Designing Quality MPI Correctness Benchmarks: Insights and Metrics

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

To cite this version:

Tim Jammer, Simon Schwitanski, Emmanuelle Saillard, Alexander Hück, Joachim Jenke, et al.. Designing Quality MPI Correctness Benchmarks: Insights and Metrics. 8th International Workshop on Software Correctness for HPC Applications (Correctness '24), Nov 2024, Atlanta, United States. hal-04878332

HAL Id: hal-04878332
https://hal.science/hal-04878332v1

Submitted on 20 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.

Designing Quality MPI Correctness Benchmarks: Insights and Metrics

Tim Jammer*, Simon Schwitanski†, Emmanuelle Saillard‡, Alexander Hück*,
Joachim Jenke†, Radjasouria Vinayagame§, Christian Bischof*

*Technical University Darmstadt. {tim.jammer,alexander.hueck,christian.bischof}@tu-darmstadt.de
†RWTH Aachen University {schwitanski,jenke}@itc.rwth-aachen.de
‡Inria emmanuelle.saillard@inria.fr
§Eviden radjasouria.vinayagame@eviden.com

***Abstract*—Several MPI correctness benchmarks have been proposed to evaluate the quality of MPI correctness tools. The design of such a benchmark comes with different challenges, which we address in this paper. First, an imbalance in the proportion of correct and erroneous codes in the benchmarks requires careful metric interpretation (recall, accuracy, F1 score). Second, tools that detect errors but do not report additional information, like the affected source line or class of error, are less valuable. We extend the typical notion of a true positive with stricter variants that consider a tool's helpfulness. We introduce a new noise metric to consider the amount of distracting error reports. We evaluate those new metrics with MPI-BugBench, on the MPI correctness tools ITAC, MUST, and PARCOACH. Third, we discuss the complexities of hand-crafted and automatically generated benchmark codes and the additional challenges of non-deterministic errors.**

***Index Terms*—Correctness Benchmark, Correctness Tools, Metrics, Error Reports**

I. Introduction

Recently, MPI correctness benchmark suites [1]–[4] have been proposed as standardized test harnesses for MPI correctness tools [5]–[7]. These suites contain small-scale tests that include both correct and erroneous uses of the MPI library, such as illegal value ranges of MPI arguments, deadlocks, and data races. As a result, they are beneficial for both correctness tool developers and users. Developers can evaluate their tools' feature support and performance against other solutions, while users can determine the best tool for their practical needs.

However, assessing MPI correctness tools with these benchmarks presents unique challenges due to inherent ambiguities in tool error reporting, where reports may be unclear about the nature and origin of errors. Furthermore, traditional evaluation metrics based on binary classification can be misleading, mainly because of an imbalance between the number of erroneous and correct test cases. Thus, evaluating error reports requires considering multiple factors, including (i) source location accuracy, (ii) error description clarity, (iii) and the presence of extraneous information or *noise*. This noise includes correctly identified errors accompanied by multiple non-existent ones, adding unnecessary complexity for the user.

Additionally, the design of correctness benchmarks introduces further complexities, requiring trade-offs between hand-crafted and automatically generated test cases. Also, the non-deterministic nature of evaluating MPI and hybrid MPI + X models, like MPI+OpenMP or MPI+CUDA [8], [9], poses significant challenges in accurately detecting and reporting errors. This work analyzes these complexities and proposes solutions and strategies for more effective evaluation of correctness tools. In Section II, we discuss the ambiguity of binary classification for correctness benchmarks and introduce new metrics for an in-depth analysis of the helpfulness of error reports. We evaluate MPI-BugBench (MBB) [10] with those new metrics in Section III. Section IV explores the intricacies of benchmark design, such as automatic test generation, weighing the benefits and drawbacks of existing approaches. In Section V, we address the challenges associated with non-deterministic behavior in MPI and hybrid MPI + X programs.

II. Correctness Benchmark Classification

Evaluating a correctness analysis tool with a correctness benchmark can be seen as a binary classification problem. For an incorrect code, a correctness tool either finds the designated error (True Positive: TP) or misses it (False Negative: FN). Similarly, for a correct code, a tool either wrongfully detects an error not present (False Positive: FP) or correctly determines the code to be error-free (True Negative: TN).

