Dynamic Symbolic Verification of MPI Programs

Dhriti Khanna[1], Subodh Sharma[2(✉)], César Rodríguez[3], and Rahul Purandare[1]

[1] IIIT Delhi, New Delhi, India
dhritik@iiitd.ac.in
[2] IIT Delhi, New Delhi, India
svs@cse.iitd.ac.in
[3] Université Paris 13, Sorbonne-Paris-Cité, LIPN, CNRS, Villetaneuse, France

Abstract. The success of dynamic verification techniques for Message Passing Interface (MPI) programs rests on their ability to address *communication nondeterminism*. As the number of processes in the program grows, the dynamic verification techniques suffer from the problem of exponential growth in the size of the reachable state space. In this work, we provide a hybrid verification technique for message passing programs that combines explicit-state dynamic verification with symbolic analysis. The dynamic verification component deterministically replays the execution runs of the program, while the symbolic component encodes a set of interleavings of the observed run of the program in a quantifier-free first order logic formula and verifies it for communication deadlocks. In the absence of property violations, it performs analysis to generate a different run of the program that does not fall in the set of already verified runs. We demonstrate the effectiveness of our approach, which is sound and complete, using our prototype tool Hermes. Our evaluation indicates that Hermes performs significantly better than the state-of-the-art verification tools for *multi-path* MPI programs.

Keywords: Dynamic verification · Message passing interface
Deadlock detection · Symbolic analysis

1 Introduction

Message passing (MP) is a prominent paradigm via which nodes of the distributed systems can communicate. Typically, the MP programs are run on large computer clusters and are developed not only by career computer professionals but also by unconventional programmers affiliated to other disciplines of science. However, designing MP programs is known to be a challenging exercise. Programmers have to anticipate the messaging patterns, perform data marshaling and compute the locations for coordination in order to design correct and efficient programs. Unsurprisingly, this design complexity lends itself to the verification complexity of MP programs. The problem of communication races, which

K. Havelund et al. (Eds.): FM 2018, LNCS 10951, pp. 466–484, 2018.
https://doi.org/10.1007/978-3-319-95582-7_28

leads to data corruption or communication deadlocks, plays a central role in the verification complexity of MP programs.

In the context of discovering communication deadlocks, the problem has been studied extensively over the years [2,11,14,17,19,20,23,25,28]. However, a scalable solution remains elusive; this is primarily due to the nondeterminism in the semantics of MP primitives. For instance, in MPI (Message Passing Interface [21], a popular standard for writing parallel programs in C/Fortran) use of the *wildcard receive* call can lead to nondeterministic matching with potential senders at runtime [30]. Presence (or absence) of buffering in MPI nodes can also contribute to nondeterminism; for instance, standard blocking sends semantics are dependent on the presence of system buffering – under no system buffering the send calls behave as synchronous calls while under infinite buffering the same send calls complete immediately without even requiring a matching receive call. MPI implementations allow nodes to provide buffering in order to improve the performance, however, the nondeterminism resulting from buffering can potentially introduce additional deadlocks [31].

It is worthwhile to note that nondeterministic communication of data can affect the control-flow of the program (*e.g.*, when the communicated data to a wildcard receive is used in a subsequent branch instruction of the program). Programs with the pattern mentioned above are termed as *multi-path* programs [9], and they significantly affect the scalability of existing verification techniques. Correspondingly, *single-path* programs are those (paraphrasing from [9]) where the program executes irrespective of the data communicated to a receive call, the same sequence of instructions, *i.e.*, the control-flow of the program remains unaffected by the communication actions in the program.

Explicit-state runtime model checkers of MPI programs, such as ISP [30] and DAMPI [33], can analyse multi-path (and single-path) MPI programs for the absence of communication deadlocks and assertion violations. However, they require the programs to be repeatedly run such that in each run a distinct communication pattern (such as a send-receive match) is explored. Though the said model checkers are exhaustive in their exploration under a fixed input and a buffering mode (viz. *zero* and *infinite* [31]), they suffer from a possible requirement of a considerable number of program re-runs. It is often the case that much of the time is spent in verifying the loops of the program containing only computation code that is of little relevance to establish the correctness of the communication structure of the program.

Recently, symbolic analysis techniques for MP program executions have been proposed. Although they do address the problem of program re-runs by symbolically encoding the interleavings to explore, they can only be applied to single-path programs [10,15,22]. These techniques are classified as *trace verification* techniques and the tools as *trace verifiers*.

Techniques which perform full-blown symbolic execution of a program can discover deadlocks and other assertion violations in multi-path programs [3,11, 28]. While they cover both the input space and communication nondeterminism,

they are known to not scale beyond a relatively small size of the program and a few processes.

We present a sound and complete technique to verify multi-path MPI that is complementary to the above-mentioned techniques. Our technique combines the strengths of trace verification and symbolic execution techniques, which, respectively, provide scalability and multi-path coverage. Furthermore, our technique is able to verify programs with not only zero- or inifinite-buffering, but also with nondeterministic buffering decisions up to a bound k. While we present the technique in the context of verifying MPI programs for the absence of deadlocks, it can be applied to other properties, such as assertion violations. We demonstrate the effectiveness of the technique by implementing it as a prototype tool HERMES and comparing it with state-of-the-art tools.

2 Overview

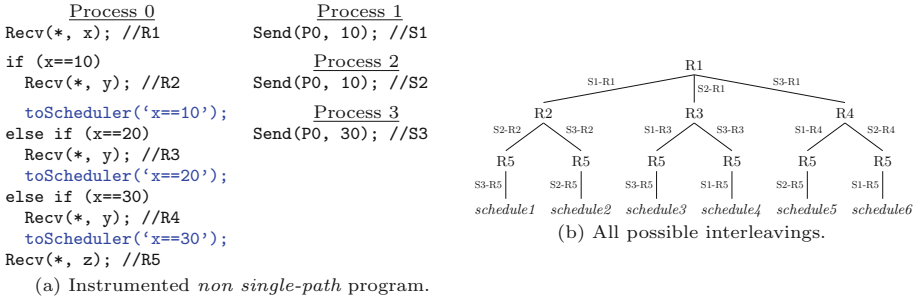
In this section, we present an overview of our hybrid method which discovers deadlocks in multi-path MPI programs. The technique exhaustively explores the executions of the program under a fixed input as follows: (i) it obtains a concrete run ρ of the program via dynamic analysis (via a scheduler that orchestrates a run); (ii) encodes symbolically the set of *feasible runs* obtained from the same set of events as observed in ρ such that each process triggers the same control-flow decisions and executes the same sequence of communication calls as in ρ (note that the encoding captures the entire set of runtime matches of communication events from ρ); (iii) check for violations of any property (in our case, communication deadlocks); and (iv) if no property is violated, then alter the symbolic encoding to explore the feasibility of taking an alternate control flow behavior which is different from ρ. In case of such a feasibility, initiate a different concrete run.

