. Then (2) the precomputation phase starts, where all message envelopes are computed and registered to the backend library. After the precomputation phase, the original global state is restored (3), before normal execution resumes (4).

15.2.1. Global side effects

In our analysis, it is crucial to consider all side effects conveyed by envelope precomputation. We detail how we handle these effects to ensure they don't impact application execution in this section.

Pointer aliasing

Pointer aliasing is a key concern for many static analysis approaches. When determining whether a memory read operation is "necessary to precompute", our analysis needs to mark any write operations to that location that might be executed before as "necessary to precompute". A key challenge in our analysis is that we need to determine all write operations in all functions that may be executed before. Therefore, our analysis does not only consider pointer usages "upwards" (in the LLVM def-use chain) to find the pointer origin (likely its allocation), but also its "downward" usages to find the write operations that are important. In Listing 15.1, this is important for the variables `numprocs` and `rank` (lines 2 and 11), as these variables are accessed via pointers by the MPI library (lines 13–14). If a pointer is passed to a function, our analysis will trace the pointer usages in this function. During this analysis, if we see that another pointer is derived and a new alias is created, we treat it the same as the original pointer. When determining the write operations that are important, our pass only considers those that can happen before the read of the value that was marked "necessary to precompute". In the case of two operations being in separate functions, their order is determined by the order of calls to the functions in question in their least common caller in the call hierarchy (at least `main`).

Memory management

If the application allocates memory during the precomputation phase, the precompute library will intercept the allocation call in order to keep track of the allocated memory, so that it will be freed after the precomputation phase. Our implementation is limited in the allocation calls it currently supports (see Section 15.3).

Global variables

For global variables, our pass will re-initialize them with their original value after the precomputation phase has finished, similar to checkpointing. This is necessary, if they could have been modified during the precomputation phase. This applies to `numprocs` in the example in Listing 15.1 (line 2).

MPI library

For the state of the MPI library, there are no relevant side effects as our analysis is aware of the side effects of MPI operations. Usually, only side-effect free operations are necessary to compute the message envelopes, like reading the rank of the current process or the number of processes (lines 13–14 in the example of Listing 15.1). Our analysis would abort with an error if execution of other MPI operations was determined necessary for envelope computation. We did not encounter this case in our evaluation, though.

Execution environment

The program may interact with its execution environment such as reading the command line arguments, environment variables or input files. For the execution environment, reading from the environment is considered "side effect free" for our purposes, as it does not modify the environment, therefore executing code interacting with the execution environment in a read-only way is not harmful in the precomputation phase, as the "real" execution of the application can read the execution environment again. Any writing to the execution environment is therefore not allowed. But modifying the execution environment should not be necessary in order to compute the message envelope used, as write operations have no "result" that can be used in order to derive a message envelope from it. We will discuss in which cases this presents a limit to our approach in Section 15.3.

15.2.2. Advanced C++

Our pass operates on LLVM IR and is therefore capable of handling C as well as C++ codes. For C++, there are some advanced language features, commonly challenging static analysis tools. Templates are already instantiated by the clang-fronted, but other language features require extra consideration at LLVM IR level, which we detail in this section.

Polymorphism

For control-flow and call-graph analysis, complex inheritance hierarchies with polymorphic objects does present a challenge.

Polymorphic objects are implemented with virtual tables (vtable).This results in indirect calls in the LLVM IR, where the function pointer to call is first loaded from the vtable and then called. The LLVM tool chain does already include a whole program de-virtualization pass that can be run during the LTO phase. We piggybacked on the analysis part of this pass to determine the possible call sites for indirect calls. If the de-virtualization does not yield a list of all possible call targets for a given call, we do consider all functions, whose pointer is captured, as possible call targets.

The load of a function pointer that can lead to a callee that is marked as "necessary to precompute" is therefore also marked. As explained above in Section 15.2, this leads to all relevant stores to also being marked. Our transformation then replaces the usage of the original function pointer with an altered one, so that instead of the original function, the altered one – containing only the instructions "necessary to precompute" – is called. This means that the pass will create an alternative vtable that is used in the precompute phase, while the original one is still used after the precompute phase.

Exceptions

Exceptions may alter the control flow of a program, therefore we need to analyze them. In most cases, our analysis actually ignores exceptions and aborts, as these are considered so harmful that the precomputation does not work correctly anyway. In these cases, for example when the process is out of memory, it is very likely that the original application will also crash. However, for the exceptions that are properly handled by the original application, this approach is not sufficient. Consider the example sketched in Listing 15.2:

```
struct input_parameters;
try{
  input_parameters = Read_Input_File(); }
catch (const std::system_error& e) {
  // catch I/O errors like file not found
  input_parameters = default_input; }
target_process=depends_on_input(input_parameters);
```

Listing 15.2: Sketch of exception that is "necessary to precompute".

In this case, the envelope to precompute depends either on the input data or is based on the default input, if there was some exception during file I/O.

This case is correctly handled by our analysis as well. In the LLVM IR, exceptions are handled with a `landingpad` instruction. Our pass discovers that the `landingpad` (analogous to the `catch` statement in line 5 in Listing 15.2) is important, as the resulting value of `target_process` does depend on it. The analysis therefore marks all operations that can raise an exception caught by the `landingpad` as "necessary to precompute", so those are executed in the precompute phase to check if the exception is raised. This approach does also work if – in contrast to the code example in Listing 15.2 – the operation that raises the exception is not "necessary to precompute" in the exception-free case.

For optimization purposes, our analysis can ignore exceptions that result in diverging the control-flow completely away from the computations of message envelopes. Examples are exceptions that may be the result of writing to the execution environment (e.g. writing to stdout). The compiler can generate exception handling code when for example a write operation to stdout fails, as C++ semantics require that certain destructors are called in this case. We discuss in which cases ignoring certain exceptions limits our approach in Section 15.3.

15.3. Discussion

In this section we discuss the current shortcomings of our approach. First, we will discuss the general limitations of our approach, before detailing some limitations in our implementation that we do not consider a general shortcoming of the precomputation idea.

Limits of the approach in general

In general, our approach only works with full LTO information, which can increase compile time. On the other hand, this also allows the application to reap the benefits obtained by LTO.

The current implementation does not include any heuristic to determine if precomputation is worth the effort. If e.g. the application termination depends on the input data (i.e. on convergence) and then some messages are sent at the end of the program, the precompute phase needs to actually run the whole computationally intensive part of the program, in order to arrive at the point where those final message envelopes are computed. In the future, we want to incorporate a heuristic that can find such cases and disable the precomputation if it does lead to too much runtime overhead.

Limits of the current proof of concept implementation

Additionally, there are some limitations in the current implementation, described in Section 15.2.

There is the possibility of a resource leak in the current implementation, for example when memory is allocated or files are read, as the corresponding free or file close is not part of the precompute phase. A solution for this problem could be the precomputation library intercepting all resource acquiring calls such as memory allocations. The precompute library can then keep track of all aquired resources and properly clean them up after the precompute phase.

Reading from `stdin` is currently not supported, as it may require interactive input by the user. Although we have not encountered a scenario where message envelopes depend on input form `stdin`, this capability could be added, by capturing `stdin` to an internal buffer for usage after the precompute phase. We did not consider programs that write to the execution environment and then later expect to read these written values for further usage, as any process could change the environment in between.

Where our implementation can currently lead to unexpected or even undefined behavior is the handling of exceptions. For example, consider an exception thrown while reading the input arguments, that cause the program to print a help message about its arguments. As discussed in Section 15.2.2, most exceptions are actually ignored. Therefore, our analysis will ignore the exception handling in that case, as no message envelope depends

on it. This can lead to a program crash without the help message being printed, resulting in a different program behavior than without the precompute phase.

Furthermore, this can lead to undefined behavior (i.e. if the wrong arguments are later used), as no such exception will be raised during the precompute phase. We do consider this as a limitation of our implementation that needs further investigation, as we want to avoid executing all exception checking code during the precompute phase for performance reasons.

However, this isn't inherent to our precomputation approach, as in the worst case, the precompute library can catch any unhandled exception thrown during the precompute phase and abort the precomputation. In this case, we can just tell the MPI library that matching conflicts may be present and proper message matching mechanisms need to be used, in order to continue like normal. We elected to stay with the current implementation, as our assumption is that the user is capable of using the optimized program correctly and most programs do not handle exception in other ways than just aborting anyway. We regard this choice similar to assertions, which are not included in release code.

15.4. Evaluation

We demonstrate the capabilities of our analysis by showcasing it on an N-body Simulation code: MUrB (MoveUrBody). The code was developed by Adrien Cassagne to showcase different implementations of N-body simulations using his MIPP SIMD wrapper [91]. MUrB has roughly 42,000 lines of C++ code across ≈150 files. MUrB uses complex C++ features such as polymorphic inheritance and templates, making it very challenging to analyse with static analysis tools. Additionally, the communication scheme used does depend on the implementation selected. The user can choose the implementation to use via a command line parameter, the usage of one of the different implementations is then managed with polymorphic inheritance. Therefore, this code allows us to sufficiently showcase the capabilities of our analysis approach, as a purely static tool would not be able to decide which implementation will be selected by the user at runtime.

In order to successfully apply our analysis to MUrB, we only needed to swap the used compiler with a compiler wrapper that passes the needed command line arguments to enable our transformation pass during the CMake build step.

Our compiler pass successfully transforms the application to include a precompute phase. This allows us to *automatically* apply the optimization developed by Jammer et al. [142]

numProc	msg size	performance gain	cost of precompute	performance gain lost to precompute overhead
2	2.28 MB	0.11% (1701 ms)	3 ms	≈ 0.18%
4	1.14 MB	15.59% (6749 ms)	3 ms	≈ 0.04%
8	0.57 MB	5.75% (6788 ms)	3 ms	≈ 0.04%

Table 15.1.: Performance gain contrasted with the run time required for precalculation.

to the complex MUrB code. The application transformed by our pass behaves like the original one. The only exception is the wrong usage of command line arguments, where the altered application just crashes, while the original one prints a help message (see Section 15.2.2).

When knowing all message envelopes, one can see that MUrB uses distinct message tags for each pair of persistent send/receive operations, so that message matching is not observe a performance benefit, detailed in Table 15.1.

We measured the performance benefit on a system consisting of eight nodes, each equipped with one AMD Epyc 7443 CPU (2.85 GHz, 24 cores), 128 GB RAM and 100 Gbit Ethernet. The setup uses CentOS 8 Stream, LLVM 16.0.1 and our modified version of Open MPI 4.1.1. We executed MUrB with one MPI process with 24 threads per node.

As the optimization only targets the communication overhead, we do not expect to see a very high overall performance benefit, considering that a high computation-communication overlap is already present in MUrB. Still, there is a performance benefit that exceeds the runtime overhead of precomputation. As seen in Table 15.1, the relative performance gain increases with a growing communication overhead caused by more processes being used for the same problem size.

15.5. Conclusion

In this work, we demonstrated the feasibility of precomputing the used MPI message envelopes at MPI initialization. This does allow for disabling of proper message matching inside the MPI library, when no matching conflicts are present. Disabling message matching can increase communication performance, while retaining correctness according to the MPI standards semantics, due to the precomputed information.

We demonstrated the capability of our implementation to handle complex C++ codes, by successfully applying it to the N-body simulation code MUrB. In this example application, we were also able to achieve a speedup by *automatically* applying the optimization developed by Jammer et al. [142]. During our measurements, the cost of precomputing the message envelopes was offset by the increased communication performance as soon as the second iteration.

In the future, we want to further address the current shortcomings of our implementation discussed in Section 15.3 with a specific focus on efficient exception handling. Additionally, we want to include a heuristic to decide if precomputation of message envelopes is worth the runtime overhead. We also want to explore how further static analysis could be used to limit the amount of message envelopes necessary to precompute. We have already done some initial investigations into this approach by using a Clang plugin that creates a global control flow graph of the program and analyses it for MPI communication patterns. The main challenge with this static approach is the possible influence of runtime dependent parameters on the message envelopes, which is tackled by our precompute approach. Nevertheless, there are communication patterns that could be analyzed correctly at compile time. In that case the Clang plugin can annotate the affected persistent operations, where it was able to show the absence of matching conflicts. By incorporating such a static approach, we could reduce the amount of precomputations necessary. Furthermore, if one could determine different phases of the application, one could apply our approach to the main phase and enable message matching again for a later cleanup phase, in order to skip the precalculation of message envelopes for the cleanup phase.

