Exploiting Epochs and Symmetries in Analysing MPI Programs

Rishabh Ranjan
IIT Delhi
India
cs1180416@iitd.ac.in

Ishita Agrawal
IIT Delhi
India
mt1190695@iitd.ac.in

Subodh Sharma
IIT Delhi
India
svs@cse.iitd.ac.in

ABSTRACT

Communication nondeterminism is one of the main reasons for the intractability of verification of message passing concurrency. In many practical message passing programs, the non-deterministic communication structure is symmetric and decomposed into epochs to obtain efficiency. Thus, symmetries and epoch structure can be exploited to reduce verification complexity. In this paper, we present a dynamic-symbolic runtime verification technique for single-path MPI programs, which (i) exploits communication symmetries by way of specifying symmetry breaking predicates (SBP) and (ii) performs compositional verification based on epochs. On the one hand, SBPs prevent the symbolic decision procedure from exploring isomorphic parts of the search space, and on the other hand, epochs restrict the size of a program needed to be analyzed at a point in time. We show that our analysis is sound and complete for single-path MPI programs on a given input. Using our prototype tool SIMIAN, we further demonstrate that our approach leads to (i) a significant reduction in verification times and (ii) scaling up to larger benchmark sizes compared to prior trace verifiers.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; **Distributed programming languages**.

KEYWORDS

symmetry breaking, MPI, verification, deadlocks

ACM Reference Format:
Rishabh Ranjan, Ishita Agrawal, and Subodh Sharma. 2022. Exploiting Epochs and Symmetries in Analysing MPI Programs. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556954

1 INTRODUCTION

The message passing (MP) paradigm is the *lingua franca* for developing large communicating distributed programs such as those in high-performance scientific computing applications developed using Message Passing Interface (MPI). A common feature in such applications is the presence of *communication nondeterminism*, which is primarily used to obtain efficiency by masking network latencies. Under the said nondeterminism, a process can post (possibly asynchronous) receive calls that can potentially match any of the messages sent. Evidently, communication nondeterminism is an important driver of intractability verification for reachability properties in message-passing programs [13, 19].

The focus of this work is programs developed using MPI. As it emerges, many MPI programs (or just programs) usually have symmetric communication patterns; for instance, in a mesh network topology, a process's communication with its top, left, right and bottom neighbor is similar to every other process's communication behavior. Also, it is not uncommon to see that communication in a program takes place in phases (called *epochs*) wherein every communication call invoked in a phase is matched within that phase.

This paper uses symmetry and epoch-styled communication patterns to design an analysis technique for detecting *communication deadlocks* in *single-path* [14] MPI programs. Communication deadlocks (or just deadlocks) are common and hard-to-detect error in MPI programs. Each member process is in an indefinite cyclic-wait for some member process to communicate with it. The presence of nondeterminism in message reception can lead to a situation where the witnessed program execution may complete successfully. However, an alternate execution with a different order of message reception may lead to a deadlock. Single-path programs are where the order of instructions from a process is deterministic. In other words, the program's control-flow is not affected by the communicated data. Indeed, the single-path property limits the degree of program nondeterminism but is still significant and has formed the basis of prior studies [14, 18, 20, 30]. The problem of detecting deadlocks is NP-complete for single-path programs [13].

In prior single-path analyzers, the set of feasible runs of a program on a concrete input is symbolically encoded into a propositional SAT or SMT formula. Inevitably, SAT/SMT solving for larger instances of such encodings ends up consuming an unreasonably long time. However, in many instances, the solver is stuck on examining symmetric runs of a program. Pruning them can potentially speed up the solving time. This observation serves as the basis for our work.

Contributions. (C1) As the first contribution of this work, we present a theoretical framework and an encoding where symmetries are exploited by way of specifying *symmetry breaking predicates* (SBPs) [6]. The additional constraints, in the form of SBPs, eliminate symmetric solutions (which are isomorphic parts of the search space). As a result, SBPs can bring about exponential savings in the constraint solver's solving times.

(C2) Detecting symmetries over the entire program can be expensive. Additionally, symmetry may not always manifest globally

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556954

through the program but may exist locally. As our second contribution, we present an analysis for detecting and decomposing a program into epochs. Epoch decompostion facilitates *compositional analysis* of a program by analysing each epoch in isolation. It also serves an additional advantage as its combinations with C1 may yield more reductions than those obtained by applying C1 or C2 in isolation.

(C3) We show that symmetry reductions and epoch decompositions are sound and complete, *i.e.*, report a deadlock if and only if there is a real deadlock in single-path MPI programs.

(C4) We implement the above two techniques as a dynamic-symbolic runtime analysis tool called SIMIAN[1]. We demonstrate empirically that SIMIAN is significantly more efficient than existing dynamic-symbolic trace verifiers such as MOPPER [14] and MOPPER-opt [13].

2 PRELIMINARIES

The following text presents the necessary definitions relating to MPI process modeling and process symmetries.

Process Model. A single-path MPI program (or just MPI program), $\mathcal{P}$, is assumed to be a collection of N processes denoted by $P_1, \ldots, P_N$. Each process executes a sequence of MPI actions. We denote an action from process P_i at index j (the position within the process) as $a_{i,j}$. The permitted MPI actions considered in this work are nonblocking *sends* and *receives*, blocking *waits*, and blocking *barriers*.

A *nonblocking send* from P_i to P_j at index $k \in \mathbb{N}$ is denoted as $S_{i,k}(j)$. The send action asynchronously transfers the data from the source process to the destination process. A nonblocking receive from P_j to P_i at index k is denoted as $R_{i,k}(j)$. The special *wildcard* receive (which can accept a message from any source process) is denoted by replacing the destination process identifier by $*$. A blocking wait action is denoted as $W_{i,k}(h_{i,j})$ where $h_{i,j}$ denotes handle of the associated nonblocking action from P_i. A wait action returns upon the successful completion of the associated nonblocking action. For instance, wait action on a nonblocking receive will return only after the sent message is copied fully into the receiver's address space. It is worth noting that *synchronous* send (resp. receive) action can be modeled by issuing a wait action immediately after the send (resp. receive) action. A barrier from a process is a system-wide synchronizing event which blocks until all processes have reached the *same* barrier. Since barriers in a program are totally ordered, the k^{th} barrier action from P_i at index j is denoted as $B_{i,j}(k)$. The collection of k^{th} barriers from each process in the system forms a barrier group.

