THE MPI BUGS INITIATIVE : EXTENTION FOR MPI HAZARD ERRORS

INTERNSHIP DISSERTATION

JULY 22, 2022

PIERRE-ANTOINE ROUBY

Master 2 Informatique Fondamental, Université of Bordeaux
Inria Bordeaux Sud-Ouest, STORM team

Internship supervisor:
EMMANUELLE SAILLARD

Inria Bordeaux Sud-Ouest, STORM team

&

MARTIN QUINSON

University of Rennes, Inria, CNRS, IRISA

ACKNOWLEDGEMENTS

First, a special thanks to my exceptional supervisors. Emmanuelle for allowing me to do this very enriching internship, the proposal of doctorate, your confidence in me, and finally for the tea. Martin for your precious advice about works and trolls.

Thanks to all the colleagues of the STORM team for the welcome, the chocolate, the cakes and restaurant. I think of Samuel, Nathalie, Amina, Mihail, Laércio, Denis, Olivier, Raymond, PAW, Marie-Christine, Sabrina, and Scott.

Thanks to all the people of the OpenSpace (or *Open Sieste*) for the atmosphere. Diane for the tea breaks. Baptiste for the beers and the political discussions. Radja for the distraction when we try to work. All the Charles (*Charles premier* and *Charles second*) for the philosophical questions and answer. Célia for the chocolate and cakes. Gwenolé for your infinite wisdom. Chiheb for your boundless charisma. *VentiloMan* for no reason… I also think of Maxime, Alice, Romain, Pélagie, Thomas, Edgar, Van Man, Kun, and Bastien.

Thanks to other team members. Hugo, for all the babyfoots games. You're not so bad now.

CONTENTS

List of Figures

List of Tables

List of Listings

I. Introduction

This dissertation is written in the context of my end-of-study internship of *master* in computer science. I made this formation with the specificity *informatique fondamental*, this specificity focuses on fundamental aspect of computer science, like algorithmic, abstract modeling and automatic verification of complex systems. In this formation I studied some theory aspect, like we study state-of-arts verification methods. In addition, during my first year of *master*, I followed an optional course on parallel programming (*Programmation des Architecture Parallèle*). In this course, I studied technology of parallel and distributed programming: *pthread*, *OpenMP* and *MPI*.

My internship take place at Inria in laboratory of Bordeaux. Inria is a French public laboratory for computer science research. During the internship, I integrated the STORM team, a research team specialized on high performance computing (HPC) problematic. HPC programming cause some specific problem, particularly in software verification. In the variety of the team research work, some works are oriented on software verification for HPC. This is not the first time I integrated the STORM team. My internship during my second year of computer science license was with Samuel THIBAULT on analysis and simulation of memory usage on hierarchical compressed matrix on a StarPU [1] application. The second one was during my first year of *master informatique fondamental* with Emmanuelle SAILLARD on the protage of Parcoach [8] on LLVM 12, a bug finding tool which uses static sources codes analysis.

The goal of my internship is to extend the MPI Bugs Initiative (MBI) with hazard errors. The MBI is a framework for MPI verification tools comparison. This comparison mainly target HPC developers who need to found bugs in MPI codes to help them choose the best tool for their needs. Emmanuelle SAILLARD and Martin QUINSON have supervised my internship. These are two of the authors of the first paper of MBI [3] and still work on the project. Emmanuelle SAILLARD works at Inria Bordeaux and Martin QUINSON works at the University of Rennes. We meet once a week. These meeting were the occasion to keep update on my internship advancement and define what to do next. In addition, we have a Discord channel to discuss technical problems or others questions on a daily.

In this document I propose an extension of MBI for MPI hazards errors. The contributions are:

1. An extended error classification of MPI errors with hazards errors.
2. New correct and uncorrect code covering the new category of errors.
3. A new metric and visualization to compare tools outputs.
4. We use our methodology and metrics to compare 10 state-of-the-art bug findings tools.

The rest of the document is organized as follows: Section II gives background of the work with definitions of important notions. Section III, decribes my contribution. Section IV, presents the results analysis and the MBI tools comparison. Section V summarizes my end-of-study internship.

II. Background

This section gives an introduction of MPI standard, followed by a presentation of the MBI framework. Related works and a description of hazard errors.

A. Message Passing Interface

Message Passing Interface [7] (MPI) is a *message-passing library interface specification*. It is the most used standard for message passing application development. There exists different implementations of MPI, well-known are *OpenMPI* and *MPICH*. MPI defines operations expressed as functions, subroutines or methods to target C or FORTRAN programming language. In this document we will stay mainly focus on C API.

In message passing application the idea is to execute the same program on different computers (or nodes) by processes. During execution, processes can exchange data through communication and are identified with an ID called rank. Listing 1 shows an example with two processes `P0` and `P1`, with respectively rank 0 and 1. In this example `P0` calls function `A` and `P1` calls function

[1] https://team.inria.fr/storm/software/starpu/

B. To make examples more readable we use in this document, except specific exception, the equivalent syntax in listing 2. In this document we use simplified MPI API syntax with exclusively relevant information for example we can omit functions arguments if they are not relevant.

```
P0:                 P1:
  if (rank = 0)       if (rank = 0)
    A()                 A()
  else                else
    B()                 B()
```

Listing 1: MPI code where processes call different functions depending on their ID.

```
P0:                 P1:
  A()                 B()
```

Listing 2: MPI code with simplified syntax where processes call different functions depending on their ID.

There are 3 types of communications in MPI: *Point-to-point*, *Collective* and RMA.

Point-to-point (P2P) communication is minimal message passing paradigm communication pattern. This is the send-receive pattern, which allows two processes to communicate. In MPI, the send and receive functions are called `MPI_Send` and `MPI_Recv`. An example is showed in Listing 3. The two functions prototype are close, but take different parameter like sending buffer (resp. receiving buffer) or destination process of message (resp. source process of message).

```
P0:                 P1:
  MPI_Send(1)         MPI_Recv(0)
```

Listing 3: Example of *Point-to-point* communication: P0 sends a message to P1.

Collective communications must be called by all processes in a communicator. The barrier `MPI_Barrier` blocks processes untill all of them call the function. There exists other collective; all to one (gather), one to all (bcast), all to all (alltoall).

```
MPI_ISend(buf, 1, &req)
...
MPI_Wait(&req)
```

Listing 4: Example of non-blocking send function.

```
MPI_Send_init(&req)
...
MPI_Start(buf, 1, &req)
...
MPI_Wait(&req)
...
MPI_Request_free(&req)
```

Listing 5: Example of persistent send function.

Point-to-point and *collective* communication can be either *Blocking* (presented below), *Non-blocking* (let see example Listing 4) or *persistent* (let see example Listing 5). A *Non-blocking* communication is split in two functions. An initialization and a completion function. The communication can be performed during the two functions but with no warranty of when. *Persistent* communication is split in four functions: *initialization*, *start*, *wait* and *free*. The operation can be performed between *start* and *wait*. This type of communication is used to increase performance of program with the possibility of doing other computation during the communication.

