The MPI BUGS INITIATIVE: a Framework for MPI Verification Tools Evaluation

Mathieu Laurent, Emmanuelle Saillard, Martin Quinson

The MPI BUGS INITIATIVE: a Framework for MPI Verification Tools Evaluation

Mathieu Laurent
Inria Rennes
Rennes, France
mathieu.laurent@ens-rennes.fr

Emmanuelle Saillard
Inria Bordeaux Sud-Ouest
Bordeaux, France
emmanuelle.saillard@inria.fr

Martin Quinson
University of Rennes, Inria, CNRS, IRISA
Rennes, France
martin.quinson@ens-rennes.fr

***Abstract*—Ensuring the correctness of MPI programs becomes as challenging and important as achieving the best performance. Many tools have been proposed in the literature to detect incorrect usages of MPI in a given program. However, the limited set of code samples each tool provides and the lack of metadata stating the intent of each test make it difficult to assess the strengths and limitations of these tools. In this paper, we present the MPI BUGS INITIATIVE, a complete collection of MPI codes to assess the status of MPI verification tools. We introduce a classification of MPI errors and provide correct and incorrect codes covering many MPI features and our categorization of errors. The resulting suite comprises 1,668 codes, each coming with a well-formatted header that clarifies the intent of each code and specifies how to execute and evaluate it. We evaluated the completeness of the MPI BUGS INITIATIVE against eight state-of-the-art MPI verification tools.**

***Index Terms*—Verification, MPI, Benchmarks, Tools**

I. INTRODUCTION

In the recent past, the vast majority of MPI applications were exhibiting rigid communication patterns and deterministic executions, to ensure both maximal performance and correctness. The focus of programmers was solely on the application performance, as application correctness was simple to test. As modern computing facilities reach tremendous dimensions, it becomes near to impossible to ensure perfectly stable and homogeneous hardware conditions, even over shorter execution time spans. Nowadays, large MPI applications must cope with variations in the environment through dynamic and non-deterministic execution patterns. As a result, the correctness of MPI programs of increasing complexity and size becomes as challenging and important as achieving the best performance [1].

MPI programs can exhibit many types of errors, that can be either classical synchronization errors (leading to deadlocks and data races) or more specific, such as mismatches between the parameters of corresponding MPI calls (data type or root of collective communications). Some MPI calls also introduce specific restrictions. For example, collective operations must be called in the same order by all processes, resource handlers should be freed appropriately, and remote memory access (RMA) fences must be correctly paired. Some potential MPI faults are more subtle, depending on obscure conditions. For example, two concurrent blocking sends from A to B and from B to A may or may not result in a deadlock depending on whether the data fits in the system buffers. MPI portability introduces another set of challenges, where a given application can fail only with some specific MPI implementations.

Many tools have been proposed in the literature to detect errors in MPI programs, leveraging diverse approaches such as static analysis [2]–[4], dynamic analysis [5]–[9], or symbolic analysis [10]–[13]. Some of these approaches are better suited to detect some classes of errors, and authors may further specialize their tool to a subset of the potential errors. This situation makes it very difficult to evaluate the proposed tools, both for the users wanting to select the tool fitting their needs, but also for the authors wanting their tool to meet users' needs. One could rely on the test suites provided by each tool, but the lack of metadata often obfuscates the intent of each test. It is thus hard to judge whether a given sample collection properly covers the possible errors or is merely a set of redundant tests for a very specific error class. In this paper, we present the MPI BUGS INITIATIVE (MBI), a systematic and practical methodology to assess the status of MPI verification tools. This work makes the following contributions:

- We introduce a new classification of MPI errors depending on their root cause.
- We propose a collection of codes covering the core features of MPI (point-to-point, collectives, etc.) and more advanced features, such as RMA. We provide both correct and incorrect codes to cover our error classification;
- We use our methodology to assess and compare eight state-of-the-art verification tools from the literature.

The rest of the paper is organized as follows. Section II discusses related work on existing MPI test suites and MPI tools comparison. Section III describes our methodology, centered around a classification of MPI errors and features. Section IV details the corresponding suite of MPI sample codes. Section V uses this methodology on existing verification tools while Section VI summarizes our findings and future work.

II. RELATED WORK

Assessing the quality of the MPI runtime implementations is a significant concern for decades, resulting in countless tools and benchmarks on the Internet. Most of the works described in the scientific literature focus on the performance of MPI implementations [14], or a subset of features [15], [16]. Thoroughly testing the correctness and standards conformance

of the MPI implementations is an engineering challenge of its own [17]. On the application side, many tracing and profiling tools have been proposed. Several surveys propose a qualitative assessment of such MPI profilers and debuggers, through an empirical evaluation of the graphical interface, the documentation, the usefulness of the provided feedback, etc [18]–[20]. Our work in contrast aims at constituting an automated test suite providing a quantitative evaluation of correctness assessment tools.

The Run-Time Error Detection suite (RTED – [21]) is very relevant, even if dated, to our work (the code samples can only be found from the Internet Archive project). It aims at proposing a quantitative evaluation of the runtime feedback to programming errors in several contexts. The authors gathered ten thousand code samples, organized in four suites containing serial, MPI, OpenMP, or UPC codes. The MPI suite proposes 1,942 code samples (in C, C++, or Fortran 90) falling into 10 categories: buffer out of bounds, buffer overlap, data type errors, rank errors, other argument errors, wrong order of MPI calls, negative message length, deadlocks, race conditions, implementation-dependent errors (such as potential deadlocks and race conditions). Each code contains exactly one described error, but no correct code is provided. Surprisingly, this work was almost never referenced in the context of MPI correctness. To the best of our knowledge, the only references to RTED in a context including MPI seems to be [22], a survey misrepresenting RTED as being limited to OpenMP.

