Verifying MPI Applications with SimGridMC

The Anh Pham, Thierry Jéron, Martin Quinson

Verifying MPI Applications with SimGridMC

The Anh Pham
IRISA laboratory (Inria / ENS Rennes / Université de Rennes 1 / CNRS), Rennes, France.
the-anh.pham@inria.fr

Thierry Jéron
IRISA laboratory (Inria / ENS Rennes / Université de Rennes 1 / CNRS), Rennes, France.
thierry.jeron@inria.fr

Martin Quinson
IRISA laboratory (ENS Rennes / Université de Rennes 1 / Inria / CNRS), Rennes, France.
martin.quinson@ens-rennes.fr

ABSTRACT

SimGridMC (also dubbed Mc SimGrid) is a stateful Model Checker for MPI applications. It is integrated to SimGrid, a framework mostly dedicated to predicting the performance of distributed applications. We describe the architecture of McSimGrid, and show how it copes with the state space explosion problem using Dynamic Partial Order Reduction and State Equality algorithms. As case studies we show how SimGrid can enforce safety and liveness properties for MPI applications, as well as global invariants over communication patterns.

CCS CONCEPTS

• **Software and its engineering** → **Model checking**; **Software verification**; **Dynamic analysis**; **Formal software verification**; *Ultra-large-scale systems*; • **Theory of computation** → *Parallel algorithms*;

ACM Reference Format:
The Anh Pham, Thierry Jéron, and Martin Quinson. 2017. Verifying MPI Applications with SimGridMC. In *Proceedings of Correctness'17: First International Workshop on Software Correctness for HPC Applications (Correctness'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3145344.3145345

1 INTRODUCTION

Writing parallel and distributed programs poses notorious correctness challenges. In addition to intrinsic concurrent programming difficulties (*e.g.*, possible race conditions), distributed programming adds some of its own, such as the lack of centralized memory and the lack of a centralized clock. Modern High Performance Computing (HPC) applications must take on all these difficulties when attempting to aggregate and fully exploit the computational power of a set of multi-core nodes. In this context, the classical approach to ensure correctness is to rely on rigid communication patterns, so as to avoid complex synchronization scenarios. Unfortunately, these rigid communication patterns scale poorly. The size of modern compute platforms thus mandates applications with communication patterns that are irregular and dynamic. However, it then becomes virtually impossible to ensure correctness of such applications via classical testing approaches. There is thus a strong need for new correctness verification tools that rely on formal methods.

Among existing formal methods, Model Checking is appealing because it can be fully automated and is therefore usable by users without any formal methods expertise. It consists in checking that all possible behaviors of the program satisfy a given property. If the property does not hold, a counter-example is produced. That helps the practitioner to understand the reason. Usually, Model Checking is applied to a system model, often manually devised. In this work, we propose a practical approach to automatically perform the Model Checking directly of HPC codes written with the Message Passing Interface (MPI). This approach makes it possible to discover bugs that would most likely remain undiscovered using classical testing methods.

The contributions of this paper are the following:

- We present McSimGrid, an extension of the SimGrid framework that allows to Model Check regular MPI applications. It builds upon SimGrid's ability to observe and steer the execution of distributed programs, as initially intended for performance prediction.
- We discuss the techniques we use to mitigate the State Space Explosion Problem, a well-known issue with Model Checking due to the use of an exhaustive search. While the verification of applications at large scale remains for now intractable, these techniques allow us to find bugs in the dynamic and irregular communication patterns of modern HPC applications thanks to a semantically exhaustive exploration at a smaller scale.
- Via several use cases we showcase the kind of properties that can be enforced with McSimGrid on unmodified MPI applications written in C/C++ or Fortran, namely, safety properties, liveness properties, and global invariants over the communication patterns.

This article is organized as follows. Section 2 introduces the basics of McSimGrid and its state space reduction techniques. Section 3 presents use cases. Section 4 presents an experimental evaluation of our technique. Section 5 compares our work to the state of the art. Finally, Section 6 summarizes our contribution and highlights future work directions.