In terms of a correctness benchmark however, there are several challenges when using the standard evaluation metrics, like precision ($\frac{TP}{TP+FP}$), recall ($\frac{TP}{TP+FN}$), accuracy ($\frac{TP+TN}{TP+TN+FP+FN}$) or F1 score ($\frac{2TP}{2TP+FP+FN}$). Evaluation metrics such as the F1 Score can be misleading on unbalanced data sets [11].

Additionally, it may be ambiguous if a given error report from a correctness tool actually represents a true positive. One question here is: "Should a rather vague report that points out that there is *some* error, but is otherwise not helpful for a programmer to understand and fix the issue be considered as a true positive (TP)?"

Current correctness benchmarks follow a unit-test-like design, with only one explicit error per test case. This design is particularly useful to determine if correctness tools support different MPI features, as they can be tested independently. It

is also valuable for developing a correctness tool, as small-scale unit tests can be used to debug correctness tools. Nevertheless, the issues discussed in this paper can be applied to different designs of correctness benchmarks (e.g., benchmarks with more complex test cases and multiple errors per test case).

A. Evaluation Metrics on Imbalanced Data Sets

Today, the usefulness of tools is evaluated by standard metrics like recall, accuracy, and F1 score. It is known that those metrics can be misleading for imbalanced data sets [11], where there are more positive (in our case erroneous) examples than negative (in our case correct) ones. For example, when many positive (erroneous) cases exist, a tool with a recall of 1 could be a perfect tool or just one that *always* reports an error. In this case, a tool that always predicts "error" would also achieve relatively high precision, as the number of negative (correct) cases to "challenge" the tool's hypothesis is rather limited. As the F1 score is the harmonic mean of recall and precision, this will also lead to a high F1, although such a tool that always predicts "error" is not of very high quality.

Unfortunately, an MPI correctness benchmark is naturally unbalanced. The reason is that for every correct MPI usage, multiple different errors can be made. Consider an `MPI_Send`, for example: Passing `NULL` as the buffer argument is a different error than specifying a too large number of elements to be sent. This means that a correctness benchmark is biased towards favoring tools that overestimate the errors present. For example, in MPI-BugBench, only $\approx 20\%$ of test cases are correct (negative) examples.

It is important for a tool's usefulness that it does not report many false positives, as the tool's users will then spend time understanding an error where there is none. Additionally, this may lead to tool users ignoring certain warnings, even if those errors really occur this time. More than precision as the key indicator is needed in this case because it is biased towards overestimation, as explained above. Therefore, we propose to always include the specificity ($\frac{TN}{TN+FP}$) or the inverse "false positive ratio" as one of the key factors in evaluating a correctness tool's quality.

B. Ambiguity of Evaluating Error Reports

Calculating the metrics mentioned in Section II-A requires that true positive (TP), true negative (TN) as well as false negative (FN), and false positive (FP) are clearly defined. If a tool only reports *"Error"* or *"No Error"*, these metrics can be clearly defined without ambiguity. While there may be at least some value in such a judgment from a correctness tool, as it tells the developer that they need to look into an issue further, it is not very helpful, in our opinion. In order to fix the issue, the application programmer first needs to understand what the actual error is, and a message just consisting of *"Error"* is not helpful in that regard.

The exact classification of error reports as "helpful" for fixing the error may be subjective, though. Therefore, a standardized evaluation of the correctness tool's error reports in a benchmark has several challenges related to the ambiguity of error reports. For example: Is "deadlock" an appropriate error description, or should a deadlock be considered as the symptom, e.g., of a message tag mismatch error? Therefore, a fair, standardized classification of error reports is challenging.

We consider three aspects of "helpful" error reports: source location, error description, and noise.

a) Source Location: As one marker for a "helpful" error message, we suggest including the source code line of at least one of the involved MPI functions in the error description. The source location information allows developers to quickly identify which part of the codebase they need to investigate to understand and in turn, resolve the issue. In terms of a standardized benchmark, the information on the source code line has the additional advantage of being rather unambiguous. Only when some MPI calls are missing, it may be ambiguous where the programmer should look to find the error. As no single MPI call exists in isolation however, we suggest that a correctness tool points to the other MPI call that provokes the error. For example, when using a non-blocking operation with a missing call to `MPI_Wait`[1], the missing `MPI_Wait` has no clear position in the source code, but the non-blocking operation that starts the request has a well-defined location.

