# Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs

Vojtěch Forejt, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma

Department of Computer Science, University of Oxford, UK

**Abstract.** The Message Passing Interface (MPI) is the standard API for high-performance and scientific computing. Communication deadlocks are a frequent problem in MPI programs, and this paper addresses the problem of discovering such deadlocks. We begin by showing that if an MPI program is *single-path*, the problem of discovering communication deadlocks is NP-complete. We then present a novel propositional encoding scheme which captures the existence of communication deadlocks. The encoding is based on modelling executions with partial orders, and implemented in a tool called MOPPER. The tool executes an MPI program, collects the trace, builds a formula from the trace using the propositional encoding scheme, and checks its satisfiability. Finally, we present experimental results that quantify the benefit of the approach in comparison to a dynamic analyser and demonstrate that it offers a scalable solution.

## 1 Introduction

The Message Passing Interface (MPI) [17] is the *lingua franca* of high-performance computing (HPC) and remains one of the most widely used APIs for building distributed message-passing applications. Given MPI's wide adoption in large-scale studies in science and engineering, it is important to have means to establish some formal guarantees, like deadlock-freedom, on the behaviour of MPI programs.

In this work, we present an automated method to discover *communication deadlocks* in MPI programs that use blocking and nonblocking (asynchronous) point-to-point communication calls (such as send and receive calls) and global synchronization primitives (such as barriers). A communication deadlock (referred to simply as "deadlock" in this paper), as described in [19], is "a situation in which each member process of the group is waiting for some member process to communicate with it, but no member is attempting to communicate with it".

Establishing deadlock-freedom in MPI programs is hard. This is primarily due to the presence of nondeterminism that is induced by various MPI primitives and the buffering/arbitration effects in the MPI nodes and the network. For instance, a popular choice in MPI programs to achieve better performance (as noted in [25]) is the use of receive calls with `MPI_ANY_SOURCE` argument; such calls are called "wildcard receives". A wildcard receive in a process can be matched with any sender targeting the process, thus the matching between senders and

receivers is susceptible to network delivery nondeterminism. MPI calls such as probe and wait are sources of nondeterminism as well. This prevalence—and indeed, preference—for nondeterminism renders MPI programs susceptible to the schedule-space explosion problem.

Additional complexity in analysing MPI programs is introduced when control-flow decisions are based on random data, or when the data communicated to wildcard receives is used to determine the subsequent control-flow of the program. We call the programs that do not bear this complexity *single-path* MPI programs. As many MPI programs are implemented as single-path programs, we focus on verifying deadlock-freedom in programs where nondeterminism is caused only by wildcard receives and where any control flow that could affect inter-process communication is deterministic.

The rationale for focussing on single-path programs is also found in numerous other domains. For instance, the single-path property is the basis of recent work on verifying GPU kernels [15].

Popular MPI debuggers and program verifiers such as [16, 11, 14, 10] only offer limited assistance in discovering deadlocks in programs with wildcard receives. The debuggers concern themselves exclusively with the send-receive matches that took place in the execution under observation: alternate matches that could potentially happen in the same execution are not explored, nor reasoned about.

On the more formal side, tools such as model checkers can detect bugs related to nondeterministic communication by exploring all relevant matchings/interleavings. However, such tools suffer from several known shortcomings. In some cases, the model has to be constructed manually [21], while some tools have to re-execute the entire program until the problematic matching is discovered [24, 26]. These limitations prevent such tools from analysing MPI programs that are complex, make heavy use of nondeterminism, or take long to run.

In contrast to established tools, we analyse MPI programs under two different buffering modes: (i) the zero-buffering model, wherein the nodes do not provide buffering and messages are delivered synchronously, and (ii) the infinite-buffering model, under which asynchronously sent messages are buffered without limit. These two models differ in their interpretation of the MPI Wait event. Under the zero-buffering model, each wait call associated with a nonblocking send blocks until the message is sent and copied into the address space of the destination process. Under the infinite-buffering model, each wait call for a nonblocking send returns immediately (see Section 2).

**Contribution** This paper presents two new results for single-path MPI programs. First, we demonstrate that even for this restricted class of programs, the problem of deadlock detection is NP-complete (Section 3).

Second, we present a novel MPI analyser that combines a dynamic verifier with a SAT-based analysis that leverages recent results on propositional encodings of constraints over partial orders [1].

Our tool operates as follows: the dynamic verifier records an execution trace in the form of a sequence of MPI calls. Then, we extract the per-process *matches-*

*before* partial order on those calls (defined in Section 2), specifying restrictions on the order in which the communication calls may match on an alternative trace. We then construct a sufficiently small over-approximate set of *potential matches* [20] for each send and receive call in the collected trace. Subsequently, we construct a propositional formula that allows us to determine whether there exists a valid MPI run that respects the matches-before order and yields a deadlock. In our implementation of the propositional encoding, the potentially matching calls are modelled by equality constraints over bit vectors, which facilitates Boolean constraint propagation (BCP) in the SAT solver, resulting in good solving times.

Our approach is sound and complete for the class of single-path MPI programs we consider (modulo the buffering models which we implement): that is, our tool reports neither false alarms nor misses any deadlock. Our experiments indicate significant speedup compared to the analysis time observed when using ISP [25] (In-situ Partial Order), which is a dynamic analyser that enumerates matches explicitly.

For programs that are not single-path, our approach can still be used as a per-path-oracle in a dynamic verifier or model checker that explores the relevant control-flow paths. Finally, we believe that the presented encoding for MPI programs has a wider applicability to other popular programming languages that provide message passing support, such as Erlang or Scala.

The paper is organized as follows: We begin by outlining the related work and then introduce the necessary definitions in Section 2. In Sections 3 and 4 we present the complexity results for the studied problem and present our SAT encoding. Then in Section 5 we present the evaluation of our work.

**Related Work** Deadlock detection is a central problem in the CCS community. As an instance, DELFIN$^+$ [8] is a model checker for CCS that uses the A$^*$ algorithm as a heuristic to detect errors early in the search. Process algebra systems, like CCS and CSP, appear to be a natural fit to analyse MPI programs. However, to the best of our knowledge, no research exists that addresses the problem of automatically building CSP/CCS models from MPI programs and analysing them using CSP/CCS tools. Tools such as Pilot [2] support the implementation of CSP models using MPI.

