MPI-BugBench: A Framework for Assessing MPI Correctness Tools

Tim Jammer, Emmanuelle Saillard, Simon Schwitanski, Joachim Jenke,
Radjasouria Vinayagame, Alexander Hück, Christian Bischof

▶ **To cite this version:**

Tim Jammer, Emmanuelle Saillard, Simon Schwitanski, Joachim Jenke, Radjasouria Vinayagame, et al.. MPI-BugBench: A Framework for Assessing MPI Correctness Tools. EuroMPI/Australia 2024, Sep 2024, Perth, Australia. pp.121-137, ⟨10.1007/978-3-031-73370-3_8⟩. ⟨hal-04878321⟩

HAL Id: hal-04878321
https://hal.science/hal-04878321v1

Submitted on 9 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

MPI-BugBench: A Framework for Assessing MPI Correctness Tools

Tim Jammer[1][0000−0003−3735−9677]⋆, Emmanuelle Saillard[2][0009−0008−6409−1673]⋆, Simon Schwitanski[3][0000−0001−7121−7205]⋆, Joachim Jenke[3][0000−0003−0640−8966], Radjasouria Vinayagame[4][0009−0001−8841−3322], Alexander Hück[1][0000−0003−1931−773X], and Christian Bischof[1][0000−0003−2711−3032]

[1] Technical University Darmstadt, Darmstadt, Germany
{tim.jammer,alexander.hueck,christian.bischof}@tu-darmstadt.de
[2] Inria, Bordeaux, France {emmanuelle.saillard}@inria.fr
[3] RWTH Aachen University, Aachen, Germany
{schwitanski,jenke}@itc.rwth-aachen.de
[4] Eviden, Bordeaux, France {radjasouria.vinayagame}@eviden.com

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.

Keywords: MPI · Correctness · Verification · Benchmarks · Tools

1 Introduction

The Message Passing Interface (MPI, [13]) 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 [1, 3].

To address this issue, several MPI correctness tools have been developed using static [1], dynamic [3, 7], or a combined analysis [15] 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

⋆ These authors contributed equally to this research.

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, [10]) and the MPI Bugs Initiative (MBI, [9]). 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, [16]) 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 [4] comparing the MPI calls and their arguments present in COBE and MBI with a dataset of real-world MPI usage [5]. 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 [3], ITAC [7], and PARCOACH [15], using MPI-BugBench.

The rest of the paper is structured as follows. In Sect. 2, we present existing MPI correctness benchmarks and discuss our definition of real-world MPI usage. Sect. 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 Sect. 4, we evaluate three state-of-the-art MPI correctness tools using MPI-BugBench. Sect. 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 Sect. 6.

2 Previous Work

In previous work [4], we evaluated COBE and MBI's real-world applicability by applying a feature-scoring method based on our previous MPI study [5] 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., [8], 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 Sect. 2.1, we provide a brief overview of the correctness benchmarks COBE, MBI, and RRB. We elaborate on our scoring algorithm in Sect. 2.2. It serves as the guideline for designing MPI-BugBench's test code generator described in Sect. 3.

2.1 Correctness Benchmarks

MPI-CorrBench COBE [10] 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 [9] 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 [16] focuses on different data race test cases for the remote memory access (RMA) models MPI RMA, OpenSHMEM [14], and GASPI [2]. 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.

2.2 MPI Feature Usage Analysis

In a previous study [5], 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 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 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 [4].

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. 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 [4]. The reasoning behind testing the same errors with several distinct MPI usage patterns is further detailed in Section 3.2.

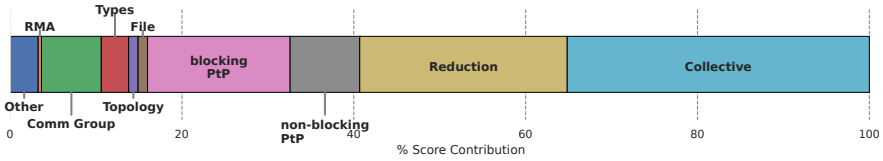

Fig. 1: Weighting of MPI categories, adapted from [4]. Collectives and reductions are predominantly used, followed by PtP operations. This indicates that correctness benchmarks should focus on these categories first to improve their scoring.