Let C be the set of all MPI actions in a program. We present the semantics of an MPI program, which builds on the notions of *matching* and ordering of actions. Informally, matching actions are those that together form a valid communication. Instances of valid communication are: (i) a send and its matching receive, (ii) a nonblocking request and its associated wait, or (iii) a barrier group. Formally, we define a total order on actions from a process (Definition 2.1) and the partial order in which actions from a process will match (Definition 2.2, borrowed from [14]).

[1]https://github.com/rishabh-ranjan/simian

Definition 2.1 (Process order). A strict total order $<$ is defined for all actions $a_{i,k}$ and $b_{j,l}$ as follows: $a_{i,k} < b_{j,l}$ iff $i = j \wedge k < l$.

Definition 2.2 (Matches-before order). A strict partial order $\prec \subseteq C \times C$ is the smallest matches-before order s.t. for all $c_1, c_2 \in C, c_1 \prec c_2$ iff $c_1 < c_2$ and one of the following is true:

- c_1 is blocking
- c_1 and c_2 are nonblocking sends (resp. receives excluding wildcards) to (from) the same destination (resp. source).
- c_1 is a nonblocking call and c_2 is an associated wait call
- c_1 is a wildcard receive and c_2 is either a wildcard receive or a receive from P_k for some k

Under system buffering, a nonblocking send, c_1, can complete before the associated wait, c_2, but may match afterwards. For such send calls, we replace their matching guarantee with completion guarantee (rationale for the third condition in Definition 2.2). The matches-before (MB) ordering comprehensively incorporates the *non-overtaking* FIFO ordering guarantee over communicated actions as defined in the MPI standard. The only case where MB ordering is not always enforceable is when c_1 is a receive from P_k (for some k) and c_2 is a wildcard receive. The *conditional* ordering exists only when both c_1 and c_2 can match the same send from P_k [14]. We leave the ordering between such receives unspecified.

MPI Semantics. Following [13], the semantics of an MPI program is given by a finite state machine: $\langle Q, q_0, \mathcal{A}, \rightarrow \rangle$ where:

- $Q \subseteq 2^C \times 2^C$ is the set of states where each state $q = \langle I, M \rangle$. The set I is the set of actions that were so far issued by the processes in the program, and $M \subseteq I$ is the set of actions that were matched so far.
- $q_0 = \langle \emptyset, \emptyset \rangle$ is the starting state.
- $\mathcal{A} \subseteq 2^C$ is the set of actions.
- $\rightarrow \subseteq Q \times \mathcal{A} \rightarrow Q$ is the transition function of the form $q \xrightarrow{\alpha} q'$, which comprises of mainly of two type of transitions:(i) *issue* and *match* transitions. The concrete semantics and the resulting state change for issue and match transitions are as follows.

$$\frac{q = \langle I, M \rangle \qquad \begin{array}{c} \alpha \subseteq \{c | \forall c_1 \in C, (c_1 < c \rightarrow c_1 \in I) \\ \wedge (c_1 \prec c \wedge \mathit{isBlocking}(c_1) \rightarrow c_1 \in M)\} \end{array}}{q' = \langle I \cup \alpha, M \rangle, q \xrightarrow{\alpha} q'} \textbf{ Issue}$$

The *issue* transition issues a new action $c \in \alpha$ only if all preceding nonblocking actions from the c's process are already issued and all preceding blocking actions (known through the predicate *isBlocking*) from the c's process are already matched.

$$\frac{q = \langle I, M \rangle \qquad \alpha \subseteq \mathit{Matchable}(q)}{q' = \langle I, M \cup \alpha \rangle, q \xrightarrow{\alpha} q'} \textbf{ Match}$$

The match transition matches the set of actions that are matchable at q. The following definitions describe which set of actions are indeed matchable at a state.

Definition 2.3 (Ready Actions). A set actions $R(q) \subseteq I \setminus M$ is ready at $q = \langle I, M \rangle$ if for every $c \in R(q)$ and for every $a \prec c$, we have that $a \in M$.

Definition 2.4 (Matchable Actions). A set of action-sets is matchable at a state q, denoted by $\mathit{Matchable}(q)$, is defined as follows:

P_0	P_1	P_2
$S_{0,0}(1)$	$R_{1,0}(*)$	$S_{2,0}(1)$
$B_{0,1}(0)$	$R_{1,1}(*)$	$B_{2,1}(0)$
$S_{0,2}(1)$	$W_{1,2}(h_{1,1})$	
	$B_{1,3}(0)$	
	$R_{1,4}(*)$	

Figure 1: Example Message Passing Program

- $\forall i, j \in \mathbb{N}$, if $c_1 = S_{i,-}(j), c_2 = R_{j,-}(i/*)$ and $c_1, c_2 \in R(q)$ then $\{c_1, c_2\} \in \mathit{Matchable}(q)$ (Note: $-$ at a position represents that the position is not relevant to the discussion)
- if $c_1 = W_{i,-}(-) \in R(q)$, then $\{c_1\} \in \mathit{Matchable}(q)$
- if $\forall i \in \{1, \ldots, N\}, \exists d.B_{i,-}(d) \in R(q)$, then $\{B_{i,-}(d)\} \in \mathit{Matchable}(q)$

Let $\mathbb{M} = \bigcup_{q \in Q} \mathit{Matchable}(q)$ be the set of all *potential* matches. It was shown in [14] that precise computation of $\mathbb{M}$ is NP-complete for single-path programs.

Definition 2.5 (Trace). Given a run of $\mathcal{P}$ from its starting state $q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} q_n$, the trace τ corresponding to this run is the sequence $\alpha_0\alpha_1 \ldots \alpha_{n-1}$. For brevity, we may omit the labels for the Issue transitions.