b) Error Description: While including the correct source code location in the error description is valuable, it may not be sufficient on its own. In order to quickly understand the error, a helpful error description is important. A helpful error report must accurately identify the present error rather than distract users with incorrect assumptions by the tool. Here, however, no standardized automatic measurement of "helpfulness" is possible. As discussed above, it is subjective to determine at what point an error description can be considered helpful. This means that it is not feasible for a standardized benchmark to define which error descriptions are helpful. But a standardized correctness benchmark can define certain classes of errors, like "Invalid Parameter" (e.g. a negative target rank for sending a message) or "Local Concurrency" (e.g. a write to the message buffer while a non-blocking operation is in progress). In order to determine if a tool's error report points the user in the right direction, one can match the error description of the tool with the error categories defined by the correctness benchmark. However, this may require some effort when evaluating a new version or a completely new tool with the correctness benchmark. MUST, for example, is able to produce 111 different error descriptions, that need to be matched to the corresponding error classes in the correctness benchmark.

c) Noise: Another aspect of ambiguity when utilizing a standardized benchmark is considering an error report, where a tool points out multiple errors, even though only one error is present in the test case. One may argue that a standardized benchmark should only check the first error reported, as other errors may be "follow-up" errors that are resolved when the first error is fixed. In our opinion, however, a high-quality correctness tool should only report the actual present

[1] or other means of completing the non-blocking request

errors and should not clutter the error report with additional *misleading* errors that are actually not present.

We, therefore, define the "noise ratio" as the ratio of "not-helpful reports" over all error reported in the tools output, with "helpful" being measured with the aspects of source location and error description, as discussed above. For example, a tool may report a given error for both involved processes, but an unrelated different error is additionally (and wrongfully) reported for process 0. In this case, the "noise ratio" is $\frac{1}{3}$ as one error reported was misleading.

We therefore propose different aspects of scrutiny when calculating the metrics mentioned in Section II-A, with a high quality correctness tool fulfilling all aspects:

- TP_{base}: Any error is reported.
- TP_{line}: The error report contains correct source line information.
- TP_{class}: The error report points to an error of the correct class.
- TP_{class_line}: The error report points to an error of the correct class as well as includes correct source line information.

Depending on the level of scrutiny, an error report is only considered true positive (TP) for calculating the metrics discussed in Section II-A, when it fulfills all relevant criteria. Otherwise, it is considered a false positive (FP) for calculation of the metrics discussed in Section II-A, as, in our opinion, a not helpful report distracts from the actual errors present in the same way a false alarm would.

III. Helpfulness of Error Reports

Figure 1 shows the helpfulness of the errors reported by the correctness tools MUST [5], ITAC [7] and PARCOACH [12], evaluated with MPI-Bugbench (MBB) [10]. The different aspects of helpfulness visualized in Figure 1 are explained in Section II-B. An error report can contain a description of the appropriate error class (blue in Figure 1), the source code line of an erroneous MPI call (green in Figure 1), or both (hatched blue and green in Figure 1). If it contains neither the appropriate error class nor appropriate source code line information, it is considered not helpful (red in Figure 1). For each tool we manually defined a mapping from the tools' error descriptions to the corresponding error classes of MBB.

While the correctness tools are not perfect, we see that most error reports can be considered helpful. The majority of reports contain an appropriate error description alongside correct information on the affected source code location. PARCOACH (Figure 1c) does not support point-to-point communication. Therefore, most error reports are not helpful for this category. The blue part of P2P corresponds to reports that contain the right error class (CallOrdering) but wrongfully assumes an error with collective operations instead of the erroneous point-to-point operation. ITAC (Figure 1a) sometimes gives not-helpful reports alongside: "Error: Unknown Error". We note that this evaluation only looks at the helpfulness of errors actually reported, in the case where errors are expected. The errors that were not found at all, as well as errors reported, although no errors are expected, can *not* be observed in the figures.