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 4.2). In this section, we first discuss the different types of errors we consider in Section 3.1, while Section 3.2 discusses different instantiations of errors that can be generated. The explanation of how the test cases are generated follows in Section 3.3.

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 [10] and MBI [9].

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.

3.2 Error Instantiations Covered

Apart from testing for different *kinds* of errors explained in Section 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 [5, 8] 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 [5].

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 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 1.

In order to determine which usage patterns occur in the real world, MBB uses the data set collected by Hück et al. [5], 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.

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

Table 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

```
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 2: Illustration of the test-case generation function for invalid buffer errors.

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 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 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 `generate` function is to set an appropriate error description in line 11 of Listing 2, before yielding the instantiated `TemplateManager` to the generation infrastructure. The generation infrastructure will then generate the resulting erroneous code (illustrated in Listing 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 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 3.2.

```
// 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 3: Excerpt of one error code produced by the generator shown in Listing 2.

4 Evaluation of Correctness Tools

This section first discusses the real-world coverage of the generated test cases in Section 4.1. We then evaluate three state-of-the-art MPI correctness tools using the generated test cases in Section 4.2.

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 3.2. The real-world applicability, as explained in Section 2.2, is illustrated in Fig. 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 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 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.

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) [7] profiles

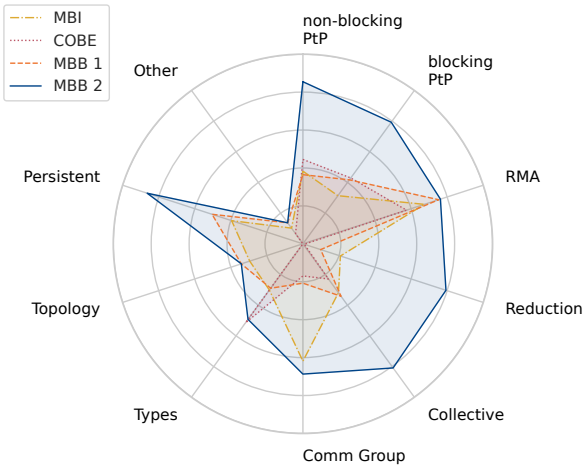

Fig. 2: Comparison of real-world applicability of MPI Correctness Benchmarks. MBI (orange) denotes the MPI Bugs Initiative. COBE (blue) denotes MPI-CorrBench. MBB (green) denotes MPI-BugBench with coverage level 2.1. We used the methodology proposed by Hück et al. [4] for evaluation, without including the Fortran codes in the real-world dataset (see Section 2.2).

Table 2: Total real-world coverage score. The scoring is described in Section 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

and analyzes MPI programs to check their correctness. It intercepts MPI calls and generates trace files that can be analyzed to understand the program's behaviors. MUST [3] 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 [6] and LLVM 14. PARCOACH [15, 19] 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.

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 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

Table 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

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 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. 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 [17] 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 [18, 19] for RMA that significantly improves the

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

Fig. 3: Tool evaluation using MPI-BugBench including a breakdown for each MPI feature category.

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 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 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.

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 [4], 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 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 [11, 12].

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.