Consider the program shown in Fig. 1(a). It is a nondeterministic, multi-path, and deadlock-free program. The non-colored lines illustrate the pseudo-code of the program. It is worthy to note that *trace verifiers* will fail to verify the program since the program has multiple control flow branches and the nondeterministic matching choice of $R1$ governs the execution of these branches.

Our approach statically discovers the code locations where the received data or message tags (a field in MPI send and receive calls that serve as a unique marker for messages) are used to branch at conditional statements. At these locations, we instrument certain calls to a *scheduler*. The *scheduler* schedules the MPI calls of the program according to the MPI semantics and drives the execution. The scheduler is also responsible for building a partially ordered *happens-before* relation between these calls (refer Sect. 3). At runtime, the instrumented code communicates the predicate expression in the branching instruction to the scheduler. The instrumented code is shown in blue color in Fig. 1(a).

Based on a trace ρ the symbolic encoding is generated from the execution of the program with instrumented code. The communication events and the branching decisions made in this trace are modeled as an SMT formula. This

formula encodes all the semantically possible schedules of events observed in ρ, which follow the same control-flow decisions as made in ρ.



(a) Instrumented *non single-path* program.

(b) All possible interleavings.

Fig. 1. Multi-path program and its interleavings. (Color figure online)

In the example shown in Fig. 1(a), `Process 0` executes the first control flow branch if $R1$ matches with either $S1$ or $S2$. If $R1$ matches $S1$, the SMT formula will encode the interleavings 1 and 2 as shown in Fig. 1(b). An SMT solver is used to solve this formula, which checks for the violation of the safety property. This verifies two interleavings. If there is no property violation, we verify another control flow path that may have been taken if $R1$ had matched with some other send. To this effect, we want to change the path condition obtained from the trace to reschedule another execution through an unvisited control-flow. Hence, we alter the path condition (in a typical symbolic execution style) and execute the program again, so that it follows the path corresponding to the altered path condition. To force the scheduler to follow a different control-flow branch, we may also have to force a wildcard receive call to match with a send call that sends data different from the send call that matched before. We repeat this process until all the paths in the program are exhausted. In the context of the above example, in the second execution, $R1$ must match $S3$, and it must avoid matching $S2$ because $S2$ is sending the same data as $S1$ ($S1$ had already matched with $R1$ in the first execution). The encoding resulting from this run will cover schedules 3 to 6.

The example program has six possible interleavings across multiple control flow paths. Our technique executes the program only twice to cover all of them and thus shows the contrasting difference from trace verification which does not provide full path coverage and from dynamic verification which executes the program as many times as there are possible interleavings.

3 MPI Model and Execution Semantics

In this section we formalize the execution of an MPI program. The MPI runtime often provides buffers to store the data of issued but unfinished calls. The

presence of buffers can introduce subtle behaviors in a program [26,31]. Due to space limitations we present a zero-buffer semantics, but our results hold for *zero-*, *infinite-*, and κ-buffer modes, with $\kappa \in \mathbb{N}$.

We consider *MPI programs* consisting of $n \in \mathbb{N}$ processes $P_1, \cdots, P_n$ in a single communication group. Each process P_i manipulates a set of *process-local variables* Var_i. Let $Var = \bigcup_i Var_i$. We model the execution of the program as a sequence of events, one for each executed MPI call. An event of process P_i is a tuple $e := \langle c, a, d \rangle$, where c is a *path constraint* over Var_i (describing the conditional branches taken by P_i to produce the event, the constraint language is left unspecified), a is an *MPI call*, and $d \in \mathbb{N}$ is a *depth* which increments monotonically with the events of P_i. We let E denote the set of all possible events, $p(e) := P_i$ the process of e, and $l(e) := a$ the MPI call of e.

Without loss of generality, we only model nonblocking MPI operations and `MPI_Wait`. Nonblocking calls return immediately with a handle that can later be passed to `MPI_Wait` for the process to block until the operation completes. An MPI call is either a nonblocking *send* (resp. *receive*) issued by process P_i to send data to (resp. receive from) process P_j with tag t, denoted by $S_{i,j,t}$ (resp. $R_{i,j,t}$); or a nonblocking *barrier* issued by process P_i, denoted by B_i; or a (blocking) *wait* issued by process P_i in order to wait for the completion of the so-called corresponding event at depth h, denoted by $W_{i,h}$. For a wait event $e := \langle c, W_{i,h}, d \rangle$ the *corresponding event*, $corr(e)$, is the only event of P_i whose depth is h.

The MPI runtime matches send and receive operations (among others) using the well-defined semantics. Given events $e := \langle c, a, d \rangle$ and $e' := \langle c', b, d' \rangle$, we say that e *matches before* e', denoted by $e_1 \prec_{mo} e_2$, iff $p(e) = p(e')$, and $d \leq d'$, and one of the following is satisfied: (i) a and b are send calls (resp. receive calls) to the same destination (resp. from the same source) with the same tags; (ii) a is a wildcard receive call and b is a receive call sourcing from the same process, or a wildcard receive and the tags of calls a and b are the same, or the tag of call a is a wildcard; (iii) a is a nonblocking call and b is an associated wait call.

We view the execution of an MPI program as a sequence of events in E^*, but not all sequences correspond to executions. We now define a Labeled Transition System (LTS) $\langle Q, \rightarrow, q_0 \rangle$ whose runs capture the valid executions. The *states* in Q are tuples of the form $\langle I, M, z \rangle$ where $I \subseteq E$ is the set of *issued* events, $M \subseteq E$ is the set of *matched* events, and $z \colon \mathbb{N} \rightarrow \mathbb{N}$ maps every process to the depth of the next event expected from that process. The *initial state* q_0 is $\langle \emptyset, \emptyset, z_o \rangle$, where z_0 maps all processes to 0.