Our code is available on GitHub: `https://github.com/tudasc/MPI-precompute`. To enhance reproducibility, the repository contains not only the developed compiler pass, but also the code of MUrB and the altered Open MPI with the performance adjustments developed in [142].

Acknowlegements:

This work was in part supported by the Hessian Ministry for Higher Education, Research and the Arts through the Hessian Competence Center for High-Performance Computing and by the Federal Ministry of Education and Research (BMBF) and the state of Hesse as part of the NHR program. The N-Body Simmulation Code MUrB was written by Adrien Cassagne and uses his MIPP SIMD wrapper [91].

16. Annotation of Compiler attributes for MPI Functions

Bibliographic Information

The contents of this chapter has previously been published in:

Tim Jammer, Adrian Schmidt, and Christian Bischof. "Annotation of Compiler attributes for MPI Functions". In: *Proceedings of the 31th European MPI Users' Group Meeting*. EUROMPI/Australia '24. Perth, Australia, Sept. 2024. DOI: 10.1007/978-3-031-73370-3_2

The contribution of the author of this thesis is summarized as follows:

As the leading author, Tim Jammer contributed the initial idea. Adrian Schmidt provided the actual implementation as part of his bachelor thesis, supervised by Tim Jammer. Adrian Schmidt drafted the initial manuscript, which was subsequently improved by Tim Jammer. Christian Bischof provided invaluable feedback on the manuscript's text and the presentation of the results. All authors agree with the use of the publication for this thesis.

Abstract

This paper explores the use of LLVM IR function and parameter attributes to enhance compiler optimizations for code that uses MPI. As MPI is usually used as a dynamically linked library, the compiler is not able to automatically infer certain function attributes like `nofree`, which signals that no memory is deallocated in this function. Therefore,

we implemented an LLVM compiler pass that annotates the used MPI functions with suitable attributes when compiling the user application. We manually derived applicable attributes based on the semantics described in the MPI standard, so that this approach is applicable to all MPI implementations.

We showcase different cases where this additional annotations impact the code generated by the compiler for the MiniApps from the Exascale Proxy Applications Project. The addition of MPI function annotations allows for a variety of compiler optimizations like reducing unnecessary memory accesses, optimizing register usage, and streamlining control flow.

The code of our annotation pass is available on GitHub: https://github.com/AdrSchm/mpi-attributes-pass

16.1. Introduction

Many libraries for C and C++, including the MPI library, are used by including header files that contain function declarations and other definitions, and later linking against a precompiled library object file. Hence, when compiling applications that use them, the compiler only knows about the declarations, but not about the semantics of these functions. This prevents certain compiler optimizations that are possible when the compiler has more information, e.g. by the function being part of the same translation unit. Part of this lost information is the memory access behavior of functions, which is especially relevant when the function takes pointer parameters. Some of those pointers are used read-only, others may only be written to. Currently, the compiler does not have access to this information to perform optimizations and needs to assume that every pointer is used for reading *and* writing. This results in the compiler needing to store the values of variables in memory before such function calls and reload them again afterwards. If the compiler knows additional access information, it may be able to keep values in registers instead, and thus use fewer memory accesses. Due to MPI being a library that relies heavily on the use of pointers, there are potentially many such opportunities for compiler optimizations in MPI applications.

A possible way to add information to the compilation process are compiler attributes. Adding attributes to declarations in header files provides compilers with some of the lost information, potentially enabling additional optimizations.

The idea of using compiler attributes between translation units is similar to the ”Header Time Optimization” proposed by Moses et al., where they propagated the attributes inferred by the compiler to different translation units via annotations in the header file [120]. As Moses et al. point out, more than half of the speedup achieved by using link time optimization (LTO) [82] can be obtained by only supplying the compiler with proper function attributes [120]. In our work, however, we do not use the compiler to infer the relevant function attributes, rather we infer them manually based on the MPI standards semantics.

In this paper, we investigate the effect that adding attributes to MPI functions has on the produced binaries of a selection of MiniApps. We investigate their impact on code generation, and how they enable formerly unused optimization possibilities.

The remainder of this paper is structured as follows: First, we provide an overview of the attributes we use in Section 16.2, and show how we annotate them to MPI functions. In Section 16.3, we showcase the differences of the generated assembly code with and without the annotations. Section 16.4 concludes our paper with a summary and an outlook on future work.

16.2. Attributing MPI Functions

We provide the compiler more information on the pointer usages of MPI functions by using LLVM IR function and parameter attributes. Usually, the compiler analyzes the function and infers the relevant attributes itself. But this only works inside of one translation unit. As the MPI library is usually dynamically linked at runtime, we need to specify the relevant annotations manually.

As the relevant attributes exist on the LLVM IR level, we implemented an LLVM optimization pass that annotates the definitions of the used MPI functions before all other optimizations happen. The attributes we employed are described in Section 16.2. Since these attributes are already part of the LLVM infrastructure, the usage of these attributes in order to perform code optimizations is already part of LLVM.

As can be seen in studies on MPI usage [105, 147], most applications use only a small subset of the available MPI functions. Therefore, we currently limit the scope of our pass to such commonly used ones that are expected to be found in most MPI applications, such as point-to-point and collective operations.

16.2.1. Function and Parameter Attributes

As the MPI API extensively uses pointer arguments, and the memory access semantics is one of the key factors in possible compiler optimizations, we use the following relevant attributes:

memory This function attribute describes which memory locations are accessed in which way.[1] For our annotations we mostly use it as `memory(argmem: readwrite, inaccessiblemem: readwrite)`. The functions annotated in this way only access memory based on pointers given as parameters, or memory that the calling context has no access to, e.g. library internal buffers. Any other memory locations that are accessible from the calling context are not used by a function with this attribute, like for example captured pointers or other local variables. Therefore, it signals that memory related side-effects are possible, but not observable in the callers context unless those memory accesses happen through the function arguments. In contrast, `memory(readwrite)` indicates that accesses to arbitrary memory locations are possible, including those not happening through the arguments, but still observable in the callers context. Using this restriction allows for possible optimizations, as the compiler can be sure that variables that are not used during the call have the same value afterwards as they had before. It is possible to further specify the memory access behavior of individual parameters using the following attributes.

readnone This parameter attribute specifies that a pointer argument is not dereferenced during the function call. So variables do not have to be stored before and loaded afterwards, but can be kept in registers, for example.

readonly This parameter attribute specifies that a pointer argument is only dereferenced for reading the pointed to data. Pointed to variables need to be written to memory before the call, but they can potentially be kept in registers and used from there after the call.

writeonly This parameter attribute specifies that a pointer argument is only dereferenced for writing to the pointed to memory. This attribute describes that *if* a memory access happens it will be to only write data, but no guarantee that an access *will* happen. In rare cases this can omit stores to the memory region before function calls.

[1] Since LLVM 16 this attribute replaces the attributes `argmemonly`, `inaccessiblememonly` and `inaccessiblemem_or_argmemonly`

nocapture This parameter attribute specifies that a pointer argument is not captured by the function call. Capturing the pointer means that the function stores any bits of the pointer somewhere else, where they are accessible by other threads or functions even after the capturing function returns. These other threads or functions may be able to use this pointer for memory accesses or other operations, potentially modifying values that are used in other contexts. Other literature also use the term pointer *escape* for this [11]. In LLVM, `nocapture` implies *noescape* and these terms are used interchangeably, as there is no distinct attribute *noescape* [159]. Being sure that the pointer is not captured and can therefore not escape, the compiler can omit memory accesses after the function call, as there are no other threads or contexts that may still have access the annotated pointer being able to modify the pointer's content.

nofree This parameter attribute specifies that the memory pointed to by a pointer argument is not freed during the call, so that accesses to the memory locations are still valid afterwards. An identically named function attribute extends this to all valid pointers in the calling context.

Other memory access related attributes of LLVM, such as `noalias`, are only used to optimize the function body and are not relevant for optimization of the code at the callsite. Thus, we focus only on the relevant attributes described above.

16.2.2. Applying Attributes to MPI Functions

Inside one translation unit, the compiler can automatically infer some functions and parameter attributes. In this work, however, we infer fitting attributes manually by analyzing the semantic of MPI functions as described in the standard [148]. This approach allows for the annotation of additional attributes that the compiler cannot infer by itself. This strategy is not dependent on any particular MPI implementation, hence our pass is valid for any standard compliant implementation. We only annotate those pointers which are part of the MPI standard. For an argument with the type `MPI_Comm` for example, OpenMPI uses a pointer, while MPICH uses an integer. As both choices are in accordance with the standard, those arguments are not annotated. The following provides examples of annotations for different MPI functions:

Utility functions such as `MPI_Comm_rank` or `MPI_Comm_size` read the status of the MPI library. Those are annotated as follows:
`nofree memory(argmem: readwrite, inaccessiblemem: readwrite)`

`@MPI_Comm_size(%comm, ptr nocapture writeonly %size)`. The pointer argument to this function is only written, which is why we annotate it with `writeonly`. As the MPI library does not capture the pointer, we can also annotate it with `nocapture`. The used `memory` attribute shows that no further memory access occur. This function does not free any memory, so we annotate it with `nofree` as well.

Blocking point-to-point operations are annotated as follows:

```
nofree memory(argmem: readwrite, inaccessiblemem: readwrite)
@MPI_Send(ptr nocapture readonly %buf, %count, %dtype, %dest,
%tag, %comm)
```

According to the MPI standard, the used send buffer is only read. Therefore, we annotate it as a `readonly` pointer. As this is a blocking operation, it has completed once the call returns. Since the pointer to the buffer is not used after the call returns, the `nocapture` attribute is appropriate. This attribute may not be inferred by the compiler alone. For the compiler to be able to infer `nocapture`, it needs to *prove* that nothing captures the pointer. As the network might be involved in the communication, the compiler may not be able to prove that all lower network layers do not capture the pointer. Since the call to `MPI_Send` does not free any memory accessible to the caller, we can annotate the whole function with `nofree` as well. The `memory` attribute, signalling no memory accesses to other reachable memory, is, strictly speaking, not a valid attribute for this function, or most other MPI functions for that matter. The reason is that it is possible to provide the MPI implementation with an additional user-allocated buffer which can then be used by the MPI library (buffered-mode). This buffer is potentially accessible from the user's program, which would require to use the attribute that *all* memory is `readwrite`. However, usage of a buffer supplied to the MPI library is undefined behavior according to the MPI standard's semantics. Therefore, for any compliant MPI application the user-provided buffer is effectively inaccessible to the user. This allows usage of the more restrictive memory attribute, potentially enabling more optimizations. Receive operations are annotated similarly, with `writeonly` being used instead of `readonly` for the buffer argument. The status parameter of a receive operation is annotated with `writeonly` and `nocapture`, as it is only written. Often it is `MPI_STATUS_IGNORE` [147] and thus not that important.

Non-blocking point-to-point operations have the same attributes as their blocking counterparts:

```
nofree memory(argmem: readwrite, inaccessiblemem: readwrite)
@MPI_Isend(ptr nocapture readonly %buf, %count, %dtype,
%dest, %tag, %comm, ptr nocapture writeonly %request)
```

The use of `nocapture` might look incorrect at first glance. Strictly speaking, the MPI library may capture the pointer to the send-buffer. But - similar to the above reasoning - the send-buffer pointer is ”out of the user-application context” while the non-blocking operation is in progress, as any write usage can result in undefined behavior according to the MPI standard’s semantics. For a send operation, read usages of the send-buffer are not harmful, but the semantics of MPI imply that the MPI library cannot write through the ”captured” pointer. This means that assuming `nocapture` is also correct in those cases. Therefore, the compiler can treat non-blocking operation as ”finished after the non-blocking call returns”, as a standard compliant user application will not harmfully access the message buffer before it ensures completion with another MPI call.