Our goals are similar but not perfectly aligned with the ones of the RTED benchmark. First, RTED is strongly focused on the quality of the produced error diagnosis while we are more interested in quantifying the error detection ability of the tools. The RTED harness does not report the tests for which the tool fails to produce a diagnosis, hiding the fact that some tools only support a small subset of MPI. Since RTED does not contain any correct code, it is also unable to evaluate whether a given tool tends to diagnose errors in correct codes. Second, the chosen error categories also denote a differing focus. Some of the RTED categories target the MPI runtime and check the buffer and memory handling or enforce that no message length is negative. Issues requiring a global view (deadlocks and message races) constitute a common concern between RTED and our work, as both are difficult to detect in production settings and often require external tools. Errors resulting from the dynamic communication patterns that become prevalent on modern systems (such as *potential* synchronization issues) are badly represented in RTED. Similarly, RTED is limited to MPI-2 features while we propose tests of the RMA features in MPI-3. Third, RTED exhibits a very strong bias toward dynamic analysis. "The tests were written so that the information needed to detect the error is not available at compile-time" [21, p3], deliberately defeating static and symbolic analysis. Our work does not have such a methodological bias and was successfully used to evaluate tools relying on static, dynamic, or symbolic analysis.

In [23], the authors present a benchmark to evaluate the ability of general-purpose model checkers to find deadlocks in MPI programs. [13] is used to automatically extract models from the considered MPI programs. Mutation techniques [24], [25] are used to generate several thousands of models by injecting artificial errors into the 10 considered MPI applications, and the resulting models are translated in the formalism of the tested model checkers (PAT, FDR, Spin, PRISM, and NuSMV). The impressive amount of models included in this benchmark is hindered by the fact that the redundancy between these models is not evaluated. Besides, this collection seems to be limited to point-to-point MPI communications only.

MPI verification tools are usually evaluated separately, without being compared to the literature. Hermes [11] is an exception, as the authors experimentally compared their tool to several competing solutions (ISP [6], Mopper [7], Aislinn [5], and CIVL [10]). This study is unfortunately limited to the detection of deadlocks in MPI programs.

MPI-CorrBench [26] is a recent MPI benchmark suite containing 510 small-scale programs and 3 mini-apps with errors. The proposed programs fall in four categories: correct code, erroneous arguments, erroneous program flow, and mismatching arguments across communicator. This benchmarks are used to compare four tools from the literature (MUST, ITAC, MPI-Check and PARCOACH). The main differences with the MPI BUGS INITIATIVE is that our error categorization contains more kind of errors (e.g. covering message races, resource leaks and concurrency issues), and that we evaluate eight tools, but the MBI does not contain tests of the size of a mini-app. As a result, the MPI-CorrBench and MBI efforts are very complementary.

We believe that the MPI BUGS INITIATIVE will be highly beneficial to the potential users and tool authors targetting. This effort results from the collaboration of two unrelated teams developing such tools, and we reported several issues found through our work to the authors of the other tools. We hope that others will join this effort in the future.

III. METHODOLOGY

The MBI main goal is to measure the qualitative and quantitative performance of MPI verification tools in a reproducible and automatic way. With that goal in mind, we first identify errors that can arise in MPI programs and classify them following the errors taxonomy described in [27]. To automate the evaluation and ensure our benchmark suite is representative of real-world applications, we annotate all programs with the expected error, if any, and feature labels.

For each tool, we provide the number of errors correctly reported (TP), missed (FN), incorrectly reported (FP) and the number of programs correctly detected as error-free (TN). We also compute five standard metrics that define tools performance [28], [29]: Recall, Specificity, Precision, Accuracy, and F1 score. Recall is the ratio of true positives out of the total number of correctly and incorrectly reported errors while Specificity gives the ratio of true negatives out of the total

MPI Feature Label		Description	Number of codes using the label	
			# Incorrect codes	# Correct codes
P2P	base calls	Use of blocking point-to-point communication)	132	14
	nonblocking	Use of nonblocking point-to-point communication	114	13
	persistent	Use of point-to-point persistent communications	65	10
COLL	base calls	Use of blocking collective communication	542	468
	nonblocking	Use of nonblocking collective communication	529	480
	tools	Use of resource function (e.g., communicators, datatypes)	102	32
RMA		Use of Remote Memory Access	42	3

TABLE I: List of MPI feature labels. The two last columns give the number of correct and incorrect programs using the labels (codes may use several labels). P2P and COLL respectively stand for point-to-point and collective.

```
P0          P1          P2
recv(any)   send(0)   send(0)
send(1)     recv(0)
recv(any)
```

(a) Potential send-recv cycle (Message Race)

```
P0              P1
                window buffer X
Win_fence       Win_fence
Put(buf,1,X)
buf[0]=8
Win_fence       Win_fence
```

(b) RMA datarace (Local Concurrency)

```
P0                P1                  P2
                  window buffer X
Win_lock_all      Win_lock_all        Win_lock_all
Put(buf1,1,X)                         Put(buf2,1,X)

Win_unlock_all    Win_unlock_all      Win_unlock_all
```

(c) Conflicting Put operations (Global Concurrency)

```
int root = 22;
MPI_Comm com = MPI_COMM_WORLD;
MPI_Reduce(&myrank,&sum,1,MPI_INT,MPI_SUM,root,com);
```

(d) Root not in the communicator (Invalid Parameter)

```
MPI_Comm com = MPI_COMM_WORLD;
MPI_Datatype conti;
...
MPI_Bcast(outbuf, 6 / (3 - rank), conti, rank, com);
```

(e) Root mismatch in a broadcast function (Parameter Matching)

```
...
MPI_Ireduce(&sum, &val, 1, MPI_INT, MPI_SUM,
    root, MPI_COMM_WORLD, &req);
...
MPI_Request_free(&req);
```

(f) Missing wait (Request Lifecycle)

```
void myOP(void* in,void* inout,int* len,MPI_Datatype* dptr);
MPI_Op newop[OP_COUNT];

for (i = 0; i < OP_COUNT; i++)
  MPI_Op_create(myOP, 1, &newop[i]);
```

(g) Missing free after an operation creation (Resource Leak)

Fig. 1: Examples of different erroneous situations.

number of true negatives and false positives. The precision metric is the ratio of true positives out of the total number of errors reported. Accuracy measures the tool ability to correctly report out of all positive and false reports. Finally, the F1 score corresponds to the harmonic mean of precision and recall. In addition to these standard metrics, we define two new metrics to evaluate tools ability to draw a diagnostic on codes (Conclusiveness) and to compile codes (Coverage).