The $\prec_{mo}$ order captures matching constraints that exclusively depend on the types of the calls involved in the check. However, matching $R_{i,*,t}$ calls requires information about the state. Given a state $s := \langle I, M, z \rangle$ and events $e := \langle c, R_{i,j,t}, d \rangle$ and $e' := \langle c', R_{i,*,t}, d' \rangle$, we say that e *conditionally matches before* e', denoted by $e \prec_{co} e'$, iff $d \leq d'$ and $\exists \hat{e} \in I$ such that $l(\hat{e}) = S_{j,i,t}$.

The transitions in $\rightarrow \in Q \times 2^E \times Q$ are labeled by sets of events representing either the issuing or completion of MPI calls to the runtime or the matching of

communication calls by the runtime. Thus, we have three classes of transitions: Issue, Match, and Complete transitions.

Issue transitions capture the call to a nonblocking MPI primitive represented by event $e := \langle c, a, d\rangle$. Formally, $\langle I, M, z\rangle \xrightarrow{\{e\}} \langle I \cup \{e\}, M, z'\rangle$ iff $d = z(p(e))$, and z' is equal to z except for $z'(p(e))$ which is $z(p(e)) + 1$, and I does not contain any event whose action is a wait from process $p(e)$.

Match transitions correspond to the MPI runtime matching a set of issued events (e.g., a send with a receive). Formally, $\langle I, M, z\rangle \xrightarrow{m} \langle I \setminus m, M \cup m, z\rangle$ exactly when there is some $m \subseteq I$ such that either of the three conditions hold: (i) $m := \{e, e'\}$ with $l(\langle e, e'\rangle) = \langle S_{i,j,t}, R_{j,i,t'}\rangle$ and $\nexists \hat{e} \in I, (\hat{e} \prec_{mo} e \vee \hat{e} \prec_{mo} e')$, and $t' \in \{t, *\}$; or (ii) $m := \{e, e'\}$ with $l(\langle e, e'\rangle) = \langle S_{i,j,t}, R_{j,*,t'}\rangle$ and $\nexists \hat{e} \in I, (\hat{e} \prec_{mo} e \vee \hat{e} \prec_{mo} e' \vee \hat{e} \prec_{co} e')$, and $t' \in \{t, *\}$; or (iii) $m := \{e_1, \ldots, e_n\}$ with $l(e_i) = B_i$ for all $i \in \{1, \ldots, n\}$ and $\nexists \hat{e} \in I, (\hat{e} \prec_{mo} e_1 \vee \ldots \vee \hat{e} \prec_{mo} e_n)$.

Finally, *complete transitions* correspond to calls to `MPI_wait` returning the control to the process because the corresponding event has already been matched. Formally, $\langle I, M, z\rangle \xrightarrow{\{e\}} \langle I \setminus \{e\}, M \setminus \{corr(e)\}, z\rangle$ iff $l(e) := W_{i,d}$ and $corr(e) \in M$.

An *execution trace* (or just *trace*) $\rho \in E^*$ is any sequence formed by the events contained in the singletons that label a run of the LTS which only uses *issue* transitions. An MPI program P has a *deadlock* if it can generate a trace ρ that ends in a *deadlocking state*, i.e., one with no successors in the LTS. Deciding whether a single-path program has a deadlock is an NP-complete problem [10], under finite and infinite system buffering. Note that our events are guarded MPI calls and thus each trace of P is essentially a single-path program. Since, P has a finite number of single-path programs, it follows that the deadlock detection problem under κ-buffering for P is also NP-complete.

4 Encoding Rules

In this section we define SMT encoding rules such that for a deadlocking execution trace ρ, the variables of the encoding get satisfied and the generated model provides information about the calls that remained unmatched in ρ. The first step is defining an (over-approximated) set of sets of events, $\mathbb{M}^+ \in 2^E$, consisting of all possible sets that the LTS could make using the *match transition* using the same events as in ρ but possibly issued in a different order.

Definition 1. *$\mathbb{M}^+$: The $\mathbb{M}^+$ set is an over-approximate set of the matching calls in a trace [10]:*

$$\begin{aligned}\mathbb{M}^+ = \{\{a, b\} \subseteq E \mid\ & a = \langle -, S_{i,j,-}, -\rangle, b = \langle -, R_{j,i/*,-}, -\rangle, \\ & \forall a' \prec_{mo} a\ \exists b' \not\prec_{mo} b : \{a', b'\} \in \mathbb{M}^+, \\ & \forall b' \prec_{mo} b\ \exists a' \not\prec_{mo} a : \{a', b'\} \in \mathbb{M}^+\} \\ \cup\ & \{\{a\} \subseteq E \mid a = \langle -, W_{i,h}, -\rangle\} \\ \cup\ & \{\{a_1, \cdots, a_N\} \subseteq E \mid\ \forall i \in \{1, \ldots, n\}, a_i = \langle -, B_i, -\rangle\}\ .\end{aligned}$$

Let $\mathbb{M}^+(a) = \bigcup\{b | \exists \alpha \in \mathbb{M}^+ : a, b \in \alpha\}$ be the set of all potential matching calls for the operation a. Let $Imm(a)$ be the set of all immediate ancestors of event a. We define it by $Imm(a) = \{x | x \prec a, \forall z : x \preceq z \preceq a \implies z \in \{x, a\}\}$. Note that $\prec = (\prec_{mo} \cup \prec_{co})$ (resp. for $\preceq$).

Following [10], we restrict our presentation to problems without barriers without introducing spurious models. In Fig. 2, we provide the list of rules that encode all feasible interleavings of a given ρ. First, we explain the meaning and purpose of each variable used in the encoding. We use *tag, src (resp. dest)*, and *val* as integer variables to encode MPI call operands such as message tag, sender's (resp. receiver's) identity, and the data payload, respectively. Note that for simplicity we assume the data payload to be of primitive types (such as Integer). In order to model an interleaved run, we use an integer variable *clk* for each call in ρ. Variables m and r are boolean variables which signify the matching and the *readiness* (all $\prec_{mo}$ ancestors of the event are matched) of an event, respectively. A boolean variable *bufferUsed* is used when an event uses the buffer provided by the MPI runtime. We refer the above mentioned variables corresponding to an event a by the variable name sub-scripted with event symbol, for instance, clock variable for event a is denoted by clk_a.

Corresponding to every $\alpha \in \mathbb{M}^+$ we have a boolean variable s_α which we set to true when the events in α occur as a match in ρ. We further define $\mathcal{I} \subseteq E_\rho$ to be the set of event pairs (a, b) such that a and b are consecutive sends from one process but with different destinations. Buffering a can potentially impact the $\prec_{mo}$ relation with respect to call b. When both a and b are send calls targeting the same destination, then, despite buffering, the $\prec_{mo}$ relation between a and b stands unmodified. This is because the FIFO matching guarantee provided by the MPI standard is impervious to the underlying system buffering.