Petri nets are another popular formalism for modelling and analysing distributed systems. McMillan presented a technique to discover deadlocks in a class of Petri nets called 1-safe Petri nets (featuring finite trace prefixes) and proved the problem to be NP-complete. Nevertheless, we are not aware of any polynomial-time reduction between this problem and the problem we study.

The work in [3, 27] presents a predictive trace analysis methodology for multithreaded C/Java programs. The authors of [27] construct a propositional encoding of constraints over partial orders and pass it to a SAT solver. They utilize the source code and an execution trace to discover a causal model of the system that is more relaxed than the causal order computed in some of the prior work in that area. This allows them to reason about a wider set of thread interleav-

ings and detect races and assertion violations which other work may miss. The symbolic causal order together with a bound on the number of context switches is used to improve the scalability of the algorithm. In our work, the concept of context switch is irrelevant. The per-process matches-before relation suffices to capture all match possibilities precisely, and consequently, there are neither false positives nor false negatives. The tool presented in [1] addresses shared-variable concurrent programs, and is implemented on top of the CBMC Bounded Model Checker [4].

MCAPI (Multicore Communications API) [12] is a lightweight message passing library for heterogeneous multicore platforms. It provides support for a subset of the calls found in MPI. For instance, MCAPI does not have deterministic receives or collective operations. Thus, the class of deadlocks found in MCAPI is a subset of the class of deadlocks in MPI. Deniz et al. provide a trace analysis algorithm that detects *potential* deadlocks and violations of temporal assertions in MCAPI [5]. The discovery of potential deadlocks is based on the construction of AND Wait-for graphs and is imprecise. The work in [13, 7] discovers assertion violations in MCAPI programs. While both present an order-based encoding, the work in [7] does not exploit the potential matches relation, and thus yields a much slower encoding [13].

Huang et al. [13] present an order-based SMT encoding using the potential matches relation. The encoding is designed to reason about violations of assertions on data, and does not allow to express the existence of deadlocks. The paper furthermore shows that the problem of discovering assertion violations on a trace is NP-complete. Due to the inherent difference of the problems studied, our proof of NP-completeness is significantly more involved than the one of [13]. In particular, for a 3-CNF formula with $n$ clauses, their work uses $n$ assertions, where each assertion itself is a disjunction of propositions (corresponding to the literals in a clause of the 3-CNF formula). In our case, the satisfiability of all clauses needs to be expressed by a possibility to form a single match.

TASS [23] is a bounded model checker that uses symbolic execution to verify safety properties in MPI programs that are implemented using a strict subset of C. It is predominantly useful in establishing the equivalence of sequential and parallel versions of a numerically-insensitive scientific computing program. TASS may report false alarms and the authors indicate that the potential deadlock detection strategy does not scale when nondeterministic wildcard receives are used [23].

## 2 Preliminaries

In this section we introduce the necessary definitions and formulate the problem we study in this paper. For brevity, we refer to single-path MPI programs as MPI programs.

**MPI Programs** An MPI program is given as a collection of $N$ processes, denoted by $P_1, \ldots, P_N$. We denote the events in process $i$ by $a_{i,j}$, where $j$ denotes

the index (i.e. the position within the process) at which the event $a$ occurs. We use the terms "event" and "MPI call" interchangeably. We define the *per-process order* $\preceq_{po}$ on events as follows: $a_{i,j} \preceq_{po} b_{k,\ell}$ if and only if events $a_{i,j}$ and $b_{k,\ell}$ are from the same process (that is, $i = k$), and the index of $a$ is lower or equal to the index of $b$ (that is, $j \leq \ell$).

The list of MPI calls/events that we permit to occur in an MPI program is as follows. A nonblocking (resp. blocking) send from $P_i$ to $P_j$ indexed at program location $k \leq |P_i|$ is denoted by $nS_{i,k}(j)$ (resp. $bS_{i,k}(j)$). Similarly, a nonblocking (resp. blocking) receive call, $nR_{i,k}(j)$ (resp. $bR_{i,k}(j)$), indicates that $P_i$ receives a message from $P_j$. A wildcard receive is denoted by writing $*$ in place of $j$. We write just $S$ and $R$ when the distinction between a blocking or nonblocking call is not important. The nonblocking calls return immediately. A blocking wait call, which returns on successful completion of the associated nonblocking call, is denoted by $W_{i,k}(h_{i,j})$, where $h_{i,j}$ indicates the index of the associated nonblocking call from $P_i$. A wait call to a nonblocking receive will return only if a matching send call is present and the message is successfully received in the destination address. By contrast, a wait call to a nonblocking send will return depending on the underlying buffering model. According to the standard [17] a nonblocking send is completed as soon as the message is copied out of the sender's address space. Thus, under the zero-buffering model the wait call will return only after the sent message is successfully received by the receiver since there is no underlying communication subsystem to buffer the message. In contrast, under the infinite-buffering model the sent message is guaranteed to be buffered by the underlying subsystem. We assume, without any loss of generality, that message buffering happens immediately after the return of the nonblocking send in which case the associated wait call will return immediately.

Let $B_{i,j}$ be a barrier call at process $i$. Since barrier calls (in a process) synchronise uniquely with a per-process barrier call from each process in the system, all barrier matches are totally ordered. Thus, we use $B_{i,j}(d)$ to denote the barrier call issued by the process $i$ that will be part of the $d$-th system-wide barrier call. The process $i$ issuing the barrier blocks until all the other processes also issue the barrier $d$. When the program location is not relevant, we replace it by "$-$".

Let $\mathcal{C}$ be the set of all MPI calls in the program, and $\mathcal{C}_i$ the set of MPI calls in $P_i$, i.e., the set of MPI calls that $P_i$ may execute. A *match* is a subset of $\mathcal{C}$ containing those calls that together form a valid communication. A set containing matched send and receive operations, or a set of matched barrier operations, or a singleton set containing a wait operation are all matches.

Furthermore, we define a *matches-before* partial order $\preceq_{mo}$ which captures a partial order among communication operations in $\mathcal{C}_i$. We refer the reader to [25] for complete details on the matches-before order. This order is different for the zero-buffering and infinite-buffering model. For the zero-buffering model, it is defined to be the smallest order satisfying that for any $a, b \in \mathcal{C}$, $a \prec_{mo} b$ if $a \prec_{po} b$ and one of the following conditions is satisfied:

- $a$ is blocking;
- $a$, $b$ are nonblocking send calls to the same destination;

– $a$ is a nonblocking wildcard receive call and $b$ is a receive call sourcing from $P_k$ (for some $k$), or a wildcard receive;
– $a$ is a nonblocking call and $b$ is the associated wait call.