A. Feature Labels

We define 7 feature labels representing the way MPI processes exchange messages. Each label is either set to *Yes*: use of the feature, or *Lacking*: the feature is missing. The list of all feature labels is given in Table I (first three columns) and a full description of each one is detailed below. Note that a code can have multiple features.

1) Point-to-point: base calls and nonblocking: A point-to-point communication involves two MPI processes. It can be either blocking (i.e., the call returns when the resources of the communication can be reused), referred as base calls, or nonblocking (i.e., the call returns as soon as it can), referred as nonblocking. An MPI nonblocking operation is composed of two procedure calls: an initialization of the form `MPI_I<operation>` (e.g., `MPI_Isend`) and a completion (`MPI_Wait` or `MPI_Test`). Between these two operations, the data buffer is in a vulnerable state. A misuse of blocking and nonblocking point-to-point communications includes unbalanced communications (e.g., a send without a corresponding receive operation) and the use of invalid arguments. When using nonblocking point-to-point communication, an error can also occur if a completion call is missing, the buffer involved in the communication is modified before the operation is completed or if requests are not correctly used.

2) Point-to-point: persistent: Persistent communication is composed of four procedure calls. The first procedure initiates the communication (e.g., `MPI_Send_init`). The second procedure starts the communication (e.g., `MPI_Start`). The third and fourth procedures respectively complete and free the operation. A persistent communication is misused when called with invalid arguments, if the completion operation is missing or if the buffer involved in the communication is modified between the start and the completion calls.

3) Collective: base calls and nonblocking: A collective is a communication that involves all processes in a communicator. These labels are restricted to usual collectives communication only (e.g., `MPI_Barrier`, `MPI_Bcast`, `MPI_Ireduce`). The MPI specification requires that all processes of a communicator have the same sequence of blocking and nonblocking collectives. These operations are incorrect if communication buffers overlap, if all processes in a communicator do not call the same collectives in the same order, or if collectives are called with incompatible or invalid arguments. Note that

it is not allowed to match a blocking collective with its nonblocking counterpart; such situation is erroneous.

4) Collective: tools: This label comprises communicator and group management operations (e.g., `MPI_Comm_create/free`), topology creation operations like `MPI_Cart_create` that virtually organize processes in a multi dimensional grid, and datatypes and operators creation operations (e.g., `MPI_Op_create` to create operators for use in reduce functions). These operations must be called by all processes in the involved communicator otherwise a deadlock can occur. Furthermore, these operations are incorrect in case of improper construction and/or destruction of the MPI resource or if they are called with invalid arguments.

5) RMA: The Remote Memory Access (RMA) allows processes to expose a part of their memory (called *window*) in order to perform one-sided communications (e.g., `MPI_Get`). These communications are performed inside an *epoch* to ensure synchronizations. MPI RMA defines two kind of synchronization: active synchronization where the target takes part in the epoch creation (e.g., with `MPI_Win_fence`) and passive synchronization (e.g., with `MPI_Win_lock` and `MPI_Win_unlock`) where the target does not take part of the epoch creation. A misuse of RMA communication includes wrong RMA initialization/destruction operation usage (e.g., if `MPI_Win_create` is not called by all processes), a concurrency in memory accesses inside an epoch or a one-sided communication called with invalid arguments (e.g., if the target memory that is not part of the window).

B. Error Labels

Many errors can occur in a MPI program. Based on what was proposed in previous work [21], [27], we define 8 types of errors that can occur in a MPI program depending on the root cause: Invalid Parameter, Resource Leak, Request Lifecycle, Local Concurrency, Message Race, Parameter Matching, Call Ordering and Global Concurrency. These errors are categorized according to the scope in which they manifest: single call, single process and multi-processes.

1) Errors in single calls: These errors are only related to local MPI functions and can be detected by only analyzing the parameters of a given MPI function.

a) Invalid Parameters: This category contains errors such as invalid root, communicator, operator, datatype, tag or buffer length. The use of invalid parameters in MPI functions is incorrect and is reported by the MPI runtime.

In Figure 1d, processes call a reduce function with process 22 as root. If the program is launched with less than 23 processes, the root would be invalid. For this code, the MPI runtime reports the following message: `MPI_ERR_ROOT: invalid root`.

2) Errors local to a process: These errors often consist of an inconsistency between the local context of a process and the parameters to a given MPI call in that process. Detecting them thus requires analysis of local process information.

a) Resource Leak: Any improper destruction of MPI resources (e.g., datatype, request, communicators) leads to a resource leaking. As an example, the code snippet in Figure 1g constructs `OP_COUNT` MPI operators without freeing them.

b) Request Lifecycle: Request lifecycle is used for a missing wait or start function. Figure 1f shows a nonblocking reduce function that does not have a `MPI_Wait` associated.

c) Local Concurrency: A local concurrency occurs when a process accesses a memory region that is being read or written. This type of error is easy to produce with nonblocking and one-sided communication. For RMA operations, we consider concurrency errors inside an epoch and within a process. Figure 1b gives an example of a local memory consistency error within an epoch: Two operations are accessing variable `buf`, the Put operation reads `buf` while a store writes on `buf`.