RMA (Remote Memory Access) or One-Sided Communications is shared memory communication. In MPI-RMA, all processes expose a part of their memory to other processes. We call this part a *window*. Processes can thus directly read and write in other processes memory. The two main operations are `MPI_Put` (Figure 1) and `MPI_Get` (Figure 2), that respectively write and read memory. These operations are asynchronous and synchronizations are needed to keep coherence in memory. There exists different synchronization methods: active synchronization and passive synchronization. We call code section between two synchronizations an *epoch*. In active target synchronization, only participants of the communication must participate in the synchronization. In passive target synchronization, all processes must participate in the synchronization. For example, the more simple syn-

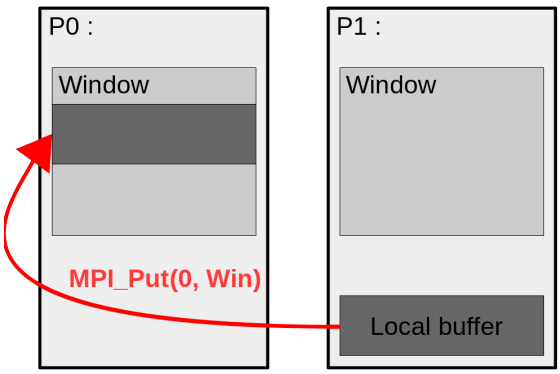


Figure 1: Example of a Put operation with 2 processes.

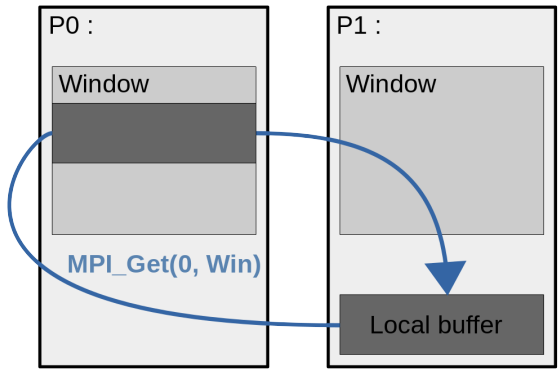


Figure 2: Example of a Get operation with 2 processes.

chronization function is `MPI_Win_fence`, is a passive target synchronization, so is a collective all processes must call it.

```
1 ...
2 MPI_Win_Fence(win); // new epoch
3 // Some RMA operations ...
4 MPI_Win_Fence(win); // new epoch
5 ...
```

Listing 6: Example of fence epoch.

MPI API is very complex and large, and because the nature of distributed paradigms there is a lot of ways to create bugs in programs. As results, detect bugs and what caused them is challenging. There exists a lot of tools to find bugs in MPI code. However, all tools are not equivalent, potentially not target the same subset of problems and don't use the same method to find bugs. It is then very difficult for an MPI application developer to choose the correct tool to use.

B. The MPI BUGS INITIATIVE *and related work*

The MPI BUGS INITIATIVE [3] proposes a classification of MPI errors, an associate benchmark suite and an automated method to compare verification tools. The MBI main goal is to measure the qualitative and quantitative performance of MPI verification tools in a reproducibility and automatic way. MBI defines 7 feature labels corresponding to the way MPI processes exchange message. The labels are given in Table 1. The first feature label is *point-to-point*, and is split in 3 labels: *base calls*, *non-blocking* and *persistent* respectively for blocking, non-blocking and persistent operations. The second features, *collective*, also split in 3 labels, *base calls*, *non-blocking* and finally *tools* for communicator and group management operations. The last feature is *RMA*, for Remote Memory Access operation.

The error classification, presented in [3], contains 8 types of errors based on their root cause: *Invalid parameter*, *Resource Leak*, *Request lifecycle*, *Local concurrency*, *Parameter matching*, *Message Race*, *Call ordering*, *Global concurrency*.

1. Errors in single call: erros in the scope of a MPI function

		Point-to-point			Collective			RMA	Unique files
		base calls	nonblocking	persistent	base calls	nonblocking	tools		
Single call	Invalid Parameter	59	46	46	33	33	73	33	194
Single process	Resource Leak	0	1	2	0	0	12	0	14
	Request lifecycle	0	4	5	0	12	0	0	18
	Epoch lifecycle	0	0	0	0	0	0	30	30
	Local concurrency	8	9	3	0	11	0	26	45
Multi-processes	Parameter matching	27	19	19	29	29	33	0	97
	Message Race	20	30	0	24	0	0	0	34
	Call ordering	55	51	0	480	444	0	0	652
	Global concurrency	6	6	0	0	0	0	32	32
System	Buffering Hazard	9	9	0	0	0	0	0	12
Data	Input Hazard	3	3	0	12	0	0	0	16
Correct codes		39	46	10	504	480	31	31	761
Total		226	224	85	1082	1009	149	152	1905

Table 1: 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).

```
1 int root = -1;
2 MPI_Reduce(&rank, &sum, root, com)
```

Listing 7: Example of a invalid root (*Invalid Parameter*)

```
1 MPI_Comm com[size];
2 MPI_Comm_dup(MPI_COMM_WORLD, &com[j]);
3 ...
4 // Missing MPI_Comm_free
```

Listing 8: Example of a *Resource Leak*

```
1 MPI_IRecv(&req)
2 ... // Missing wait
3 MPI_Request_free(&req)
```

Listing 9: Example of a missing wait (*Request Lifecycle*)

```
1 P0:                    P1:
2   int root = rank        int root = rank
3   Reduce(root)           Reduce(root)
```

Listing 10: Example of a Reduce function with unmatched root parameter (*Parameter Marching*)

```
1 P0:                P1:
2   Barrier()          Reduce()
3   Reduce()           Barrier()
```

Listing 11: Example of a *Call Ordering*

- *Invalid parameter*: is a bad parameter in a MPI function call. I can be an *invalid root, communicator, operator, datatype, tag or buffer length*. An example Listing of an invalid root is presented in 7.

2. Errors local to a process: Incoherence between local MPI context and MPI function calls

   - *Resource Leak*: MPI resource which is not correctly freed. An example is presented in Listing 8 with a missing `MPI_Comm_free` call.
   - *Request Lifecycle*: missing function on MPI request sequence. In Listing 9, the completion is missing.
   - *Local concurrency*: concurrent memory accesses in a process. This can append with non-blocking operations. Is presented in Listing 12, the buffer is written before the completion is called.

3. Multi-processes errors: Errors involving multi processes

   - *Message Race*: multiple message with no warranty of arrival order. In Listing 14, a deadlock can happen depending on which send matches the receive on any source in `P0`.
   - *Parameter Matching*: wrong parameter matching in MPI function calls. In Listing 10, the root ID for collective operation is not the same between different processes.
   - *Call Ordering*: different call order between processes, this can cause a deadlock. Listing 11 present two processes with two collective operations with different order.
   - *Global Concurrency*: occurs when two or more processes make concurrent accesses on same memory region. These errors can append with RMA operation. In Listing 13 we present a data race with two write operation on same memory region.

The features labels and errors classification propose a formal description to define benchmark. In MBI, all codes are generated with a python script to ensure a good coverage of MPI. All codes contain at most one error and some codes are correct. The number of codes are given un Table 1.

MBI proposes a comparison of 8 tools: Aislinn, CIVL, ISP, ITAC, Mc SimGrid, MPI-SV, MUST and PARCOACH. For each test, the tool outputs are analyzed with specific keywords research and results are set as True Positive (TP), False Negative (FN), False Positive (FP) or True Negative (TN) depending of test expected results. In case of errors, the number of in Compilation Error (CE), Timeout (TO) or Runtime Error (RE) is reported. The running time limit for each test is 5 minutes (300 second). After this time the test is noted as Timeout. This is a reasonable time for our short benchmark codes.