A trace is *extensible* if from the last state of the run corresponding to the trace and after zero or more issue transition moves there exists a match transition move. A trace which contains all calls in $\mathcal{P}$ is called a *complete* trace (notice that for single-path programs, every full run of the program can be viewed as a complete trace). We denote the set of all traces of $\mathcal{P}$ as T.

Definition 2.6 (Deadlocking trace). $\tau \in T$ is deadlocking if τ is not complete and is not extensible.

The set of all deadlocking traces is denoted as T_d. Finally, we describe the problem we tackle in the paper: *deadlock detection* for a program $\mathcal{P}$ is a decision problem to determine whether $T \cap T_d = \emptyset$.

3 EXAMPLE

Consider an example program shown in Figure 1. The program consists of three processes P_0, P_1 and P_2. For brevity, we refer to the actions without their arguments. The synchronization from the matching barriers $B_{0,1}, B_{1,3}$, and $B_{2,1}$ splits the communication into two phases. Before the synchronization, P_0 and P_2 send a message each to P_1. The second receive from P_1 is blocking – modeled with $W_{1,2}$ immediately following $R_{1,1}$. The MPI runtime mandates that both the receives match before $B_{1,3}$ can even be issued. After the synchronization, P_0 sends a message to P_1.

The *process order* for each process is given by: $S_{0,0} < B_{0,1} < S_{0,2}$, $R_{1,0} < R_{1,1} < W_{1,2} < B_{1,3} < R_{1,4}$, and $S_{2,0} < B_{2,1}$. The *matches-before* order is given by $B_{0,1} \prec S_{0,2}$ and $R_{1,0} < R_{1,1} \prec W_{1,2} \prec B_{1,3} \prec R_{1,4}$. Noticeably, there is no $\prec$ ordering between $S_{0,0}$ and $B_{0,1}$ or between $S_{2,0}$ and $B_{2,1}$. This is due to an absence of corresponding waits intervening between the send and barrier actions.

At state $q = \langle I, \emptyset \rangle$ with $I = \{S_{0,0}, R_{1,0}, S_{2,0}\}$, the ready action set is $R(q) = I$. Thus, $\mathit{Matchable}(q) = \{\{S_{0,0}, R_{1,0}\}, \{S_{2,0}, R_{1,0}\}\} = \mathbb{M}$.

There are two *complete traces* up to reordering of *Issue* transitions – $\tau_1 = \langle \{S_{0,0}, R_{1,0}\}, \{S_{2,0}, R_{1,1}\}, \{W_{1,2}\}, \{B_{0,1}, B_{1,3}, B_{2,1}\}, \{S_{0,2}, R_{1,4}\} \rangle$,

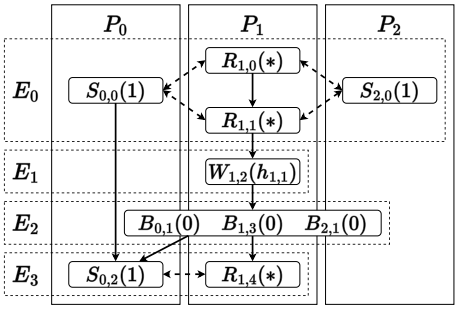


Figure 2: Program Graph for Running Example

$\tau_2 = \langle \{S_{0,0}, R_{1,1}\}, \{S_{2,0}, R_{1,0}\}, \{W_{1,2}\}, \{B_{0,1}, B_{1,3}, B_{2,1}\}, \{S_{0,2}, R_{1,4}\} \rangle$ – which differ only in the matching of the first two wildcard receives from P_1. Any proper prefix of τ_1 or τ_2 is an *incomplete* but *extensible* trace.

Deadlocking variant. If we replace $R_{1,1}(*)$ with $R_{1,1}(0)$, then $\tau' = \langle \{S_{0,0}, R_{1,0}\} \rangle$ is incomplete and inextensible, hence a *deadlocking trace*. In our approach, the SAT encoding would detect this deadlock by encoding the conditions of Definition 2.6. The SAT solver will be able to detect that at state $q = \langle I, M \rangle$ where M captures τ' and $I \setminus M = \{S_{2,0}, R_{1,1}\}$, the set $R(q)$ is empty.

Revisiting the original non-deadlocking example program, we recall our objective to verify the program for the presence of deadlocks by pruning away symmetric executions. It is worthwhile to note that the program as a whole is clearly not symmetric. However, if we only consider the communication before the synchronization, then P_0 and P_1 play symmetric roles. This highlights the importance of epoch decomposition in discovering local symmetries. Our analysis proceeds by first representing the program as a digraph (as shown in Figure 2). In this graph, the bi-direction dashed edges denote potential matches between sends and receive actions. The solid directed edges capture $\prec$ ordering among actions. As a next step, the program is decomposed into epochs. The annotated dotted rectangular boxes denote the epochs in the example program (refer to Figure 2). Each epoch is analyzed in isolation for deadlocks. Our approach constructs a quotient search space during the analysis of an epoch by pruning symmetric behaviors. Only epoch E_0, containing actions $S_{0,0}, R_{1,0}, R_{1,1}, S_{2,0}$, which shows the possibility of producing symmetric behaviors. The SAT solver is provided with constraints that stops the solver from exploring the match combinations $\{S_{0,0}, R_{1,1}\}, \{S_{2,0}, R_{1,0}\}$ and only explore $\{S_{0,0}, R_{1,0}\}, \{S_{2,0}, R_{1,1}\}$.

4 EPOCH DECOMPOSITION

Evidently, extracting and exploiting global symmetries in a program is not always feasible as none may exist. Keeping this in mind, we proceed by slicing a program into *epochs* such that each communication action in an epoch finds a match within the same epoch. We begin by representing the program as a digraph. The epoch decomposition is specified on this program graph.

Definition 4.1 (Program Graph). A program graph is a graph $G = (C, E)$ where E is a collection of edges s.t. $\forall c_1, c_2 \in C$, $c_1 \prec c_2$

iff $(c_1, c_2) \in E$ and $\{c_1, c_2\} \in \mathbb{M}$ iff $(c_1, c_2), (c_2, c_1) \in E$. Additionally all sets of matching barriers are collapsed to single nodes.

Figure 2 illustrates the program graph for the example in Figure 1.