3) Multi-processes errors: The next four errors result from multiple processes of the application, either local to a given communicator or globally.

a) Message Race: The use of wildcard receive calls can lead to nondeterministic matching with potential senders at runtime. An example is presented in Figure 1a. In this code, a deadlock arises if P0 receives from P2 first. We label these situations as message races.

b) Parameter Matching: Parameter matching corresponds to MPI calls matched with incompatible arguments. This happens for example when each rank gives its ID as root in a broadcast function as illustrated in Figure 1e. This code results either in a numerical instability or a deadlock, depending on the MPI implementation.

c) Call Ordering: Different scenarios can lead to a call mismatch: Any pattern causing a cyclic dependency (e.g., send-receive cycle) and a point-to-point or collective mismatch.

d) Global Concurrency: Global concurrency occurs when two or more processes access the same memory region and at least one access is a write. Such errors are produced with RMA operations inside an epoch and between processes. Figure 1c shows an example of two conflicting `MPI_Put` operations. Processes 0 and 2 access the same window buffer X, located on process 1.

IV. The MPI Bugs Initiative

We built the MPI Bugs Initiative to cover all errors and features categorization described in the previous section. Particular attention was paid to ensure the suite is representative and complete.

A. Benchmark Description

All codes in MBI are automatically generated from Python scripts to ensure a systematic coverage of the errors and features. Our benchmark contains $1,668$ original programs. 986 programs have errors and 682 are known to be error-free. Note that error-free programs are important for false

		Point-to-point			Collective			RMA	Unique files
		base calls	nonblocking	persistent	base calls	nonblocking	tools		
Single call	Invalid Parameter	48	36	36	33	33	55	12	154
Single process	Resource Leak	0	1	2	0	0	14	0	16
	Request lifecycle	0	4	5	0	12	0	0	18
	Local concurrency	2	3	3	0	11	0	18	37
Multi-processes	Parameter matching	27	19	19	29	29	33	0	97
	Message Race	3	3	0	0	0	0	0	4
	Call ordering	52	48	0	480	444	0	0	648
	Global concurrency	0	0	0	0	0	0	12	12
Correct codes		14	13	10	468	480	32	3	682
Total		146	127	75	1010	1009	134	45	1668

TABLE II: Errors classification. Numbers indicate the number of codes with the feature and the error. The last row shows the number of correct codes with each feature (a code can have multiple features).

positives reporting. All codes are written in C and have a formatted header providing a textual description of the problem, the list of MPI features used, the expected errors, and how to evaluate them. Each incorrect code produces only one error: all codes were verified with the Valgrind tool and we manually checked the output of all bug finding tools. We made the suite extensible so new codes, errors and features can easily be added. The MBI can be downloaded from https://gitlab.com/MpiBugsInitiative.

B. Features and Errors Coverage

As mentioned in section III-B, the occurrence of an error may depend on several parameters including the MPI implementation, the number of processes, the system buffering and the program input data. We associate each code in the benchmark to one test. A test corresponds to an execution command line with specific parameters including the number of processes. Table I gives the number of correct and incorrect programs using each feature (a program can have multiple feature labels) and Table II shows the feature usage among all correct and incorrect codes. The last row of table II depicts the total number of correct programs and incorrect programs per error. It is worth mentioning that some features are irrelevant to some errors (e.g., RMA operations cannot produce a Request Lifecycle error). It may be observed that all MPI features are represented in the benchmark with a predominance of basic collectives and point-to-point operations which correspond to the most used features in HPC applications [30].

V. Tools Comparison

We use the MBI to evaluate Aislinn (v3.12), CIVL (v1.20), ISP (v0.3.1), ITAC [31] (v2021.3), Mc SimGrid [9] (v3.28), MPI-SV [13] (v1), MUST (v1.6) and PARCOACH (v1.2) as they are active projects, available and use different techniques.

Aislinn focuses on deadlock detection caused by run-time scheduling problem among operations. To do so, it realizes a symbolic execution with Partial Order Techniques. Aislinn supports both zero and infinite buffering modes. CIVL combines symbolic execution and model checking to detect communication deadlocks. It can detect data races and assertion violations but does not support nonblocking operations. Like CIVL, MPI-SV uses symbolic execution and model checking to detect communication deadlocks in MPI programs. ISP checks if a program satisfies a given property (e.g., liveness, communication determinism) by considering all possible executions. Like all model checkers, it faces a state space explosion problem. Mc SimGrid is a model checker integrated to SimGrid, a framework mostly dedicated to predicting the performance of distributed applications. It explores all the possible execution paths of an MPI application, starting from an initial application configuration with a fixed set of inputs. Mc SimGrid verifies safety properties, liveness properties, and other properties such as communication determinism. MUST is the successor of Marmot [32] and Umpire [33]. It intercepts all MPI operations to perform online checking. It is based on GTI (generic tool infrastructure) and can detect multiple errors like deadlocks, type mismatches or resource leaking. Intel Trace Analyzer and Collector (ITAC) uses a similar approach to MUST. It is implemented as a library that performs error detection at runtime and records error reports for later analysis. PARallel COntrol flow Anomaly CHecker (PARCOACH) combines a static analysis with a code instrumentation to detect misuse of MPI collectives [34] as well as nonblocking and persistent communications [3]. Although it uses a precise data- and control-flow interprocedural analysis to pinpoint root cause problems, it may lead to many false positives. In this work we only consider the PARCOACH static phase which is implemented as a LLVM pass.

A. Experimentation Setup

The experiments were performed in a Docker image with Ubuntu version 20.04 containing all tools dependencies, except for Aislinn which was run with Ubuntu 18.04 due to dependency issues in the newer Ubuntu version and MPI-SV which was run with Ubuntu 14.04.6 (we use the docker image provided by the authors). All tests use at most 6 processes to produce the expected errors. The use of Docker facilitates the benchmark modifications and the installation of the tools we have selected. All results were obtained with MPICH v3.3.2.