The MBI project is not the first one to propose a benchmark suite for MPI applications, MPI-Corrbench [4] proposes a benchmark suite to evaluate the capacity of MPI verification tools to detect errors. This project evaluate 4 tools: *MUST*, *ITAC*, *Parcoach*, *MPI-Checker*. For error classification, MPI-Corrbench, uses a different classification than MBI: *Erroneous Arguments*, *Mismatching Arguments* and *Erroneous Program Flow*. MPI-Corrbench proposes 4 levels of benchmark: level 0 is a code in a single file, level 1 is a code with multiple files, level 2 is a mini application and level 3 is for applications. In addition, MPI-Corrbench uses another way to classify tools results. Authors make a difference between errors and warnings and propose two different metrics W^+ and W^- for warnings if we considered as errors or ignored.

Another project, more dated, is RTED [6]. The error classification of RTED is composed of 10 different 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* and *implementation dependent errors*. This classification include some hazard errors and some MBI codes are inspired by RTED codes. In addition, RTED evaluted tools feedblack by giving them a score. This score is between 0 and 5 and take into account if tools give the type of error, the line number, filename and some other information. RTED proposes code in different language: C, C++ and Fortrant. In addition, RTED not only includes MPI errors, but also classic serial errors, OpenMP errors and UPC errors.

DataRaceBench [5] (DRB) is an OpenMP bench-

mark suite focused on data races, with 4 OpenMP bug finding tools are evaluated with it. As *data races* in MPI, in OpenMP a *data race* is hazard error and does not appear at each execution. To force a *data race* durring execution, DataRaceBench launches code with different parameters, number of threads and table size. In addition, each benchmark is executed 5 times. Unlike MBI, RTED and DRB include codes written in C, C++ and FORTRANT.

III. Contribution

Hazard errors are more challenging to find for bug finding tools. This type of error is nondeterministic and thus difficult to reproduce. Hazard errors include error dependent to processes scheduling, network latency, software environment or software data. They can cause a deadlock in a program, a crash or a silence error which gives false results without any sign.

A. Hazard errors classification

The MBI project defines 8 types of errors in MPI applications. Three of them are already hazard errors: *Local Concurrency*, *Global Concurrency* and *Message Race*. Initially, the codes corresponding to these three errors were deterministic (the errors happend in all execution). In fact, we can define at least 4 types of hazard errors: data race (*Local Concurrency* and *Global Concurrency*), *Message Race*, *Buffer Hazard* and *Input Hazard*.

```
1 P0:
2   MPI_ISend(buf)
3   buf[0] += 1
4   MPI_Wait()
```

Listing 12: Example of a *Local concurrency*

```
1 P0:            P1:           P2:
2   Win_fence()   Win_fence()   Win_fence()
3                 Put(0)        Put(0)
4   Win_fence()   Win_fence()   Win_fence()
```

Listing 13: Example of a *Global Concurrency*

```
1 P0:              P1:         P2:
2   Recv(ANY_SRC)    Send(0)
3   Recv(1)                      Send(0)
```

Listing 14: Example of a *Message race*

```
1 P0:             P1:
2   MPI_Send(1)     MPI_Send(0)
3   MPI_Recv(1)     MPI_Recv(0)
```

Listing 15: Example of a *Buffer hazard*

a. Data Race

DataRaceBench [5] defines a Data Race with these words:

> A data race can occur when two concurrent threads access a shared variable and when at least one access is a write operation, and the threads use no explicit mechanism to prevent the accesses from being simultaneous.

In the message passing paradigm and MPI context, the problem is quite different. We can separate the problem into two different categories: *Local Concurrency* and *Global Concurrency*. The first category is a data race on a unique process. The second one is a data race caused by two (or more) processes. With RMA feature by example. The solution to prevent this type of error is synchronization, like `MPI_Wait` for /point-to-point/ operations or *epoch* for RMA operations.

- *Local concurrency*: A process accesses a memory region that is being read or write. This type of error can be produce with non-blocking and one-side communication. We can see an example in Listing 12. In this example, the buffer used

```
1 P0(x):
2   MPI_ISend(buf)
3   if (x < 0)
4     buf[0] += 1
5   MPI_Wait()
```

Listing 16: Example of a *Input hazard*

in Isend is modified before the completion, so we don't know which value is sent.

- *Global concurrency*: Two or more processes access the same memory region and at least one access is a write operation. This type of error can be produce with RMA communication on the same epoch. In the example Listing 13, there are two processes (`P1` and `P2`) doing a put on the same memory location in `P0`.

b. Message race

A message race occur when two message (or more) are sent and there is a receive on ANY_SRC, the order of send arrivals is not guarented. Listing 14 presents a message race. If message of `P1` matches with the first receive in `P0`, a deadlock occurs. Indeed the message sent by `P2` does not match with the second receive call in `P0`.

c. Buffer hazard

Buffer hazard is an error that is related to the buffer usage of MPI implementations. All MPI blocking communication can use a buffer for performance purpose. But this can change the program behaviors and produce non-deterministic executions. The usage of buffering depends on MPI implementation, system configuration and system usage at the program execution.

In reel world, system has a limited size of buffer, but some bug finding tools use only *zero-buffer* (we consider the implementation have do not use buffer) or *infinit-buffer* (we consider the implementation have infinit size of buffer). The tool Hermes [2] is the only one, is able to verify programs with a fixed buffer size k, a *k-buffer* parameter.

In Listing 15, with *zero-buffer*, the program deadlocks because send functions are blocking. With infinit buffering, the program works fine. In addition, the size of messages can produce a deadlock too (if the message size is greater than the system buffer size).

d. Input hazard

Input hazard is an error that occurs only for a subset of program input. This error is interesting to study in the context of MBI, because it needs an additional effort for debugging with some tools. Dynamic tools rely on one execution and are input depend. It is not the case with static analyses and symbolic execution paradigm. In Listing 16, we assume the variable `x` is given as an input, so if `x < 0` is true, there is a *Local Concurrency* error, otherwise there is no error during the execution.

Error category	**Number of codes**	**Number of tests**
Invalid parameter	194	194
Resource leak	14	14
Request lifecycle	18	18
Epoch lifecycle	30	30
Local concurrency	**45**	**225**
Parameter matching	97	97
Message race	**34**	**170**
Call ordering	652	656
Global concurrency	**32**	**160**
Buffering hazard	**12**	**24**
Input Hazard	**16**	**32**
Correct execution	761	781
Total	1905	2401

Table 2: Number of codes and tests for each error types. Hazard errors are in bold.

B. Codes generation

MBI	RTED	MPI-Corrbench	*total*
1825	16	64	*1905*

Table 3: Number of codes by origins

Codes are generated by a python script. All codes are written in C and have a formatted header providing a textual description of the problem, command line, expected result and feature used as presented Listing 17. The MBI extension covers the same MPI features as previously 1. All previous, hazard codes have been changed (we remove the synchronization) to obtain true nondeterminism hazard codes. Some new code have been added with with mixed features mostly with point to point communication and RMA operations. Some new codes is inspired by RTED or MPI-Corrbench codes, Table 3 present the number of code by origin.