When $a$ is a nonblocking receive call sourcing from $P_k$ and $b$ is a nonblocking wildcard receive call and the MPI program is at a state where both the calls are issued but not matched yet, then $a \prec_{mo} b$ is *conditionally dependent* on the availability of a matching send for $a$ (as noted in [25]). Due to its schedule-dependent nature, we ignore this case in the construction of our encoding. In our experience, we have not come across a benchmark that issues a conditional matches-before edge.

In the case of the infinite-buffering model, the only change is that the last rule does not apply when $a$ is the non-blocking send; this corresponds to the fact that all nonblocking sends are immediately buffered, and so all the waits for such sends return immediately.

Since the only difference between the finite- and infinite-buffering model is the way the order $\prec_{mo}$ is defined, most of the constructions we present apply for both models. When it is necessary to make a distinction, we will point this out to the reader.

**Semantics of MPI Programs** We now define the behaviour of MPI programs. The current *state* $q = \langle I, M\rangle$ of the system is described by the set of calls $I$ that have been issued, and a set of calls $M \subseteq I$ that were issued and subsequently matched. To formally define a transition system for an MPI program, we need to reason about the calls that can be issued or matched in $q$. The first is denoted by the set $\mathit{Issuable}(q)$, which is defined as

$$\mathit{Issuable}(\langle I, M\rangle) = \{x \mid \forall y \prec_{po} x : y \in I \wedge \forall y \prec_{mo} x : \text{if } y \in \mathcal{B}, \text{ then } y \in M\}$$

where $\mathcal{B}$ is the set of all *blocking* calls from $\mathcal{C}$, i.e., it contains all waits, barriers and blocking sends and receives. We call a set $m \subseteq I \setminus M$ of calls *ready* in $q = \langle I, M\rangle$ if for every $a \in m$ and every $s \prec_{mo} a$ we have $s \in M$. We then define

$$\begin{aligned}\mathit{Matchable}(q) \;=\; & \{\{a, b\} \text{ ready in } q \mid \exists i, j \;\; a = S_{i,-}(j), b = R_{j,-}(i/*)\} \cup \\ & \{\{a\} \text{ ready in } q \mid \exists i : \;\; a = W_{i,-}(h_{i,-})\} \cup \\ & \{\{a_1, \cdots, a_N\} \text{ ready in } q \mid \exists d \, \forall i \in [1, N] : \;\; a_i = B_{i,-}(d)\}\end{aligned}$$

The semantics of an MPI program $\mathcal{P}$ is given by a finite state machine $\mathcal{S}(\mathcal{P}) = \langle \mathcal{Q}, q_0, \mathcal{A}, \delta\rangle$ where

– $\mathcal{Q} \subseteq 2^{\mathcal{C}} \times 2^{\mathcal{C}}$ is the set of states where each state $q$ is a tuple $\langle I, M\rangle$ satisfying $M \subseteq I$, with $I$ being the set of calls that were so far issued by the processes in the program, and $M$ being the set of calls that were already matched.
– $q_0 = \langle \emptyset, \emptyset\rangle$ is the starting state.
– $\mathcal{A} \subseteq 2^{\mathcal{C}}$ is the set of actions.
– $\delta \subseteq \mathcal{Q} \times \mathcal{A} \to \mathcal{Q}$ is the transition function which is the union of two sets of transitions (i) *issue transitions*, denoted by $\rightarrow_i$, and (ii) *match transitions*, denoted by $\rightarrow_m$.

- $\langle I, M\rangle \xrightarrow{\alpha}_i \langle I \cup \alpha, M\rangle$, if $\alpha \subseteq \mathit{Issuable}(\langle I, M\rangle)$ and $|\alpha| = 1$.
- $\langle I, M\rangle \xrightarrow{\alpha}_m \langle I, M \cup \alpha\rangle$, if $\alpha \subseteq \mathit{Matchable}(\langle I, M\rangle)$.

We then use $q \xrightarrow{\alpha} q'$ to denote that $(q, \alpha, q') \in \delta$.

The set of *potential matches* $\mathbb{M}$ is defined by $\mathbb{M} = \bigcup_{q\in\Sigma} \mathit{Matchable}(q)$, where $\Sigma \subseteq \mathcal{Q}$ is the set of states that can be reached on some trace starting in $q_0$. A *trace* is a sequence of states and transitions, $q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} q_n$ beginning with $q_0$ such that $q_i \xrightarrow{a_i} q_{i+1}$ for every $0 \leq i < n$.

**The Deadlock Detection Problem** A state $\langle I, M\rangle$ is *deadlocking* if $M \neq \mathcal{C}$ and it is not possible to make any (issue or match) transition from $\langle I, M\rangle$. A trace is deadlocking if it ends in a deadlocking state. In this paper, we are interested in finding deadlocking traces and the problem we study is formally defined as follows.

**Definition 1.** *Given an MPI program $\mathcal{P}$, the* deadlock detection problem *asks whether there is a deadlocking trace in $\mathcal{S}(\mathcal{P})$.*

## 3 Complexity of the Problem

In this section we prove the following theorem.

**Theorem 1.** *The deadlock detection problem is NP-complete, for both the finite- and infinite-buffering model.*

The membership in NP follows easily. All traces are of polynomial size, because after every transition, new elements are added to the set of issued or matched calls, and maximal size of these sets is $|\mathcal{C}|$. Hence, we can guess a sequence of states and actions, and check that they determine a deadlocking trace. This check can be performed in polynomial time, because the partial order $\preceq_{mo}$ can be computed in polynomial time, as well as the sets $\mathit{Issuable}(q)$ and $\mathit{Matchable}(q)$, for any given state $q$.

Proving the lower bound of Theorem 1 is more demanding. We provide a reduction from 3-SAT; the reduction applies to both finite- and infinite-buffering semantics, because it only uses the calls whose semantics is the same under both models. Let $\Psi$ be a 3-CNF formula over propositional variables $x_1, \ldots, x_n$ with clauses $c_1, \ldots, c_m$. We create processes $Ppos_i$, $Pneg_i$ and $Pdec_i$ for each $1 \leq i \leq n$. As the names suggest, communication in process $Ppos_i$ (or $Pneg_i$) will correspond to positive (or negative) values of $x_i$. The process $Pdec_i$ will ensure that at most one of $Ppos_i$ and $Pneg_i$ can communicate before a certain event, making sure that a value of $x_i$ is simulated correctly.