In Fig. 2, most SMT rules are similar to the propositional rules from [10], except Rules 2, 5, 10, 11 and 14. Rule 1 encodes the $\prec_{mo}$ relation with an exception – the order between a pair of send calls $(a, b) \in \mathcal{I}$ is encoded in Rule 10. Rule 5 encodes the semantics of $\prec_{co}$ ordering. We start with an over-approximate set $\mathcal{E}$ that has pairs of receive calls (deterministic receive call, a, followed by a wildcard receive call b). An order is established between such a pair only when there is a *ready* send call that can match a but no send call that can match b. To record the notion of time at which calls become ready, we use the variable r_{clk}. Rules 12 and 13 encode the deadlock detection constraints.

Encoding for κ-Buffer Mode Semantics: Rules 10 and 11 encode the behavior of a program with κ-buffer semantics. The maximum number of buffer slots, κ, available with the program is provided by the user. If a buffer slot is available, the partial order relation between some of the send calls can be relaxed (as explained before).

Encoding for Path Condition: Rule 14 encodes the guards of each event in E_ρ. The guards are the path constraints obtained from the expressions of conditional statements encountered along the program execution.

1. Partial Order: $\bigwedge_{b \in E_\rho} \bigwedge_{a \in Imm(b):(a,b) \notin \mathcal{I}} clk_a < clk_b$
2. Match Pair: $\bigwedge_{a:(s,r) \in \mathbb{M}^+} s_a \rightarrow (clk_s = clk_r) \wedge (data_s = data_r) \wedge (tag_s = tag_r)$
3. Unique Match for Send: $\bigwedge_{(a,b) \in \mathbb{M}^+} \bigwedge_{(a,c) \in \mathbb{M}^+} s_{ab} \rightarrow !s_{ac}$
4. Unique Match for Receive: $\bigwedge_{(a,b) \in \mathbb{M}^+} \bigwedge_{(c,b) \in \mathbb{M}^+} s_{ab} \rightarrow !s_{cb}$
5. Conditional Partial Order:
   (a) $\bigwedge_{a \in E_\rho} r_a \rightarrow r\_clk_a = clk_{Imm(a)} + 1$
   (b) $\bigwedge_{(a,b) \in \mathcal{E}} \bigwedge_{c \in \mathbb{M}^+(b)} (r\_clk_c > r\_clk_a \wedge \bigwedge_{d \in \mathbb{M}^+(a)} r\_clk_c > r\_clk_d) \rightarrow clk_a < clk_b$
6. Match Correct: $\bigwedge_{a \in r} (m_a \rightarrow \bigvee_{(b,a) \in \mathbb{M}^+} (s_{ba})) \wedge \bigwedge_{a \in s} (m_a \rightarrow \bigvee_{(a,b) \in \mathbb{M}^+} (s_{ab}))$
7. Matched only: $\bigwedge_{a:(a_1,a_2,\ldots,a_n) \in \mathbb{M}^+} s_a \rightarrow \bigwedge_{a_i \in a} m_{a_i}$
8. All Ancestors Matched: $\bigwedge_{b \in E_\rho} (r_b \leftrightarrow \bigwedge_{a=Imm(b)} m_a)$
9. Match Only Issued: $\bigwedge_{a \in E_\rho} (m_a \rightarrow r_a)$
10. Use Buffer: $\bigwedge_{(a,b) \in \mathcal{I}} (r_a \wedge k > 0 \rightarrow (bufferUsed_a \wedge dec(k))) \wedge (k = 0 \rightarrow !bufferUsed_a \wedge clk_a < clk_b)$
11. Free buffer: $\bigwedge_{a \in E_\rho} k = ite(m_a \wedge bufferUsed_a, inc(k), k)$
12. No Match Possible: $\bigwedge_{a \in \mathbb{M}^+} (\bigvee_{t \in a} (m_t \vee !r_t))$
13. Not All Matched: $\bigvee_{a \in E_\rho} !m_a$
14. Path Condition: π_ρ

Fig. 2. SMT rules.

Theorem 1. *Given a trace ρ of program P, ρ ends in a deadlocking state* iff *there is a satisfying assignment of the variables to values in the SMT encoding of the deadlock detection problem.*

The proof of this theorem is similar to the one used in [10]. It suffices to show that (i) for every deadlocking trace of the program, the encoding rules are satisfied, and (ii) whenever the rules are satisfied, the trace deadlocks. For this, we construct a trace corresponding to the model generated by the solver (which conforms to the MPI semantics) and prove that it is deadlocking. Theorem 1 formally establishes the correctness of our SMT encoding.

5 Design

Hermes comprises of three components: program instrumenter, constraint generator and solver, and instruction rescheduler. In this section, we describe the functionality of these components and present the algorithm that Hermes implements. Figure 3 gives an architectural overview of Hermes.

5.1 Components

Program Instrumenter: The instrumenter is developed using Clang [5]. Clang is the front-end for the LLVM compiler and provides features to analyse, optimize, and instrument C/C++ source code. It targets the locations in the code where the data or the tag received in a wildcard `receive` is decoded in the predicate expression of a conditional construct, the body of which issues an MPI call. It instruments these locations with TCP calls to the scheduler which is responsible for driving the execution of the program. The instrumented calls are used to communicate the predicate expressions to the scheduler at runtime. We use ISP's dynamic verification engine as the scheduler.

Constraint Generator and Solver: The instrumented program is input to the ISP scheduler which executes it. The execution of the program drives the instrumentation to generate a path condition π which corresponds to the expressions of the conditional constructs encountered at target locations during the run. We also generate a set of potential matches, $\mathbb{M}^+$, and a sequence of MPI events, ρ, from the run. $\mathbb{M}^+$, π, and ρ are used to encode the trace of the program in the form of SMT rules given in Sect. 4, which conform to the MPI semantics. The satisfiability of the rules signifies the presence of a deadlock. Please note that the technique presented in this paper is a general verification technique to cover all possible schedules for a given input. The encoding rules provided in Sect. 4, however, are targeted for deadlock detection.