The code suite have been extended with new codes, the large majority have hazard errors (*Local Concurrency*, *Global Concurrency*, *Message Race*, *Buffer*

Hazard and *Input Hazard*). Table 2 shows the number of codes and tests for each error. Each hazard code was tested at least two times, which explains the different number of codes and tests. *Buffering Hazard* codes are tested with two different parameters: *Zero Buffer* and *Infinite Buffer*. *Input hazard* codes are executed with input producing an error and not. *Local Concurrency*, *Global Concurrency* and *Message Race* are executed 5 times to have a good chance to produce the error.

We assume a code contains only one error independently of the input parameters, so if an execution do not cause the error, we assume the error still exists in the code and a perfect tool must see it. For non-determinist codes, there is no warranty the error append at each execution. Determinist errors are executed just one time. For codes with hazard errors, we have two different strategies: code with *Local Concurrency*, *Global Concurrency* and *Message race* are executed five times each. Five is a reasonable number to have good a chance to produce the error, but not to mush to increase unreasonably the total execution time of all codes. It's also the number of executions used by DataRaceBench [5] to detect data races in OpenMP programs. For *Buffer Hazard* and *Input Hazard* we can control error appearance. For *Buffer Hazard*, some tools take the buffering mode as argument, so we make two executions for each code. For *Input Hazard*, we use also two tests, one with a parameter which causes the error and another one with a parameter which does not cause the error.

C. Tools output evaluation and metrics

The metrics used in MBI were adapted to determinist codes. This metrics are presented Table 4. With hazard errors, metrics must be reconsidered. A first solution is to use 3 metrics: *Correct diagnostic*, *Can be correct diagnostic* and *Incorrect diagnostic*. But these metrics are an over approximation for a good analysis of tools reports.

We then, define new evaluation metrics, with 6 possible results: Systematic True Negative (*STN*), Systematic False Negative (*SFN*) , Can be False Positive (*CFP*), Can be True Positive (*CTP*), Systematic False Positive (*SFP*) and Systematic True Prositive (*STP*). Based on these results, 2 metrics are computed: Accuracy$^+$ (A^+) and Accuracy$^-$ (A^-). The metrics are presented Table 5.

Systematic results correspond to a tool that returns always the same results for the same code, *Can be* results correspond to a tool returns different results for a code. This results classification works because we assume an error in a code is always present, the expected result for a code is then always the same. If a code expect an error, tests results must be exclusively in $\{TP, FN, CE, RE, TO\}$, respectively, a code without error must have tests results into $\{TN, FP, CE, RE, TO\}$. The new metrics are used to compare behaviors of tools for hazard errors.

Accuracy$^+$ and Accuracy$^-$ give the proportion of correct diagnostics over all tests. Accuracy$^+$ represents incorrect codes that can be dectected as incorrect. Accuracy$^-$ represents incorrect codes that one systematically detected as incorrect. The difference between the two metrics is the part of errors coverage for which a tool is not able to detect systematically.

IV. Use cases

We propose a comparison of 10 state-of-the-art verification tools using MBI: Aislinn (v3.12), CIVL (v1.21), Hermes [2] (git-3113718), ISP (v0.3.1), ITAC (v2021.6), Mc Simgrid (git-e32f58bcd8), MPI-SV (v1), MUST (v1.7.2), MPI-Checker [1] (llvm-11) and Parcoach (git-485166f). Tools use different verification methods as presented Table 6. Each method has advantages and disadvantages to detect errors types.

The experiments were done in a Docker image with Ubuntu 20.04 with all tools and them dependencies. Except for Aislinn and MPI-SV which use their own docker image. The docker image uses MPICH implementation in version v3.3.2. We use Ubuntu 18.04 for Aislinn due to dependency issues, and we use the docker image provided by the authors for MPI-SV. Experiments were conducted on a Grid5000 super computer node and all tests use at most 6 MPI processes.

A. Tools comparison

Tools evaluation and comparison is challenging, because of the diversity of errors root cause and methods used by the tools. Table 7 shows the results of all 10

```
// DO NOT EDIT: this file was generated by InputHazardGenerator.py. DO NOT EDIT.
/* ///////////////////////// The MPI Bugs Initiative /////////////////////////

  Origin: MBI

  Description: Missing collective function call.
    Missing collective function call for a path depending to input, a deadlock is created.

  Version of MPI: Conforms to MPI 1.1, does not require MPI 2 implementation

BEGIN_MPI_FEATURES
  P2P!basic: Lacking
  P2P!non-blocking: Lacking
  P2P!persistent: Lacking
  COLL!basic: Yes
  COLL!non-blocking: Lacking
  COLL!persistent: Lacking
  COLL!tools: Lacking
  RMA: Lacking
END_MPI_FEATURES

BEGIN_MBI_TESTS
  $ mpirun -np 2 ${EXE} 1
  | ERROR: IHCallMatching
  | P2P mistmatch. Missing MPI_Reduce at InputHazardCallOrdering_Reduce_nok.c:73.
  $ mpirun -np 2 ${EXE} 2
  | ERROR: IHCallMatching
  | P2P mistmatch. Missing MPI_Reduce at InputHazardCallOrdering_Reduce_nok.c:73.
END_MBI_TESTS
//////////////////////       End of MBI headers        ////////////////// */

// Includes ...
int main(int argc, char **argv) {
  // Variables declaration and MPI init ...
  if (nprocs < 2)
    printf("MBI ERROR: This test needs at least 2 processes to produce a bug!\n");
  if (argc < 2)
    printf("MBI ERROR: This test needs at least 1 argument to produce a bug!\n");
  // ...
  int n = atoi(argv[1]);
  if (rank == 0) {
    if ((n % 2) == 0) { /* MBIERROR */
      // Missing collective
    } else {
      MPI_Reduce(&val1, &sum1, 1, type, op, root, newcom);
    }
  } else  {
    dest=0, src=0;
    MPI_Reduce(&val2, &sum2, 1, type, op, root, newcom);
  }
  // MPI finalize ...
  return 0;
}
```

Listing 17: Example of *InputHazard* file (`InputHazardCallOrdering_Reduce_nok.c`) generated by `InputHazardGenerator.py`.

Result	Meaning
Compilation Error (CE)	The code uses a feature not supported by the tool.
Timeout (TO)	Timeout (limit: 300s).
Runtime Error (RE)	Tool failure during the evaluation of this code.
True Positive (TP)	Error correctly detected.
False Negative (FN)	Error missed.
False Positive (FP)	Correct code reported as faulty.
True Negative (TN)	Correct code reported as such.
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 4: Tests output evaluation [3].

Result	Meaning
Systematic Error (SE)	Include CE, RE, and TO.
Systematic True Positive (STP)	Error correctly detected for each exection.
Can be True Positive (**CTP**)	Error correctly detected for some execution.
Systematic False Negative (SFN)	Error missed for each exection.
Can be False Positive (**CFP**)	Correct code reported as faulty for some exection.
Systematic False Positive (SFP)	Correct code reported as faulty for each exection.
Systematic True Negative (STN)	Correct code reported as such for each exection.
Metric	**Definition and meaning**
Overall accuracy^{+}	Proportion of correct diagnostics over all tests. $A^{+} = \frac{TP+TN+CTP}{total}$
Overall accuracy^{-}	Proportion of correct diagnostics over all tests. $A^{-} = \frac{TP+TN}{total}$

Table 5: New output evaluation and new metrics for non-deterministic results.