Further, for each $1 \leq j \leq m$ we create a process $Pc_j$, and we also create three distinguished processes, $Pv$, $Pr$ and $Ps$. Hence, the total number of processes is $3 \cdot n + m + 3$.

The communication of the processes is defined in Figure 1. In the figure, the expression $\forall c_k \ni x_i : bS_{pos,-}(c_k)$ is a shorthand for several consecutive sends, one

| $Ppos_i$ | $Pneg_i$ | $Pdec_i$ | $Pc_j$ | $Pv$ | $Pr$ | $Ps$ |
|---|---|---|---|---|---|---|
| $bS_{pos_i,1}(dec_i)$ | $bS_{neg_i,1}(dec_i)$ | $bR_{dec_i,1}(*)$ | $bR_{c_j,1}(*)$ | $bS_{v,1}(r)$ | $bR_{r,1}(*)$ | $bR_{s,1}(c_1)$ |
| $\forall c_k \ni x_i :$ | $\forall c_k \ni \neg x_i :$ | $bS_{dec_i,2}(v)$ | $bS_{c_j,2}(s)$ | $bR_{v,2}(*)$ | $bR_{r,2}(s)$ | $\vdots$ |
| $bS_{pos_i,-}(c_k)$ | $bS_{neg_i,-}(c_k)$ | $bR_{dec_i,3}(*)$ | $bR_{c_j,3}(*)$ | $\vdots$ | | $bR_{s,m}(c_m)$ |
| | | | $bR_{c_j,4}(*)$ | $bR_{v,m+1}(*)$ | | $bS_{s,m+1}(r)$ |

**Fig. 1.** The MPI program $\mathcal{P}(\Psi)$. Here $i$ ranges from 1 to $n$, and $j$ ranges from 1 to $m$.

to each $Pc_k$ such that $x_i \in c_k$. The order in which the calls are made is not essential for the reduction.

To establish the lower bound for Theorem 1, we need to prove the following.

**Lemma 1.** *A 3-CNF formula $\Psi$ is satisfiable if and only if the answer to the deadlock detection problem for $\mathcal{P}(\Psi)$ is yes.*

The crucial observation for the proof of the lemma is that for a deadlock to occur, the call $bS_{s,m+1}(r)$ must be matched with $bR_{r,1}(*)$: in such a case, the calls $bR_{r,2}(s)$ and $bS_{v,1}(r)$ cannot find any match. In any other circumstance a deadlock cannot occur, in particular note that any $S_{pos_i,-}(c_k)$, and $S_{neg_i,-}(c_k)$ can find a matching receive, because there are exactly 3 sends sent to every $Pc_k$.

For $bS_{s,m+1}(r)$ and $bR_{r,1}(*)$ to form a match together, calls $bR_{s,j}(c_j)$, $1 \leq j \leq m$, must find a match before $Pv$ starts to communicate. To achieve this, having a satisfying valuation $\nu$ for $\Psi$, for every $1 \leq i \leq n$ we match $bS_{pos_i,1}(dec_i)$ or $bS_{neg_i,1}(dec_i)$ with $bR_{dec,1}(*)$, depending on whether $x_i$ is true or false under $\nu$. We then match the remaining calls of $Ppos_i$ or $Pneg_i$, and because $\nu$ is satisfying, we know that eventually the call $bS_{c_j,2}(s)$ can be issued and matched with $bR_{s,j}(c_j)$, for all $j$.

On the other hand, if there is no satisfying valuation for $\Psi$, then unless for some $i$ both the calls $bS_{pos_i,1}(dec_i)$ and $bS_{neg_i}(dec_i)$ find a match, some $bS_{c_j,2}(s)$ (and hence also $bR_{s,j}(c_j)$) remains unmatched. However, for both $bS_{pos_i,1}(dec_i)$ and $bS_{neg_i}(dec_i)$ to match, $bS_{dec_i,2}(v)$ must match some receive in $Pv$, which violates the necessary condition for the deadlock to happen, i.e. that $Pv$ does not enter into any communication.

## 4 Propositional Encoding

In this section we introduce a propositional encoding for solving the deadlock detection problem. Intuitively, a satisfying valuation for the variables in the encoding provides a set of calls matched on a trace, a set of unmatched calls that can form a match, and a set of matches together with a partial order on them, which contains enough dependencies to ensure that the per-process partial order is satisfied.

We will restrict the presentation to the problem without barriers, since barriers can be removed by preprocessing, where for barrier calls $B_{i,-}(d)$ and $B_{j,-}(d)$ and for any two calls $a$ and $b$ such that $a \prec_{mo} B_{i,-}(d)$ and $B_{i,-}(d) \prec_{mo} b$ we assume $a \prec_{mo} b$. The barrier calls can then be removed without introducing spurious models.

Our encoding contains variables $m_a$ and $r_a$ for every call $a$. Their intuitive meaning is that $a$ is matched or ready to be matched whenever $m_a$ or $r_a$ is true, respectively. Supposing we correctly identify the set of matched and issued calls on a trace, we can determine whether a deadlock has occurred. For this to happen, there must be some unmatched call, and no potential match can take place (i.e. for any potential match, some call was either used in another match, or was not issued yet). Thus, we must ensure that we determine the matched and issued calls correctly. We impose a preorder on the calls, where $a$ occurs before $b$ in the preorder if $a$ finds a match before $b$. To capture the preorder, we use the variables $t_{ab}$ to denote that $a$ matches before $b$, and $s_{ab}$ which stipulate that a call $a$ matches a receive $b$ and hence they must happen at the same time; note that this applies in the infinite buffering case as well.

Finally, we must ensure that $t_{ab}$ and $s_{ab}$ correctly impose a preorder. We use a bit vector $clk_a$ of size $\lceil \log_2 |\mathcal{C}| \rceil$ for every call $a$, denoting the "time" at which the call $a$ happens, and stipulate that $clk_a < clk_b$ (resp. $clk_a = clk_b$) if $t_{ab}$ (resp. $s_{ab}$) is true.

As part of the input, our encoding requires a set $\mathbb{M}^+ \supseteq \mathbb{M}$ containing sets of calls which are type-compatible (i.e. all $\alpha$ that can be contained in some $Matchable(q)$ if we disregard the requirement for $\alpha$ to be ready). The reason for not starting directly with $\mathbb{M}$ is that the problem of deciding whether a given set $\alpha$ is a potential match, i.e. whether $\alpha \in \mathbb{M}$, is NP-complete. This result can be obtained as a simple corollary of our construction for Lemma 1. Hence, in any practical implementation we must start with $\mathbb{M}^+$, since computing the set $\mathbb{M}$ is as hard as the deadlock detecting problem itself. We will give a reasonable candidate for $\mathbb{M}^+$ in the next section.