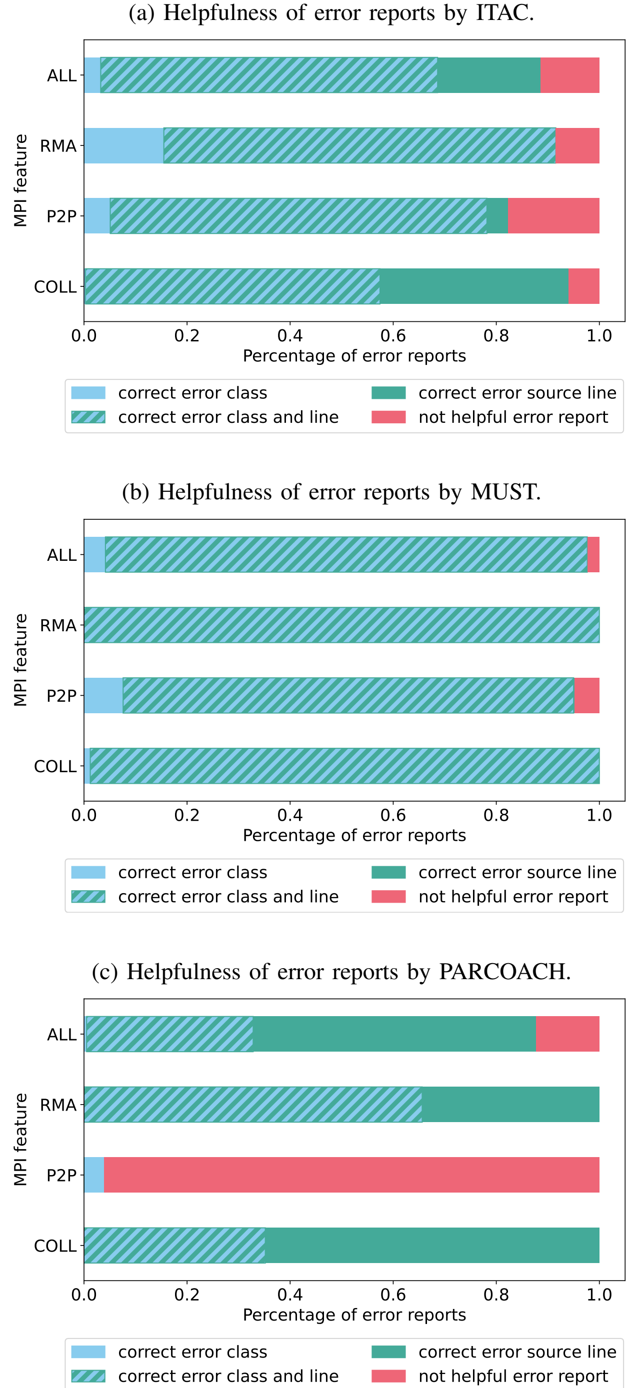


Fig. 1: Helpfulness of error evaluated on MPI-Bugbench. COLL and P2P represent codes misusing collective and point-to-point communications. RMA represents codes with errors when using the MPI-RMA feature. ALL gathers codes using the three features but also other features such as datatypes.

TABLE I: The noise ratio of errors reported (i.e., the ratio of misleading error reports in addition to the helpful ones compared to the overall number of error reports) only considers test cases where errors are expected. Lower is better.

MPI feature	ITAC	MUST	PARCOACH
ALL	0.19	0.08	0.25
RMA	0.32	0.00	0.43
P2P	0.31	0.11	1.00
COLL	0.08	0.06	0.11

Table I shows the noise ratio of the observed error reports. The noise ratio is determined as the ratio of misleading error reports compared to the overall number of errors reported. For this purpose, a misleading error report is one that contains incorrect information about either the error class or the affected source code line. We see that MUST has a very low noise ratio, and most error reports are rather helpful in finding the issue. Again, the lack of support for point-to-point communication errors explains PARCOACH's large noise ratio.

IV. Complexity of Correctness Benchmarks

When designing a correctness benchmark, the complexity necessary for creation and evaluation should also be taken into account. In general, two approaches exist: hand-crafted test codes [1], [4], [13] and automatically generated ones [2], [10]. Automatically generated codes have the advantage of potentially covering a wider range of usage patterns, as codes for one error can be generated for all possible usage patterns. As we described in [14], this allows for better real-world applicability of correctness benchmarks, as all the relevant usage patterns can be covered, not only a few hand-selected ones. On the other hand, hand-crafted codes have the advantage of enabling new test cases from other contributors to be more easily integrated. As automatically generated codes offer the possibility to test the tool to find one error in several different usage patterns, they are better at assessing the implementation quality of a tool. This helps correctness tool developers to "stress-test" their implementation with many different examples of the same error in different circumstances, fostering a more reliable implementation.