Tool	Static analysis	Model checking	Symbolic execution	Testing
ITAC				✓
MUST				✓
Mc SimGrid		✓		
ISP		✓		
CIVL		✓	✓	
MPI-SV		✓	✓	
Aislinn			✓	
PARCOACH	✓			
Hermes			✓	✓
MPI-Checker	✓		✓	

Table 6: Tools description

tools on hazards codes. The last row presents the *Ideal tool* results. Best results for each error is in bold.

For *Local concurrency*, ITAC is the best with an Accuracy$^+$ of 0.44 and an Accuracy$^-$ of 0.38, followed by ISP with an Accuracy$^+$ and Accuracy$^-$ of 0.36. CIVL, MC Simgrid, Parcoach, MPI-SV and MUST, do not detect any of *Local Concurrency* errors we define.

No tool is able to detect *Global Concurrency*. This results show a lack of RMA support in tools.

For *Message race*, ISP has the same results than an *ideal tool* with an Accuracy$^+$ and an Accuracy$^-$ of 1. Followed by ITAC, MUST and MC Simgrid. PARCOACH has a lot of STP because it does not find an error in correct codes. ISP detects *Message race* errors at each execution, Mc SimGrid, Hermes, Aislinn and CIVL too, but with a smaller coverage. ITAC, MUST and MPI-SV do not detect *message race* at each execution. PARCOACH has a lot of STP because it can find errors in incorrect codes. However, it detects deadlocks related to collectives instead of detecting the actual message races. This could be detected with a tools feedback analysis.

For *Buffering hazard*, ITAC and MUST have the same results as an *ideal tool*. Aislinn, Mc Simgrid and Hermes have also an Accuracy$^+$ at 1. The difference with Accuracy$^+$ and Accuracy$^-$ depends on usage or not of the *zero-buffer* option or not. For MUST, errors are detected in the majority of cases.

For *Input hazard*, ISP, ITAC and MC Simgrid have the same results and detect all errors. However the Accuracy$^-$ is equal to 0, so these tools are dependent of input data to detect an error. Aislinn, doesn't detect all errors, and is also dependent to input data. PARCOACH is less accurate tool, but has the same score for Accuracy$^+$ and Accuracy$^-$, because it doesn't execute the programe, so is not dependent to input data.

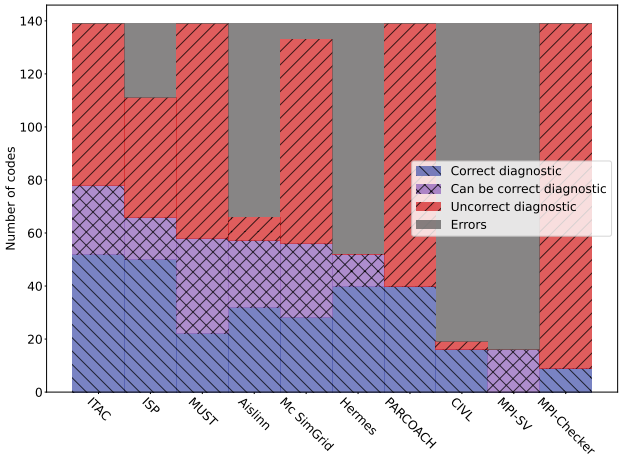


Figure 3: Tools output evaluation on hazard errors

Figure 3 present tools output evaluation on hazard errors. This visualization presents the part of errors (SE) includes CE, RE and TO, incorrect diagnostic (SFN), can be correct diagnostic (CTP) and correct diagnostic (STP) for all codes. The tools order is sort by the number of correct diagnostic and can be correct, form the left to the right.

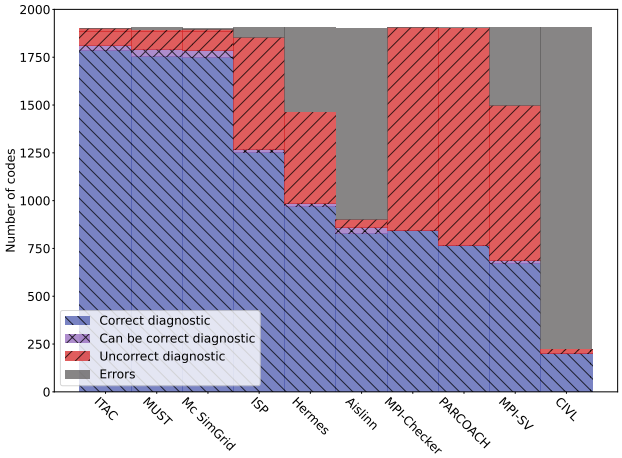


Figure 4: Tools output evaluation on all errors

Table 8 gives tools evaluation results on all codes for all of errors, with the following metrics: SE, STP, CTP,

	Local concurrency						*Global concurrency*						*Message race*						*Buffering hazard*						*Input Hazard*					
	SE	STP	CTP	SFN	Accuracy$^+$	Accuracy$^-$	SE	STP	CTP	SFN	Accuracy$^+$	Accuracy$^-$	SE	STP	CTP	SFN	Accuracy$^+$	Accuracy$^-$	SE	STP	CTP	SFN	Accuracy$^+$	Accuracy$^-$	SE	STP	CTP	SFN	Accuracy$^+$	Accuracy$^-$
Aislinn	32	8	0	5	0.18	0.18	32	**0**	**0**	**0**	**0**	**0**	6	24	0	4	0.71	0.71	**0**	0	**12**	**0**	**1**	0	3	0	13	**0**	0.81	0
CIVL	45	0	0	**0**	0	0	32	**0**	**0**	**0**	**0**	**0**	30	4	0	**0**	0.12	0.12	9	0	0	3	0	0	6	10	0	**0**	0.62	0.62
ISP	8	16	0	21	0.36	0.36	8	**0**	**0**	24	**0**	**0**	**0**	**34**	0	**0**	**1**	**1**	12	0	0	**0**	0	0	**0**	0	**16**	**0**	**1**	0
ITAC	**0**	**17**	**3**	25	**0.44**	**0.38**	**0**	**0**	**0**	32	**0**	**0**	**0**	24	6	4	0.88	0.71	**0**	**12**	0	**0**	**1**	**1**	**0**	0	**16**	**0**	**1**	0
Mc SimGrid	4	0	0	41	0	0	2	**0**	**0**	30	**0**	**0**	**0**	28	0	6	0.82	0.82	**0**	0	**12**	**0**	**1**	0	**0**	0	**16**	**0**	**1**	0
MPI-SV	45	0	0	**0**	0	0	32	**0**	**0**	**0**	**0**	**0**	18	0	16	**0**	0.47	0	12	0	0	**0**	0	0	16	0	0	**0**	0	0
MUST	**0**	0	0	45	0	0	**0**	**0**	**0**	32	**0**	**0**	**0**	12	**18**	4	0.88	0.35	**0**	**12**	0	**0**	**1**	**1**	**0**	0	**16**	**0**	**1**	0
Hermes	31	14	0	**0**	0.31	0.31	32	**0**	**0**	**0**	**0**	**0**	8	26	0	**0**	0.76	0.76	**0**	0	**12**	**0**	**1**	0	16	0	0	**0**	0	0
PARCOACH	**0**	0	0	45	0	0	**0**	**0**	**0**	32	**0**	**0**	**0**	28	0	6	0.82	0.82	**0**	0	0	12	0	0	**0**	**12**	0	4	0.75	**0.75**
MPI-Checker	**0**	9	0	36	0.2	0.2	**0**	**0**	**0**	32	**0**	**0**	**0**	0	0	34	0	0	**0**	0	0	12	0	0	**0**	0	0	16	0	0
Ideal tool	*0*	*45*	*0*	*0*	*1*	*1*	*0*	*32*	*0*	*0*	*1*	*1*	*0*	*34*	*0*	*0*	*1*	*1*	*0*	*12*	*0*	*0*	*1*	*1*	*0*	*16*	*0*	*0*	*1*	*1*