	Correct execution				Invalid parameter				Resource leak				Request lifecycle				Local concurrency				Parameter matching				Message race				Call ordering				Global concurrency			
	Build error	Runtime error	False **Positive**	True **Negative**	Build error	Runtime error	False Negative	True Positive	Build error	Runtime error	False Negative	True Positive	Build error	Runtime error	False Negative	True Positive	Build error	Runtime error	False Negative	True Positive	Build error	Runtime error	False Negative	True Positive	Build error	Runtime error	False Negative	True Positive	Build error	Runtime error	False Negative	True Positive	Build error	Runtime error	False Negative	True Positive
Aislinn	385	0	0	297	42	0	0	112	0	0	16	0	6	0	0	12	24	0	5	8	24	11	22	40	0	0	4	0	367	4	0	277	**12**	**0**	**0**	**0**
CIVL	534	14	1	133	100	47	3	4	13	1	0	2	18	0	0	0	37	0	0	0	58	21	0	18	3	0	0	1	521	5	3	119	**12**	**0**	**0**	**0**
ISP	0	1	479	202	0	5	8	141	0	0	8	8	0	0	12	6	**0**	**0**	**21**	**16**	0	1	19	77	**0**	**0**	**0**	**4**	0	1	6	641	**0**	**0**	**12**	**0**
ITAC	**0**	**0**	**0**	**682**	**0**	**0**	**2**	**152**	0	0	14	2	0	0	4	14	0	0	24	13	0	0	3	94	0	0	1	3	**0**	**0**	**0**	**648**	**0**	**0**	**12**	**0**
Mc SimGrid	0	7	0	675	0	12	0	142	0	0	2	14	**0**	**0**	**2**	**16**	0	18	14	5	0	0	4	93	**0**	**0**	**0**	**4**	0	4	261	383	**0**	**12**	**0**	**0**
MPI-SV	494	3	4	181	36	16	94	8	0	0	16	0	12	0	5	1	11	18	8	0	31	0	55	11	0	0	4	0	459	4	83	102	**0**	**12**	**0**	**0**
MUST	0	0	1	681	0	4	0	150	**0**	**0**	**0**	**16**	0	0	4	14	0	0	32	5	**0**	**0**	**0**	**97**	0	0	3	1	0	5	0	643	**0**	**0**	**12**	**0**
PARCOACH	441	0	182	59	36	0	118	0	0	0	16	0	11	0	7	0	17	0	20	0	28	0	69	0	0	0	4	0	407	0	24	217	**4**	**0**	**8**	**0**
Ideal tool	*0*	*0*	*0*	*682*	*0*	*0*	*0*	*154*	*0*	*0*	*0*	*16*	*0*	*0*	*0*	*18*	*0*	*0*	*0*	*37*	*0*	*0*	*0*	*97*	*0*	*0*	*0*	*4*	*0*	*0*	*0*	*648*	*0*	*0*	*0*	*12*

TABLE III: Results of each tool per error category.

Tool	Errors			Results				Robustness		Usefulness				**Overall accuracy**
	CE	**TO**	**RE**	**TP**	**TN**	**FP**	**FN**	**Coverage**	**Conclusiveness**	**Specificity**	**Recall**	**Precision**	**F1 Score**	
Aislinn	860	**0**	15	449	297	**0**	47	0.4844	0.4754	**1**	0.9052	**1**	0.7644	0.4472
CIVL	1296	**0**	88	144	133	1	**6**	0.223	0.1703	0.9925	**0.96**	0.9931	**0.9381**	0.1661
ISP	**0**	7	1	893	202	479	86	**1**	0.9952	0.2966	0.9122	0.6509	0.2535	0.6565
ITAC	**0**	**0**	**0**	**926**	**682**	**0**	60	**1**	**1**	**1**	0.9391	**1**	0.8256	**0.964**
Mc SimGrid	**0**	**0**	53	657	675	**0**	283	**1**	0.9682	**1**	0.6989	**1**	0.8319	0.7986
MPI-SV	1043	**0**	53	122	181	4	265	0.3747	0.3429	0.9784	0.3152	0.9683	0.517	0.1817
MUST	**0**	4	5	**926**	681	1	51	**1**	0.9946	0.9985	0.9478	0.9989	0.8277	0.9634
PARCOACH	944	**0**	**0**	217	59	182	266	0.4341	0.4341	0.2448	0.4493	0.5439	0.2225	0.1655
Ideal tool	*0*	*0*	*0*	*986*	*682*	*0*	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*

TABLE IV: Tools Evaluation against the MPI BUGS INITIATIVE benchmark.

Result	**Meaning**	**Result**	**Meaning**
True Positive (TP)	Error correctly detected.	Compilation Error (CE)	The code uses a feature not supported by the tool.
False Negative (FN)	Error missed.	Timeout (TO)	Timeout (limit: 300s).
False Positive (FP)	Correct code reported as faulty.	Runtime Error (RE)	Tool failure during the evaluation of this code.
True Negative (TN)	Correct code reported as such.	Total	Total = CE+RE+TO + TP+FP + TN+FN

Metric	**Definition and meaning**
Coverage	Ability to compile codes. $Cov = 1 - \frac{CE}{total}$
Conclusiveness	Ability to draw a diagnostic on codes. $Cc = 1 - \frac{CE+RE+TO}{total}$
Specificity	Ability to not find errors in correct codes. $S = \frac{TN}{TN+FP}$
Recall	Ability to find existing errors. $R = \frac{TP}{TP+FN}$
Precision	Potential confidence when a code is reported as correct. $P = \frac{TP}{TP+FP}$
F1 Score	Overall bug-finding quality. $F1 = \frac{2 \times P \times R}{P+R}$
Overall accuracy	Proportion of correct diagnostics over all tests. $A = \frac{TP+TN}{total}$

TABLE V: Results, Metrics and Abbreviations used. Metrics do not account for codes leading to a tool error.