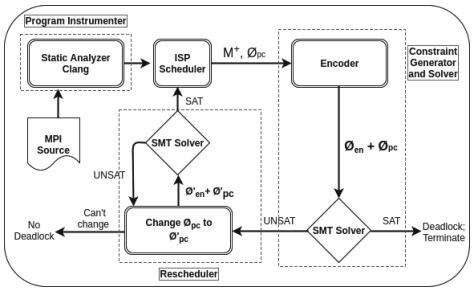


Fig. 3. Architecture of the approach.

Instruction Rescheduler: If the SMT solver cannot generate a model (UNSAT query), we modify π in a way similar to the concolic execution to try and infer the existence of another executable control flow path. In a non-chronological order, we perform a depth-first search over a tree of control flow paths where the branching points in the tree map to the conditional constructs in the target program. The resulting formula is denoted by ϕ. Although we modify the path condition in a fashion similar to concolic execution, we do not inherit its legacy problem of path complexity. This is because (i) unlike concolic execution, we do not symbolically interpret the entire program, and (ii) in our experience, the conditional expressions in multi-path MPI programs are simple equality checks (we have not come across benchmark programs where the relevant conditional expressions were complex).

Algorithm 1. Deadlock Detection.

```
   Data: Instrumented Program: P
   Result:
   1. Guarantee that a deadlock does not occur
   2. Model of the MPI calls if a deadlock is present
 1 <Path Condition: π; Trace: ρ; Potential Matches: M+ > = execute P
 2 while true do
 3     φ_en = encode(M+, ρ)
 4     res = solver(φ_en ∪ φ_π)
 5     if res == SAT then
 6         report 'Deadlock'; exit
 7     else
 8         φ = SearchDifferentPath(ρ, π, M+)
 9         if φ = ∅ then
10             report 'No Deadlock'; exit
11         else
12             < π, ρ, M+ > = execute P conforming to φ
```

5.2 Deadlock Detection

Algorithm 1 formally presents the functionality of the components described in Sect. 5.1. The input to the algorithm is an instrumented MPI program. Execution of the program at line 1 generates a program trace: ρ, a path condition: π, and the potential send-receive match set generated from ρ: $\mathbb{M}^+$. The `while` loop at line 2 executes the program repeatedly until either all of the possible control flow paths are explored or a deadlock is reported. In every iteration of the loop, we encode the trace ρ into a set of SMT rules and check their satisfiability. If we get a model from the SMT solver, then we report the deadlock as the output and exit as shown in lines 3–7. Otherwise, we search for a different control flow path by calling the procedure `SearchDifferentPath` at line 9. If that procedure is unable to find any other feasible path, we report an absence of the deadlock at line 11 and exit. Otherwise, we repeat the entire process.

We describe in Algorithm 2 the procedure `SearchDifferentPath`, to change the path condition and to generate constraints ϕ. The input to the algorithm is π, ρ, $\mathbb{M}^+$. Incrementally, we start negating the expressions in π from last to first in the loop starting at line 2. In line 3, we remove the expressions in the path condition from $(i{+}1)^{th}$ position until the end and invert the i^{th} last conditional in π to get the altered conditional expression c at line 5. This c is clubbed with the already present constraints in a maintained constraint stack to get ϕ at line 7.

We check if it is possible to drive the execution through the altered path in lines 8–11. For this, we issue a query to the solver with constraints formed from the rules given in Sect. 4 (ϕ'_{en}) and the constraints accumulated in the constraint stack (ϕ).

In order to compute ϕ'_{en}, we require (i) a subset of $\mathbb{M}^+$ which we denote by $\mathbb{M}^+_{clip}$, and (ii) a subsequence of the execution trace which is denoted by ρ_{clip}.

Algorithm 2. Searching different control flow path.

Data: ρ, π, $\mathbb{M}^+$
Result: ϕ
1 size = length(π)-1
2 **forall** $i \in size : 0$ **do**
3 remove($\pi[i+1] : \pi[size]$)
4 $\pi' = \text{negate}(\pi[i])$
5 c = $\pi'[i]$
6 add c in constraintStack
7 $\phi = \bigcup_{j=1}^{j=top} constraintStack[j]$
8 $\phi'_{en} = \text{encode}(\mathbb{M}^+_{clip}, \sigma_{clip})$ - ϕ_{safety}
9 res = solver($\phi'_{en} \cup \phi$)
10 **if** $res == SAT$ **then**
11 return ϕ
12 return $\emptyset$

Both $\mathbb{M}^+_{clip}$ and ρ_{clip}, are formed over the set of ordered MPI calls until the point of the conditional block whose predicate is negated in $\pi'(c)$. For brevity, we have only shown the details for chronological backtracking without any optimizations. An optimization strategy is presented in Sect. 6.1.

5.3 Correctness and Termination

The encoding presented in [10] is shown to be sound and complete. The encoding in Algorithm 1 is similar to the encoding from [10], hence the proof of soundness and completeness is also similar. We omit the proof here due to space consideration, but it can be found at Hermes's site.[1] The procedure `SearchDifferentPath` (shown in Algorithm 2) is sound and complete since we assume the traces are of finite length and the number of distinct control-flow paths in a program is bounded. Thus, by composition, Hermes is sound and complete for a single input.

Algorithm 2 returns $\emptyset$ and terminates at line 12 when it finishes its exhaustive search corresponding to the complete tree of possible control-flow execution paths. Note that the tree of possible control-flow paths is of finite height and width. Algorithm 1's termination is either contingent on the termination of Algorithm 2 or when a deadlock is found at line 6. We conclude that the analysis performed by Hermes terminates.

6 Implementation

We have implemented the proposed technique in a prototype tool called Hermes. We used Clang [5] to instrument the program, ISP [30] as the execution engine or

[1] https://github.com/DhritiKhanna/Hermes.

the scheduler, and Z3 [6] as the constraint solver. In the following subsections, we describe the optimizations that we have implemented to limit infeasible control flow paths. These optimizations are conservative and do not affect the soundness and completeness of the approach.