Table 7: Results with hazard tool output evaluation

Tool	Errors	Results						Overall	
	SE	**STP**	**CTP**	**SFN**	**SFP**	**CFP**	**STN**	**Accuracy$^+$**	**Accuracy$^-$**
Aislinn	1000	495	25	46	**0**	**0**	339	0.45	0.44
CIVL	1666	123	0	8	10	**0**	98	0.12	0.12
ISP	54	971	16	104	479	**0**	281	0.67	0.66
ITAC	1	**1032**	25	86	3	**0**	**758**	**0.95**	**0.94**
Mc SimGrid	6	1002	28	108	4	**0**	757	0.94	0.92
MPI-SV	405	0	16	810	**0**	**0**	674	0.36	0.35
MUST	12	1007	**34**	91	6	**0**	755	0.94	0.92
Hermes	447	967	12	**0**	479	**0**	0	0.51	0.51
PARCOACH	**0**	664	0	480	636	**0**	125	0.41	0.41
MPI-Checker	**0**	321	0	823	240	**0**	521	0.44	0.44
Ideal tool	*0*	*1144*	*0*	*0*	*0*	*0*	*761*	*1*	*1*

Table 8: Tools output evaluation on all errors

SFN, SFP, CFP, STN, Accuracy$^+$ and Accuracy$^-$. The best results are in bold. Again, the last row gives the results of an *ideal tool*. PARCOACH and MPI-Checker have no errors, followed by ITAC with only one error and MC SimGrid with 6 errors, MUST has 12 errors, and ISP has 54 errors. The rest of tools has a number of errors greater than 400. ITAC can detect most errors. It found 1032 errors systematically and 25 errors could be found. It's also the greater number of Systematic True Negative. MUST, MC SimGrid, ISP and Hermes follow with close results. In addition, all tools detects a nonexistent error systematically, there are no CFP.

With tools feedback analysis we have detected some false True Positive. In the order of 300 false diagnostics for MPI-Checker, and in the order of 150 false diagnostics of PARCOACH. The other tools have a lot of less or no bad diagnostics.

B. The case of MC Simgrid

MBI is not only made to help users choose the right bug finding tool, but also to help tools developers to improve their tools. In this section, we compare two versions of MC Simgrid. To compare the performance of two tools or two versions of same tools, we propose a representation on radar plots by errors categories. Errors categories are separated by determinist errors on left part of the plots and hazard errors at the right. We present on this visualization 3 metrics: the overall accuracy for all metrics, and A^+ noted *can be detected* and A^- noted *Always detected* for hazard errors. Larger is the colored zone, the better is errors detections.

We compare MC Simgrid version 3.27, to the actual git master branch (e32f58bcd8). Results are presented in Figure 5 and 6. We choose MC Simgrid because we know the developers use MBI to found and fix bugs in the tool. For determinist errors we can see a partial detection of *Invalid Parameter*, *Call Ordering* and *Parameter Matching* for the version 3.27. *Resource Leak*, *Request Lifecycle* and *Epoch Lifecycle* are not detected or have a small accuracy, (less than 0.2). The recent version detects a large set of *Invalid parameter*, *Resource leak*, *Call ordering*, *Parameter matching* and *Request lifecycle*. For Hazard errors, results are close between the two versions, with a small amelioration for *Input hazard*.

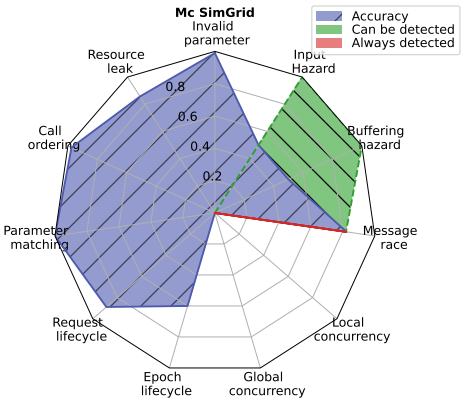


Figure 5: MC Simgrid capacity for each category.

Table 9 presents the MBI metrics for all MC Simgrid version from v3.27 to the actual version. Since version v3.31, the coverage has reduced, and there is an increase number of errors. The accuracy has increased from 0.62 in version v3.27 to 0.8 in the last version.

We can see a significant improvement of Simgrid detection capacity. Because of MBI was initially mainly focused on determinist errors the majority of improvement is visible on these errors. We can expect new improvements on nondeterminism errors in the near future.

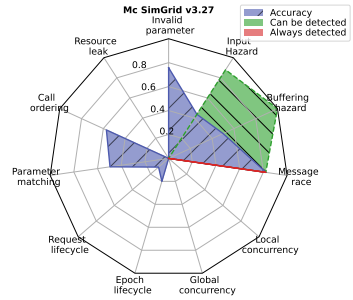
The results of all tools with the radar plots is show in Figure 7. Table 10 presents results with determinist metrics for all tools.

V. Conclusion

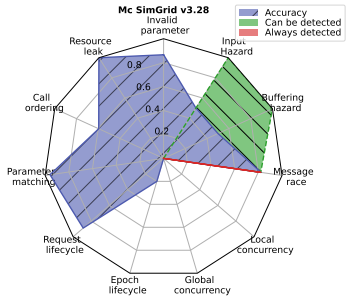
In this document we proposed an extension of MBI for hazards errors. We proposed an extension of the error classification with two new classes: *Input Hazard* and *Buffer Hazard*, and a new tool outputs evaluation to analyse tools behavior on hazard errors. This evaluation is able to detect input dependent and execution dependent tools. In addition, we proposed a tool comparison with 2 new state-of-art tools: MPI-Checker and Hermes. We proposed a comparison of different versions of MC SimGrid. MC SimGrid developers used the MBI benchmark suite to detect and fix bugs in the tool. We have noted significant improvements between the 2 versions, principally on deterministic errors.

Tool	Errors			Results				Robustness		Usefulness				Overall
	CE	**TO**	**RE**	**TP**	**TN**	**FP**	**FN**	**Coverage**	**Conclusiveness**	**Specificity**	**Recall**	**Precision**	**F1 Score**	**accuracy**
Mc SimGrid v3.27	**0**	**0**	399	744	746	**0**	512	**1**	0.8338	**1**	0.5924	**1**	0.7447	0.6206
Mc SimGrid v3.28	**0**	**0**	378	856	746	**0**	421	**1**	0.8426	**1**	0.6703	**1**	**0.7578**	0.6672
Mc SimGrid v3.29	**0**	**0**	378	856	746	**0**	421	**1**	0.8426	**1**	0.6703	**1**	**0.7578**	0.6672
Mc SimGrid v3.30	**0**	**0**	378	856	746	**0**	421	**1**	0.8426	**1**	0.6703	**1**	**0.7578**	0.6672
Mc SimGrid v3.31	4	258	**0**	**1143**	752	8	**236**	0.9983	0.8909	0.9895	**0.8289**	0.993	0.7307	0.7893
Mc SimGrid	4	30	**0**	**1143**	**772**	8	444	0.9983	**0.9858**	0.9897	0.7202	0.993	0.6946	**0.7976**
Ideal tool	*0*	*0*	*0*	*1620*	*781*	*0*	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*