One big challenge is to automate the evaluation and comparison of tools that use different techniques and outputs. To address this issue, we analyze the output of each tool independently and we have developed a Python script that associates messages reported by the tools with our errors categorization (e.g., a tool can state an error in many different ways but still be understandable by a user). More specifically, a runner compiles each tool except CIVL which is distributed in an executable form. Each program in the MBI is then compiled and executed with specific parameters as defined in programs header. To deal with time out, we limit the execution of each test at 5 minutes. This is a reasonable threshold as our tests do not take much time. All outputs are registered and analyzed by the runner: it searches for particular keywords specific to each tool. For instance, the message "`Fatal error in PMPI_Cart_get: Invalid topology`" from ISP is labelled with Invalid Parameter and any warning emitted by PARCOACH is considered as a Call Matching error.

B. Qualitative and Performance Evaluation

Table III shows the results of each tool on all tests per error category. For each error, we report the number of Build errors (i.e., number of codes that did not compile), Runtime

errors (i.e., number of codes that did not execute), the number of False Negative (the tool did not detect the error) and the number of True Positive (the tool detected the expected error). For correct codes, we report the number of Build errors and Runtime errors but also the number of False Positive (the tool reported an error on a correct code) and True Negative (the tool detected no error on a correct code). This allows to pinpoint specific features not supported by tools and highlight what errors a given tool can detect. In the table, results in bold indicate which tool has the best results for a type of error. The Errors columns of table IV regroup the number of compilation errors (Build), Timeouts and runtime errors (failure). The Results columns gather the total number of True positive (TP), False negative (FN), False positive (FP), and True negative (TN) per tool. The last row of the table depicts results an ideal tool would have. Basically, an ideal tool should have no error, false positive or false negative and detect all correct tests as true negatives and all incorrect tests as true positives. Table IV also gives the Coverage, Conclusiveness, Recall, Specificity, Precision, F1 Score and Overall Accuracy results. How to compute these metrics is detailed in Table V.

The first thing to notice is how hard it is to cover everything. Actually, no tool is able to detect all errors in the MBI programs and they all return false positives or negatives. Call ordering is the most commonly detected error type, not necessarily because it is the easiest, but certainly because it is the most common result of a MPI feature misuse. ITAC is the best performing tool with the highest overall accuracy, followed closely by MUST. ITAC and MUST both detect a wide range of errors and outperform on several types of errors: ITAC has the best results for correct execution, invalid parameter and call ordering while MUST is the best to detect resource leak and parameter matching. Despite achieving well on some error types, MUST appears to be bad on message races and local and global concurrency and ITAC has difficulties to find resource leak and global concurrency errors. Overall, table III recognizes a lack of RMA support (responsible of global concurrency) in all tools which could be something to improve. Mc SimGrid and ISP seem to be a good compromise with a few number of compilation and runtime errors and a large detection of many kind of errors. They fail on less than 1% of the tests. ISP does well on local concurrency detection and Mc SimGrid is the best on request lifecycle and message race. Aislinn reports about 46% of errors, detects 44% of correct tests and fails on 52% of codes. Few features are supported by CIVL, MPI-SV and PARCOACH. CIVL syntactic analysis returns a compilation failure on 78% of the tests. As it is focused on collective mismatch detection, PARCOACH returns a lot of false negatives and few true negatives. It fails on 57% and is able to detect 22% of errors. Note that recent work from [3] on point-to-point and persistent operations are not integrated to the latest version of the tool yet. MPI-SV detects 12% of errors and 27% of correct codes and fails on 63% of the tests. These results are related to the use of the multi-threaded library for MPI-1 called AzequiaMPI [35]. All results are available on the MBI website with meta-data on the tests and links to every test execution log. Our goal is to provide a precise feedback to help developers improve their tools.

The tools capacity to cover many errors is strongly related to the precision of their feedback. Indeed, Table III is built using a precise study of the messages returned by all tools. Aislinn and MUST both generate html outputs with execution graphs. On the opposite, the other tools require more effort to understand what is detected. We plan to work with tools developers in order to consolidate our study. To begin, we already report issues we found to tools developers. We also reported an error related to asynchronous collectives in MPICH. Beyond being able to detect errors, a tool should be capable of helping the developer to correct programs quickly. Experience has shown that few tools can precisely pinpoint what caused an error.

Our evaluation tends to reveal how user-friendly and easy to use the tools are. MUST is one of the easiest tools to use: a code is compiled with an usual compiler and launched with a single command line (`mustrun`). CIVL follows a comparable workflow, directly working on the source code. The other tools require a specific compiling pass and a run-time execution. Even if this isn't an issue while performing over a single file, it may harden the task of testing bigger projects.

VI. Conclusion

In this paper, we present the MPI Bugs Initiative, an MPI benchmark suite for the evaluation of MPI bug-finding and verification tools. We propose a new classification of MPI errors based on their root causes. Our benchmark provides both correct and incorrect code samples to cover this classification. Each code comes with a well-formatted header describing its intent and specifying how to test it. We used our benchmark on eight major state-of-the-art MPI verification tools: Aislinn, CIVL, ISP, ITAC, Mc SimGrid, MPI-SV, MUST, and PARCOACH. The evaluation compares the tools in terms of error coverage. Our results show that no tool is currently able to support all MPI features and detect all classes of errors. In our findings, ITAC achieves the best scores on the overall accuracy metric. The code samples and tools evaluated in this work have been dockerized with our test harness to improve the reproducibility of our findings. We plan to integrate more tools in our initiative and organize an MPI verification tools contest to stimulate tools development. We hope that the MPI Bugs Initiative will foster the verification and bug-finding tools that the users need to tame the application complexity induced by the large-scale and dynamic modern platforms. The MPI Bugs Initiative is an ongoing effort to build a comprehensive correctness benchmark suite and can be extended in many ways. First, we will add missing MPI features such as MPI IO and extend our errors categorization with errors depending on the buffer mode. Then, we intent to add new codes to further improve our coverage of features and errors. This could be achieved by using mutants or machine learning techniques. Finally, we plan to add larger codes to evaluate the tools scalability, either through larger synthetic codes or through a collection of real bugs in full-scale applications.