2 MODEL-CHECKING WITH SIMGRID

2.1 Principles

SimGrid is a simulator of distributed platforms that can emulate existing MPI applications using two virtualization mechanisms. First, the distributed application is *folded* into a single image. All

Correctness'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5127-0/17/11…$15.00
https://doi.org/10.1145/3145344.3145345

MPI processes are compiled as threads within a single OS process. Second, all communication is mediated through the simulator by a specific reimplementation of the MPI standard. This virtualization, detailed below, was initially proposed as a way to leverage SimGrid's simulation model so as to predict the performance of unmodified MPI applications. McSimGrid exploits it to formally assess the correctness of the application.

When running in Model-Checking mode, SimGrid explores all the possible execution paths of the application, starting from an initial application configuration and for a fixed set of inputs. This last restriction makes the tool better are adapted to application that are not data-dependent.

We consider a fully distributed model, in which each process executes sequentially and interacts with others only through message exchanges. In our model, execution paths are completely determined by the communications. The only indecision points at which the execution can branch are situations where a given process is waiting for a message that can come from more than one sender. For example, if two processes (*i.e.*, MPI ranks) $rank_1$ and $rank_2$ send a message to process $rank_0$, which accepts any incoming message, then McSimGrid will (1) completely explore all scenarios where the message of $rank_1$ arrives first, then (2) rewind the application and (3) completely explore all scenarios where the message of $rank_2$ arrives before that of $rank_1$. Similarly, the communication order can induce indecision points when functions such as `MPI_Waitany()` or `MPI_Testsome()` are used.

McSimGrid is limited to single-threaded applications, even though modern HPC applications tend to be multi-threaded. The major technical hurdles to overcome this limitation would be to *observe* the memory accesses, and accordingly *steer* the thread scheduling to explore all the possible interleavings of thread actions. The Divine [19] tool enables such exploration, but does not handle distributed computations based on message passing. Divine's and our approach are thus complementary and could in principle be combined.

2.2 Implementation Considerations

McSimGrid leverages the SimGrid' virtualization facilities, that are fully described in [4]. SimGrid is organized as an operating system, where the simulated processes run isolated from their environment. Any interactions (communications, computations which duration is reported to the simulator or regular synchronization through mutex or semaphore) can only be done through so-called *simcalls* [1]. Historically, this isolation was introduced to enable the execution of simulated processes, and it proves fundamental to the implementation of McSimGrid on top of the simulator. A transition in the model checker represents the local execution of a given process between two simcalls. It is considered to be atomic, and its potential impact on the environment is limited to the final simcall.

Our main focus when reimplementing MPI on top of SimGrid was on the prediction quality of the application's performance. We carefully validated our predicted timings to the one observed with real implementations [4]. Since the collective operations play a critical role in the accuracy of the predictions, we decided to reuse the source code of the main open-source implementations: OpenMPI and MPICH. SimGrid switches between the possible algorithms for each collective operation in the exact same conditions than the mimicked implementations of the standard.

This naturally helps ensuring McSimGrid: our MPI semantic can only diverge from the real implementation at the point-to-point level, since the collective operations are exactly the same than in real implementations. At the moment, we use a simple semantic for point-to-point communication, where communications always progress once started, regardless of their size or other parameters.

To rewind the verified application, SimGrid implements an efficient system-level checkpoint and rollback mechanism, detailed in [8]. The application and the model-checker run in separate UNIX processes. On checkpoints and rollbacks, the model-checker directly reads and write into the application process. Both processes communicate through the local network to synchronize.

For further efficiency, the snapshots are decomposed into memory pages, that are shared between snapshots if their contents remain unmodified. This proves very efficient because most transitions only modify a small fraction of the application's memory, resulting in highly similar snapshots.

2.3 Dynamic Partial Ordering Reduction

The Model Checking approach suffers from the well known *state space explosion problem*, meaning that the number of execution paths to explore can easily become intractable in practice. However, many of the execution paths are redundant in practice. SimGrid provides two reduction techniques to detect and avoid those redundant paths: Dynamic Partial Ordering Reduction (DPOR) and State Equality.