The formal definition of the encoding is presented in Figure 2. In the figure, $S$ and $R$ are the sets containing all send and receive calls, respectively, $Imm(a) = \{x | x \prec_{mo} a, \forall z : x \preceq_{mo} z \preceq_{mo} a \Rightarrow z \in \{x, a\}\}$ stands for the set of immediate predecessors of $a$, and $\mathbb{M}^+(a) = \bigcup\{b \mid \exists \alpha \in \mathbb{M}^+ : a, b \in \alpha\} \setminus \{a\}$ is the set of all calls with which $a$ can form a match. Further, $clk_a = clk_b$ (resp. $clk_a < clk_b$) are shorthands for the formulae that are true if and only if the bit vector for $a$ encodes the value equal to (resp. lower than) the value of the bit vector for $b$. The formula constructed contains $\mathcal{O}(|\mathcal{C}|^2)$ variables, and its size is in $\mathcal{O}(|\mathcal{C}|^3)$.

**Correctness of the Encoding** The correctness of the encoding is formally established by Lemmas 2 and 4.

**Lemma 2.** *For every deadlocking trace there is a satisfying assignment to the variables in the encoding.*

*Proof.* Given a deadlocking trace, we construct the satisfying assignment as follows. We set $m_a$ to true if and only if $a$ is matched on the trace, and $r_a$ true if

$$\text{Partial order} \qquad \bigwedge_{b\in\mathcal{C}} \bigwedge_{a\in Imm(b)} t_{ab} \tag{1}$$

$$\text{Unique match for send} \qquad \bigwedge_{(a,b)\in\mathbb{M}^+} \bigwedge_{c\in\mathbb{M}^+(a),c\neq b} \big(s_{ab} \to \neg s_{ac}\big) \tag{2}$$

$$\text{Unique match for receive} \qquad \bigwedge_{(a,b)\in\mathbb{M}^+} \bigwedge_{c\in\mathbb{M}^+(b),c\neq a} \big(s_{ab} \to \neg s_{cb}\big) \tag{3}$$

$$\text{Match correct} \qquad \bigwedge_{a\in R} \big(m_a \to \bigvee_{b\in\mathbb{M}^+(a)} s_{ba}\big) \wedge \bigwedge_{a\in S} \big(m_a \to \bigvee_{b\in\mathbb{M}^+(a)} s_{ab}\big) \tag{4}$$

$$\text{Matched only} \qquad \bigwedge_{\alpha\in\mathbb{M}^+} \big(s_\alpha \to \bigwedge_{a\in\alpha} m_a\big) \tag{5}$$

$$\text{No match possible} \qquad \bigwedge_{\alpha\in\mathbb{M}^+} \big(\bigvee_{a\in\alpha} (m_a \vee \neg r_a)\big) \tag{6}$$

$$\text{All ancestors matched} \qquad \bigwedge_{b\in\mathcal{C}} \big(r_b \leftrightarrow \bigwedge_{a\in Imm(b)} m_a\big) \tag{7}$$

$$\text{Not all matched} \qquad \bigvee_{a\in\mathcal{C}} \neg m_a \tag{8}$$

$$\text{Match only issued} \qquad \bigwedge_{a\in\mathcal{C}} \big(m_a \to r_a\big) \tag{9}$$

$$\text{Clock equality} \qquad \bigwedge_{(a,b)\in\mathbb{M}^+\cap(S\times R)} \big(s_{ab} \to (clk_a = clk_b\big) \tag{10}$$

$$\text{Clock difference} \qquad \bigwedge_{a,b\in\mathcal{C}} \big(t_{ab} \to (clk_a < clk_b)\big) \tag{11}$$

**Fig. 2.** The SAT encoding for the deadlock detection. Here, empty conjunctions are true and empty disjunctions are false.

and only if it is matched or if for every $b \prec_{mo} a$, $m_b$ is true. This makes sure the conditions (6)–(9) are satisfied.

We assign $s_{ab}$ to true if and only if $\{a, b\}$ occurs as a match on the trace. This ensures satisfaction of conditions of (2)–(5). Further, let $\alpha_1\alpha_2\ldots$ be the sequence of actions under which match transitions are taken on the trace. We stipulate $t_{ab}$ if $a \in \alpha_i$ and $b \in \alpha_j$ for $i < j$. We also set $clk_a = i$ for every $a \in \alpha_i$ and every $i$. This ensures satisfaction of the remaining conditions. □

The following lemma follows easily from conditions (2) and (3).

**Lemma 3.** *In every satisfying assignment to the variables in the encoding we have that for every $a$, if $s_{ab}$ and $s_{ab'}$ are true, then $b = b'$, and also if $s_{ba}$ and $s_{b'a}$ are true, then $b = b'$.*

**Lemma 4.** *For every satisfying assignment to the variables in the encoding there is a deadlocking trace.*

*Proof.* Given a satisfying assignment, we construct the trace as follows. Let $A$ be the set of all sends and waits such that $a \in A$ if and only if $m_a$ is true, and let $a_1 \dots a_K$ be an ordered sequence of elements in $A$ such that for any $a_i$ and $a_j$, if $clk_{a_i} < clk_{a_j}$, then $i < j$. We further define a sequence $\theta = \alpha_1 \dots \alpha_K$, where every $\alpha_i$ contains $a_i$, and if $a_i$ is a send, then $\alpha_i$ also contains the unique receive $b_i$ such that $s_{a_i b_i}$ is true. Such $b_i$ always exists, and is unique by Lemma 3. By (10) the sequence $\theta$ satisfies that whenever $a \in \alpha_i$ and $b \in \alpha_j$ and $clk_a < clk_b$, then $i < j$. Moreover, for any $c$ we have that the proposition $m_c$ is true if and only if $c$ occurs in some $\alpha_i$; this follows by the construction of $A$ and by (4) and (5).