Persistent operations are annotated as follows:

```
nofree memory(argmem: readwrite, inaccessiblemem: readwrite)
@MPI_Send_init(ptr readnone %buf, %count, %dtype,
%dest, %tag, %comm, ptr nocapture writeonly %request)
```

In this case, the message buffer is captured by the MPI library. The above reasoning does not apply, as the MPI library will use the captured buffer when the user instructs it to do so by using `MPI_Start`. As the initialization of a persistent request does not start communication, the buffer is annotated with `readnone`, as it is only going to be used later.

MPI_Start As this function starts the actual communication of a persistent request, we cannot specify any restrictions on the memory access of this function. The available attributes are not able to express any connections between the communication request and the accessed buffers, as the same method is used to start send and receive operations.

Collective operations have similar annotations as point-to-point operations:

```
nofree memory(argmem: readwrite, inaccessiblemem: readwrite)
@MPI_Allreduce(ptr nocapture readonly %sendbuf,
ptr nocapture %recvbuf, %count, %dtype, %op, %comm)
```

Some collective operations can be specified to happen in-place by using `MPI_IN_PLACE` as the send buffer. The compiler is able to detect the usage of this special MPI pointer value and could theoretically adjust the annotations, based on which pointers are read and which ones are written, by the given usage pattern. This is not implemented in our pass yet, as it currently only supports annotations on a per-function level and it is possible to have some operations in place and others with separate send and

receive buffers. Therefore, our pass uses the most conservative annotations that match usage of in-place operations as well as distinct memory regions, to limit the tool's complexity. The `nocapture` attribute holds for both uses.

16.3. Evaluation

16.3.1. Impact on Exascale Proxy apps

Our pass is not meant to be used when compiling an MPI library itself, but intended to be included when compiling user applications that make use of the library. To evaluate the impact of the additional added attributes, we employ our pass during the compilation process of 14 MiniApps by the Exascale Proxy Applications Project [141]. We tested all Proxy apps that use C or C++ and could be built without the need for external dependencies.

In Fig. 16.1 we show how many MPI functions are annotated in the MiniApps. Even though our pass only annotates the most used subset of all MPI functions, it covers most functions (68.2%) and callsites (77.2%) found in the Mini-Apps. Some functions used in the MiniApps are not yet part of our pass, those include functions to create derived MPI datatypes or communicators with topology information (e.g. `MPI_Cart_create`), as well as MPI I/O functions, as those MPI functions are not used in the majority of MPI applications [105, 147]. This can also be seen in Fig. 16.1, considering that our pass annotates a majority of the used MPI functions for a majority of the MiniApps.

Table 16.1.: Comparison of annotated functions and callsites

	miniQMC	Kripke	PENNANT	Quicksilver	miniXyce	SimpleMOC	HPCCG	miniTri	Comb	LULESH	miniVite	miniAMR	hypar	sw4lite
Functions not annotated	0	2	0	5	1	5	2	7	15	3	14	4	10	14
Functions annotated	5	10	12	16	8	10	9	12	17	11	18	15	16	17
Callsites not annotated	0	2	0	5	1	10	7	7	24	17	47	8	18	127
Callsites annotated	7	16	19	16	30	23	34	57	54	125	96	149	194	103

We show the difference in the assembly instruction count of the resulting binary in Fig. 16.2. For some apps like LULESH [52], the annotations optimize the code in a way that leads to fewer instructions. For other apps like SimpleMOC [62], the optimizations results in more instructions, due to code duplication performed by the compiler when branches are eliminated, which was not possible without the annotations.

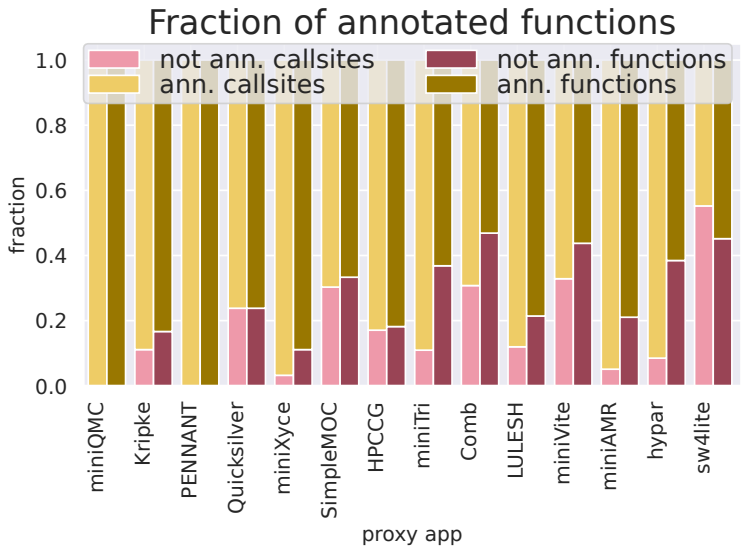


Figure 16.1.: Fraction of MPI Functions and callsites annotated. The left bar shows the fraction of the annotated functions. The right bar shows the fraction of annotated callsites. Table 16.1 shows the absolute values.

We detail the observable optimizations using the examples of LULESH [52] and SimpleMOC [62] in the following section. The other MiniApps trigger similar compiler optimizations.

16.3.2. Assembly-level differences

We compare the code generated without annotation from our pass with the assembly code generated including our pass. We use `clang` version 17.0.6 for the x86-64 architecture on a machine equipped with Intel Xeon Platinum 9242 CPUs running Red Hat 8.5.0-18 with openMPI version 4.1.6.

To be able to see additional optimizations we compile the programs using the respective default optimization flags, i.e. `O2` or `O3`, depending on the program. The memory access related optimizations discussed are part of the optimization level `O2`.

We note that the assertions lead to the compiler using different registers for computations, as well as different memory locations for variables on the stack. In the following

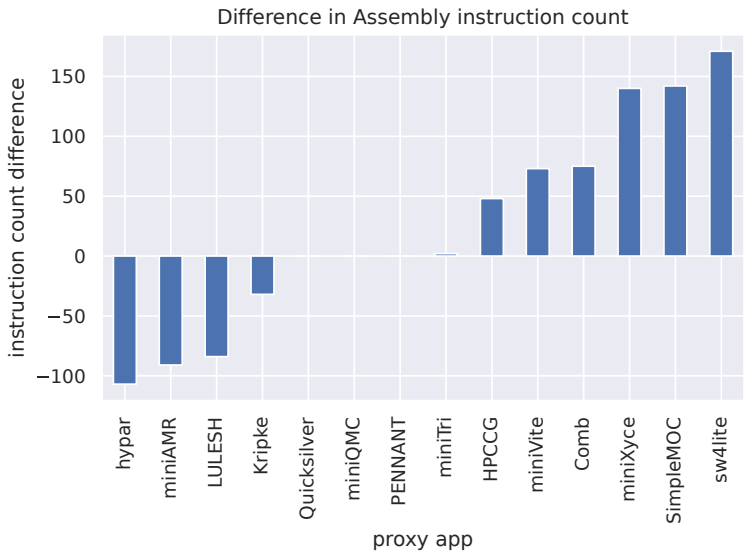


Figure 16.2.: Differences in assembly count for ECP proxy apps.

observations however, we will focus on the more pronounced differences regarding the number of generated instructions and the resulting control flow.

Example: LULESH

The left side of Listing 16.1 shows an excerpt of the original assembly generated, while the right side of Listing 16.1 shows the resulting assembly for the same source code location, when our annotation pass was enabled.

When comparing the assembly code in Listing 16.1, it is possible to observe the additional optimizations that are enabled by annotating attributes to the functions `MPI_Comm_size` and `MPI_Comm_rank`. In Line 9 in Listing 16.1, the generated assembly code loads the size of the communicator into a register. The optimized code (right in Listing 16.1) uses a different register, in order to reuse the communicator size from the register, which can be seen in Line 24 of Listing 16.1. In Line 13 of Listing 16.1, the previously initialized communicator rank is loaded from memory. As above, the un-optimized version (left in Listing 16.1) needs to re-load the same value from memory in Line 19.

```
mov     rsi, rsp
mov     rdi, rbx
call    MPI_Comm_size@PLT
lea     rsi, [rsp + 4]
mov     rdi, rbx
call    MPI_Comm_rank@PLT
; ...
; Loads the value of size:
mov     eax, dword ptr [rsp]
add     eax, 10
; ...
; Loads the value of rank:
mov     edx, dword ptr [rsp + 4]
lea     rcx, [rsp + 20]


call    ;Mangled function name
; rank is reloaded:
mov     esi, dword ptr [rsp + 4]
mov     eax, dword ptr [rsp + 40]
or      eax, esi
; ...
; size reloaded from memory:
mov     esi, dword ptr [rsp]
```

```
lea     rsi, [rsp + 76]
mov     rdi, rbx
call    MPI_Comm_size@PLT
lea     rsi, [rsp + 72]
mov     rdi, rbx
call    MPI_Comm_rank@PLT
; ...
; Uses separate register to load size:
movsxd  rbx, dword ptr [rsp + 76]
lea     eax, [rbx + 10]
; ...
; Uses separate register to load rank:
mov     ebp, dword ptr [rsp + 72]
lea     rcx, [rsp + 12]
; move between registers needed:
mov     edx, ebp
call    ;Mangled function name

; Reuse of the loaded rank
mov     eax, dword ptr [rsp + 32]
or      eax, ebp
; ...
; Reuse of the loaded size
mov     esi, ebx
```

Listing 16.1: Comparison of assembly code generated from `lulesh.cc` with and without attributes. The left side shows the assembly generated without attributes. The right side shows the assembly generated with our pass enabled, which leads to the removal of unnecessary memory accesses, as indicated with the comments.

As both of these values are reused again several times during the course of the entire program, our annotations result in 12 memory accesses being saved by this optimization.

The reason the compiler is able to perform these optimizations is the usage of the `nocapture` attribute, annotated to the pointer parameter of both `MPI_Comm_size` and `MPI_Comm_rank`. This attribute signals the compiler that the values of the `size` and `rank` variables stay the same. Without this attribute, the compiler needs to assume that the MPI library may change the variables at any time, e.g. using a background thread and therefore needs to fetch the current value (again) before each usage.

Another exemplary difference is found in the produced assembly code generated from the code in Listing 16.2. Again, the compiler is able to perform additional optimizations because of the `nocapture` attribute annotated to the pointer parameter of `MPI_Comm_rank`. Despite the same attribute, the optimization is of a different kind. Essentially, the compiler is able to optimize the usage of variable `i` declared in Line 6 of Listing 16.2. As the compiler now knows that the value of `myRank` does not change, it can perform an index

```
1  int myRank;
2  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
3  // ...
4  while (regionNum == lastReg) {
5    regionVar = rand() % costDenominator;
6    i = 0; // loop index can be transformed to start at myRank
7    while (regionVar >= regBinEnd[i])
8      i++;
9    // a transformed index will save an addition in this line:
10   regionNum = ((i + myRank) % numReg()) + 1;
11 }
```

Listing 16.2: Excerpt of `lulesh-init.cc` from LULESH [52]

```
1  int myRank;
2  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
3  // ...
4  while (regionNum == lastReg) {
5    regionVar = rand() % costDenominator;
6    array_addr = &regBinEnd[0];
7    tmp =  myRank - 1; // keep in register for outer while loop
8    do { tmp++; // increment is cheaper than reading from memory
9    } while (regionVar >= *array_addr++);// no implicit add in operator[]
10   regionNum = (tmp % numReg()) + 1; // at this point tmp equals myRank + i
11 }
```

Listing 16.3: Illustration of the idea for the optimization in Listing 16.2

transformation on the inner loop by basically initializing `i` with `myRank - 1`. This will save the addition of `i + myRank` after the loop (Line 10 of Listing 16.2). This implies that the compiler keeps a copy of `myRank - 1` on the stack, to use in the next iteration of the outer loop. Listing 16.3 shows a source code representation of the optimization done by the compiler. Due to other optimizations this does not cause an additional addition in the inner loop even though Listing 16.3 looks like it. This fact is only observable when looking at the generated assembly directly.