References

[1] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, and G. Bronevetsky, "Formal Analysis of MPI-Based Parallel Programs," *Commun. ACM*, 2011.

[2] H. Ma, L. Wang, and K. Krishnamoorthy, "Detecting Thread-Safety Violations in Hybrid OpenMP/MPI Programs," in *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*. IEEE Computer Society, 2015, pp. 460–463.

[3] V. M. Nguyen, E. Saillard, J. Jaeger, D. Barthou, and P. Carribault, "PARCOACH Extension for Static MPI Nonblocking and Persistent Communication Validation," in *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications*, 2020, pp. 31–39.

[4] S. F. Siegel and T. K. Zirkel, "Automatic Formal Verification of MPI-Based Parallel Programs," *SIGPLAN Not.*, vol. 46, no. 8, 2011.

[5] S. Böhm, O. Meca, and P. Jančar, "State-space reduction of non-deterministically synchronizing systems applicable to deadlock detection in MPI," in *FM 2016: Formal Methods*, J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou, Eds. Cham: Springer International Publishing, 2016, pp. 102–118.

[6] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "ISP: a tool for model checking MPI programs," in *PPOPP*, 2008.

[7] V. Forejt, S. Joshi, D. Kroening, G. Narayanaswamy, and S. Sharma, "Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs," *ACM Trans. Program. Lang. Syst.*, vol. 39, 2017.

[8] T. Hilbrich, M. Weber, J. Protze, B. R. de Supinski, and W. E. Nagel, "Runtime Correctness Analysis of MPI-3 Nonblocking Collectives," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: ACM, 2016, pp. 188–197.

[9] T. A. Pham, T. Jéron, and M. Quinson, "Verifying MPI Applications with Mc SimGrid," in *Correctness 2017 - First International Workshop on Software Correctness for HPC Applications*, 2017.

[10] 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*, Nov 2015, pp. 1–12.

[11] D. Khanna, S. Sharma, C. Rodríguez, and R. Purandare, "Dynamic symbolic verification of MPI programs," in *Formal Methods*, K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, Eds. Cham: Springer International Publishing, 2018, pp. 466–484.

[12] S. F. Siegel and G. S. Avrunin, "Verification of halting properties for MPI programs using nonblocking operations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, F. Cappello, T. Herault, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 326–334.

[13] H. Yu, Z. Chen, X. Fu, J. Wang, Z. Su, J. Sun, C. Huang, and W. Dong, "Symbolic Verification of Message Passing Interface Programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020.

[14] R. Reussner, P. Sanders, L. Prechelt, and M. Müller, "SKaMPI: A detailed, accurate MPI benchmark," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, V. Alexandrov and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[15] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07. Association for Computing Machinery, 2007.

[16] M. G. F. Dosanjh, T. L. Groves, R. E. Grant, R. Brightwell, and P. G. Bridges, "RMA-MT: A benchmark suite for assessing MPI multi-threaded RMA performance," in *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*. IEEE Computer Society, 2016.

[17] J. Hursey, E. Mallove, J. M. Squyres, and A. Lumsdaine, "An Extensible Framework for Distributed Testing of MPI Implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, F. Cappello, T. Herault, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 64–72.

[18] S. Moore, D. Cronk, K. London, and J. Dongarra, "Review of performance analysis tools for MPI parallel programs," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2001, pp. 241–248.

[19] A. De Sarkar and N. Mukherjee, "A Study on Performance Analysis Tools for Applications Running on Large Distributed Systems," *arXiv preprint arXiv:1006.2650*, 2010.

[20] G. R. Luecke, B. M. Groth, N. T. Weeks, and M. Kraeva, "Comparing Allinea's and Intel's performance tools for HPC," in *Proceedings of the 25th High Performance Computing Symposium*, 2017, pp. 1–12.

[21] G. R. Luecke, J. Coyle, J. Hoekstra, M. Kraeva, Y. Xu, M. Park, E. Kleiman, O. Weiss, A. Wehe, and M. Yahya, "The Importance of Run-Time Error Detection," in *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*. Springer, 2009, pp. 145–155.

[22] A. M. Alghamdi and F. E. Eassa, "Software testing techniques for parallel systems: A survey," *Int. J. Comput. Sci. Netw. Secur.*, vol. 19, no. 4, pp. 176–186, 2019.

[23] W. Hong, Z. Chen, H. Yu, and J. Wang, "Evaluation of Model Checkers by Verifying Message Passing Programs," *Sci. China Inf. Sci.*, vol. 62, no. 200101, 2019.

[24] H. Yu, "Combining Symbolic Execution and Model Checking to Verify MPI Programs," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. Association for Computing Machinery, 2018, p. 527530.

[25] R. Just, M. D. Ernst, and G. Fraser, "Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. Association for Computing Machinery, 2014.

[26] 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. New York, NY, USA: Association for Computing Machinery, 2020, p. 6980. [Online]. Available: https://doi.org/10.1145/3431379.3460652

[27] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov, "Automated, Scalable Debugging of MPI Programs with Intel Message Checker," in *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, ser. SE-HPCS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 7882.

[28] G. Verma, Y. Shi, C. Liao, B. M. Chapman, and Y. Yan, "Enhancing DataRaceBench for Evaluating Data Race Detection Tools," in *4th IEEE/ACM International Workshop on Software Correctness for HPC Applications, Correctness@SC, Atlanta, GA, USA, November 11, 2020*, I. Laguna and C. Rubio-González, Eds. IEEE, 2020, pp. 20–30.

[29] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz, "BegBunch: Benchmarking for C bug detection tools," in *Proceedings of the 2nd Intl Workshop on Defects in Large Software Systems: In conjunction with the ACM SIGSOFT Intl Symposium on Soft. Testing and Analysis (ISSTA 2009)*, 2009, pp. 16–20.

[30] 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*. Association for Computing Machinery, 2019.

[31] "Intel Trace Analyzer and Collector," https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/trace-analyzer.html, accessed: 2021-07-15.