6.1 Non-chronological Backtracking

P_i	P_j
$R_{i,1}(\text{from} *, x)$	$R_{j,1}(\text{from} *, y)$
$if(x == 10)$	$if(y == 5)$
$R_{i,2}(\text{from } P_k)$	$S_{j,2}(\text{to } P_i, 10)$

Performing chronological backtracking to alter the path condition π may result in generating queries that cannot be satisfied by the solver. Consider the call $R_{i,1}$ in the adjoining example matches with $S_{j,2}$. π will contain the constraint `assume`$(x == 10)$ which, at some point during backtracking, will be inverted. However, inverting this condition alone will generate no new feasible control-flow path unless $R_{i,1}$ matches with a `send` other than $S_{j,2}$. Thus, inverting the constraint `assume`$(x == 10)$ will require the inversion of `assume`$(y == 5)$. These two conditions can be a part of a chain of dependencies that form an *unsat-core* of the unsatisfied formula. Hence, instead of chronological backtracking, we find the culpable conflict and backjump directly to the root of this dependency chain and negate the expression of the root node. The static analysis component of the program instrumentation block identifies the conditional expressions which introduce these dependencies.

In a tree of control flow paths, let there be a dependency chain of size d on path p. After verifying p, the number of SAT queries required (in the worst case) to find another feasible control-flow path in chronological backtracking is d. However, with non-chronological backtracking, only one SAT query should suffice.

6.2 Terminated Interleavings

Discovering a new control-flow path in Algorithm 2 requires a change of received data of the wildcard receive corresponding to the negated assume expression. In order to ascertain whether such an execution exists SMT solver is invoked with the modified path condition along with the other constraints. The constraints of the modified path condition depend on the over-approximate match-set $\mathbb{M}^+$, but the program actually runs with ISP's scheduler which makes the matches based on the ample-set. Since $|\textit{ample-set}| \leq |\mathbb{M}^+|$, there may be cases when the SMT solver returns a model from line 10 in Algorithm 2, but an actual run satisfying ϕ may not be possible. We handle these scenarios in our implementation by terminating such runs and restoring the state of the previous correct run. We provide an example of a terminated interleaving at our tool's site.[2]

[2] https://github.com/DhritiKhanna/Hermes.

7 Evaluation

The purpose of evaluating Hermes is to assess its efficiency and effectiveness in verifying message passing programs. We set this evaluation in the context of C/C++ MPI programs (see footnote 2) and compare Hermes against the state-of-the-art verification tools. To guide the evaluation, we ask the following research questions: [RQ1] How well does the proposed approach fair against state-of-the-art techniques for single-path and multi-path programs? [RQ2] Is the proposed approach effective in discovering deadlocks exposed under finite buffer mode?

Artifacts and Runtime Environment: We used the FEVS test-suite [27] and benchmarks from prior research papers [2,13,29,36]. The multi-path benchmarks include Matrix Multiply, Integrate, Workers, and Monte Carlo for *pi* calculation. A majority of the benchmarks are based on the client-server architecture. The experiments were performed on a 64 bit, Intel quad-core machine with 1.7 GHz processors, 4 GB of RAM, and Ubuntu version 14.04. We used ISP version 3.1 and Z3 version 4.4.2. All timings reported are in seconds and are averaged over 20 runs. `TO` signifies the time-out after 30 min. Note that the number of executions of Hermes also include the runs which were terminated.

[RQ1] We compared Hermes against the state-of-the-art tools - Mopper (a trace verifier), Aislinn and ISP (dynamic verifiers), and CIVL (a stateful symbolic model checker). We summarize the results in Table 1. On most single-path

Table 1. Performance comparison for single-path MPI programs.

B'mark	#P	B	D	ISP			Mopper		Aislinn		CIVL		Hermes	
				Detect	Runs	Time	Detect	Time	Detect	Time	Detect	Time	Detect	Time
DTG	5	0	✔	Yes	3	2.135	Yes	0.007	Yes	0.830	Yes	8.72	Yes	0.0365
	5	∞			3	2.257		0.043		0.815				0.077
	8	0	✔	Yes	3	2.220	Yes	0.011	Yes	1.135	Yes	8.78	Yes	0.040
	8	∞			3	2.307		0.044		1.139				0.080
Gauss Elim	4	0			1	0.529		0.210		8.936		TO		0.300
	8	0			1	0.371		0.295		14.322		TO		0.423
	16	0			1	2.041		0.408		TO		TO		0.659
	32	0			1	5.457		0.856		TO		TO		1.163
Heat	4	0	✔	Yes	7	8.572	Yes	0.365		⋆		⋆	Yes	0.389
	8	0	✔	-	>1K	TO	Yes	0.593		⋆		⋆	Yes	0.660
	16	0	✔	-	>1K	TO	Yes	0.927		⋆		⋆	Yes	1.063
	32	0	✔	-	>1K	TO	Yes	1.709		⋆		⋆	Yes	2.036
2D Diffusion	4	0	✔	Yes	1	0.008		NI			Yes	8.523		NI
	4	∞			90	123.733		0.388		0.908				0.451
	8	0	✔	Yes	1	0.05		NI			Yes	12.461		NI
	8	∞			>1.1K	TO		TO		16.020				TO
Floyd (5)	4	0			1	0.005		0.020		0.640		TO		0.078
	8	0			>1.6K	TO		0.116		1.391		TO		0.218
	16	0			>1.6K	TO		0.128		2.998		TO		0.540
	32	0			>1.6K	TO		3.836		6.424		TO		2.829

B'mark: Benchmark; #P: Number of Processes, B: Buffering Mode; D: Deadlock Exists
⋆: Benchmark not supported; NI: Not Invoked

Table 2. Performance comparison for multi-path MPI programs.

B'mark	#P	B	D	ISP		Aislinn	CIVL	Hermes	
				Runs	Time	Time	Time	Runs	Time
Monte (0.15)	4	0		6	6.025	0.971	⋆	3	2.326
	5	0		24	28.346	1.668	⋆	4	3.472
	6	0		120	151.598	5.028	⋆	5	4.819
	8	0		>1.2K	TO	10.173	⋆	7	7.434
MatrixMul (2×2 matrices)	4	∞		36	39.669	0.866	17.93	1	7.252
	5	∞		144	163.277	1.101	25.307	1	9.993
	6	∞		720	837.633	1.334	48.068	1	12.925
	8	∞		∼1.5K	TO	2.206	258.86	1	19.670
Integrate	4	0		27	32.755	0.858	910.36$^{\diamond}$	1	0.206
	5	0		256	332.362	3.030	156.82$^{\diamond}$	1	0.302
	6	0		3125	TO	27.839	157.63$^{\diamond}$	1	0.497
	8	0		>1.5K	TO	TO	173.27$^{\diamond}$	1	0.852
Workers (8 jobs; size of job = 2)	4	0		18	20.975	1.549	37.76	1	0.360
	5	0		24	28.433	1.368	72.333	1	0.384
	6	0		120	151.525	2.286	1027.31	1	0.510
	8	0		∼1.3K	TO	9.113	TO	1	1.021

B'mark: Benchmark; #P: Number of Processes, B: Buffer Mode; D: Deadlock Exists
⋆: Benchmark not supported; $\diamond$: Result = null (probably an internal CIVL error: the theorem prover has not said anything)

benchmarks the performance of Hermes is comparable to Mopper and considerably better than the other state-of-the-art explicit-state model checkers without compromising on error-reporting. Benchmark `2D Diffusion` exhibits a complex communication pattern and a high degree of nondeterminism which leads to a huge M^+. Hence, symbolic analysis tools do not perform well for such benchmarks. We use `--potential-deadlock` option of CIVL which verifies the program irrespective of the buffering mode.