Interestingly, the load instruction for initialization of `i` uses one byte less than the immediate-using instruction it replaces. Therefore, the compiler does not generate a subsequent padding in the machine code, which makes the code smaller and omits a following `nop` instruction.

Looking at a different part of LULESH, it is possible to see another effect of our annotated

attributes. Listing 16.4 is an excerpt from a synchronization function that gets called for every simulated time step.

```
1  if (planeMax) {
2    srcAddr = &domain.commDataRecv[pmsg * maxPlaneComm];
3    MPI_Wait(&domain.recvRequest[pmsg], &status);
4    for (Index_t fi = 0 ; fi < xferFields; ++fi) {
5      Domain_member dest = fieldData[fi];
6      for (Index_t i = 0; i < opCount; ++i) {
7        (domain.*dest)(dx*dy*(dz - 1) + i) = srcAddr[i];
8      }
9      srcAddr += opCount;
10   }
11   ++pmsg;
12 }
```

Listing 16.4: Excerpt of `lulesh-comm.cc` from LULESH [52]

In Listing 16.4, the variable `fieldData` in Line 5 is an array of function pointers that is initialized earlier and effectively constant, as it is never written thereafter. Since the variable `xferFields` has the constant value of `6`, the compiler is able to unroll the outer loop beginning in Line 4. This loop unrolling happens with and without our additional annotations. Ideally, the compiler should be able to statically resolve the function pointers and call the pointed-to functions in Line 7 directly. Compiling this program reveals that the compiler is not able to do this. The reason for this is that the compiler has no information on the MPI functions called beforehand, namely `MPI_Wait` in Line 3 and `MPI_Comm_rank` in a line not shown. Therefore, it has to assume the worst case possible, which in these circumstances means that these functions modify the array contents to now point to different functions. Using our annotation pass, `MPI_Wait` and `MPI_Comm_rank` both get annotated with the `memory(argmem: readwrite, inaccessiblemem: readwrite)` attribute. By including this annotation, the compiler is able to determine that the array contents are indeed not modified and can statically resolve the function call. This only works in the first iteration however, as the called function itself is not marked with such attributes and therefore prevents further usage of this optimization. Since this is part of the internals of the application, additional attributes to the present MPI functions are not able to resolve this issue.

Nevertheless, even the single statically resolved function call causes major observable differences in the generated assembly. Since the function pointers point to member functions of a class, the compiler generates additional instructions to deal with the possibility of virtual function calls. The general idea is, that the compiler first need to

check, if the pointer loaded from the array is a direct function pointer or a pointer to the virtual table. This is not necessary if the function pointer can be resolved directly. For the single resolved function pointer, the compiler is able to save 22 instructions in total. If the calls in the other iterations could be resolved as well, the same would apply to them.

Additionally, using the annotated attributes allows the compiler to keep other variables in registers, removing the need to reload values. In total there are another 5 memory accesses less during the execution of the rest of this unrolled loop.

Unfortunately, we were not able to measure a significant performance benefit of resolving the call to a direct one in the first loop iteration. As the application prevents resolving the calls in the other five iterations of the loop, the performance benefit is limited in this case.

Nevertheless, this shows that valuable optimizations can be triggered by the addition of relevant attributes to MPI functions. We see that the addition of attributes to MPI functions enables the compiler to statically resolve a function call to a direct one, which has the potential to enable further even more valuable optimizations, although they are not triggered in the case of LULESH. As our pass uses the attributes that are already part of the LLVM Infrastructure, it does not add any significant complexity to the toolchain. Considering that compiler optimizations are a difficult endeavor, huge performance improvements, without extra toolchain complexity are very difficult to realize. Therefore, we propose to use our pass to not let the optimization opportunities enabled by our pass unused, no matter how small they might be.

Example: SimpleMOC

Another set of possible optimizations results in more instructions being generated. An example for this can be found when looking at the code in Listing 16.5, where several function calls to print routines are only made by the process with rank 0.

Due to the compiler having the additional information that the rank variable is not captured, it is able to eliminate branches. This is done by duplicating the code that is executed by all processes. In one of these duplicated regions the code specific for the process with rank 0 is included, whereas the other duplicated region does not have this code. In total, this omits three branches for processes with a rank different from 0, and several tests for the value of rank in the other region. The reason that this is not possible when compiling without our pass, is that the compiler has to assume that

```c
int mype = 0;
// ...
MPI_Comm_rank(MPI_COMM_WORLD, &mype);
// ...
if( mype == 0 )
  logo(version);
// ...
if( mype == 0 )
  print_input_summary(input);
// ...
for( int i = 0; i < num_iters; i++) {
  // Loop code omitted
  // Mainly this part gets duplicated
  if( mype == 0 )
    printf("keff = %f\n", keff);
  }
// ...
if( mype == 0 ) {
  border_print();
  center_print("RESULTS SUMMARY", 79);
  border_print();
  // ...
}
```

Listing 16.5: Excerpt of `main.c` from SimpleMOC [62]

between different tests for the value of rank the value has been changed by a different thread or context, making multiple tests necessary. Knowing that the rank variable is not captured allows the compiler to omit these unnecessary tests. Separating both cases additionally allows the compiler to use 32 bytes less stack space as it is able to keep more values in registers.

Even though this optimization increases the size of the produced object file, fewer instructions are actually executed during runtime. As the code is only executed once in the initial setup of the application, the performance benefit is only minimal. However, this shows that such optimizations are possible, with the potential of a bigger impact when employing our pass on other programs.

Even though the performance improvement of this optimization is only minimal for the MiniApp, we can showcase that this kind of optimizations can lead to a substantial performance improvement by using an example program, designed to extract the effect of this optimization. The simple example[2] we used is shown in Listing 16.6. With this

[2]The example is also part of our GitHub repository: https://github.com/AdrSchm/mpi-attribute

example, we are able to show a performance improvement of more than 16% (of 2.7 seconds) by annotating the MPI functions, as the information of the annotations allows the compiler to hoist the rank conditional from Line 7 of Listing 16.6 out of the loop.

```
1  int rank;
2  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
3  starttime = MPI_Wtime();
4  for (int n = 0; n < num_iters; ++n) {
5    for (int i = 0; i < ARRAY_SIZE; ++i) {
6      array[i] = array[i] + (n * 42.1337); } // dummy calculation
7    if (n % ITER_TO_PRINT == 0 && rank == 0) {
8      printf("Iter %d : %f\n", n, array[0]); } // rank 0 prints progress
9    }
10 endtime = MPI_Wtime();
11 // Use MPI_Reduce to find the slowest process and report this timing
```

Listing 16.6: Example to showcase performance benefit

16.4. Conclusion

In this work we demonstrated that the compiler is able to further optimize applications using MPI, if it is provided additional information about the MPI functions. We supplied the compiler with this information via an LLVM compiler pass that annotates the used MPI functions with relevant attributes at the LLVM IR level. The attributes have been manually derived according to the MPI standards semantics. Thus, our approach is compatible with *all* standard-compliant MPI implementations.

We showed that the addition of attributes enables many different kinds of compiler optimizations, including the removal of unnecessary memory accesses and the simplification or elimination of branches.

The performance impact on the considered MiniApps of the Exascale Proxy Applications Project [141] is quite limited, as the optimizations mostly target the ¨management code¨ and not the core compute intensive part. As the runtime performance can only improve, and, depending on the interplay of MPI calls and the rest of the code, possibly in a substantial fashion, we advocate the use of these compiler attributes. Our approach does not add complexity to the LLVM infrastructure, as we employ well-established LLVM attributes.

`s-pass`

Considering that the MPI Forum is currently discussing a standardization of an ABI for MPI, we think that this work can inform ABI discussion by showcasing how compilers can leverage standardized information for code optimization.

In the future, we plan to investigate how the use of MPI wrapper functions influences the effects of added attributes. We also want to explore how the effects of the attributes are proliferated to other compiler environments. In particular, we want to explore how our findings can be transferred to Fortran applications, as Fortran can also be compiled with the LLVM toolchain. Additionally, we plan to extend our approach to also cover less frequently used MPI functions such as one sided operations.

The code of our annotation Pass is available on GitHub: https://github.com/AdrSchm/mpi-attributes-pass.

Acknowledgements: This work was supported by the Hessian Ministry for Higher Education, Research and the Arts through the Hessian Competence Center for High-Performance Computing and by the Federal Ministry of Education and Research (BMWF) and the states of Hesse as part of the NHR program. Measurements for this work were conducted on the Lichtenberg high performance computer of the TU Darmstadt.

Bibliography

[1] A. Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 (cit. on pp. 14, 83).

[2] Edmund M. Clarke and E. Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Logics of Programs*. Ed. by Dexter Kozen. Springer, 1982, pp. 52–71. ISBN: 978-3-540-39047-3. DOI: 10.1007/BFb0025774 (cit. on pp. 14, 83).

[3] Mark Weiser. "Program Slicing". In: *IEEE Transactions on Software Engineering* SE-10.4 (July 1984), pp. 352–357. ISSN: 0098-5589. DOI: 10.1109/tse.1984.5010248 (cit. on p. 215).

[4] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. https://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps. Accessed: 2024-08-09. May 5, 1994 (cit. on p. 2).

[5] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 1.1*. https://www.mpi-forum.org/docs/mpi-1.1/mpi-11.ps. Accessed: 2024-08-09. June 12, 1995 (cit. on p. 85).

[6] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. "MPI-2: Extending the message-passing interface". In: *Euro-Par'96 Parallel Processing*. Ed. by Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert. Springer, 1996, pp. 128–135. ISBN: 978-3-540-70633-5. DOI: 10.1007/3-540-61626-8_16 (cit. on p. 85).

[7] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997. ISBN: 978-0-201-63392-4 (cit. on p. 117).

[8] G. J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521 (cit. on p. 88).

[9] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. https://www.mpi-forum.org/docs/mpi-2.0/mpi-20.ps. Accessed: 2024-08-09. July 18, 1997 (cit. on p. 2).

[10] L. Dagum and R. Menon. "OpenMP: An Industry Standard API for Shared-Memory Programming". In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313 (cit. on pp. 54, 117).

[11] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. "Escape analysis for Java". In: *Acm Sigplan Notices* 34.10 (1999), pp. 1–19. DOI: 10.1145/320385.320386 (cit. on pp. 44, 229).

[12] Rolf Rabenseifner. "Automatic Profiling of MPI Applications with Hardware Performance Counters". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Jack Dongarra, Emilio Luque, and Tomàs Margalef. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 35–42. ISBN: 978-3-540-48158-4. DOI: 10.1007/3-540-48158-3_5 (cit. on p. 5).

[13] J.S. Vetter and B.R. de Supinski. "Dynamic Software Testing of MPI Applications with Umpire". In: *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. 2000, pp. 51–51. DOI: 10.1109/SC.2000.10055 (cit. on pp. 89, 91, 115, 141).

[14] William Gropp. "MPICH2: A New Start for MPI Implementations". In: *9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2002, p. 7. ISBN: 3540442960. DOI: 10.1007/3-540-45825-5_5 (cit. on p. 99).

[15] William Gropp. "MPICH2: A new start for MPI implementations". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 9th European PVM/MPI Users' Group Meeting*. Springer. 2002, pp. 7–7. DOI: 10.1007/3-540-45825-5_5 (cit. on p. 145).

[16] Glenn R. Luecke, Yan Zou, James Coyle, Jim Hoekstra, and Marina Kraeva. "Deadlock detection in MPI programs". In: *Concurrency and Computation: Practice and Experience* 14.11 (2002), pp. 911–932. DOI: 10.1002/cpe.701 (cit. on p. 14).

[17] Glenn Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. "MPI-CHECK: a tool for checking Fortran 90 MPI programs". In: *Concurrency and Computation: Practice and Experience* 15.2 (2003), pp. 93–100. DOI: 10.1002/cpe.705 (cit. on pp. 89, 172).