Definition 4.2 (Communication Epoch). Given a program graph G. Let G_e, with set of calls C_e, be a strongly connected component (SCC) of G. This subgraph G_e is an epoch. The matches-before relation $\prec_e$ is a restriction of $\prec$ to C_e. The allowed matches $\mathbb{M}_e \subseteq \mathbb{M}$ contains sets of C_e actions.

Given a graph G, SCCs can be computed in $O(n + m)$ (n be the number of calls and $m = |\mathbb{M}|$) using standard algorithms such as Tarjan's [39] or Kosaraju-Sharir's [35]. The computation of precise epochs, as SCCs of a graph, is fundamentally dependent on the precision of match-set $\mathbb{M}$. Since the precise computation of $\mathbb{M}$ is established to be an intractable problem, we provide a series of heuristics to construct a refinement of $\mathbb{M}$. We denote this refinement by $\mathbb{M}^+$. Notice that replacing $\mathbb{M}$ with $\mathbb{M}^+$ does not affect the correctness of deadlock detection.

4.1 Matchset Refinement

The most obvious choice for $\mathbb{M}^+$ is where (i) the matching barrier actions (at the same depth) are merged into a single action set, and (ii) all send and receive actions that are type-compatible are set as potential matches.

Recursive matches-before order pruning. An observation for refinement is that for an action c_1 to match with c_2, then under $\prec$ ordering all actions ancestor to c_1 must find a match with actions that are not successors of c_2.

$$M^+_{rec} \leftarrow \{\{c_1, c_2\} \in M^+ : \forall c'_1 \in C, c'_1 \prec c_1, \\ \exists c'_2 \in C, c_2 \not\prec c'_2, \{c'_1, c'_2\} \in M^+_{rec}\}$$

The recursive matches-before pruning heuristic has been mentioned in [14]; however, we note that the technique is not implemented in the publicly available version of their tool, MOPPER.

P_0	P_1
$S_{0,0}(1)$	$R_{1,0}(*)$
$S_{0,1}(1)$	$R_{1,1}(*)$

Consider a simple example program shown above. Here, $\mathbb{M}^+ = \{\{S_{0,0}, R_{1,0}\}, \{S_{0,0}, R_{1,1}\}, \{S_{0,1}, R_{1,0}\}, \{S_{0,1}, R_{1,0}\}\}$. Since $S_{0,0} \prec S_{0,1}$, it is evident that $S_{0,1}$ can not match $R_{1,0}$ in any execution. Consider $c_1 = S_{0,1}$ and $c_2 = R_{1,0}$. Then $c'_1 = S_{0,0}$. We see that there does not exist a c'_2 s.t.$c_2 \not\prec c'_2$ that can match c'_1. Thus, $\{S_{0,1}, R_{1,0}\}$ is not included in $\mathbb{M}^+_{rec}$. Similary, one can reason about the invalid match pair $\{S_{0,0}, R_{1,1}\}$.

Barrier-led pruning. Note that in the program graph construction, matching barriers at a depth were collapsed and represented by a single barrier (see Definition 4.1). If a barrier node is responsible for ordering a potential matching pair under $\prec$, then clearly they are fit to be pruned away. The declarative definition for barrier-led pruning is as follows:

$$M^+_{anc} \leftarrow M^+_{rec} \setminus \{\{c_1, c_2\} \in M^+_{rec} : \exists \{b\} \in M^+_{rec}, c_1 \prec b \prec c_2\}$$

Consider the example in Figure 2. Notice that $R_{1,1} \prec B_{1,3}$ through the wait call, $W_{1,2}$. But $R_{1,1}$ is not yet ordered with its potential match $S_{0,2}$. Only because the barrier group $\{B_{0,1}, B_{1,3}, B_{2,1}\}$ is considered a single node, a relation $R_{1,2} \prec S_{0,2}$ is induced. Thereafter, according to the Barrier-led pruning heuristic, it is determined that $R_{1,2}$ and $S_{0,2}$ cannot be matched in any execution and can be safely removed from further consideration.

Counting heuristic. If the number of sends is equal to the number of matching receives, then match pairs involving these sends with receives outside the above-mentioned set can be pruned away. The counting heuristics have been found in prior works [20, 36]. Below we give a declarative definition of this heuristic.

Let $C_1 \subseteq C$ and $C_2 = \{c_2 \in C : \exists c_1 \in C_1 : \{c_1, c_2\} \in M^+\}$. If $|C_1| = |C_2|$, then we can prune all $\{c, c_2\}$ from M^+ for all c such that $c_1 \prec c$ for all $c_1 \in C_1$. This process can be repeated for multiple such C_1 sets.

Consider the example in Figure 2 again. Consider $C_1 = \{R_{1,0}, R_{1,1}\}$. The unique set C_2 for such a C_1 is $C_2 = \{S_{0,0}, S_{2,0}\}$. Note that $S_{0,2} \notin C_2$ since it was pruned away by the barrier-led pruning heuristic. Since $|C_1| = |C_2|$, we find a c s.t. $c_1 \prec c, \forall c_1 \in C_1$. We note that $c = R_{1,4}$. This implies that we prune $\{S_{0,0}, R_{1,4}\}$ (which could not otherwise be pruned by the other two pruning heuristics).

4.2 Caching

Verified epochs (free from deadlocks) are cached to avoid redundant verification. The program graphs representing the epochs are hashed using BLISS [23] (a tool that checks for graph isomorphism) and stored in a hash table. To ascertain a match (since hashes may not be reliable in theory), the current epoch's and the cached epoch's program graphs must be tested for graph isomorphism. This is again done using BLISS. The graph isomorphism problem is not known to be in P or in NP-complete [32]. However, in practice, it is easy to solve for most common cases. We find in our experiments that time spent in isomorphism checks with cached epochs is insignificant compared to the time required for new verification.

4.3 Correctness

We establish the correctness of epoch decomposition by the following theorem.

Theorem 4.3. *P has a deadlocking trace τ if and only if some communication epoch $e \in E$ has a deadlocking trace τ_e.*

The forward proof follows the argument that pruning applied to $\mathbb{M}^+$ is conservative. Thus, SCCs will contain a set of match edges that is over-approximate and can realize parts of any trace of P. For the backward direction, observe that since P was obtained from dynamic execution, every epoch is reachable; so a deadlocking trace in some epoch readily gives a deadlocking trace for P.