Evaluation with multi-path programs required us to compare Hermes with all tools except Mopper, since Mopper is constrained to work with only single-path programs. The basis for comparing against ISP is the number of times a program is executed while the basis for comparing with other tools is the time taken to complete the verification. The results of our comparison are summarized in Table 2. On all benchmarks ISP times out for as few as 8 processes. Aislinn on most benchmarks (barring `MatrixMul`) either takes longer execution time in comparison with Hermes or times out. The results indicate that when the number of processes increases, the growth in execution time is relatively reasonable in Hermes in comparison with ISP and Aislinn. The scalability of Hermes regarding number of processes comes from the fact that it prunes out the redundant paths and explores only the feasible ones. CIVL is a powerful and a heavy stateful model checker that can backtrack as soon as it witnesses a visited state. An advantage of CIVL over the other tools is that it can verify

programs on complex correctness properties over a wide range of inputs. However, CIVL was consistently slower than Hermes on all benchmarks barring `Heat`. The `Heat` benchmark contained MPI calls that are not supported by CIVL yet.

[RQ2] MPI standard allows flexibility in the ways `send` calls are buffered. Aislinn buffers `send` calls if the size of the sent message is not greater than a parameter provided by the user. On the other hand, we follow the approach taken by Siegel et al. [26] and model the buffer as a limited resource. In other words, `send` calls use the buffer if it is available. Due to these differences, we cannot compare Hermes with Aislinn in the context of κ-buffer mode.

To demonstrate the importance of κ-buffer mode, we ran Hermes on the benchmarks used in [2,31]. Hermes detected the deadlocks when a buffer of size one was provided. The deadlock in the program disappears under zero-buffer and infinite-buffer modes.

8 Related Work

Deadlock detection in message passing programs is an active research domain with a rich body of literature. There are numerous popular debuggers or program correctness checkers which provide features such as monitoring the program run [16,18,32]. However, they fall short to verify the space of communication nondeterminism.

Predictive trace analysis for multi-threaded C programs is another popular area of work. The central idea in these techniques is to encode the thread interleavings of a program execution [34,35]. These techniques rely on the computation of a symbolic causal relation in the form of a partial order. The work in [34] motivated the predictive trace analysis work for MPI, MCAPI, and CSP/CCS programs [7,9,14,15,22]. The encoding presented by Forejt et al. [9] is similar to the encoding for a static bounded model checking approach to analyse shared memory programs [1] but is restricted to *single-path* programs.

Hermes's contribution on selective program instrumentation is motivated by the work in [12], which identified, using taint analysis, the relevant input sources and shared accesses that influence control-flow decisions. The use of taint analysis to extract input sources and shared accesses is an important step for covering relevant program behaviors.

Marques et al. developed a type-based strategy to statically verify MPI programs [17,20]. They verify the annotated MPI program against the protocol specifications capturing the behavior of the program using session types. Unlike model checking, their approach is not influenced by the number of processes and problem size and is insensitive to scalability issues. But they consider only deterministic and loop-free programs.

Concolic Testing (Dynamic Symbolic Execution) combines concrete execution and symbolic execution [4,24] to overcome the limitation of SAT/SMT solvers when faced with nonlinear constraints. Sherlock [8] is a deadlock detector for Java concurrent programs which combines concolic execution with constraint

solving. While we use SMT encoding to scan through all permissible schedules of a path, Sherlock instead permutes the instructions of one schedule to get another schedule. A fair comparison of Hermes with concolic execution techniques cannot be performed since Hermes does not consider every conditional statement to be included in the path condition.

CIVL [19] is an intermediate language to capture concurrency semantics of a set of concurrency dialects such as OpenMP, Pthreads, CUDA, and MPI. The back-end verifier can statically check properties such as functional correctness, deadlocks, and adherence to the rules of the MPI standard. CIVL creates a model of the program using symbolic execution and then uses model checking. Hermes creates a model of a single path of the program and uses symbolic encoding to verify that path of the program. It also uses data-aware analysis to prune irrelevant control-flow branches of the program. CIVL does not handle non-blocking MPI operations.

Vakkalanka et al. proposed POE_{MSE} algorithm [31] to dynamically verify MPI programs under a varying buffer capacity of MPI nodes. To this effect they employ an enumerative strategy to find a minimal set of culprit sends which, if buffered, can cause deadlocks. Siegel had proposed a model checking approach where the availability of buffers is encoded in the states of model itself [26].

Aislinn [2] is an explicit-state model checker which verifies MPI programs with arbitrary-sized system buffers and uses POR to prune the redundant state space. Aislinn models the buffer as an unlimited entity and the send calls use the buffers when their message size is bigger than a user provided value.

9 Conclusion and Future Work

We combined constraint solving with dynamic verification to discover communication deadlocks in *multi-path* MPI programs. For C/C++ MPI programs, the technique is concretized in a tool Hermes. It formulates an SMT query from the program's trace by conforming to the MPI runtime semantics and the non-deterministic buffer modes provided by MPI implementations. By capturing the flow of data values through communication calls, Hermes restricts the dynamic scheduler to explore a significantly fewer number of traces as compared to the explorations performed by a dynamic verifier. We tested our proposed technique on FEVS test suite and other benchmarks used in past research and found that our technique compares favorably with the state-of-the-art dynamic verification tools.

In the future, we plan to focus on ensuring the correctness of MPI programs with collective operations. Currently, we have implemented our tool with the assumption that the data which is received in a wildcard `receive` call and used in a conditional statement is only an integer variable or tag. This is a limitation of the implementation, which we plan to address in future work. However, a more subtle limitation that we impose is that the received data is not modified between the point from where it is received to the point where it is used (in the conditional statement). The constraint was motivated by analysing a large

number of benchmarks. We, in our experience, did not find that the received data underwent a transformation before it was decoded in a control statement. Note, however, that this limitation can be relaxed by allowing assignment statement in the trace language that we defined in Sect. 3. Extending the technique on these lines will possibly allow us to analyse a larger and more complex set of MPI programs.