[18] Rob F vanderWijngaart and Jin Haopiang. “NAS parallel benchmarks, multi-zone versions”. In: (2003). Accessed: 2024-08-13. URL: https://www.nas.nasa.gov/publications/npb.html (cit. on p. 91).

[19] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *11th European PVM/MPI Users’ Group Meeting*. Sept. 2004, pp. 97–104. DOI: 10.1007/978-3-540-30218-6_19 (cit. on p. 82).

[20] Stephen F. Siegel and George S. Avrunin. “Verification of MPI-Based Software for Scientific Computation”. In: *Model Checking Software*. Ed. by Susanne Graf and Laurent Mounier. Springer, 2004, pp. 286–303. ISBN: 978-3-540-24732-6. DOI: 10.1007/978-3-540-24732-6_20 (cit. on p. 88).

[21] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. “Automated, Scalable Debugging of MPI Programs with Intel™ Message Checker”. In: *2nd Intl. Workshop on Software Engineering for High Performance Computing System Applications*. SE-HPCS ’05. St. Louis, Missouri: ACM, 2005, pp. 78–82. ISBN: 1595931171. DOI: 10.1145/1145319.1145342 (cit. on p. 83).

[22] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. “Pin: Customized Program Analysis Tools with Dynamic Instrumentation”. In: *ACM SIGPLAN Notices* 40.6 (2005), pp. 190–200. DOI: 10.1145/1064978.1065034 (cit. on p. 122).

[23] Stephen F. Siegel and George S. Avrunin. “Modeling Wildcard-Free MPI Programs for Verification”. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’05. Chicago, IL, USA: ACM, 2005, pp. 95–106. ISBN: 1595930809. DOI: 10.1145/1065944.1065957 (cit. on p. 88).

[24] Michael Süß and Claudia Leopold. “Common mistakes in OpenMP and how to avoid them: A collection of best practices”. In: *International Workshop on OpenMP*. Springer. 2005, pp. 312–323. DOI: 10.1007/978-3-540-68555-5_26 (cit. on pp. 19, 115, 120).

[25] Keith D Underwood, K Scott Hemmert, Arun Rodrigues, Richard Murphy, and Ron Brightwell. "A hardware acceleration unit for MPI queue processing". In: *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2005, 10–pp. DOI: 10.1109/IPDPS.2005.30 (cit. on pp. 190, 213).

[26] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. "RDMA read based rendezvous protocol for MPI over InfiniBand". In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '06*. ACM Press, 2006. DOI: 10.1145/1122971.1122978 (cit. on p. 196).

[27] Anthony Danalis, Lori Pollock, and Martin Swany. "Automatic MPI application transformation with ASPhALT". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370486 (cit. on pp. 23, 182).

[28] William Gropp and Rajeev Thakur. "Thread-safety in an MPI implementation: Requirements and analysis". In: *Parallel Computing* 33.9 (2007). Selected Papers from EuroPVM/MPI 2006, pp. 595–604. DOI: 10.1016/j.parco.2007.07.002 (cit. on p. 117).

[29] K. Scott Hemmert, Keith D. Underwood, and Arun Rodrigues. "An architecture to perform NIC based MPI matching". In: *2007 IEEE International Conference on Cluster Computing*. IEEE, 2007. DOI: 10.1109/clustr.2007.4629234 (cit. on p. 213).

[30] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100. DOI: 10.1145/1273442.1250746 (cit. on p. 122).

[31] Stephen F. Siegel. "Model Checking Nonblocking MPI Programs". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Byron Cook and Andreas Podelski. Springer, 2007, pp. 44–58. ISBN: 978-3-540-69738-1. DOI: 10.1007/978-3-540-69738-1_3 (cit. on p. 88).

[32] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224. URL: http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf (cit. on p. 89).

[33] Torsten Hoefler and Andrew Lumsdaine. “Message progression in parallel computing - to thread or not to thread?” In: *2008 IEEE International Conference on Cluster Computing*. IEEE, Sept. 2008. DOI: 10.1109/clustr.2008.4663774 (cit. on p. 196).

[34] Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos. “MPI-aware compiler optimizations for improving communication-computation overlap”. In: *Proceedings of the 23rd international conference on Supercomputing*. 2009, pp. 316–325. DOI: 10.1145/1542275.1542321 (cit. on pp. 23, 172, 182, 190).

[35] Tobias Hilbrich, Bronis R de Supinski, Martin Schulz, and Matthias S Müller. “A graph based approach for MPI deadlock detection”. In: *Proceedings of the 23rd international conference on Supercomputing*. 2009, pp. 296–305. DOI: 10.1145/1542275.1542319 (cit. on p. 14).

[36] Glenn Luecke, James Coyle, James Hoekstra, Marina Kraeva, Ying Xu, Mi-Young Park, Elizabeth Kleiman, Olga Weiss, André Wehe, and Melissa Yahya. “The Importance of Run-Time Error Detection”. In: Jan. 2009, pp. 145–155. DOI: 10.1007/978-3-642-11261-4_10 (cit. on p. 166).

[37] Matthias S. Müller, Matthijs van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C. Brantley, Chris Parrott, Tom Elken, Huiyu Feng, and Carl Ponder. “SPEC MPI2007 — an application benchmark suite for parallel systems using MPI”. In: *Concurrency and Computation: Practice and Experience* 22.2 (2009), pp. 191–205. DOI: 10.1002/cpe.1535 (cit. on p. 91).

[38] R. Rabenseifner, G. Hager, and G. Jost. “Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes”. In: *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. 2009, pp. 427–436. DOI: 10.1109/PDP.2009.43 (cit. on p. 117).

[39] Mohammad J. Rashti and Ahmad Afsahi. “A Speculative and Adaptive MPI Rendezvous Protocol Over RDMA-enabled Interconnects”. In: *International Journal of Parallel Programming* 37.2 (Mar. 2009), pp. 223–246. DOI: 10.1007/s10766-009-0094-9 (cit. on p. 200).

[40] Ahmed Bukhamsin, Mohamad Sindi, and Jallal Al-Jallal. “Using the Intel MPI Benchmarks (IMB) to Evaluate MPI Implementations on an Infiniband Nehalem Linux Cluster”. In: *Proceedings of the 2010 Spring Simulation Multiconference*. SpringSim ’10. Orlando, Florida: Society for Computer Simulation International, 2010. ISBN: 9781450300698. DOI: 10.1145/1878537.1878787 (cit. on p. 83).

[41] Kevin Dowd and Charles Severance. *High Performance Computing*. https://archive.org/details/cnx-org-col11136. Accessed: 2024-08-13. 2010 (cit. on p. 178).

[42] Tobias Hilbrich, Martin Schulz, Bronis R de Supinski, and Matthias S Müller. "MUST: A scalable approach to runtime error detection in MPI programs". In: *Tools for high performance computing 2009*. Springer, 2010, pp. 53–66. DOI: 10.1007/978-3-642-11261-4_5 (cit. on pp. 11, 55, 76, 83, 89, 115, 117, 141, 150, 151, 161, 171).

[43] Torsten Hoefler, Greg Bronevetsky, Brian Barrett, Bronis R. de Supinski, and Andrew Lumsdaine. "Efficient MPI Support for Advanced Hybrid Programming Models". In: *Recent Advances in the Message Passing Interface*. Ed. by Rainer Keller, Edgar Gabriel, Michael Resch, and Jack Dongarra. Berlin, Heidelberg: Springer, 2010, pp. 50–61. DOI: 10.1007/978-3-642-15646-5_6 (cit. on p. 127).

[44] Matthew Small, Zheng Gu, and Xin Yuan. "Near-Optimal Rendezvous Protocols for RDMA-Enabled Clusters". In: *2010 39th International Conference on Parallel Processing*. IEEE, Sept. 2010. DOI: 10.1109/icpp.2010.72 (cit. on pp. 200, 201).

[45] William Gropp, Torsten Hoefler, Rajeev Thakur, and Jesper Larsson Träff. "Performance expectations and guidelines for MPI derived datatypes". In: *Recent Advances in the Message Passing Interface: 18th European MPI Users' Group Meeting, EuroMPI 2011. Proceedings 18*. Springer. 2011, pp. 150–159. DOI: 10.1007/978-3-642-24449-0_18 (cit. on p. 55).

[46] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. "Dynamic race detection with LLVM compiler: Compile-time instrumentation for ThreadSanitizer". In: *International Conference on Runtime Verification*. Springer. 2011, pp. 110–114. DOI: 10.1007/978-3-642-29860-8_9 (cit. on p. 122).

[47] Jeff A. Stuart, Pavan Balaji, and John D. Owens. "Extending MPI to Accelerators". In: *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*. ASBD '11. Galveston Island, Texas, USA: ACM, 2011, pp. 19–23. DOI: 10.1145/2377978.2377981 (cit. on p. 117).

[48] Francesca Arcelli Fontana, Andrea Caracciolo, and Marco Zanoni. "DPB: A benchmark for design pattern detection tools". In: *16th European Conference on Software Maintenance and Reengineering*. IEEE. 2012, pp. 235–244. DOI: 10.1109/CSMR.2012.32 (cit. on pp. 14, 83).

[49] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. Accessed: 2024-08-09. Sept. 21, 2012 (cit. on p. 2).

[50] Masayuki Hatanaka, Atsushi Hori, and Yutaka Ishikawa. "Optimization of MPI persistent communication". In: *Proceedings of the 20th European MPI Users' Group Meeting*. 2013, pp. 79–84. DOI: 10.1145/2488551.2488566 (cit. on p. 209).

[51] Tobias Hilbrich, Bronis R. de Supinski, Wolfgang E. Nagel, Joachim Protze, Christel Baier, and Matthias S. Müller. "Distributed wait state tracking for runtime MPI deadlock detection". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503237 (cit. on p. 14).

[52] Ian Karlin, Jeff Keasler, and Rob Neely. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. Accessed: 2024-08-13. Livermore, CA, Aug. 2013, pp. 1–9. URL: https://www.osti.gov/servlets/purl/1090032 (cit. on pp. 45, 98, 232, 233, 236, 237).

[53] Timo Schneider, Fredrik Kjolstad, and Torsten Hoefler. "MPI datatype processing using runtime compilation". In: *Proceedings of the 20th European MPI Users' Group Meeting*. 2013, pp. 19–24. DOI: 10.1145/2488551.2488552 (cit. on p. 32).

[54] Masamichi Takagi, Yuichi Nakamura, Atsushi Hori, Balazs Gerofi, and Yutaka Ishikawa. "Revisiting rendezvous protocols in the context of RDMA-capable host channel adapters and many-core processors". In: *Proceedings of the 20th European MPI Users' Group Meeting on - EuroMPI '13*. ACM Press, 2013. DOI: 10.1145/2488551.2488571 (cit. on p. 209).

[55] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms". In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917. DOI: 10.1016/j.jpdc.2014.06.008 (cit. on p. 89).

[56] Jan Felix Münchhalfen, Tobias Hilbrich, Joachim Protze, Christian Terboven, and Matthias S. Müller. "Classification of Common Errors in OpenMP Applications". In: *Using and Improving OpenMP for Devices, Tasks, and More*. Ed. by Luiz DeRose, Bronis R. de Supinski, Stephen L. Olivier, Barbara M. Chapman, and Matthias S. Müller. Cham: Springer International Publishing, 2014, pp. 58–72. DOI: 10.1007/978-3-319-11454-5_5 (cit. on pp. 19, 115, 120).

[57] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. "PARCOACH: Combining static and dynamic validation of MPI collective communications". In: *The International Journal of High Performance Computing Applications* 28.4 (2014), pp. 425–434. DOI: 10.1177/1094342014552204 (cit. on pp. 11, 55, 83, 87, 99, 115, 141, 150, 151, 161, 172).

[58] Jeffrey Svajlenko and Chanchal K Roy. "Evaluating modern clone detection tools". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 321–330. DOI: 10.1109/ICSME.2014.54 (cit. on pp. 14, 83).