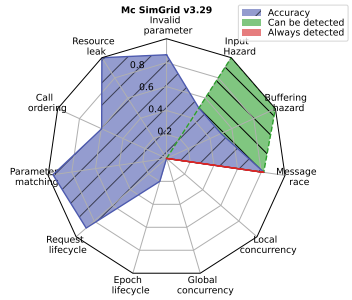
Table 9: Metric for all version of MC Simgrid



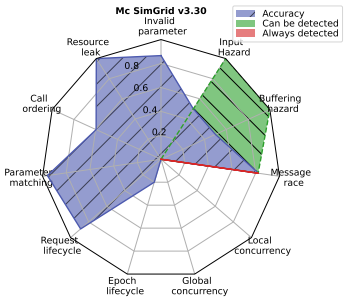
(a) Simgrid version 3.27 capacity for each categories



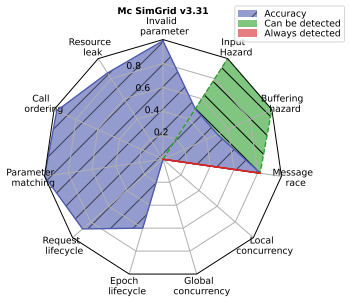
(b) Simgrid version 3.28 capacity for each categories



(c) Simgrid version 3.29 capacity for each categories



(d) Simgrid version 3.30 capacity for each categories



(e) Simgrid version 3.31 capacity for each categories

Figure 6: Simgrid other version results.

MBI is the only project to propose a large errors classification based on root cause and specific tools outputs evaluation metrics for hazard errors. It's also the project with the larger number of bugs finding tools with 10 state-of-art tools.

DataRaceBench [5] proposes a formal description of a nice benchmark suite on 7 points: *Representative*, *Scalable*, *General*, *Accessible*, *Extensible*, *Easy to use*, *Qualitative* and *Correct*. We can compare MBI with this description:

- *Representative*: MBI include codes with various MPI features, and 11 errors classes. MBI does not cover all the features API but represent for common MPI errors.
- *Scalable*: MBI does not test the scalability of tools and include only micro-benchmark without multiple possible scale execution.
- *General*: The test suite is designed to be used with static and dynamic analyses. It compares tools using there are a large variety of methods and mixed methods.
- *Accessible*: MBI is accessible online on a free

software license [2].

- *Extensible*: The Benchmark suite can be extended by adding new code generators. All script are written in Python with some utility functions already written.
- *Easy to use*: All MBI features are centralized in a unique script which can take different input commands and the execution environment is available into a Docker image. User can run micro-benchmarks in the Docker without setting up any software environment on his computer.
- *Qualitative*: MBI uses various metrics : *Recall*, *Specificity*, *Precision*, *Accuracy*, *F1 Score*, *Robustness*, *API Coverage* and Accuracy^{+} and Accuracy^{-} for hazard errors.
- *Correct*: MBI uses hand made verification of tests correctness (it checks what all tools returned on the codes are valgrind tool).

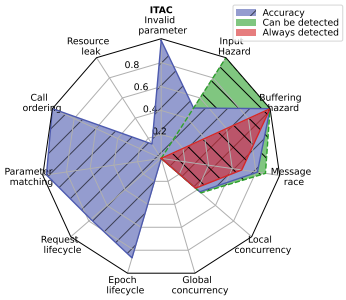
This analysis highlights some possible improvement for MBI. First, MBI does not include notion of scalability. Secondly, MBI ensures the tests correctness with human verification and an automatic way to ensure this correctness could be interesting. Indeed, we have detected many false True Positive. An automatic method can be a future work. This report is accented by our tools feedback analysis which show a consequent number of false True Positive.

I already had the occasion to see the research work environment, but this internship, has been the occasion for me to discover the work of researchers. In addition, we are, with Emmanuelle SAILLARD and Martin QUINSON, writing a paper to submit at the Correctness workshop 2022 [3] or at the TACAS (Tools and Algorithms for the Construction and Analysis of Systems) conference 2022 [4].

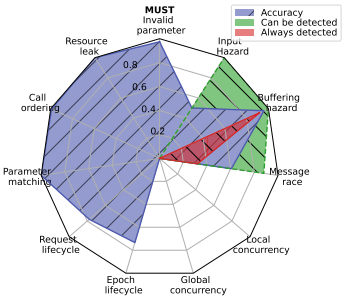
[2] https://gitlab.com/MpiBugsInitiative

[3] https://correctness-workshop.github.io/2022/

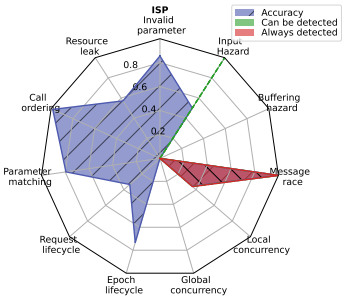
[4] https://etaps.org/2022/tacas



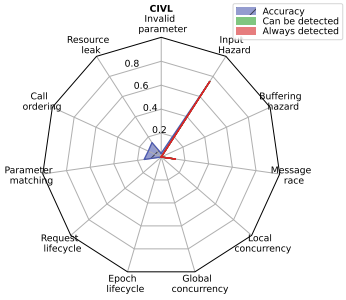
(a) ITAC capacity for each categories



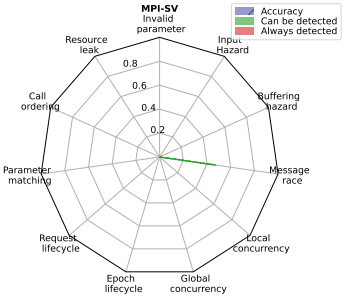
(b) MUST capacity for each categories



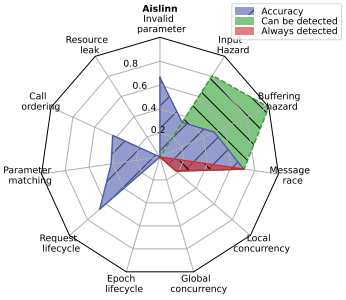
(c) ISP capacity for each categories



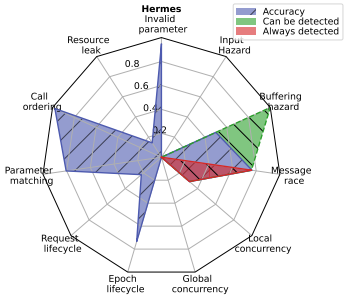
(d) CIVL capacity for each categories



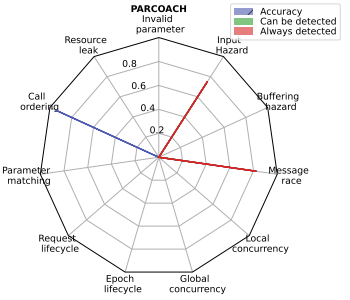
(e) MPISV capacity for each categories



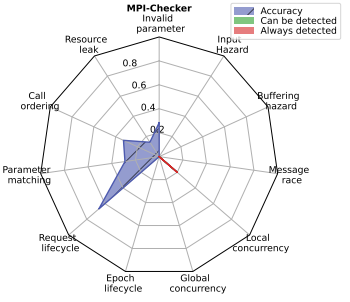
(f) AISLINN capacity for each categories



(g) HERMES capacity for each categories



(h) PARCOACH capacity for each categories



(i) MPI-CHECKER capacity for each categories

Figure 7: Other tool result.