5 SYMMETRY IN MPI PROGRAMS

The use of symmetries in reducing the search space requires two steps. The first step is to discover symmetries in a program. This reduces to the graph automorphism problem, which, like graph isomorphism, is not known to be in P or NP-complete [32]. The second step is to make use of discovered symmetries by directing the search procedure to examine only a representative member of each symmetry class. In this section, we formalize the notion of

symmetry in the context of MPI programs and describe how to formulate constraints that allow a SAT solver to break symmetries.

5.1 Symmetry Detection

We describe the symmetries in $\mathcal{P}$ by computing automorphisms of G (the program graph), which are structure preserving *vertex permutations*.

Definition 5.1 (Graph Automorphism). A bijection $\pi : C \rightarrow C$ is an automorphism of G if $(c_1, c_2) \in E \iff (\pi(c_1), \pi(c_2)) \in E$ for all $c_1, c_2 \in C$.

It is known that the set of automorphisms of a graph forms a group under composition. We denote the group of all automorphisms of G by $\mathcal{G}$. We show that these automorphisms when applied to traces of $\mathcal{P}$ preserve the relation $\prec$ over the actions of $\mathcal{P}$. As a result, T_d (the set of deadlocking traces) too remains preserved.

Let $\pi \in \mathcal{G}$ be an arbitrary automorphism of G. π is naturally extended to other structures over C by replacing all occurrences of c in the structure with $\pi(c)$. In particular, for trace $\tau \in T$, $\pi(\tau)$ is naturally defined. In the proofs that follow, wherever applicable, we only prove the forward implication. The backward implication follows by considering the automorphism π^{-1} instead of π.

Lemma 5.2. *The automorphism π preserves allowed matches,* i.e., $\{c_1, c_2\} \in M \iff \{\pi(c_1), \pi(c_2)\} \in M$.

Proof. $\{c_1, c_2\} \in \mathbb{M} \implies (c_1, c_2), (c_2, c_1) \in E \implies (\pi(c_1), \pi(c_2)), (\pi(c_2), \pi(c_1)) \in E \implies \{\pi(c_1), \pi(c_2)\} \in \mathbb{M}$ (by Definition 4.1) □

Lemma 5.3. *The automorphism π preserves the matches-before order $\prec$,* i.e., $c_1 \prec c_2 \iff \pi(c_1) \prec \pi(c_2)$

Proof. Follows from the definitions of E and π. □

Theorem 5.4. *The automorphism π preserves traces,* i.e., $\tau \in T \iff \pi(\tau) \in T$.

Proof. Assuming $\tau \in T$, we show that $\pi(\tau)$ has elements only from $\mathbb{M}$ only, each call $c \in C$ occurs at most once in it, and that causality, as specified by $\prec$, is preserved. Since τ contains only elements from $\mathbb{M}$, we have that for each $\{c_1, c_2\} \in \tau$. From Lemma 5.2, we have $\{\pi(c_1), \pi(c_2)\} \in \mathbb{M}$. If $c \in C$ occurs more than once in $\pi(\tau)$, then $\pi^{-1}(c)$ occurs more than once in τ, which is a contradiction. Finally, suppose there exists c_2 in $\pi(\tau)$ and $\exists c_1 \in C, c_1 \prec c_2$ s.t. c_1 does not occur earlier in $\pi(\tau)$ than c_2. Then $\pi^{-1}(c_2)$ exists in τ with $\pi^{-1}(c_1) \prec \pi^{-1}(c_2)$ such that $\pi^{-1}(c_1)$ does not occur earlier in τ than $\pi^{-1}(c_2)$ (by Lemma 5.3). This is a contradiction since $\tau \in T$. □

Theorem 5.5. *The automorphism π preserves deadlocks,* i.e., $\tau \in T_d \iff \pi(\tau) \in T_d$.

Proof. Assume $\tau \in T_d$. Let $R \subseteq C$ be the set of calls which do not occur in τ. $\pi(R)$ is obtained from R by replacing occurrences of c with $\pi(c)$. Note that $\pi(R) \subseteq C$ is the set of calls which do not occur in $\pi(\tau)$. Since, $R \neq \phi$, it implies $\pi(R) \neq \phi$. This proves that $\pi(\tau)$ is not complete.

Let $\{c_1, c_2\}$ be an arbitrary element in $\mathbb{M}$ such that $c_1, c_2 \in \pi(R)$. Then $\{\pi^{-1}(c_1), \pi^{-1}(c_2)\} \in \mathbb{M}$ (by Lemma 5.2) and $\pi^{-1}(c_1), \pi^{-1}(c_2) \in R$. Since $\tau \in T_d$, there exists $c \in R$ such that $c \prec \pi^{-1}(c_1)$ or $c \prec \pi^{-1}(c_2)$. Then for $\pi(c) \in \pi(R)$, we have $\pi(c) \prec c_1$ or $\pi(c) \prec c_2$ (by Lemma 5.3). This proves that $\pi(\tau)$ is inextensible; thus, completes the proof for $\pi(\tau) \in T_d$. □

We now show that automorphisms, as discussed above, facilitate the pruning of the search space while preserving the correctness of deadlock detection in an MPI program.

Definition 5.6 (Trace Equivalence $\equiv$). For $\tau_1, \tau_2 \in T$, the group of automorphisms leads to an equivalence relation $\equiv \subseteq T \times T$, such that $\tau_1 \equiv \tau_2$ if there exists $\pi \in \mathcal{G}$ and $\tau_2 = \pi(\tau_1)$.

The equivalence relation $\equiv$ partitions the set of traces T into equivalence classes called *orbits*. The following theorem establishes that any reduction in the search space by way of realizing $\equiv$ and exploring only the representative traces from each orbit will not affect the correctness of deadlock detection.