[32] B. Krammer, M. S. Müller, and M. M. Resch, "Runtime checking of MPI applications with MARMOT," in *Mini-Symposium Tools Support for Parallel Programming, ParCo*, 2005, pp. 12–16.

[33] J. S. Vetter and B. R. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2000.

[34] P. Huchant, E. Saillard, D. Barthou, H. Brunie, and P. Carribault, "PARCOACH Extension for a Full-Interprocedural Collectives Verification," in *2nd Intl Workshop on Soft. Correctness for HPC Applications*, 2018.

[35] J. A. Rico-Gallego, J. M. Alvarez-Llorente, F. J. Perogil-Duque, P. P. Antnez-Gmez, and J. C. Daz-Martn, "A Pthreads-Based MPI-1 Implementation for MMU-Less Machines," in *2008 International Conference on Reconfigurable Computing and FPGAs*, 2008.

APPENDIX

A. Software availability and dependencies

The MPI Bugs Initiative (MBI) is available on Gitlab at https://gitlab.com/MpiBugsInitiative. The source code of all evaluated tools is available in the MBI repository, that is self contained. No additional files are needed beside of the Ubuntu online repositories that are used automatically.

Tool	Version	Compiler	Docker image
Aislinn	v3.12	GCC 7.4.0	ubuntu:18.04
CIVL	v1.20	*used with JDK-14*	MBI (ubuntu:20.04)
ISP	0.3.1	GCC 10.2.0	MBI (ubuntu:20.04)
ITAC	2021.3	2021.3	MBI (ubuntu:20.04)
Mc SimGrid	v3.28	GCC 10.2.0	MBI (ubuntu:20.04)
MPI-SV	v1	GCC 4.8.4	MPI-SV (ubuntu:14.04)
MUST	v1.6	GCC 10.2.0	MBI (ubuntu:20.04)
PARCOACH	v1.2	LLVM 9	MBI (ubuntu:20.04)

TABLE VI: Tools information: version, compiler and Docker image used.

B. Installation

We use Docker to facilitate our benchmark modifications and the installation of the tools we have selected. A user can launch all tests outside the docker image by using the `./test-all` command, that runs all steps described below.

The MBI experiments were performed in a Docker image with Ubuntu version 20.04 containing all tools dependencies, except for Aislinn which was run with Ubuntu version 18.04 due to dependency issues in the newer Ubuntu version and MPI-SV which was run in the docker image provided by the authors. A user can create the MBI docker image from the provided Dockerfile as follows:

```
docker build -f Dockerfile -t mbi:latest .
docker run -it mbi bash
```

Once inside the docker, the `/MBI/MBI.py` script provides a centralized interface to all tool-specific scripts (located under `/MBI/scripts/tools/<tool>.py`). It handles both the data provenance (producing execution logs for all tools) and the data analysis (producing figures out of the logs)

C. Data provenance

The source code of all tools are included in the Docker image, and they are compiled on need. The binaries are persistent out of the Docker environment, as a cache mechanism.

The test cases are generated using the following command:

```
python3 ./MBI.py -c generate
```

One can either run the tests for all tools, or chose a specific tool as follows:

```
python3 ./MBI.py -c run
python3 ./MBI.py -c run -x <tool>
```

All logs are produced under `/MBI/logs/<tool>`, that is persistent out of the Docker. In each directory, the following files are produced for the data analysis:

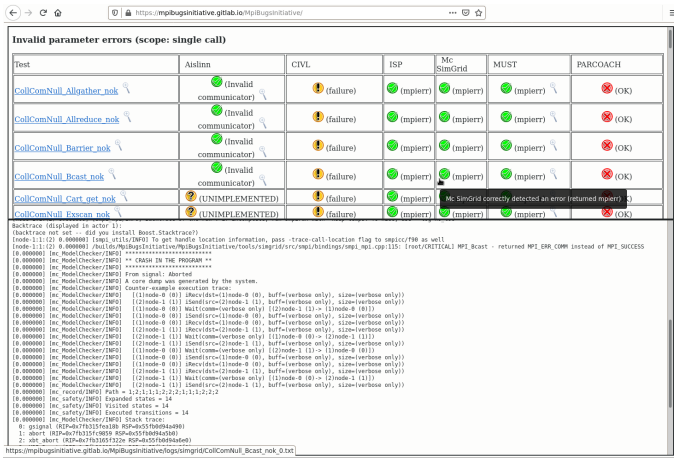

Fig. 2: Screenshot of the MBI internal dashboard.

- *test_name*.txt: that contains the tool output for that test
- *test_name*.elapsed: that gives the wallclock time

In addition, a *test_name*.md5sum file is used to detect changes in the test codes, and cache the test results when the code is unmodified.

D. Data analysis

Once all logs are in cache, the LaTeX tables included in this article are regenerated as follows:

```
python3 ./MBI.py -c latex
```

This is an error-prone component, and we manually checked the results on all generated codes, also comparing the observed outcomes of all bug-finding tools.

A web dashboard can be generated to explore the logs in cache as follows.

```
python3 ./MBI.py -c html
```

This is useful to debug the tool-specific scripts that parse the textual output of that tool, i.e. the component in charge of categorizing a given run as either as 'True Positive', 'False Negative' and so on. This dashboard provides two views presented in Figure 2. The upper part gives a quick glance on the categorization of each tool and test code (using colored icons), while the lower part allows to see the detail of all logs. This tool is only described in this reproducibility section, as it is meant as an internal tool to ensure the quality of the tool-specific parsers.

E. Continuous Integration

We rely on GitLab Continuous Integration features to enforce the reproducibility of the provided tooling. All results can be visualized at https://mpibugsinitiative.gitlab.io/MpiBugsInitiative/. MPI-SV and ITAC are not included in this dashboard, because of a technical difficulty. They fail with a SIGILL error when executed in the docker-in-docker settings that is mandated gitlab-ci. They must be run manually to produce the logs.