[59] Alexander Droste, Michael Kuhn, and Thomas Ludwig. "MPI-Checker: Static Analysis for MPI". In: *2nd Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–10. DOI: 10.1145/2833157.2833159 (cit. on pp. 11, 55, 83, 87, 91, 115, 141, 150, 172).

[60] X. Fu, Z. Chen, Y. Zhang, C. Huang, W. Dong, and J. Wang. "MPISE: Symbolic Execution of MPI Programs". In: *IEEE 16th International Symposium on High Assurance Systems Engineering*. 2015, pp. 181–188. DOI: 10.1109/HASE.2015.35 (cit. on p. 89).

[61] Ryan Grant, Anthony Skjellum, and Purushotham V Bangalore. *Lightweight threading with MPI using Persistent Communications Semantics*. Tech. rep. Accessed: 2024-08-13. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2015. URL: https://www.osti.gov/servlets/purl/1328651 (cit. on pp. 30, 182).

[62] Geoffrey Gunow, John Tramm, Benoit Forget, Kord Smith, and Tim He. "SimpleMOC – A PERFORMANCE ABSTRACTION FOR 3D MOC". In: *ANS & M&C 2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*. Accessed: 2024-08-13. 2015. URL: https://dspace.mit.edu/handle/1721.1/110238 (cit. on pp. 47, 232, 233, 239).

[63] Tobias Hilbrich, Bronis R de Supinski, Andreas Knüpfer, Robert Dietrich, Christian Terboven, Felix Münchhalfen, and Wolfgang E Nagel. *A Deadlock Detection Concept for OpenMP Tasks and Fully Hybrid MPI-OpenMP Applications*. http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/poster_files/post149s2-file3.pdf. Accessed: 2024-08-13. 2015 (cit. on p. 127).

[64] Julien Jaeger, Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. "Correctness analysis of MPI-3 Non-blocking communications in PARCOACH". In: *Proceedings of the 22nd European MPI Users' Group Meeting*. 2015, pp. 1–2. DOI: 10.1145/2802658.2802674 (cit. on p. 172).

[65] Adam J Kunen, Teresa S Bailey, and Peter N Brown. *KRIPKE-a massively parallel transport mini-app*. Tech. rep. Accessed: 2024-08-13. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2015. URL: https://www.osti.gov/servlets/purl/1229802 (cit. on p. 98).

[66] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.1*. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. Accessed: 2024-08-09. June 4, 2015 (cit. on pp. 82, 85, 86, 95, 171, 174, 175).

[67] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers. "CIVL: the concurrency intermediate verification language". In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12. DOI: 10.1145/2807591.2807635 (cit. on p. 89).

[68] Karthikeyan Vaidyanathan, Dhiraj D. Kalamkar, Kiran Pamnany, Jeff R. Hammond, Pavan Balaji, Dipankar Das, Jongsoo Park, and Bálint Joó. "Improving concurrency and asynchrony in multithreaded MPI applications using software offloading". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Nov. 2015. DOI: 10.1145/2807591.2807602 (cit. on p. 196).

[69] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H Ahn, Ignacio Laguna, Martin Schulz, Gregory L Lee, Joachim Protze, and Matthias S Müller. "ARCHER: Effectively Spotting Data Races in Large OpenMP Applications". In: *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE. 2016, pp. 53–62. DOI: 10.1109/IPDPS.2016.68 (cit. on p. 123).

[70] Tim Cramer, Felix Münchhalfen, Christian Terboven, Tobias Hilbrich, and Matthias S Müller. "Extending MUST to Check Hybrid-Parallel Programs for Correctness Using the OpenMP Tools Interface". In: *Tools for High Performance Computing 2015*. Springer, 2016, pp. 85–101. DOI: 10.1007/978-3-319-39589-0_7 (cit. on pp. 19, 120).

[71] Hoang-Vu Dang, Marc Snir, and William Gropp. "Towards millions of communicating threads". In: *Proceedings of the 23rd European MPI Users' Group Meeting*. 2016, pp. 1–14. DOI: 10.1145/2966884.2966914 (cit. on pp. 24, 26, 171).

[72] Mario Flajslik, James Dinan, and Keith D Underwood. “Mitigating MPI message matching misery”. In: *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*. Springer. 2016, pp. 281–299. DOI: 10.1007/978-3-319-41321-1_15 (cit. on pp. 35, 201).

[73] Mario Flajslik, James Dinan, and Keith D. Underwood. “Mitigating MPI Message Matching Misery”. In: *High Performance Computing*. Springer International Publishing, 2016, pp. 281–299. ISBN: 9783319413211. DOI: 10.1007/978-3-319-41321-1_15 (cit. on p. 213).

[74] Jichi Guo, Qing Yi, Jiayuan Meng, Junchao Zhang, and Pavan Balaji. “Compiler-assisted overlapping of communication and computation in MPI applications”. In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2016, pp. 60–69. DOI: 10.1109/CLUSTER.2016.62 (cit. on pp. 23, 182, 190).

[75] Roger Kowalewski and Karl Fürlinger. “Nasty-MPI: Debugging synchronization errors in MPI-3 one-sided applications”. In: *European Conference on Parallel Processing*. Springer. 2016, pp. 51–62. DOI: 10.1007/978-3-319-43659-3_4 (cit. on p. 90).

[76] Emmanuelle Saillard, Hugo Brunie, Patrick Carribault, and Denis Barthou. “PARCOACH Extension for Hybrid Applications with Interprocedural Analysis”. In: *Tools for High Performance Computing 2015*. Ed. by Andreas Knüpfer, Tobias Hilbrich, Christoph Niethammer, José Gracia, Wolfgang E. Nagel, and Michael M. Resch. Springer, 2016, pp. 135–146. ISBN: 978-3-319-39589-0. DOI: 10.1007/978-3-319-39589-0_11 (cit. on pp. 11, 87).

[77] Jesper Larsson Träff. “A Library for Advanced Datatype Programming”. In: *EuroMPI*. ACM, 2016, pp. 98–107. DOI: 10.1145/2966884.2966904 (cit. on p. 122).

[78] Hadia Ahmed, Anthony Skjellumh, Purushotham Bangalore, and Peter Pirkelbauer. “Transforming blocking MPI collectives to non-blocking and persistent operations”. In: *Proceedings of the 24th European MPI Users’ Group Meeting*. 2017, pp. 1–11. DOI: 10.1145/3127024.3127033 (cit. on pp. 23, 29, 182, 183).

[79] Vojtundefinedch Forejt, Saurabh Joshi, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. “Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs”. In: *ACM Trans. Program. Lang. Syst.* 39.4 (Aug. 2017). ISSN: 0164-0925. DOI: 10.1145/3095075 (cit. on pp. 14, 89).

[80] Vojtundefinedch Forejt, Saurabh Joshi, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. "Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs". In: *ACM Trans. Program. Lang. Syst.* 39.4 (Aug. 2017). ISSN: 0164-0925. DOI: 10.1145/3095075 (cit. on p. 14).

[81] GASPI Forum. *GASPI: Global Address Space Programming Interface 17.1*. Accessed: 2024-05-28. 2017. URL: https://raw.githubusercontent.com/GASPI-Forum/GASPI-Forum.github.io/master/standards/GASPI-17.1.pdf (cit. on pp. 14, 153).

[82] Teresa Johnson, Mehdi Amini, and Xinliang David Li. "ThinLTO: scalable and incremental LTO". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 111–121. DOI: 10.1109/CGO.2017.7863733 (cit. on pp. 43, 227).

[83] Benjamin Klenk, Holger Froening, Hans Eberle, and Larry Dennison. "Relaxations for High-Performance Message Passing on Massively Parallel SIMT Processors". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2017. DOI: 10.1109/ipdps.2017.94 (cit. on p. 213).

[84] Benjamin Klenk and Holger Fröning. "An overview of MPI characteristics of exascale proxy applications". In: *High Performance Computing: 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017, Proceedings 32*. Springer. 2017, pp. 217–236. DOI: 10.1007/978-3-319-58667-0_12 (cit. on pp. 5, 54).

[85] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. "DataRaceBench: A Benchmark Suite for Systematic Evaluation of Data Race Detection Tools". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: ACM, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126958 (cit. on pp. 14, 83, 115, 166).

[86] LLNL. *AMG2013 Mini-App*. https://computing.llnl.gov/projects/co-design/amg2013. Accessed: 2024-08-13. 2017 (cit. on p. 98).

[87] Ziqing Luo, Manchun Zheng, and Stephen F Siegel. "Verification of MPI programs using CIVL". In: *24th European MPI Users' Group Meeting*. 2017, pp. 1–11. DOI: 10.1145/3127024.3127032 (cit. on p. 89).

[88] Anh Pham, Thierry Jéron, and Martin Quinson. "Verifying MPI applications with SimGridMC". In: *1st Intl. Workshop on Software Correctness for HPC Applications*. 2017, pp. 28–33. DOI: 10.1145/3145344.3145345 (cit. on p. 89).

[89] Joachim Protze, Jonas Hahnfeld, Dong H. Ahn, Martin Schulz, and Matthias S. Müller. "OpenMP Tools Interface: Synchronization Information for Data Race Detection". In: *IWOMP*. Vol. 10468. Lecture Notes in Computer Science. Springer, 2017, pp. 249–265. DOI: 10.1007/978-3-319-65578-9_17 (cit. on p. 123).

[90] Min Si and Pavan Balaji. "Process-Based Asynchronous Progress Model for MPI Point-to-Point Communication". In: *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, Dec. 2017. DOI: 10.1109/hpcc-smartcity-dss.2017.27 (cit. on p. 196).

[91] Adrien Cassagne, Olivier Aumage, Denis Barthou, Camille Leroux, and Christophe Jégo. "MIPP: a Portable C++ SIMD Wrapper and its use for Error Correction Coding in 5G Standard". In: *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*. WPMVP'18. Vienna, Austria: Association for Computing Machinery, 2018. ISBN: 9781450356466. DOI: 10.1145/3178433.3178435 (cit. on pp. 42, 222, 224).

[92] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. "Characterization of MPI usage on a production supercomputer". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 386–400. DOI: 10.1109/SC.2018.00033 (cit. on pp. 5, 54).

[93] Y. Gu and J. Mellor-Crummey. "Dynamic Data Race Detection for OpenMP Programs". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2018, pp. 767–778. DOI: 10.1109/SC.2018.00064 (cit. on pp. 83, 122, 123).

[94] Alexander Hück, Jan-Patrick Lehr, Sebastian Kreutzer, Joachim Protze, Christian Terboven, Christian Bischof, and Matthias S Müller. "Compiler-aided type tracking for correctness checking of MPI applications". In: *IEEE/ACM 2nd Intl. Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE. 2018, pp. 51–58. DOI: 10.1109/Correctness.2018.00011 (cit. on p. 171).

[95] Alexander Hück, Jan-Patrick Lehr, Sebastian Kreutzer, Joachim Protze, Christian Terboven, Christian Bischof, and Matthias S. Müller. "Compiler-aided type tracking for correctness checking of MPI applications". In: *IEEE/ACM 2nd Intl. Workshop on Software Correctness for HPC Applications (Correctness)*. 2018, pp. 51–58. DOI: 10.1109/Correctness.2018.00011 (cit. on pp. 55, 99, 111, 115, 122, 161).

[96] Marc Jasper, Malte Mues, Maximilian Schlüter, Bernhard Steffen, and Falk Howar. "RERS 2018: CTL, LTL, and Reachability". In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification*. Ed. by Tiziana Margaria and Bernhard Steffen. Springer, 2018, pp. 433–447. ISBN: 978-3-030-03421-4. DOI: 978-3-030-03421-4_27 (cit. on pp. 14, 83).

[97] Paulo Nunes, Ibéria Medeiros, José C Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. "Benchmarking static analysis tools for web security". In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 1159–1175. DOI: 10.1109/TR.2018.2839339 (cit. on pp. 14, 83).

[98] Amit Ruhela, Hari Subramoni, Sourav Chakraborty, Mohammadreza Bayatpour, Pouya Kousha, and Dhabaleswar K Panda. "Efficient asynchronous communication progress for MPI without dedicated resources". In: *Proceedings of the 25th European MPI Users' Group Meeting*. 2018, pp. 1–11. DOI: 10.1145/3236367.3236376 (cit. on p. 196).