We define a trace from the sequence $\theta$ by stipulating that it visits the states

$$q_i = \langle I_i, M_i \rangle = \langle\ \{y \mid \exists x \succeq_{po} y : x \in \bigcup_{1 \le \ell \le i} \alpha_\ell\}\ ,\ \bigcup_{1 \le \ell \le i} \alpha_\ell\ \rangle$$

for $0 \le i \le K$, where the part of the trace from $q_i$ to $q_{i+1}$ is defined to be

$$q_i \xrightarrow{\{b_{i,1}\}}_i \langle I_i \cup \{b_{i,1}\}, M_i\rangle \xrightarrow{\{b_{i,2}\}}_i \dots \xrightarrow{\{b_{i,n}\}}_i \langle I_i \cup \{b_{i,1}, \dots b_{i,n_i}\}, M_i\rangle \xrightarrow{\alpha_{i+1}}_m q_{i+1}$$

for $\{b_{i,1}, \dots, b_{i,n_i}\} = \{y \mid \exists x \succeq_{po} y : x \in \alpha_{i+1}\} \setminus \{y \mid \exists x \succeq_{po} y : x \in \bigcup_{1 \le \ell \le i} \alpha_\ell\}$, and where if $b_{i,j} \prec_{po} b_{i,\ell}$, then $j < \ell$.

We now argue that the sequence above is indeed a valid trace in $\mathcal{S}(\mathcal{P})$. Firstly, $q_0 = \langle \emptyset, \emptyset \rangle$. Let $i$ be largest number such that the sequence from $q_0$ up to $q_i$ is a valid trace. Let $j$ be largest number such that the extension of this trace from $q_i$ up to $\langle I, M\rangle = \langle I_i \cup \{b_{i,1}, \dots b_{i,j}\}, M_i\rangle$ is a valid trace. We analyse the possible values of $j$, showing that each leads to a contradiction.

- Suppose $0 \le j < n_i$. First, note that $b_{i,j+1} \notin I \cup M$, because $b_{i,j+1}$ does not occur in $\{y \mid \exists x \succeq_{po} y : x \in \bigcup_{1 \le \ell \le i} \alpha_\ell\}$. We need to show that $b_{i,j+1} \in \mathit{Issuable}(\langle I, M\rangle)$.
  If $a \prec_{po} b_{i,j+1}$, then by the definition of the sequence $b_{i,1}, \dots b_{i,n_i}$ the element $a$ has been issued already. Further, if $a \prec_{mo} b_{i,j+1}$, then by (1) we have that $t_{ab_{i,j+1}}$ is true, and so $clk_a < clk_{b_{i,j+1}}$. By the conditions (7) and (9) we have that $m_a$ is true, and so $a$ must occur in some $\alpha_\ell$. We have argued that if $clk_a < clk_{b_{i,j+1}}$, then $a \in \alpha_\ell$ for $\ell \le i$, and so $a \in M$.
  Hence by definition $b_{i,j+1} \in \mathit{Issuable}(\langle I, M\rangle)$.
- Suppose $j = n_i$. We have argued above that for every element $b \in \alpha_{i+1}$ and every $a \prec_{mo} b$ we have $a \in M$. Also, $b \in I \setminus M$, and so $\alpha_{i+1}$ is ready in $\langle I, M\rangle$. Finally, we defined $\alpha_{i+1}$ to be either a singleton set containing a wait, or a set containing compatible send and receive, hence, $\alpha_{i+1} \in \mathit{Matchable}(\langle I, M\rangle)$.

Finally, we argue that the trace is deadlocking. By (8) and the construction of the sequence $\theta$ we have that $M_K \subsetneq \mathcal{C}$. We show that from $q_K = \langle I_K, M_K\rangle$ it is not possible to make a match transition, even after possibly making a number of issue transitions. This proves that there is a deadlocking trace. Suppose that it is possible to make a match transition, and let us fix a suffix $q_K \xrightarrow{\{b_1\}}_i \hat{q}_1 \xrightarrow{\{b_2\}}_i \hat{q}_2 \dots \xrightarrow{\{b_n\}}_i \hat{q}_n \xrightarrow{\alpha}_m \bar{q}$. Note that because $\hat{q}_n = \langle I_K \cup \{b_1, \dots, b_n\}, M_K\rangle$, for

the transition under $\alpha$ to exist it must be the case that for any $b \in \alpha$ and any $a \prec_{mo} b$ we have $a \in M_K$. But then by (7) all $b \in \alpha$ satisfy that $r_b$ is true. Then by (6) we get that there is $b \in \alpha$ for which $m_b$ is true, and so $b \in M_K$, which contradicts that the match transition under $\alpha$ can be taken in $\hat{q}_n$. □

## 5 Implementation and Experimental Results

The MOPPER deadlock detection tool takes as input an MPI program and outputs the result of the deadlock analysis. MOPPER first compiles and executes the input program using ISP (In-Situ Partial order) [24]. The ISP tool outputs a canonical trace of the input program, along with the *matches-before* partial order $\preceq_{mo}$. MOPPER then computes the $\mathbb{M}^+$ overapproximation as follows. The intial $\mathbb{M}^+$ is obtained by taking the union of all sets whose elements are type-compatible (i.e., singleton sets containing a wait call, sets of barrier calls containing individual calls from each process, and sets containing $S_{i,-}(j)$ together with $R_{j,-}(i/*)$), and then refining the set by removing the sets which violate some basic rules implied by $\preceq_{mo}$. Formally, the $\mathbb{M}^+$ we use is the largest set satisfying

$$\begin{aligned}\mathbb{M}^+ = \{\{a,b\} \mid\ & a = S_{i,-}(j), b = R_{j,-}(i/*),\\ & \forall a' \prec_{mo} a\ \exists\, b' \not\succ_{mo} b : \{a',b'\} \in \mathbb{M}^+,\\ & \forall b' \prec_{mo} b\ \exists\, a' \not\succ_{mo} a : \{a',b'\} \in \mathbb{M}^+\}\\ \cup\ & \{\{a\} \mid a = W_{i,l}(h_j)\}\\ \cup\ & \{\{a_1,\cdots,a_N\} \mid\ \forall i \in [1,n], a_i = B_{i,-}\}\ .\end{aligned}$$

The partial order $\preceq_{mo}$ and the over-approximation of $\mathbb{M}$ (*matchsetapp*) are then used by MOPPER to construct the prepositional formula as explained in the previous section. This prepositional formula is then passed to the SAT solver, and when the computation finishes, the result is presented to the user, possibly with a deadlocking trace.