Dynamic Partial Ordering Reduction was first proposed by Flanagan *et Al.* in [5]. The key idea is that some events are *independent* of each other, meaning that their relative ordering has no impact on the final outcome. For example, local events occurring on separate hosts are independent. If two events t_1 and t_2 are independent, then two histories only differing by the order of t_1 and t_2 are semantically equivalent. It is then sufficient to explore only one execution path in each such equivalence class (called Mazurkiewicz traces). Several techniques have been proposed in the literature to avoid re-exploring redundant paths. For example in [1], the authors provide an optimal algorithm that explores exactly one ordering per Mazurkiewicz trace, but unfortunately proves computationally intensive. The authors propose another algorithm that may explore more than one ordering per Mazurkiewicz class, but that proves more efficient in practice.

McSimGrid implements a classical DPOR algorithm, at the level of point-to-point (P2P) communications. The collective operations that are implemented on top of it benefit of this reduction with no extra modeling effort. We presented in [14] a formal specification of such communications and determined sufficient conditions under which two given communications are independent. Working only at the level of P2P communications is much simpler than the previous attempt to model the semantic of the whole MPI standard, which

[1] Although processes can also interact through mutexes and semaphores in SimGrid, this is not yet handled by McSimGrid that is thus limited to purely distributed applications.

relied on almost 200 pages of TLA+ specification [24]. Indeed, McSimGrid would constitute a very convenient environment to study the semantics of collective algorithms, such as the non-blocking collectives.

DPOR cannot be used to enforce liveness properties in McSimGrid: our implementation of DPOR breaks the cycles, which could prevent to detect some liveness violations.

2.4 State Equality Detection

State Equality is another reduction technique is based on the simple idea that there is no need to explore twice the outcome of a given state. In abstract models, detecting that the verified system has returned to an already visited state is as simple as computing a hash of all known variables' value. In the case of MPI programs, the problem is much more challenging. Our approach, detailed in [8], is to partially reconstruct the semantics of the process memory (global variables, heap, and stack) using an array of tools and libraries that were initially intended for use by debuggers. It is then possible to design a heuristic that only considers relevant (*i.e.*, actually used) bits during state comparison. Our heuristic is efficient in practice even if it remains fallible because memory semantics cannot always be perfectly reconstructed (e.g., the heap area). Unfortunately, it is currently impossible to activate both DPOR and State Equality reduction at the same time in McSimGrid.

3 MPI VERIFICATION USE CASES

This section highlights several typical uses cases enabled by McSimGrid, namely, the discovery of safety and liveness bugs, as well as determining global invariants over communication patterns. These use cases are for unmodified MPI applications at small- to mid-range scale.

3.1 Safety Properties

Safety properties are the simplest type of properties that can be checked with McSimGrid, as they consist of simple assertions. To find a counter-example, a model checker simply searches for a state in which the assertion does not hold. McSimGrid is an explicit-state Model Checker that explores the state space by systematically interleaving process executions in depth-first order, storing a stack that represents the schedule history.

In previous work, we showed how McSimGrid can efficiently find bugs in several MPI programs that were specifically written as experimental test cases [14]. The *DPOR* technique was shown to greatly reduce the size of the explored state space during exhaustive verifications. McSimGrid also found a bug in our implementation of the Chord P2P protocol that proved challenging to isolate through testing.

3.2 Liveness Properties

These properties are often expressed in *Linear Time Logic* (LTL). As such, they combine first-order propositions with quantifiers over time. For example, LTL allows one to express that a given property P holds true forever (noted $\Box P$), or that it eventually becomes true at some point in the future (noted $\Diamond P$). The fact that once you press on the brake pedal the car will slow down after a finite number of events is a classical liveness property.

Counter-examples to such properties are infinite execution paths taken by the application without reaching the expected state. Since actual computer systems are finite, such infinite paths must contain cycles. The Model Checker must thus search for cycles in the execution occurring after an eventual triggering event ("the brake pedal is pressed" in our previous example) and before the expected event ("the car slows down").

The classical approach is to build a Bücchi automaton that represents the opposite of the considered property (such automata encode and recognize infinite sequences) and to explore in a double-DFS the cross-product of the automaton with the application. Once the exploration reaches an accepting state (*i.e.*, a state satisfying the triggering condition) and until the ending event, the Model Checker actively searches for loops: if it explores again a state that was already explored since the accepting state, then an infinite loop that violates the property has been found.