[99] Whit Schonbein, Matthew G. F. Dosanjh, Ryan E. Grant, and Patrick G. Bridges. "Measuring Multithreaded Message Matching Misery". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2018, pp. 480–491. ISBN: 9783319969831. DOI: 10.1007/978-3-319-96983-1_34 (cit. on p. 213).

[100] Whit Schonbein, Matthew GF Dosanjh, Ryan E Grant, and Patrick G Bridges. "Measuring multithreaded message matching misery". In: *European Conference on Parallel Processing*. Springer. 2018, pp. 480–491. DOI: 10.1007/978-3-319-96983-1_34 (cit. on pp. 171, 181, 190, 208).

[101] Nawrin Sultana, Anthony Skjellum, Purushotham Bangalore, Ignacio Laguna, and Kathryn Mohror. "Understanding the usage of mpi in exascale proxy applications". In: *2018 SC Conference Workshop*. Accessed: 2024-08-13. 2018. URL: https://www.sc18.supercomputing.org/proceedings/workshops/workshop_files/ws_exampi103s2-file1.pdf (cit. on p. 5).

[102] Qingqing Xiong, Anthony Skjellum, and Martin C Herbordt. "Accelerating MPI message matching through FPGA offload". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2018, pp. 191–1914. DOI: 10.1109/FPL.2018.00039 (cit. on pp. 190, 213).

[103] Kurt Ferreira, Ryan E. Grant, Michael J. Levenhagen, Scott Levy, and Taylor Groves. "Hardware MPI message matching: Insights into MPI matching behavior to inform design". In: *Concurrency and Computation: Practice and Experience* 32.3 (Feb. 2019). ISSN: 1532-0634. DOI: 10.1002/cpe.5150 (cit. on p. 213).

[104] Ryan E Grant, Matthew GF Dosanjh, Michael J Levenhagen, Ron Brightwell, and Anthony Skjellum. "Finepoints: Partitioned multithreaded MPI communication". In: *International Conference on High Performance Computing*. Springer. 2019, pp. 330–350. DOI: 10.1007/978-3-030-20656-7_17 (cit. on pp. 27, 30, 182, 185, 187).

[105] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. "A Large-Scale Study of MPI Usage in Open-Source HPC Applications". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. ACM, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356176 (cit. on pp. 5–8, 54, 57, 58, 84, 92, 97, 99, 115, 117, 152, 157, 182, 227, 232).

[106] Scott Levy and Kurt B. Ferreira. "Evaluating tradeoffs between MPI message matching offload hardware capacity and performance". In: *Proceedings of the 26th European MPI Users' Group Meeting*. EuroMPI 2019. ACM, Sept. 2019. DOI: 10.1145/3343211.3343223 (cit. on pp. 190, 213).

[107] P. Lin, C. Liao, M. Schordan, and I. Karlin. "Exploring Regression of Data Race Detection Tools Using DataRaceBench". In: *IEEE/ACM 3rd Intl. Workshop on Software Correctness for HPC Applications (Correctness)*. 2019, pp. 11–18. DOI: 10.1109/Correctness49594.2019.00007 (cit. on p. 83).

[108] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard 2019 Draft Specification*. https://www.mpi-forum.org/docs/drafts/mpi-2019-draft-report.pdf. Accessed: 2024-08-09. Nov. 20, 2019 (cit. on pp. 171, 178).

[109] Rohit Zambre, Megan Grodowitz, Aparna Chandramowlishwaran, and Pavel Shamis. "Breaking band: a breakdown of high-performance communication". In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–10. DOI: 10.1145/3337821.3337910 (cit. on pp. 36, 203, 213).

[110] Ayesha Afzal, Georg Hager, and Gerhard Wellein. "Desynchronization and wave pattern formation in MPI-parallel and hybrid memory-bound programs". In: *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings 35*. Springer. 2020, pp. 391–411. DOI: 10.1007/978-3-030-50743-5_20 (cit. on p. 36).

[111] Arm Limited. *ARM DDT*. https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt. Accessed: 2024-08-13. 2020 (cit. on p. 90).

[112] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E. Grant, Thomas Naughton, Howard P. Pritchard, Martin Schulz, and Geoffroy R. Vallee. "A survey of MPI usage in the US exascale computing project". In: *Concurrency and Computation: Practice and Experience* 32.3 (2020), e4851. DOI: 10.1002/cpe.4851 (cit. on pp. 5, 54).

[113] Mikhail Brinskii. *UCX Active Messages*. https://ucfconsortium.org/wp-content/uploads/2021/02/Mikhail_Brinskii_UCX_Active_Messages.pdf. Accessad: 2024-08-13. Dec. 2020 (cit. on pp. 197, 198).

[114] A. Hück, J. Protze, J. -P. Lehr, C. Terboven, C. Bischof, and M. S. Müller. "Towards compiler-aided correctness checking of adjoint MPI applications". In: *IEEE/ACM 4th Intl. Workshop on Software Correctness for HPC Applications (Correctness)*. 2020, pp. 40–48. DOI: 10.1109/Correctness51934.2020.00010 (cit. on pp. 91, 99).

[115] Tim Jammer, Christian Iwainsky, and Christian Bischof. "A Comparison of the Scalability of OpenMP Implementations". In: *Euro-Par 2020: Parallel Processing*. virtual: Springer International Publishing, 2020, pp. 83–97. DOI: 10.1007/978-3-030-57675-2_6 (cit. on p. viii).

[116] Tim Jammer, Christian Iwainsky, and Christian Bischof. "Automatic Detection of MPI Assertions". In: *High Performance Computing. ISC High Performance 2020. Lecture Notes in Computer Science*. virtual: Springer International Publishing, 2020, pp. 34–42. DOI: 10.1007/978-3-030-59851-8_3. This publication is included in Chapter 12 (cit. on pp. vii, 83, 91, 170, 182, 190, 195).

[117] Jan-Patrick Lehr, Alexander Hück, Moritz Fischer, and Christian Bischof. "Compiler-Assisted Type-Safe Checkpointing". In: *High Performance Computing*. Ed. by Heike Jagode, Hartwig Anzt, Guido Juckeland, and Hatem Ltaief. Springer, 2020, pp. 5–18. ISBN: 978-3-030-59851-8. DOI: 10.1007/978-3-030-59851-8_1 (cit. on p. 94).

[118] Jan-Patrick Lehr, Alexander Hück, Yannic Fischler, and Christian Bischof. "MetaCG: annotated call-graphs to facilitate whole-program analysis". In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Tools for Automatic Program Analysis*. ACM, Nov. 2020. DOI: 10.1145/3427764.3428320 (cit. on p. 209).

[119] Alexey V Medvedev. "Towards benchmarking the asynchronous progress of non-blocking MPI operations". In: *Parallel Computing: Technology Trends, Proceedings of the International Conference on Parallel Computing*. Vol. 36. 2020, pp. 419–428. DOI: 10.3233/APC200067 (cit. on p. 203).

[120] William Moses and Johannes Doerfert. *"Header Time Optimization": Cross-Translation Unit Optimization via Annotated Headers*. Accessed: 2024-04-23. San Diego, CA, USA, 2020. URL: https://c.wsmoses.com/posters/HTO_Poster.pdf (cit. on pp. 43, 227).

[121] Van Man Nguyen, Emmanuelle Saillard, Julien Jaeger, Denis Barthou, and Patrick Carribault. "Automatic code motion to extend MPI nonblocking overlap window". In: *International Conference on High Performance Computing*. Springer. 2020, pp. 43–54. DOI: 10.1007/978-3-030-59851-8_4 (cit. on p. 190).

[122] OpenMP Architecture Review Board. *OpenMP - Application Programming Interface Version 5.1*. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf. Accessed: 2024-08-13. 2020 (cit. on pp. 19, 21, 117, 120, 128).

[123] OpenSHMEM Committee. *OpenSHMEM: Application Programming Interface Version 1.5*. Accessed: 2024-05-28. 2020. URL: http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.5.pdf (cit. on pp. 14, 153).

[124] Perforce Software Inc. *TotalView HPC Debugging Software*. https://totalview.io/products/totalview. Accessed: 2024-08-13. 2020 (cit. on p. 90).

[125] Joachim Protze, Marc-André Hermanns, Ali Demiralp, Matthias S Müller, and Torsten Kuhlen. "MPI detach-asynchronous local completion". In: *Proceedings of the 27th European MPI Users' Group Meeting*. 2020, pp. 71–80. DOI: 10.1145/3416315.3416323 (cit. on p. 50).

[126] Emmanuelle Saillard, Julien Jaeger, Denis Barthou, Patrick Carribault, et al. "Automatic code motion to extend MPI nonblocking overlap window". In: *International Conference on High Performance Computing*. Springer. 2020, pp. 43–54. DOI: 10.1007/978-3-030-59851-8_4 (cit. on pp. 23, 182, 183).

[127] Adrian Schmitz, Joachim Protze, Lechen Yu, Simon Schwitanski, and Matthias S. Müller. "DataRaceOnAccelerator – A Micro-benchmark Suite for Evaluating Correctness Tools Targeting Accelerators". In: *Euro-Par 2019: Parallel Processing Workshops*. Ed. by Ulrich Schwardmann, Christian Boehme, Dora B. Heras, Valeria Cardellini, Emmanuel Jeannot, Antonio Salis, Claudio Schifanella, Ravi Reddy Manumachu, Dieter Schwamborn, Laura Ricci, Oh Sangyoon, Thomas Gruber, Laura Antonelli, and Stephen L. Scott. Springer, 2020, pp. 245–257. ISBN: 978-3-030-48340-1. DOI: 10.1007/978-3-030-48340-1_19 (cit. on p. 83).

[128] Julian Seward et al. *Memcheck: a memory error detector*. https://valgrind.org/docs/manual/mc-manual.html. Accessed: 2024-08-13. 2020 (cit. on p. 185).

[129] Jannek Squar, Tim Jammer, Michael Blesel, Michael Kuhn, and Thomas Ludwig. "Compiler Assisted Source Transformation of OpenMP Kernels". In: *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*. virtual: IEEE, July 2020. DOI: 10.1109/ispdc51135.2020.00016 (cit. on pp. viii, 23, 50, 187).

[130] Ayesha Afzal, Georg Hager, and Gerhard Wellein. "Analytic Modeling of Idle Waves in Parallel Programs: Communication, Cluster Topology, and Noise Impact". In: (Mar. 2021). DOI: 10.1007/978-3-030-78713-4_19 (cit. on p. 209).

[131] Ayesha Afzal, Georg Hager, and Gerhard Wellein. "Analytic modeling of idle waves in parallel programs: communication, cluster topology, and noise impact". In: *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24–July 2, 2021, Proceedings 36*. Springer. 2021, pp. 351–371. DOI: 10.1007/978-3-030-78713-4_19 (cit. on p. 36).

[132] Atsushi Hori, Emmanuel Jeannot, George Bosilca, Takahiro Ogura, Balazs Gerofi, Jie Yin, and Yutaka Ishikawa. "An international survey on MPI users". In: *Parallel Computing* 108 (2021). ISSN: 0167-8191. DOI: 10.1016/j.parco.2021.102853 (cit. on pp. 5, 54).

[133] Tim Jammer and Christian Bischof. "Automatic Partitioning of MPI Operations in MPI+OpenMP Applications". In: *High Performance Computing. ISC High Performance 2021. Lecture Notes in Computer Science*. virtual: Springer International Publishing, 2021, pp. 191–198. DOI: 10.1007/978-3-030-90539-2_12. This publication is included in Chapter 13 (cit. on pp. vii, 180).

[134] Mathieu Laurent, Emmanuelle Saillard, and Martin Quinson. "The MPI Bugs Initiative: a Framework for MPI Verification Tools Evaluation". In: *IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*. 2021, pp. 1–9. DOI: 10.1109/Correctness54621.2021.00008 (cit. on pp. 15, 55, 76, 77, 115, 118, 129, 141, 150, 152, 155).