Generating many different test cases for a thorough evaluation of a tool may, however, lead to a high runtime of the evaluation. For a small set of hand-crafted test cases, it may be possible to manually verify whether the correctness tool found the error and produced a helpful report. Although this still is a *subjective* evaluation process, it alleviates some challenges discussed in Section II-B. For a more extensive set of codes, a manual evaluation is not feasible. In MPI-BugBench, we tackled the trade-off between hand selected and auto-generated test cases by introducing different generation levels, with generation level 0 essentially equivalent to a set of hand-crafted codes. For a quick overview of the tool's capabilities, fewer test cases may be sufficient, but a thorough test of the tool's implementation can only be facilitated with automatically generated codes.

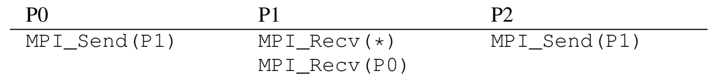
V. Challenges of Non-Determinism

Some programs may only show errors in specific execution paths. For example, all codes in MBB first check if they are launched with a sufficient number of processes and abort if not. In that case, a dynamic correctness checking tool would not report any error as the observed MPI usage is correct in this context. While it may be unfair to expect a dynamic tool to flag such not observable errors, these tests can showcase the potential of static tools to detect all possible errors, even if they are not observed in a single execution.

Apart from errors that are deterministically not observable, complex distributed programs have several possibilities for non-deterministic behavior, especially when considering MPI+X programming models, like MPI+OpenMP, as also investigated in [3]. This leads to challenges regarding the correctness of the application, as all possible interleaving of the operations performed by the different processes or threads should be correct. For non-deterministic errors, however, the erroneous behavior may not be observable, depending on the interleaving of the processes involved, the MPI implementation used, compiler optimizations or input parameters.

Our MPI-only error classification of MPI-BugBench [10] already has three non-determinism errors: Message race, local concurrency, and global concurrency. An example of a message race error due to an MPI wildcard receive is shown in Figure 2. A high quality correctness checking tool may be able to distinguish between a code that was correct in one specific interleaving and a code that is actually correct (in all possible interleavings). In order to test this, a correctness benchmark should therefore be designed in a way that some executions are very likely to produce the error, while others are unlikely to do so. Currently, all incorrect codes in MPI-BugBench have an error that is expected to be always "observable". In the future, we want to extend MBB to test for the tool's ability to detect the non-deterministic errors. This will require further refinement of the evaluation metrics, to account for tools that always, occasionally, or never report a non-deterministic error. Also, some erroneous usages of MPI result in undefined behavior, e.g., data races. In those cases, the compiler is allowed to completely change the behavior, potentially in a way that no error is observable. This inherent challenge in testing tools with undefined behavior ultimately cannot be solved by a correctness benchmark.

The considerations regarding non-determinism are also important for other correctness benchmarks such as DataRaceBench [13], especially, as most erroneous cases of DataRaceBench result in undefined behaviour [15].



P0	P1	P2
MPI_Send(P1)	MPI_Recv(*) MPI_Recv(P0)	MPI_Send(P1)

Fig. 2: MPI example with non-deterministic behavior: If the message of P2 matches the wildcard receive first, then the execution runs through. If the message of P0 matches the wildcard receive first, the execution deadlocks.

VI. Conclusion

This paper addresses the different challenges of designing correctness benchmarks, namely imbalanced test sets, ambiguity in the evaluation of error reports, complexity of the test case generation, and evaluation of non-deterministic errors, using MPI-BugBench as an example. Current MPI correctness benchmarks consist of more incorrect than correct codes. This imbalance should be considered with care, as metrics like accuracy favor tools that report more errors, including false positives. To quantify the helpfulness of tool error reports, we proposed new metrics that consider the reported error source location, error class, and the number of unrelated error reports (noise). The evaluation of ITAC, MUST, and PARCOACH against MPI-BugBench shows that most error reports help identify and fix errors. MUST achieves the highest number of helpful reports with roughly 97%, while ITAC and PARCOACH achieve roughly 88%. However, there is still room to improve the feedback quality for all tools.

Non-deterministic errors in MPI programs are another challenge, requiring refined evaluation metrics. In future work, we plan to incorporate tests in MPI-BugBench where the error is not always observable and verify if a tool always, occasionally, or never reports such errors. The considerations in this paper are not limited to MPI and can also apply to other correctness benchmarks like DataRaceBench [13].

Artifact Description

This section gives details on how to reproduce the results presented in the paper. The experiments are based on the MPI-BugBench, available at https://git-ce.rwth-aachen.de/hpc-public/mpi-bugbench (commit 986f0fff5ed72dfb4975d2b2f40e55c6ce214e4f).