Acknowledgements. This work is partly supported by the Tata Consultancy Services grant and Infosys Centre for Artificial Intelligence at IIIT Delhi. The authors thank the anonymous reviewers for their valuable feedback.

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_9
2. Böhm, S., Meca, O., Jančar, P.: State-space reduction of non-deterministically synchronizing systems applicable to deadlock detection in MPI. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 102–118. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_7
3. Bucur, S., Ureche, V., Zamfir, C., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: Proceedings of the Sixth Conference on Computer Systems. EuroSys 2011, pp. 183–198. ACM (2011)
4. Cadar, C., Dunbar, D., Engler, D.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. OSDI 2008, pp. 209–224. USENIX Association (2008)
5. Clang: A C language family frontend for LLVM. http://clang.llvm.org/
6. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
7. Elwakil, Mohamed, Yang, Zijiang, Wang, Liqiang: CRI: symbolic debugger for MCAPI applications. In: Bouajjani, Ahmed, Chin, Wei-Ngan (eds.) ATVA 2010. LNCS, vol. 6252, pp. 353–358. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15643-4_27
8. Eslamimehr, M., Palsberg, J.: Sherlock: scalable deadlock detection for concurrent programs. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2014, pp. 353–365. ACM (2014)
9. Forejt, V., Joshi, S., Kroening, D., Narayanaswamy, G., Sharma, S.: Precise predictive analysis for discovering communication deadlocks in MPI programs. ACM Trans. Program. Lang. Syst. **39**(4), 15:1–15:27 (2017)
10. Forejt, V., Kroening, D., Narayanaswamy, G., Sharma, S.: Precise predictive analysis for discovering communication deadlocks in MPI programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 263–278. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_19
11. Fu, X., Chen, Z., Yu, H., Huang, C., Dong, W., Wang, J.: Symbolic execution of MPI programs. In: Proceedings of the 37th International Conference on Software Engineering, vol. 2. ICSE 2015, pp. 809–810. IEEE Press (2015)

12. Ganai, M., Lee, D., Gupta, A.: DTAM: dynamic taint analysis of multi-threaded programs for relevancy. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. FSE 2012, pp. 46:1–46:11 (2012)
13. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-passing Interface, 2nd edn. MIT Press, Cambridge (1999)
14. Huang, Y., Mercer, E.: Detecting MPI zero buffer incompatibility by SMT encoding. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 219–233. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_16
15. Huang, Y., Mercer, E., McCarthy, J.: Proving MCAPI executions are correct using SMT. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, pp. 26–36. IEEE (2013)
16. Krammer, B., Bidmon, K., Müller, M.S., Resch, M.M.: MARMOT: an MPI analysis and checking tool. In: PARCO. Advances in Parallel Computing. Elsevier (2003)
17. López, H.A., Marques, E.R.B., Martins, F., Ng, N., Santos, C., Vasconcelos, V.T., Yoshida, N.: Protocol-based verification of message-passing parallel programs. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2015, pp. 280–298. ACM (2015)
18. Luecke, G.R., Zou, Y., Coyle, J., Hoekstra, J., Kraeva, M.: Deadlock detection in MPI programs. Concurrency Comput. Pract. Experience **14**(11), 911–932 (2002)
19. Luo, Z., Zheng, M., Siegel, S.F.: Verification of MPI programs using CIVL. In: Proceedings of the 24th European MPI Users' Group Meeting. EuroMPI 2017, pp. 6:1–6:11. ACM (2017)
20. Marques, E.R.B., Martins, F., Vasconcelos, V.T., Ng, N., Martins, N.: Towards deductive verification of MPI programs against session types. In: Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software. PLACES 2013, pp. 103–113 (2013)
21. MPI: A message-passing interface standard version 3.1. http://mpi-forum.org/docs/mpi-3.1/
22. Narayanaswamy, G.: When truth is efficient: analysing concurrency. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. ISSTA 2015, pp. 141–152. ACM (2015)
23. Sato, K., Ahn, D.H., Laguna, I., Lee, G.L., Schulz, M., Chambreau, C.M.: Noise injection techniques to expose subtle and unintended message races. In: Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP 2017, pp. 89–101 (2017)
24. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ESEC/FSE 2013, pp. 263–272 (2005)
25. Sharma, S.V., Gopalakrishnan, G., Kirby, R.M.: A survey of MPI related debuggers and tools. Technical Report UUCS-07-015, University of Utah, School of Computing (2007). http://www.cs.utah.edu/research/techreports.shtml
26. Siegel, S.F.: Efficient verification of halting properties for MPI programs with wildcard receives. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 413–429. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_27
27. Siegel, S.F., Zirkel, T.K.: Fevs: a functional equivalence verification suite for high-performance scientific computing. Math. Comput. Sci. **5**, 427–435 (2011)

28. Siegel, S.F., Zirkel, T.K.: TASS: the toolkit for accurate scientific software. Math. Comput. Sci. **5**(4), 395–426 (2011)
29. Vakkalanka, S.: Efficient Dynamic Verification Algorithms for MPI Applications. Ph.D thesis (2010)
30. Vakkalanka, S., Gopalakrishnan, G., Kirby, R.M.: Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 66–79. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_9
31. Vakkalanka, S., Vo, A., Gopalakrishnan, G., Kirby, R.M.: Precise dynamic analysis for slack elasticity: adding buffering without adding bugs. In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds.) EuroMPI 2010. LNCS, vol. 6305, pp. 152–159. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15646-5_16
32. Vetter, J.S., de Supinski, B.R.: Dynamic software testing of MPI applications with umpire. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing. SC 2000. IEEE Computer Society (2000)
33. Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., Supinski, B.R.D., Schulz, M., Bronevetsky, G.: A scalable and distributed dynamic formal verifier for MPI programs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. SC 2010, pp. 1–10. IEEE Computer Society (2010)
34. Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 256–272. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_17
35. Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2010, pp. 328–342 (2010)
36. Xue, R., Liu, X., Wu, M., Guo, Z., Chen, W., Zheng, W., Zhang, Z., Voelker, G.: Mpiwiz: subgroup reproducible replay of mpi applications. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP 2009, pp. 251–260 (2009)