[135] Jan-Patrick Lehr, Tim Jammer, and Christian Bischof. "MPI-CorrBench: Towards an MPI Correctness Benchmark Suite". In: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. virtual: ACM, June 2021. DOI: 10.1145/3431379.3460652. This publication is included in Chapter 8 (cit. on pp. vi, 55, 76, 77, 81, 115, 118, 127, 129, 141, 150, 152, 155).

[136] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf. Accessed: 2024-08-09. June 9, 2021 (cit. on pp. 2, 10, 21, 23, 26, 54, 55, 57, 65, 114–118, 124–127, 129, 138, 141, 142, 181, 190, 208, 212).

[137] Kaiming Ouyang, Min Si, Astushi Hori, Zizhong Chen, and Pavan Balaji. "Daps: A Dynamic Asynchronous Progress Stealing Model for MPI Communication". In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2021, pp. 516–527. DOI: 10.1109/Cluster48925.2021.00027 (cit. on pp. 196, 197).

[138] Joseph Schuchart, Philipp Samfass, Christoph Niethammer, José Gracia, and George Bosilca. "Callback-based completion notification using MPI Continuations". In: *Parallel Computing* 106 (2021), p. 102793. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2021.102793 (cit. on p. 50).

[139] Nawrin Sultana, Martin Rüfenacht, Anthony Skjellum, Purushotham Bangalore, Ignacio Laguna, and Kathryn Mohror. "Understanding the use of message passing interface in exascale proxy applications". In: *Concurrency and Computation: Practice and Experience* 33.14 (2021), e5901. DOI: 10.1002/cpe.5901 (cit. on p. 54).

[140] Department of Computer Science, TU Darmstadt. *Cumulative Theses and Self-Citations in Theses*. Accessed: 2024-08-12. 2022. URL: https://www.informatik.tu-darmstadt.de/media/informatik/fb20_dekanat/dokumente_promotion/20221201_TUDa_FB20_KumulativeDissertation_Eigenzitate_DE_EN.pdf (cit. on p. vii).

[141] Exascale Computing Project. *Exascale Proxy Applications*. Accessed: 2024-04-23. 2022. URL: https://proxyapps.exascaleproject.org/ (cit. on pp. 45, 232, 240).

[142] Tim Jammer and Christian Bischof. "Compiler-enabled optimization of persistent MPI Operations". In: *2022 IEEE/ACM International Workshop on Exascale MPI (ExaMPI)*. Dallas, TX: IEEE, Nov. 2022. DOI: 10.1109/exampi56604.2022.00006. This publication is included in Chapter 14 (cit. on pp. vii, 55, 189, 213, 214, 222, 224).

[143] Tim Jammer, Alexander Hück, Jan-Patrick Lehr, Joachim Protze, Simon Schwitanski, and Christian Bischof. "Towards a Hybrid MPI Correctness Benchmark Suite". In: *EuroMPI/USA'22: 29th European MPI Users' Group Meeting*. Chattanooga, TN: ACM, Sept. 2022. DOI: 10.1145/3555819.3555853. This publication is included in Chapter 9 (cit. on pp. vi, 55, 76, 77, 113, 141).

[144] Alexey V Medvedev. "IMB-ASYNC: a revised method and benchmark to estimate MPI-3 asynchronous progress efficiency". In: *Cluster Computing* 25.4 (2022), pp. 2683–2697. DOI: 10.1007/s10586-021-03452-8 (cit. on pp. 36, 196, 203).

[145] Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, and Matthias S. Müller. "On-the-Fly Data Race Detection for MPI RMA Programs with MUST". In: *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 2022, pp. 27–36. DOI: 10.1109/Correctness56720.2022.00009 (cit. on p. 164).

[146] Alexander Hück, Tim Jammer, Joachim Jenke, and Christian Bischof. "Investigating the Real-World Applicability of MPI Correctness Benchmarks". In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W 2023. Denver, CO: ACM, Nov. 2023. DOI: 10.1145/3624062.3624091. This publication is included in Chapter 10 (cit. on pp. vi, 140, 151–154, 161, 166).

[147] Alexander Hück, Tim Jammer, Joachim Jenke, and Christian Bischof. "Investigating the Usage of MPI at Argument-Granularity in HPC Codes". In: *Proceedings of the 30th European MPI Users' Group Meeting*. EUROMPI '23. Bristol, UK: ACM, Sept. 2023. DOI: 10.1145/3615318.3615322. This publication is included in Chapter 7 (cit. on pp. vi, 53, 141, 142, 145, 151, 153, 157, 158, 214, 227, 230, 232).

[148] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. Accessed: 2024-08-09. Nov. 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf (cit. on pp. 150, 229).

[149] Simon Schwitanski, Joachim Jenke, Sven Klotz, and Matthias S. Müller. "RMARaceBench: A Microbenchmark Suite to Evaluate Race Detection Tools for RMA Programs". In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W '23. ACM, 2023, pp. 205–214. DOI: 10.1145/3624062.3624087 (cit. on pp. 14, 150, 153).

[150] Radjasouria Vinayagame, Emmanuelle Saillard, Samuel Thibault, Van Man Nguyen, and Marc Sergent. "Rethinking Data Race Detection in MPI-RMA Programs". In: *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, pp. 196–204. DOI: 10.1145/3624062.3624086 (cit. on p. 164).

[151] Philippe Virouleau, Emmanuelle Saillard, Marc Sergent, and Pierre Lemarinier. "Highlighting PARCOACH Improvements on MBI". In: *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W 2023. 2023. DOI: 10.1145/3624062.3624093 (cit. on pp. 161, 164).

[152] Department of Computer Science, TU Darmstadt. *Nutzung von KI-Hilfsmitteln bei Prüfungsleistungen im Studium*. Accessed: 2024-08-12. 2024. URL: https://www.informatik.tu-darmstadt.de/media/informatik/fb20_studium/infos_dozenten/20240126_KI_Hilfsmittel.de.pdf (cit. on p. 262).

[153] Intel. *Intel Trace Analyzer and Collector*. https://www.intel.com/content/www/us/en/developer/articles/release-notes/intel-trace-analyzer-and-collector-release-notes-linux.html?wapkw=ITAC. Accessed: 2024-08-13. 2024 (cit. on pp. 11, 55, 83, 115, 117, 122, 141, 150, 151, 160).

[154] Tim Jammer, Tim Heldmann, Michael Blesel, Michael Kuhn, and Christian Bischof. "Compiler-Based Precalculation of MPI Message Envelopes". In: *High Performance Computing. ISC High Performance 2024. Lecture Notes in Computer Science*. Hamburg, Germany: Springer International Publishing, May 2024. DOI: 10.1007/978-3-031-73716-9_1. This publication is included in Chapter 15 (cit. on pp. vii, 211).

[155] Tim Jammer, Christian Iwainsky, and Christian Bischof. "Survey of OpenMP Practice in General Open Source Software". In: *OpenMP: Advanced Task-Based, Device and Compiler Programming. IWOMP 2024. Lecture Notes in Computer Science*. Perth, Australia, Sept. 2024. DOI: 10.1007/978-3-031-72567-8_7 (cit. on pp. viii, 51).

[156] Tim Jammer, Emmanuelle Saillard, Simon Schwitanski, Joachim Jenke, Radjasouria Vinayagame, Alexander Hück, and Christian Bischof. "MPI-BugBench: A Framework for Assessing MPI Correctness Tools". In: *Proceedings of the 31th European MPI Users' Group Meeting*. EUROMPI 2024. Perth, Australia, Sept. 2024. DOI: 10.1007/978-3-031-73370-3_8. This publication is included in Chapter 11 (cit. on pp. vi, 149).

[157] Tim Jammer, Adrian Schmidt, and Christian Bischof. "Annotation of Compiler attributes for MPI Functions". In: *Proceedings of the 31th European MPI Users' Group Meeting*. EUROMPI/Australia '24. Perth, Australia, Sept. 2024. DOI: 10.1007/978-3-031-73370-3_2. This publication is included in Chapter 16 (cit. on pp. vii, 225).

[158] Tim Jammer, Simon Schwitanski, Emmanuelle Saillard, Alexander Hück, Joachim Jenke, Radjasouria Vinayagame, and Christian Bischof. "Designing Quality MPI Correctness Benchmarks: Insights and Metrics". In: *Proceedings of the SC '24 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W 2024. Atlanta, GA: ACM, Nov. 2024. DOI: 10.1109/SCW63240.2024.00034 (cit. on p. viii).

[159] *LLVM Language Reference Manual*. Accessed: 2024-05-04. 2024. URL: https://llvm.org/docs/LangRef.html%5C#pointercapture (cit. on p. 229).

[160] Bronis R. de Supinski and Michael Klemm. *OpenMP Technical Report 13: Version 6.0 Public Comment Draft*. Accessed: 2024-08-22. 2024. URL: https://www.openmp.org/wp-content/uploads/openmp-TR13.pdf (cit. on p. 51).

[161] *Dyninst*. https://www.dyninst.org/. Accessed: 2022-08-15 (cit. on p. 122).

[162] *Intel Inspector*. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/inspector.html. Accessed: 2022-01-20 (cit. on p. 122).

[163] *Intel MPI Benchmarks*. https://github.com/intel/mpi-benchmarks. Accessed: 2021-01-24 (cit. on p. 83).

[164] Message Passing Interface Forum. *MPI 4.2*. https://www.mpi-forum.org/mpi-42/. Accessed: 2024-08-09 (cit. on p. 2).

[165] Message Passing Interface Forum. *MPI 5.0*. https://www.mpi-forum.org/mpi-50/. Accessed: 2024-08-09 (cit. on p. 2).

[166] *MPI Forum Issue Tracker*. https://github.com/mpi-forum/mpi-issues/issues/558. Accessed: 2024-08-19 (cit. on p. 49).

[167] *MPI Forum Issue Tracker*. https://github.com/mpi-forum/mpi-issues/issues/117. Accessed: 2022-08-15 (cit. on p. 138).

[168] *Rigorous Examination of Reactive Systems (RERS) Challenge*. http://rers-challenge.org. Accessed: 2021-01-24 (cit. on pp. 14, 83).

[169] *ThreadSanitizer*. https://clang.llvm.org/docs/ThreadSanitizer.html. Accessed: 2022-08-15 (cit. on p. 132).

Statement of AI usage

In accordance with the department's policy [152], during the writing of the synopsis and some of the included publications, generative AI ChatGPT (https://chat.openai.com) was utilized, to improve language, grammar and spelling. The input prompts to the AI tool contained text drafts manually written by the thesis author. Formulation suggestions by generative AI were only retained after careful consideration. Most prompts were of the form "This text, but a few words shorter: [text draft written by this thesis author]", to achieve more concise formulation, where applicable. For example, an initial draft used

the phrase “switch off message matching”, which was subsequently changed to “disable message matching.” Prompts used, alongside the AI systems responses can be found here: https://github.com/ttm02/dissertation-ai-use.
Additionally, languagetool.org was used for the same purpose.

For translating the English abstract into German, ChatGPT was used to generate a preliminary German version of the abstract.

List of Figures

List of Tables

List of Listings

List of Acronyms

HPC high performance computing

MPI Message Passing Interface

OpenMP Open Multi-Processing

SHMEM Shared Memory Programming Model

GASPI Global Address Space Programming Interface

API application programming interface

ABI application binary interface

RMA remote memory access

LTO link time optimization

CFG control-flow graph

IDE integrated development environment

IR intermediate representation

UCX Unified Communication X

MBB MPI-BugBench

MBI MPI Bugs Initiative

COBE MPI-CorrBench

CATO Compiler Assisted Source Transformation of OpenMP Kernels

PARCOACH PARallel COntrol flow Anomaly CHecker

MUST Marmot Umpire Scalable Tool

ITAC Intel Trace Analyzer and Collector

IMB Intel MPI Benchmarks

RERS Rigorous Examination of Reactive Systems

LTL Linear Temporal Logic

CTL Computation Tree Logic