When considering real applications, a key difficulty is to evaluate whether the application has reached a state that was already explored. To address this difficulty, McSimGrid leverages the features that form the basis for the State Equality reduction technique, presented earlier. We used McSimGrid to verify several applications from the MPICH3 testsuite (consisting of up to 1,300 lines in C or Fortran), exhaustively searching for non-progressive loops (*i.e.*, livelocks) in various scenarios involving 2 to 6 processes. McSimGrid was able to verify these scenarios in less than a day with State Equality reduction enabled, proving the absence of livelocks in these applications. McSimGrid was also able to find a bug in an erroneous implementation of mutual exclusion in which the request of a given host were deliberately never answered. These results are detailed in a research report [8].

To the best of our knowledge, McSimGrid is the only tool available today that can formally assess liveness properties over unmodified MPI applications, even if several tools exist for other systems [12]. We would like to use it to enforce more interesting liveness properties on HPC code. This work is our first step toward engaging the HPC community in a view to identifying such liveness properties as well a relevant production code to which our tool could be applied.

3.3 Invariants over the Communication Pattern

In [2], the authors manually inspected 27 HPC applications to verify that the MPI ranks always send messages in the exact same order. Their motivation was that highly efficient checkpointing algorithms can be used on applications that exhibit this *Send Determinism* property. More formally, this property means that there exists a local order of *send* events (local to each MPI rank) that holds for every possible execution path. Specifying such properties require a branching-time logic, such as *Computational Tree Logic* (CTL) instead of LTL.

To enforce this property, McSimGrid first explores a random execution path to discover an event order, and then verifies whether this order holds for every other paths. The results previously obtained in [2] were perfectly reproduced: our automatic evaluation reached the same conclusion that the manual inspection found for each application whose source code was available to us. To the best

of our knowledge, McSimGrid is the only tool with the ability to automatically verify such properties for MPI applications despite their importance to efficiently yet correctly checkpoint the applications.

4 EVALUATION

This section evaluates the efficiency of the Dynamic Partial Ordering Reduction technique while conducting liveness analysis. We selected several small programs from the MPICH3 test suite. These programs are very small (some hundreds of lines each), written in either C or Fortran. Most of the MPICH3 test suite runs without any modification in SimGrid (and thus in McSimGrid too), while some tests leverage MPI-3 functions that are not reimplemented in SimGrid yet.

In this evaluation, we selected applications that test both point-to-point communications and group communications. We run an exhaustive verification, searching for deadlocks and non-progression cycles (which is a very simple liveness property). No experiment is allowed to run longer than 24 hours. We compare the time and space performance without any reduction, with the state equality reduction (but not the DPOR), and with both the state equality reduction and the memory compression. Technical reasons in McSimGrid make it impossible to activate DPOR while conducting such a liveness analysis, or to activate both DPOR and the state equality reduction at the same time.

The timings were obtained on an host equipped with a 48-cores Intel(R) Xeon(R) CPU E7540 @ 2.00GHz processor with 512GiB RAM, running under Debian wheezy 3.2.0-4-amd64. The results are presented in Table 1.

Even on such small applications, the amount of visited states growth very quickly if the state space is not reduced. However, our state equality heuristic detects many of these states as semantically equal. This makes it possible to explore scenarios that were simply impossible without reduction.

For example, the time to exhaustively explore `dup` for 4 processes drops from 7 hours and 36 minutes without reduction to only 1 minute with reduction. That way, the application can be explored with 5 processes (in 6 hours and 32 minutes). This is however not sufficient to explore the scenario with 6 processes in less than 24 hours.

For the `sendrecv2` application, the exhaustive exploration failed to complete in less than 24 hours while the reduction makes it possible to explore this scenario in a few seconds. Similar observations can be made with the `inplacef` application (that is written in Fortran).

Concerning the memory, the several verification consume dozens or hundreds of GiB to complete. The proposed memory compression schema reduces this to less than 16 GB, at the cost of a small time overhead (less than 10%). These experiments show that the state space reduction is an efficient answer to the exploration time issue while the memory compression is an efficient answer to the memory consumption problem posed. Both techniques are thus mandated to conduct the exhaustive verification of liveness properties on less powerful machines.