Theorem 5.7. *For $\tau_1, \tau_2 \in T$, $\tau_1 \equiv \tau_2 \implies (\tau_1 \in T \iff \tau_2 \in T) \wedge (\tau_1 \in T_d \iff \tau_2 \in T_d)$.*

Proof. Follows from Theorems 5.4 and 5.5, and Definition 5.6. □

Example. We revisit the example shown in Figure 1. Let us consider epoch E_0. Consider the trace for E_0, $\tau : \langle \{S_{0,0}, R_{1,0}\}, \{S_{2,0}, R_{1,1}\} \rangle$. For brevity, we have suppressed showing the arguments of actions in the trace. Consider $\pi : E \rightarrow E$, given as:$S_{0,0} \mapsto S_{2,0}$, $S_{2,0} \mapsto S_{0,0}$, $R_{1,0} \mapsto R_{1,0}$, and $R_{1,1} \mapsto R_{1,1}$.

Under this automorphism, we obtain a trace: $\pi(\tau) : \langle \{S_{2,0}, R_{1,0}\}, \{S_{0,0}, R_{1,1}\} \rangle$. Note that $\tau \equiv \pi(\tau)$ and examination of $\pi(\tau)$ can be avoided.

Computing Automorphisms for MPI Programs. We use BLISS which is particularly efficient for sparse graphs. BLISS characterizes the automorphism by producing an irredundant set of group *generators* (elements of a group that can generate all the elements of a group). In a graph with n vertices, the automorphism group can be specified by no more than $n - 1$ generators and BLISS satisfies this bound.

BLISS also allows setting a time-out and only the generators found till then will be used to encode symmetry breaking constraints (step 2). We found this especially convenient for our formulation, although we did not find graph automorphism to be the bottleneck in our experiments – we ran BLISS without setting time limits.

5.2 Symmetry Breaking Predicates

We encode the symmetries, as discovered in the previous step through BLISS, via a collection of SBPs. Each SBP is chosen in such a way that for each $\pi \in \mathcal{G}$ they are satisfied for exactly one trace in the equivalence class induced by π.

Symmetry Constraint ϕ_s. We introduce a *lexicographic ordering* on the traces in T. We then design a predicate that is true of only the smallest trace under this ordering within each equivalence class. More formally, we introduce a constraint ϕ_s, which is satisfied by one (or few) representative traces for each orbit. Before presenting the shape of ϕ_s, we briefly revisit the SAT encoding for deadlocks proposed in [13]; the proposed SBPs augment this encoding with ϕ_s.

Definition 5.8 (Propositional encoding for symmetry-aware deadlock verification). For a trace $\tau \in T$, $\phi(\tau) = \phi_t \wedge \phi_d \wedge \phi_s$, where:

(1) ϕ_t: satisfied by all valid traces arising from feasible matchings of actions witnessed in τ under $\mathbb{M}$
(2) ϕ_s: satisfied by one (or few) representative traces for each orbit of symmetric traces
(3) ϕ_d: satisfied by deadlocking traces

The encoding contains Boolean variables m_a and r_a for each action $a \in C$ denoting whether the action is matched or is ready to be matched, respectively. Boolean variables t_{ab} and s_{ab} denote that $a \prec b$ and a matches with b, respectively. The encoding also uses a bit-vector to capture the logical time at which each action a happens. Thus, if t_{ab} is true then $clk_a < clk_b$.

Since, $\mathbb{M}$ computation is NP-complete, the encoding works with an over-approximation $\mathbb{M}^+$, which contains all type-compatible actions. For instance, $\mathbb{M}^+$ contains all those matching $S_{i,-}(j)$ and $R_{j,-}(i/*)$ which do not violate $\prec$. Note that this may still include infeasible send-receive match pairs.

Thus, ϕ_t is captured by the following constraints:

$$\bigwedge_{b \in C} \bigwedge_{a \prec b} t_{ab} \tag{1}$$

$$\bigwedge_{a,b \in C} \left(t_{ab} \rightarrow (clk_a < clk_b)\right) \tag{2}$$

Constraints (1) and (2) encode the relation $\prec$ for each $a, b \in C$.

$$\bigwedge_{\{a,b\} \in \mathbb{M}^+} \left(s_{ab} \rightarrow \bigwedge_{c \in \mathbb{M}^+(a), c \neq b} \neg s_{ac} \wedge \bigwedge_{c \in \mathbb{M}^+(b), c \neq a} \neg s_{cb}\right) \tag{3}$$

$$\bigwedge_{(a,b) \in \mathbb{M}^+} \left(s_{ab} \rightarrow (clk_a = clk_b)\right) \tag{4}$$

Constraints (3) and (4) encode a unique match for each send and receive in C along with the constraint on their occurrence time.

$$\bigwedge_{a \in C} \left(m_a \rightarrow \bigvee_{b \in \mathbb{M}^+(a)} s_{ba} \wedge \bigvee_{b \in \mathbb{M}^+(a)} s_{ab}\right) \tag{5}$$

$$\bigwedge_{\alpha \in \mathbb{M}^+} \left(s_\alpha \rightarrow \bigwedge_{a \in \alpha} m_a\right) \tag{6}$$

Constraints (5) and (6) capture that each matched send (resp. receive) should appear in some match in $\mathbb{M}^+$. Also, every constituent action a in each match α have the status of having found a match.

$$\bigwedge_{a \in C} \left(m_a \rightarrow r_a\right) \tag{7}$$

$$\bigwedge_{b \in C} \left(r_b \leftrightarrow \bigwedge_{a \prec b} m_a\right) \tag{8}$$

Constraints (7) and (8) encode that for an action a to be matched it is necessary for it to be ready first. An action b is ready if all ancestors under $\prec$ of b are already matched.