Tool	Errors			Results				Robustness		Usefulness				**Overall accuracy**
	CE	**TO**	**RE**	**TP**	**TN**	**FP**	**FN**	**Coverage**	**Conclusiveness**	**Specificity**	**Recall**	**Precision**	**F1 Score**	
Aislinn	1275	**0**	17	649	351	4	105	0.469	0.4619	0.9887	0.8607	**0.9939**	0.6618	0.4165
CIVL	2127	**0**	6	144	89	24	11	0.1141	0.1116	0.7876	0.929	0.8571	0.6748	0.097
ISP	**0**	104	26	1191	301	479	300	**1**	0.9459	0.3859	0.7988	0.7132	0.2941	0.6214
ITAC	**0**	1	6	**1256**	769	11	358	**1**	0.9971	0.9859	0.7782	0.9913	0.682	**0.8434**
Mc SimGrid	4	30	**0**	1143	**772**	8	444	0.9983	0.9858	**0.9897**	0.7202	0.993	**0.6946**	0.7976
MPI-SV	5	**0**	900	0	673	**0**	823	0.9979	0.6231	1	0.0	(error)	(error)	0.2803
MUST	1	7	6	1129	766	14	478	0.9996	0.9942	0.9821	0.7026	0.9878	0.686	0.7893
Hermes	**0**	86	689	1143	0	483	**0**	**1**	0.6772	0.0	**1**	0.703	0.0	0.4761
PARCOACH	4	**0**	**0**	788	108	672	829	0.9983	0.9983	0.1385	0.4873	0.5397	0.1284	0.3732
MPI-Checker	**0**	**0**	**0**	357	541	240	1263	**1**	**1**	0.6927	0.2204	0.598	0.3545	0.374
Ideal tool	*0*	*0*	*0*	*1620*	*781*	*0*	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*

Table 10: Metric for all tools

REFERENCES

[1] Alexander Droste, Michael Kuhn, and Thomas Ludwig. Mpi-checker: Static analysis for mpi. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Dhriti Khanna, Subodh Sharma, César Rodríguez, and Rahul Purandare. Dynamic symbolic verification of mpi programs. In *International Symposium on Formal Methods*, pages 466–484. Springer, 2018.

[3] Mathieu Laurent, Emmanuelle Saillard, and Martin Quinson. The mpi bugs initiative: a framework for mpi verification tools evaluation. In *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 1–9. IEEE, 2021.

[4] Jan-Patrick Lehr, Tim Jammer, and Christian Bischof. Mpi-corrbench: Towards an mpi correctness benchmark suite. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, pages 69–80, 2021.

[5] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. Dataracebench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[6] Glenn R. Luecke, James Coyle, James Hoekstra, Marina Kraeva, Ying Xu, Mi-Young Park, Elizabeth Kleiman, Olga Weiss, André Wehe, and Melissa Yahya. The Importance of Run-Time Error Detection. In *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 145–155. Springer, 2009.

[7] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[8] 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*, pages 31–39, 2020.

A. Reproductibility

The MPI Bugs Initiative (MBI) is available on Gitlab at `https://gitlab.com/MpiBugsInitiative`. All verification tools are free except for ITAC who source code are not available, but a binary executable is freely available. All evaluation script and code generator source code is available in the repository.

Because of the nature non-determinist of hazard errors, results is not fully reproducible, but the results must be close for each reproduction.

A. Installation

MBI use a Docker to run script with controlled environment and facilitate usage for user. Container usage guaranty the user software environment and system configuration as no effect on experiments, is easier to reproduce results, because of user as no need to set up a specific software environment expect a working Docker installation on system. In addition, evaluation processes is fully automated and can produce some plots, latex and HTML files for results visualization.

The MBI experiments are performed in a Docker image derived to Ubuntu version 20.04 with all tools and dependency, except for Aislinn and MPI-SV who use their own docker image, Ubuntu 18.04 for Aislinn due to dependency issues, and we use the Docker image provided by the authors for MPI-SV.

```
docker build -f Dockerfile -t mbi:latest .
docker run -it --shm-size=512m mbi bash
```

In the Docker, the `/MBI/MBI.py` script provides a centralized interface for all MBI features: benchmarks generator, tools execution and data analysis.

B. 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>
python3 ./MBI.py -c run -x
↪  <tool1>,<tool2>,...
```

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:

- *test_name*.txt: that contains the tool output for that test
- *test_name*.elapsed: hat gives the wall clock 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.

C. Data analysis

Once all logs are in cache, the LaTeXtables 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 8. The upper part gives a quick glance on the categorization of each tool and test code (using colored icons), while the lower

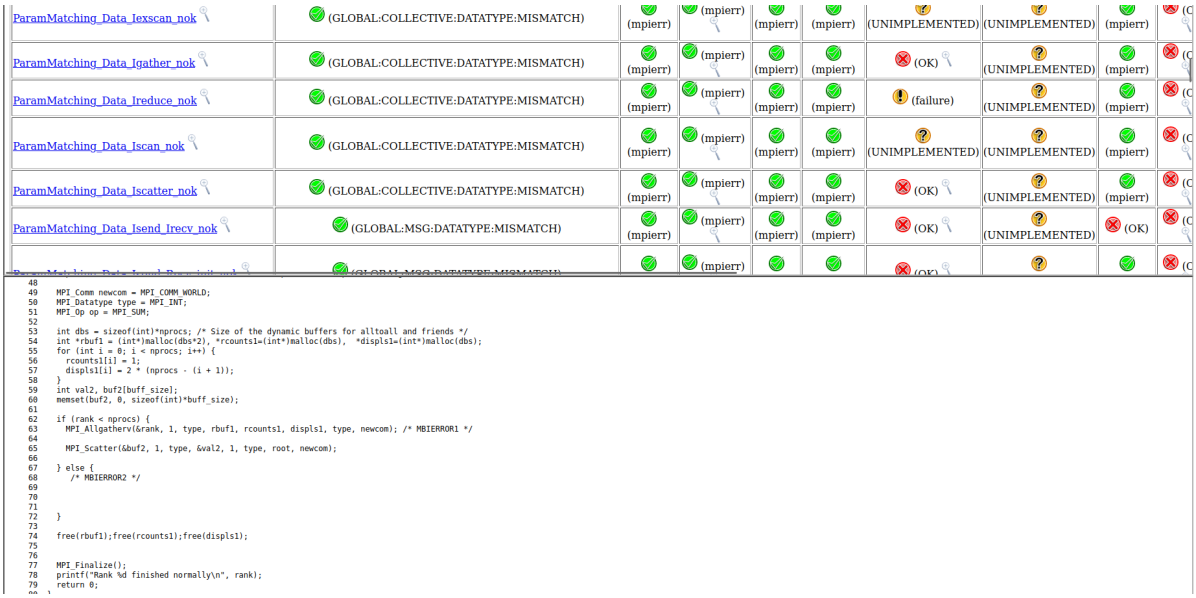


Figure 8: Screenshot of the MBI internal dashboard.

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.

A set of plots can be generated to explore the logs in cache as follows.

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

This command generate some bars and radar plots useful to visualize for tools outputs results summary. All plots used in this document are generate with this command.

D. 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.