5 STATE OF THE ART

The idea of applying model checking to actual programs originated in the late 90s [6, 15]. It was later applied to many systems and interfaces such as Java [25], multithreaded programms [16] or distributed programms [13]. In the context of MPI-based parallel programs also, several tools have been proposed [7].

Runtime instrumentation tools such as Marmot [11] or MUST [9] intercept and verify every MPI call. This catches API misusages, such as type mismatch between a send and the corresponding receive. In addition, a dependency graph of the calls (either centralized in Marmot or distributed in MUST) can catch some deadlocks. Unfortunately, such testing tools only explore *some* of the possible execution paths.

Several static tools based on code analysis were proposed. TASS [22] and CIVL [23] rely on symbolic execution and state enumeration techniques to propagate the interval of values taken by the application variables. It requires source code annotations specifying bounds on input variables to reduce the amount of false positives. Similarly, PARCOACH [20] detects through static analysis potentially problematic sections of code, and adds assertions which are then checked at runtime. This does not require any code annotations but will miss failures that occur on unexplored execution paths. Formal approaches, such as that implemented in this work, are needed for an exhaustive coverage.

One of the first formal tools specifically designed for MPI programs was MPI-Spin [21], an extension to the classical Spin Model-Checker. But it requires the user to manually build an abstracted model of the application. Gauss [17] is an automated model extractor, but it remains limited to small applications because much information that is required to build an efficient and accurate model of the application are only known at runtime.

Many tools use the PMPI instrumentation layer of MPI to observe and steer the application. Nasty-MPI [10] delays the calls to experience pessimistic schedules that often trigger bugs in the application. ISP [26] and DAMPI [27] are formal tools that dynamically explore all the possible execution paths while applying adequate reduction techniques to not re-explore Mazurkiewicz traces when possible.

These last two efforts are the closest to our work. The main difference is that instead of mediating the calls on top of a real MPI implementation through the PMPI layer, we leverage the SimGrid reimplementation of MPI. Our solution is more demanding since our MPI implementation must faithfully reproduce the behavior of production MPI runtimes. McSimGrid leverages the major modeling effort in SimGrid toward realistic performance modeling of MPI applications [4].

This provides many advantages. Since the application is locally folded into a single process, the verification remains centralized and fast where PMPI-based solution have to rely on distributed and potentially complex algorithms. We have perfect knowledge of the application state whereas PMPI-based solutions have no information on the runtime internals. Verifying an application using non-blocking collective operations is very complex at the PMPI level, because the model checker is not aware of the algorithms used for these collectives, nor of their internal state during the execution. By contrast, McSimGrid verifies the collective implementations and

Application (language)	# P	Without any Reduction			With StateEq Reduction			With StateEq Reduction and Memory Compression		
		# States	Time	Memory	# States	Time	Memory	# States	Time	Memory
bcasttest (C)	3	> 1 million	> 24 h	-	4 823	18 min 31 s	37 GB	4 823	25 min 23 s	1.01 GB
bcastzerotype (C)	5	12 135 948	1 h 22 min	0.35 GB	4 734	6 min 35 s	5.83 GB	4 734	6 min 50 s	0.84 GB
	6	> 263 millions	> 24 h	-	56 054	9 h 03 min	62.4 GB	56 054	10 h 02 min	7.16 GB
commcreate1 (C)	4	102 289	44 s	0.35 GB	1 556	1 min 19 s	2.48 GB	1 556	1 min 28 s	0.49 GB
	5	12 710 034	1 h 23 min	0.35 GB	8 359	23 min 22 s	10.56 GB	8 359	25 min	1.48 GB
	6	> 274 millions	> 24 h	-	99 235	23 h 14 min	108.36 GB	99 235	24 h 55 min	14.18 GB
dup (C)	2	907	2 s	0.35 GB	81	2 s	0.45 GB	81	2 s	0.35 GB
	3	138 678	43 s	0.35 GB	405	5 s	0.77 GB	405	7 s	0.36 GB
	4	78 082 843	7 h 36 min	0.35 GB	2 352	1 min 04 s	2.99 GB	2 352	1 min 16 s	0.62 GB
	5	> 276 millions	> 24 h	-	39 263	6 h 32 min	47.63 GB	39 263	7 h 06 min	5.83 GB
groupcreate (C)	4	102 289	31 s	0.35 GB	1 205	40 s	1.86 GB	1 205	44 s	0.44 GB
	5	12 710 034	1 h 22 min	0.35 GB	6 237	11 min	7.62 GB	6 237	11 min 21 s	1.21 GB
	6	> 272 millions	> 24 h	-	80 878	16 h 15 min	89.31 GB	80 878	17 h 35 min	11.47 GB
inplacef (Fortran)	3	> 182 millions	> 24 h	-	2 941	1 min 07 s	3.87 GB	2 941	1 min 15 s	0.73 GB
op_commutative (C)	3	358	2 s	0.35 GB	94	2 s	0.46 GB	94	2 s	0.35 GB
	4	102 289	31 s	0.35 GB	1 545	1 min 16 s	2.47 GB	1 545	1 min 20 s	0.48 GB
	5	12 710 034	1 h 23 min	0.35 GB	10 998	48 min 42 s	14.25 GB	10 998	53 min 29 s	1.79 GB
sendrecv2 (C)	2	> 156 millions	> 24 h	-	1 877	28 s	3.25 GB	1 877	30 s	0.49 GB