The deadlocking constraint, ϕ_d, encodes a state from where trace is inextensible and not all actions have matched:

$$\bigwedge_{\alpha \in \mathbb{M}^+} \left(\bigvee_{a \in \alpha} (m_a \vee \neg r_a)\right) \tag{9}$$

$$\bigvee_{a \in C} \neg m_a \tag{10}$$

Finally, to specify ϕ_s for a trace τ, we introduce a bit-vector of Boolean variables s_α for each $\alpha \in \tau$. We denote such a bit-vectors by $[\tau]$. In order to establish total ordering on traces, we resort to defining a lexicographic ordering constraint on the bit-vectors. This ordering is also known in SBP literature as *Lex-Leader* constraints [6]. Given bit-vectors for traces $[\tau_1] = [s_{\alpha_1}, s_{\alpha_2}, \ldots, s_{\alpha_n}]$ and $[\tau_2] = [s_{\beta_1}, s_{\beta_2}, \ldots, s_{\beta_n}]$. We define a predicate $P_i(V, W), 0 \leq i \leq n$ as:

$$V_{i-1} = W_{i-1} \rightarrow s_{\alpha_i} \leq s_{\beta_i}$$

where $V_{i-1} = [s_{\alpha_1}, \ldots, s_{\alpha_{i-1}}]$ and $W_{i-1} = [s_{\beta_1}, s_{\beta_2}, \ldots, s_{\beta_{i-1}}]$. Building on the above predicate, we define the lexicographic constraint $[\tau_1] \leq [\tau_2]$ by the following encoding:

$$\bigwedge P_i([\tau_1], [\tau_2])$$

The intuition behind the above encoding is that given an assignment for $[\tau_1]$ and $[\tau_2]$ as binary numbers, $[\tau_1] \leq [\tau_2]$ only when the respective binary numbers also hold the inequality. The above encoding is not efficient. We chose **AND-CSE**, which we found most promising with respect to performance and formula size among the nine encodings surveyed and evaluated in [48], to encode the above Lex-Leader constraints.

Moving forward, we obtain a set $B \subseteq \mathcal{G}$ of automorphisms from BLISS. For each $\pi \in B$, we apply the Lex-Leader constraint on τ and $\pi(\tau)$. The constraint obtained is:

$$\phi_s \qquad \bigwedge_{\pi \in B} [\tau] \leq [\pi(\tau)] \tag{11}$$

Theorem 5.9. *The Lex-Leader ordering constraints for $\mathcal{G}$ are sound and complete for symmetry breaking.*

Proof. We prove soundness and completeness of symmetry breaking separately: Consider an orbit Q. Let τ be the minimal trace in Q by $\leq$. Then τ satisfies $\tau \leq \pi(\tau), \pi \in \mathcal{G}$ for any $\pi \in Q$. Hence, τ is present in the constrained search space, which proves soundness.

Consider an orbit Q and any trace $\tau \in Q$ which is also in the constrained search space. Let τ' be an arbitrary trace in Q. Then there exists $\pi \in \mathcal{G}$ such that $\tau' = \pi(\tau)$. The ordering constraint $\tau \leq \pi(\tau)$ implies that $\tau \leq \tau'$. Hence, τ must be the unique minimal trace in Q. This proves completeness. □

5.3 Complexity Analysis

The size of SAT formula (excluding the ϕ_s constraints) for a trace with n actions is shown to have $O(n^3)$ and $O(n^2)$ clauses and variables, respectively [13]. With ϕ_s, the Lex-Leader constraints for all $\pi \in \mathcal{G}$ risk become exponential in number. Instead, we use the Corollary to Theorem 5.9 and choose B to be the set of generators of the automorphism group $\mathcal{G}$. It has been shown in [1] that using group generators for exploiting symmetry directly from SAT formulas is sufficient. Thus, working with at most g generators (where $g < n$), the number of automorphisms to consider in ϕ_s is in $O(n)$. **AND-CSE** introduces $O(n^2)$ many new variables resulting in a formula that is linear in the size of the trace (*i.e.*, $O(n^2)$). Combining this with the number of automorphisms, we observe that the asymptotic complexity of the encoding remains unchanged, *i.e.*, $O(n^3)$.

6 RESULTS

In this section, we present our experimental setup and results to validate the efficacy of our proposed method.

6.1 Implementation

We implement the presented technique in a tool called SIMIAN. SIMIAN can verify MPI programs written in C/C++ for deadlocks. As a pre-processing step, the source code is compiled using ISP [42]'s compiler to produce a binary suitable for ISP's execution engine. SIMIAN itself has the following 3 components:

Scheduler. The compiled binary is run on a particular set of inputs / arguments with the specified number of processes and buffering semantics (0- or ∞-) by ISP's execution engine (also called scheduler). If the dynamic run terminates successfully, the scheduler outputs the set of MPI actions C recorded during the execution, along with the process order $<$, the matches-before order $\prec$, and the initial over-approximate matchset $\mathbb{M}^+$ on C. On the other hand, if the dynamic run deadlocks, the scheduler detects this and SIMIAN terminates. This component is common to HERMES, MOPPER and MOPPER-Opt.

Analyzer. The analyzer implements the core logic for epoch decomposition and symmetry-breaking. It performs matchset refinements, constructs the program graph and identifies epochs. For each epoch to be verified, it invokes BLISS to detect symmetry, and generates the SAT formula. Further, it implements the caching mechanism described in Section 4.2. The analyzer runs statically since the output of the scheduler is sufficient for all the analysis required.

Solver. SIMIAN calls the Z3 SMT solver [7] using its C++ API to solve the generated SAT formulas. If a formula is satisfiable, a deadlocking trace is implied, and SIMIAN terminates. Otherwise, the analyzer proceeds to the next epoch, if available, performing symmetry detection and formula generation as usual.

6.2 Baselines

We compare SIMIAN with MOPPER [14], MOPPER-Opt [13], and HERMES [27]. MOPPER and MOPPER-Opt are trace verifiers with the same core codebase. MOPPER-Opt differs from MOPPER as it works on an abstraction that interprets an unbroken sequence of wildcard receive actions as a single receive action. SIMIAN is designed without these abstractions, but can be augmented with such abstractions.

HERMES is a tool that uses a modified encoding to analyse both single-path and multi-path programs. Additionally, it departs from a propositional SAT encodings (used in MOPPER and MOPPER-Opt) to a SMT formulation.

A recent tool presented in [20] also detects communication deadlocks by first predicting a set of candidate deadlocking traces, and then pruning the infeasible candidates through a SMT formulation. Unfortunately, we could not empirically evaluate our work againt this tool as it did not accompany preprocessing, compilation and running scripts for benchmarks. We did not receive a response to our follow-up request on this matter to the authors of [20]. Note, however, that the contributions of SIMIAN are orthogonal and complementary to the contributions in [20].