MPI-BugBench contains three Docker images that automatically install each tool with their dependencies. We use python scripts to compile and execute all codes, analyze tools feedbacks and generate metrics. To build and run a docker image (<tool> should be replaced by parcoach, must or itac):

```
docker build -f Dockerfile.<tool> -t mbb:latest .
docker run -v `pwd`:/MBB -it mbb:latest bash
cd /MBB
```

Once in the docker image and in the MBB folder, level 2 codes can be generated with the command

```
python3 MBB.py -c generate --level 2
```

Then you can run a tool on the codes with the command

```
python3 MBB.py -c run -x <tool> -t <timeout in
    sec> -l <logs dir> -n <number of workers>
```

<timeout in seconds> gives the time limit when executing a code, <logs dir> is the path where log files will be saved, and <number of workers> is the size of the pool of workers that execute the tests in parallel. For the experiments of this paper, we used a time limit of 120 seconds and 16 workers. As executing the three tools on all codes takes several hours, all tools results can be found in https://git-ce.rwth-aachen.de/hpc-public/mpi-bugbench-results/-/tree/main/logs-20240723-151721.

After downloading our results or generating them as described above, statistics of a tool can be collected with

```
python MBB.py -l <logs dir> -x <tool> -c csv
```

Figure 1 in Section III is generated with the command

```
python MBB.py -l <logs dir> -x <tool> -c plots
```

References

[1] J.-P. Lehr, T. Jammer, and C. Bischof, "MPI-CorrBench: Towards an MPI Correctness Benchmark Suite," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC'21. ACM, 2021, pp. 69–80.

[2] M. Laurent, E. Saillard, and M. 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.

[3] T. Jammer, A. Hück, J.-P. Lehr, J. Protze, S. Schwitanski, and C. Bischof, "Towards a Hybrid MPI Correctness Benchmark Suite," in *Proceedings of the 29th European MPI Users' Group Meeting*. ACM, 2022, pp. 46–56.

[4] S. Schwitanski, J. Jenke, S. Klotz, and M. 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*, ser. SC-W '23. ACM, 2023, pp. 205–214.

[5] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. 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.

[6] E. Saillard, H. Brunie, P. Carribault, and D. Barthou, "PARCOACH Extension for Hybrid Applications with Interprocedural Analysis," in *Tools for High Performance Computing 2015*, A. Knüpfer, T. Hilbrich, C. Niethammer, J. Gracia, W. E. Nagel, and M. M. Resch, Eds. Springer, 2016, pp. 135–146.

[7] Intel, "Intel Trace Analyzer and Collector," https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/trace-analyzer.html, 2023, accessed: 2024-05-05.

[8] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. 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*, ser. SC '19. ACM, 2019.

[9] A. Hück, T. Jammer, J. Protze, and C. Bischof, "Investigating the Usage of MPI at Argument-Granularity in HPC Codes," in *Proceedings of EuroMPI2023: the 30th European MPI Users' Group Meeting*, ser. EuroMPI2023. ACM, 2023, pp. 1–10.

[10] T. Jammer, E. Saillard, S. Schwitanski, J. Jenke, R. Vinayagame, A. Hück, and C. Bischof, "MPI-BugBench: A Framework for Assessing MPI Correctness Tools," in *Proceedings of the 31st European MPI Users' Group Meeting*, ser. EuroMPI/Australia '24. Springer, to appear.

[11] Q. Gu, L. Zhu, and Z. Cai, "Evaluation measures of the classification performance of imbalanced data sets," in *Computational Intelligence and Intelligent Systems: 4th International Symposium, ISICA 2009, Huangshi, China, October 23-25, 2009. Proceedings 4*. Springer, 2009, pp. 461–471.

[12] P. Huchant, E. Saillard, D. Barthou, and P. Carribault, "Multi-Valued Expression Analysis for Collective Checking," in *EuroPar*, Göttingen, Germany, Aug. 2019.

[13] P.-H. Lin and C. Liao, "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)*, 2021, pp. 1–8.

[14] A. Hück, T. Jammer, J. Jenke, and C. 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*, ser. SC-W '23. ACM, 2023, pp. 230–233.

[15] J. Jenke and S. Schwitanski, "Improve and Stabilize the Classification Results of DataRaceBench," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 234–237.