Table 1: Exhaustive exploration of MPI applications coming from the MPICH3 integration test suite.

their internal state together with the application using them. In addition, the state equality detection (which is mandatory for verifying liveness properties) is simply impossible with PMPI mediation.

6 CONCLUSION AND FUTURE WORK

SimGrid is a stable simulator of distributed applications, which can be used to study MPI programs written in C/C++ or Fortran. In this paper, we presented a full-featured model-checker that extends this framework, McSimGrid. McSimGrid can be used to verify safety properties, but also liveness properties, or even other properties such as communication determinism. To the best of our knowledge, McSimGrid is the only tool able to enforce formal properties, beyond just safety properties, for unmodified MPI applications. It implements two well known techniques (DPOR and *State Equality*) to mitigate high exploration times. In spite of these techniques, formally verifying HPC applications at full scale remains intractable. However, we believe that many bugs, especially given the dynamic and irregular communication patterns exhibited by modern HPC applications, can be found via exhaustive explorations at small scale.

Even if McSimGrid is already usable in practice, many extensions remain possible. We plan to add new reduction techniques, leveraging symmetries between the MPI ranks or using true concurrency models such as event structures and their unfoldings [18]. The semantic of point-to-point communications should be enriched to catch all subtleties of the MPI standard. The state equality mechanism could be configurable, leaving to users the possibility to partially abstract their system state. We plan to verify student-produced MPI projects with McSimGrid. This would be particularly helpful along with the MPI pedagogic projects provided in [3] since testing and or validating (via code inspection) MPI code written by students is both tedious and error-prone. We also plan to use McSimGrid on larger applications and gather user feedback. In particular, throughout the upcoming year we would like to collect

examples of safety and liveness properties that are generally useful in typical modern HPC applications.

REFERENCES

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. *ACM SIGPLAN Notices*, 49(1):373–384, January 2014. ISSN 0362-1340. doi: 10.1145/2535838.2535845. URL http://doi.acm.org/10.1145/2578855.2535845.

[2] Franck Cappello, Amina Guermouche, and Marc Snir. On Communication Determinism in Parallel HPC Applications. In *Proceedings of the 19th International Conference on Computer Communications and Networks (IEEE ICCCN 2010)*, pages 1–8, 2010. doi: 10.1109/ICCCN.2010.5560143.

[3] Henri Casanova. The SMPI CourseWare. https://simgrid.github.io/SMPI_CourseWare/.

[4] Augustin Degomme, Arnaud Legrand, George Markomanolis, Martin Quinson, Mark Stillwell, and Frédéric Suter. Simulating MPI Applications: The SMPI Approach. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2387–2400, August 2017. doi: 10.1109/TPDS.2017.2669305.

[5] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *ACM Sigplan Notices*, volume 40, pages 110–121. ACM, 2005.

[6] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3. doi: 10.1145/263699.263717. URL http://doi.acm.org/10.1145/263699.263717.