Our experiments were performed on a 64-bit, quad-core, 3 GHz Xeon machine with 16 GB of memory, running Linux version 3.5. MOPPER uses ISP version 0.2.0 [24] to generate the trace and MiniSat version 2.2.0 [6] to solve the propositional formula. All our benchmarks are C MPI programs and the sources of the benchmarks and the MOPPER tool can be found at `http://www.cprover.org/mpi`.

We compare the performance of MOPPER with the dynamic verifier that is integrated in ISP. We instruct ISP to explore the matches exhaustively with a time-out of two hours. We use a time-out of 30 minutes for MOPPER. We also compare the bounded model checker TASS [23] with MOPPER; TASS is configured to time-out after 30 minutes.

The results of the experiments are tabulated in Table 1. The table presents the results under different buffering assumptions only for those benchmarks where buffering had an impact. Note that the MOPPER running time does not

include the time it takes to generate the trace with ISP; the MOPPER numbers do include the constraint generation and SAT solving time. Comparison of the execution time of both tools is meaningful only when the benchmarks are single-path. For the benchmarks where this is not the case MOPPER only explores a subset of the scenarios that ISP explores.

To estimate the degree of match nondeterminism in the collected program trace, we introduce a new metric $\rho = |\mathbb{M}^+|/mcount$, where $mcount$ is the number of send and receive matches in the trace. Benchmarks with a high value of $\rho$ have a large set of potential matches. Since the metric relies on potential matches, $\rho$ could be greater than 1 even for a completely deterministic benchmark.

**Benchmarks** The benchmarks Diffusion2d and Integrate_mw are a part of the FEVS benchmark suite [22]; these benchmarks exhibit high degree of nondeterminism, as indicated by their value of $\rho$. The Diffusion2d benchmark solves the two-dimensional diffusion equation. In Diffusion2d, each node communicates its local computation results with its neighbouring nodes which are laid out in a grid fashion. The Integrate_mw benchmark estimates the integral of a sine or a cosine function in a given range. The integration tasks are dynamically allotted to worker nodes by a master node. Due to this dynamic load balancing by the master node, Integrate_mw is not a single-path MPI program. In order to make Integrate_mw a single path benchmark, we modified the source to implement static load balancing. In this single-path variant of the Integrate_mw benchmark, the schedule space grows as $n!/n$ where $n$ is the number of processes.

The benchmarks Floyd and Gauss Elimination are from [28] and both are single-path MPI programs. Floyd implements the all-pairs shortest path algorithm and employs a pipelined communication pattern where each process communicates with the process immediately next in a ranking.

Monte is a benchmark from [9] that implements the Monte Carlo method to compute the value of *pi*. It is implemented in a classic master-worker communication pattern with dynamic load balancing. We have run this benchmark without modification and thus cannot claim the results to be complete.

We have a set of 10 synthetic benchmarks with various deadlocking patterns that are not discovered by the MPI runtime even after repeated runs. Among them, we include only the DTG (dependence transition group [24]) benchmark. The benchmark has seemingly unrelated pair of matches at the start state that do not commute. Thus, selecting one match-pair over the other leads to a deadlock. A run of ISP with optimization fails to discover the deadlock, however, when the optimization is turned off, ISP discovers the deadlock after 3 runs.

A pattern similar to DTG exists in the Heat-errors benchmark [18]. This benchmark implements the solution of the heat conduction equation. ISP discovers the deadlock (when this benchmark is run on eight processes) in just over two hours after exploring 5021 interleavings. The same deadlock is detected in under a second by MOPPER.

For comparison of MOPPER with TASS we used the 64-bit Linux binary of TASS version 1.1. Since TASS accepts only a limited subset of C, our exper-

**Table 1.** Experimental Results

| | | | | | | MOPPER | | | ISP | |
|---|---|---|---|---|---|---|---|---|---|---|
| B'mark | #Calls | Procs | $\rho$ | B | Dl[a] | #Vars | #Clauses | time | #Runs | time |
| [s]DTG[†] | 16 | 5 | 1.33 | 0 | ✔ | 266 | 739 | 0.01 | 3 | 0.08 |
| | | | | $\infty$ | | 483 | 1389 | 0.01 | 3 | 0.08 |
| [s]Gauss Elim | 92 | 8 | 1.86 | 0 | | 2.7K | 8.4K | 0.01 | 1 | 0.27 |
| | 188 | 16 | 1.93 | 0 | | 6.3K | 19.9K | 0.02 | 1 | 0.36 |
| | 380 | 32 | 1.97 | 0 | | 14.3K | 45.2K | 0.04 | 1 | 0.58 |
| [s]Heat | 152 | 8 | 1.8 | 0 | ✔ | 8.9K | 27.2K | 0.03 | >2.5K | TO |
| | 312 | 16 | 1.84 | 0 | ✔ | 20K | 60.9K | 0.06 | >2.5K | TO |
| | 632 | 32 | 1.86 | 0 | ✔ | 44.9K | 136.9K | 0.18 | >2.5K | TO |
| [s]Floyd | 120 | 8 | 7 | $\infty$ | | 14K | 51K | 1.4 | >20K | TO |
| | 256 | 16 | 7.53 | 0 | | 35.09K | 128K | 16.37 | >20K | TO |
| | | | | $\infty$ | | 34.6K | 127.2K | 32.5 | >20K | TO |
| | 528 | 32 | 7.8 | 0 | | 79.34K | 292K | 161.26 | >20K | TO |
| | | | | $\infty$ | | 78.28K | 288.5K | 122.39 | >20K | TO |
| [s]Diffusion2d | 52 | 4 | 2.82 | $\infty$ | | 2.9K | 9.6K | 0.01 | 90 | 29.1 |
| | 108 | 8 | 5.7 | $\infty$ | | 13.6K | 49.9K | TO | >10.5K | TO |
| [s]Pingping | 2370 | 4 | 2.0 | $\otimes$ | | 336K | 1.16M | 1.15 | >1k | TO |
| [m]Integrate | 28 | 4 | 3.0 | $\otimes$ | | 1.9K | 6K | 0.01 | 6 | 0.04 |
| | 36 | 8 | 4.0 | $\otimes$ | | 1.8K | 6.2K | 0.05 | 5040 | 216.72 |
| | 46 | 10 | 5.0 | $\otimes$ | | 3.2K | 11.6K | 20.4 | >13K | TO |
| | 76 | 16 | 7.0 | $\otimes$ | | 10.7K | 40.5K | TO | >13K | TO |
| Monte | 35 | 4 | 2.42 | $\infty$ | | 1K | 3K | 0.00 | 6 | 0.76 |
| | 75 | 8 | 4.6 | $\infty$ | | 3.6K | 12.3K | 0.43 | 5040 | 1928.28 |
| | 155 | 16 | 8.7 | $\infty$ | | 15.6K | 58K | TO | >5.4K | TO |

[a] Deadlock present [†] ISP misses the deadlock under optimized run
[s] single-path [$\otimes$] Buffering model irrelevant [m] Modified to single-path

imentation with TASS is restricted to only few benchmarks, namely Integrate and the synthetic benchmarks. With these few benchmarks, the scalability of TASS cannot be evaluated in an objective manner. We observed, however, that the potential deadlock detection of TASS on our benchmarks was particularly slow: the analysis of Integrate with TASS timed out when run for ten processes. On the synthetic benchmarks, TASS was one order of magnitude slower than MOPPER.