Bibliography

[1] Droste, A., Kuhn, M., Ludwig, T.: MPI-checker: static analysis for MPI. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM '15, ACM (2015), `https://doi.org/10.1145/2833157.2833159`

[2] GASPI Forum: GASPI: Global Address Space Programming Interface 17.1 (2017), URL `https://raw.githubusercontent.com/GASPI-Forum/GASPI-Forum.github.io/master/standards/GASPI-17.1.pdf`, [online; accessed 28-May-2024]

[3] Hilbrich, T., Schulz, M., de Supinski, B.R., Müller, M.S.: MUST: A scalable approach to runtime error detection in MPI programs. In: Tools for High Performance Computing 2009, pp. 53–66, Springer (2010), `https://doi.org/10.1007/978-3-642-11261-4_5`

[4] Hück, A., Jammer, T., Jenke, J., Bischof, C.: 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, pp. 230–233, SC-W '23, ACM (2023), `https://doi.org/10.1145/3624062.3624091`

[5] Hück, A., Jammer, T., Protze, J., Bischof, C.: Investigating the Usage of MPI at Argument-Granularity in HPC Codes. In: Proceedings of EuroMPI2023: the 30th European MPI Users' Group Meeting, pp. 1–10, EuroMPI2023, ACM (2023), https://doi.org/10.1145/3615318.3615322

[6] Hück, A., Lehr, J.P., Kreutzer, S., Protze, J., Terboven, C., Bischof, C., Müller, M.S.: Compiler-aided type tracking for correctness checking of MPI applications. In: IEEE/ACM 2nd Intl. Workshop on Software Correctness for HPC Applications (Correctness), pp. 51–58 (2018), https://doi.org/10.1109/Correctness.2018.00011

[7] Intel: Intel Trace Analyzer and Collector. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/trace-analyzer.html (2023), [online; accessed 28-May-2024]

[8] Laguna, I., Marshall, R., Mohror, K., Ruefenacht, M., Skjellum, A., Sultana, N.: 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), https://doi.org/10.1145/3295500.3356176

[9] Laurent, M., Saillard, E., Quinson, M.: The MPI Bugs Initiative: a Framework for MPI Verification Tools Evaluation. In: IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness), pp. 1–9 (2021), https://doi.org/10.1109/Correctness54621.2021.00008

[10] Lehr, J.P., Jammer, T., Bischof, C.: MPI-CorrBench: Towards an MPI Correctness Benchmark Suite. In: Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, pp. 69–80, HPDC'21, ACM (2021), https://doi.org/10.1145/3431379.3460652

[11] Lin, P.H., Liao, C.: High-Precision Evaluation of Both Static and Dynamic Tools using DataRaceBench. In: 2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness), pp. 1–8 (2021), https://doi.org/10.1109/Correctness54621.2021.00011

[12] Luecke, G., Coyle, J., Hoekstra, J., Kraeva, M., Xu, Y., Park, M.Y., Kleiman, E., Weiss, O., Wehe, A., Yahya, M.: The Importance of Run-Time Error Detection. pp. 145–155 (01 2009), https://doi.org/10.1007/978-3-642-11261-4_10

[13] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 4.1 (Nov 2023), URL https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf, [online; accessed 28-May-2024]

[14] OpenSHMEM Committee: OpenSHMEM: Application Programming Interface Version 1.5 (2020), URL http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.5.pdf, [online; accessed 28-May-2024]

[15] Saillard, E., Carribault, P., Barthou, D.: PARCOACH: Combining static and dynamic validation of MPI collective communications. The International Journal of High Performance Computing Applications **28**(4), 425–434 (2014), https://doi.org/10.1145/2488551.2488555

[16] Schwitanski, S., Jenke, J., Klotz, S., Müller, M.S.: 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, pp. 205–214, SC-W '23, ACM (2023), https://doi.org/10.1145/3624062.3624087

[17] Schwitanski, S., Jenke, J., Tomski, F., Terboven, C., Müller, M.S.: 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), pp. 27–36, IEEE (2022), https://doi.org/10.1109/Correctness56720.2022.00009

[18] Vinayagame, R., Saillard, E., Thibault, S., Nguyen, V.M., Sergent, M.: 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, pp. 196–204 (2023), https://doi.org/10.1145/3624062.3624086

[19] Virouleau, P., Saillard, E., Sergent, M., Lemarinier, P.: Highlighting PARCOACH Improvements on MBI. In: Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023 (2023), https://doi.org/10.1145/3624062.3624093