[7] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R. De Supinski, Martin Schulz, and Greg Bronevetsky. Formal Analysis of MPI-based Parallel Programs. *Communication of the ACM*, 54(12):82–91, December 2011. ISSN 0001-0782. URL http://doi.acm.org/10.1145/2043174.2043194.

[8] Marion Guthmuller, Gabriel Corona, and Martin Quinson. System-Level State Equality Detection for the Formal Dynamic Verification of Legacy Distributed Applications. Inria Research Report, January 2015. URL https://hal.inria.fr/hal-01558049.

[9] Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. *MUST: A Scalable Approach to Runtime Error Detection in MPI Programs*, pages 53–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-11261-4. doi: 10.1007/978-3-642-11261-4_5. URL https://doi.org/10.1007/978-3-642-11261-4_5.

[10] Roger Kowalewski and Karl Fürlinger. *Nasty-MPI: Debugging Synchronization Errors in MPI-3 One-Sided Applications*, pages 51–62. Springer International Publishing, Cham, 2016. ISBN 978-3-319-43659-3. doi: 10.1007/978-3-319-43659-3_4.

[11] B. Krammer, K. Bidmon, M.S. Müller, and M.M. Resch. MARMOT: An MPI analysis and checking tool. *Advances in Parallel Computing*, 13:493 – 500, 2004. Parallel Computing.

[12] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off go: Liveness and safety for channel-based programming. *ACM SIGPLAN Notices*, 52(1):748–761, January 2017. ISSN 0362-1340.

[13] Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *6th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2009. URL https://www.usenix.org/legacy/events/nsdi09/tech/full_papers/yang/yang_html/.

[14] Stephan Merz, Martin Quinson, and Cristian Rosa. SimGrid MC: Verification Support for a Multi-API Simulation Platform. In *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, pages 274–288, 2011.

[15] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, December 2002. ISSN 0163-5980. doi: 10.1145/844128.844136. URL http://doi.acm.org/10.1145/844128.844136.

[16] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, volume 8, pages 267–280, 2008. URL https://www.usenix.org/legacy/event/osdi08/tech/full_papers/musuvathi/musuvathi_html/.

[17] Robert Palmer, Steve Barrus, Yu Yang, Ganesh Gopalakrishnan, and Robert M Kirby. Gauss: A Framework for Verifying Scientific Computing Software. *Electronic Notes in Theoretical Computer Science*, 144(3):95 – 106, 2006. ISSN 1571-0661. URL http://dx.doi.org/10.1016/j.entcs.2006.01.007. Proceedings of the Workshop on Software Model Checking (SoftMC 2005).

[18] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 456–469, 2015.

[19] Petr Ročkai. *Model Checking Software*. PhD thesis, Faculty of Informatics, Masaryk University, 2015.

[20] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. PARCOACH: Combining static and dynamic validation of MPI collective communications. *International Journal of High Performance Computing Applications*, 2014.

[21] Stephen F. Siegel. *Model Checking Non-blocking MPI Programs*, pages 44–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-69738-1. doi: 10.1007/978-3-540-69738-1_3. URL https://doi.org/10.1007/978-3-540-69738-1_3. 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007).

[22] Stephen F. Siegel and Timothy K. Zirkel. Automatic Formal Verification of MPI-based Parallel Programs. *SIGPLAN Not.*, 2011.

[23] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. Civl: The concurrency intermediate verification language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 61:1–61:12, New York, NY, USA, 2015. ACM. doi: 10.1145/2807591.2807635. URL http://doi.acm.org/10.1145/2807591.2807635.

[24] Sarvani Vakkalanka, Anh Vo, Ganesh Gopalakrishnan, and Robert M. Kirby. *Reduced Execution Semantics of MPI: From Theory to Practice*, pages 724–740. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-05089-3. doi: 10.1007/978-3-642-05089-3_46. URL https://doi.org/10.1007/978-3-642-05089-3_46.

[25] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, Apr 2003. ISSN 1573-7535. doi: 10.1023/A:1022920129859. URL https://doi.org/10.1023/A:1022920129859.

[26] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Formal Verification of Practical MPI Programs. *ACM SIGPLAN Notices*, 44(4):261–270, February 2009. ISSN 0362-1340.

[27] Anh Vo, S. Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'10, pages 1–10. IEEE Computer Society, Nov 2010.