**Discussion** Our results show that the search for deadlocks using SAT and our partial-order encoding is highly efficient compared to an existing, state-of-the-art dynamic verifier. However, there is room for improvement in several directions. Our encoding times out on three benchmarks. To address the time-out problem, we can restrict our analysis to calls that match within a window enclosed by barriers. Additionally, we can further refine $\mathbb{M}^+$ by discovering additional constraints under which matches really take place. Furthermore, our benchmarks (and MPI programs in general) contain a high degree of communication symme-

try (groups of processes that follow the same control flow). We conjecture that by exploiting this symmetry we can successfully perform a sound reduction of the trace (i.e., without missing deadlocks). We also aim to support a larger class of MPI programs by (i) extending the encoding for nondeterministic calls such as waitsome and waitany, and (ii) covering data-dependent MPI programs.

## 6 Conclusion

We have investigated the problem of deadlock detection for a class of MPI programs with no control-flow nondeterminism. We have shown that finding a deadlock in such programs is NP-complete. We have further devised a SAT-based encoding that can be successfully used to find deadlocks in real-world programs. We have implemented the encoding as part of a new tool, called MOPPER, and have provided an evaluation on benchmarks of various sizes. Our experiments show that the tool outperforms the state-of-the-art model checker in the area.

There are several directions in which our tool can be improved, such as handling larger subset of the MPI language, or reducing the size of the traces. We plan to investigate these in our future work.

**Acknowledgements** The authors would like to thank Martin Brain, Alex Horn and Saurabh Joshi for helpful discussions on the topic.

The authors were in part supported by EPSRC H017585/1 and J012564/1, the EU FP7 STREP PINCETTE and ERC 280053. G. Narayanaswamy is a Commonwealth Scholar, funded by the UK government. V. Forejt is also affiliated with Masaryk University, Czech Republic.

## References

1. J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.
2. J. D. Carter, W. B. Gardner, and G. Grewal. The Pilot library for novice MPI programmers. In *PPoPP*, pages 351–352. ACM, 2010.
3. F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: A predictive runtime analysis tool for Java. In *ICSE*, pages 221–230. ACM, 2008.
4. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
5. E. Deniz, A. Sen, and J. Holt. Verification and coverage of message passing multicore applications. *ACM Trans. Design Autom. Electr. Syst.*, 17(3):23, 2012.
6. N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
7. M. Elwakil and Z. Yang. Debugging support tool for MCAPI applications. In *PDATAD*, pages 20–25. ACM, 2010.
8. S. Gradara, A. Santone, and M. L. Villani. DELFIN$^+$: An efficient deadlock detection tool for CCS processes. *J. Comput. Syst. Sci.*, 72(8):1397–1412, Dec. 2006.

9. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. MIT Press, 1999.
10. W. Haque. Concurrent deadlock detection in parallel programs. *Int. J. Comput. Appl.*, 28(1):19–25, Jan. 2006.
11. T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. MPI runtime error detection with MUST: advances in deadlock detection. In *SC*, page 30, 2012.
12. J. Holt, A. Agarwal, S. Brehmer, M. Domeika, P. Griffin, and F. Schirrmeister. Software standards for the multicore era. *IEEE Micro*, 29(3):40–51, May 2009.
13. Y. Huang, E. Mercer, and J. McCarthy. Proving MCAPI executions are correct using SMT. In *ASE*, pages 26–36. IEEE, 2013.
14. B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. MARMOT: An MPI analysis and checking tool. In *PARCO*, volume 13 of *Advances in Parallel Computing*, pages 493–500. Elsevier, 2003.
15. A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner. Verifying GPU kernels by test amplification. In *PLDI*, pages 383–394. ACM, 2012.
16. G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14(11):911–932, 2002.
17. Message Passing Interface. `http://www.mpi-forum.org/docs/mpi-2.2`.
18. M. S. Mueller, G. Gopalakrishnan, B. R. de Supinski, D. Lecomber, and T. Hilbrich. Dealing with MPI bugs at scale: Best practices, automatic detection, debugging, and formal verification. `http://sc11.supercomputing.org/schedule/event_detail.php?evid=tut131`.
19. N. Natarajan. A distributed algorithm for detecting communication deadlocks. In *FSTTCS*, volume 181 of *LNCS*, pages 119–135. Springer, 1984.
20. S. Sharma, G. Gopalakrishnan, E. Mercer, and J. Holt. MCC: A runtime verification tool for MCAPI user applications. In *FMCAD*, pages 41–44, 2009.
21. S. F. Siegel. Model checking nonblocking MPI programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 4349 of *LNCS*, pages 44–58. Springer, 2007.
22. S. F. Siegel and T. K. Zirkel. FEVS: A functional equivalence verification suite for high-performance scientific computing. *Mathematics in Computer Science*, 5(4):427–435, 2011.
23. S. F. Siegel and T. K. Zirkel. The Toolkit for Accurate Scientific Software. Technical Report UDEL-CIS-2011/01, Department of Computer and Information Sciences, University of Delaware, 2011.
24. S. Vakkalanka. *Efficient dynamic verification algorithms for MPI applications*. PhD thesis, University of Utah, Salt Lake City, UT, USA, 2010. AAI3413092.
25. S. S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *CAV*, LNCS, pages 66–79. Springer, 2008.
26. A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. de Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *SC*, pages 1–10. IEEE, 2010.
27. C. Wang, S. Kundu, M. K. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, volume 5850 of *LNCS*, pages 256–272. Springer, 2009.
28. R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker. MPIWiz: subgroup reproducible replay of MPI applications. In *PPoPP*, pages 251–260. ACM, 2009.