We do not compare against full symbolic verifiers like MPI-SV [47], as such a comparison would be inequitable. While trace verifiers assume access to a dynamic execution of the program, allowing them to work with very succinct models, symbolic verifiers must perform all the analysis statically. Further, a trace verifier only verifies the program for the given input, whereas symbolic verifiers seek to provide guarantees for large symbolic input spaces. The choice of verifier depends on the runtime vs guarantee requirements, and as such, these techniques are not direct competitors.

6.3 Benchmark Summary

Here we briefly describe the benchmarks used in our experiments.
Adder [38]. It is a benchmark that adds an array of numbers in parallel using master-worker communication pattern. It is obtained from the FEVS suite.
Floyd [46]. It implements the all-pairs shortest path Floyd-Warshall algorithm with a pipelined communication pattern where each process communicates with the adjacent processes in the ranking.
GaussElim [46]. It is a parallel implementation of the Gauss-Jordan Elimination to obtain the row-echelon form of a matrix, using pairwise communication between processes.
Heat and Heat Errors [34]. The two benchmarks implement the heat conduction equation on a 2D grid. The latter benchmark is seeded with communication bug.
Integrate [38]. It is an implementation of computing an numerical integration of an input function using the *trapezoid* rule.
Diffusion [38]. It implements an iterative algebraic solver for 2D difussion equation on a mesh of shape $N \times M$ with t time steps. It is a particularly interesting benchmarks with many epochs.
MatMul [38]. It is an implementation of matrix multiplication of $N \times L$ and $L \times M$ matrices. Communication pattern is a block-distribution over rows.

6.4 Variations

Most prior works show only the variation in benchmarks with the number of processes or the buffering mode while fixing the values for the grid/matrix shapes/sizes and a single timestep. We demonstrate the scalability of our methods to the verification of programs over different parameters, as it shows the robustness and applicability of epochs and symmetry-analysis to different variants. We consider the following variations over **Diffusion** and **MatMul**:

(1) Diffusion with fixed timestep (t=1), but varying processes (p) and grid size $N \times M$ with $N \times M = p$. For every p, we consider all factorizations $N \times M$ and report the average runtimes. We find that our techniques are applicable to different grid shapes, thus allaying the fear that symmetry might be a rare occurrence and highly sensitive to the configurations under which the program is run.
(2) Diffusion on a 2x2 grid, with timesteps varying. Here, we show that the epoch identification is strong enough to detect the redundancy in the communication patterns seen in different timesteps. While SIMIAN can use its cache of verified epochs to achieve constant solver time with timestep variations, other tools end up solving ever-larger formulas for the growing traces due to larger number of timesteps.
(3) MatMul on fixed size matrices (N=L=M=8) with varying number of processes.

Table 1: Runtimes for ∞-buffering (in s)

Name vs X	X	Deadlock	Mopper	Mopper-Opt	Hermes	Simian
Adder	8	No	0.268	**0.042**	0.212	0.061
vs	16	No	TO	**0.212**	TO	1.157
Processes	32	No	TO	**1.497**	TO	1.912
	64	No	TO	**4.116**	TO	7.257
Floyd	8	No	2.761	6.132	1.453	**0.628**
vs	16	No	283.524	399.156	2.148	**1.939**
Processes	32	No	TO	TO	5.021	**2.491**
	64	No	TO	TO	10.312	**6.081**
GaussElim	8	No	0.243	0.233	0.187	**0.186**
vs	16	No	0.628	1.655	0.258	**0.283**
Processes	32	No	4.314	4.282	1.993	**2.334**
	64	No	10.033	6.159	3.912	**3.226**
Heat	8	No	0.666	0.395	0.406	**0.325**
vs	16	No	1.581	0.845	0.636	1.506
Processes	32	No	6.543	1.623	2.005	4.255
	64	No	14.927	10.597	4.464	**3.232**
HeatErrors	8	Yes	0.523	0.395	**0.309**	0.353
vs	16	Yes	1.191	0.779	0.662	**0.565**
Processes	32	Yes	5.706	2.574	2.712	**2.191**
	64	Yes	10.392	**4.799**	6.435	5.051
Integrate	8	No	0.256	**0.038**	0.209	0.062
vs	16	No	TO	0.232	TO	**0.131**
Processes	32	No	TO	3.581	TO	**1.854**
	64	No	TO	**4.212**	TO	7.583
Diffusion	4	No	TO	3.729	239.77	**0.122**
(Timesteps=1)	8	No	TO	14.854	TO	**0.716**
vs	16	No	TO	49.069	TO	**8.566**
Processes	24	No	TO	271.247	TO	**36.308**
Diffusion	2	No	TO	1.598	TO	**0.225**
(Grid=2x2)	4	No	TO	26.347	TO	**1.023**
vs	8	No	TO	842.951	TO	7.197
Timesteps	16	No	TO	TO	TO	**67.854**
MatMul	8	No	2.719	0.083	1.815	**0.076**
(N=L=M=8)	16	No	3.572	0.274	3.032	**0.114**
vs	32	No	4.448	**0.734**	4.014	1.477
Processes	64	No	8.625	4.001	5.482	**2.766**
MatMul	8	No	2.708	0.086	1.918	**0.076**
(N=L=M=p)	16	No	TO	0.328	TO	**0.224**
vs	32	No	TO	3.728	TO	**2.822**
Processes	64	No	TO	**4.808**	TO	16.175
MatMul	4	No	0.061	**0.062**	0.071	0.065
(p=8)	6	No	0.101	**0.066**	0.112	0.079
vs	8	No	2.704	0.092	1.794	**0.076**
Size (N=L=M)	12	No	TO	9.772	TO	**4.938**

Table 2: Runtimes for 0-buffering (in s)

Name vs X	X	Deadlock	Mopper	Mopper-Opt	Hermes	Simian
Adder	8	No	0.281	**0.043**	0.285	0.061
vs	16	No	TO	**1.249**	TO	1.301
Processes	32	No	TO	1.684	TO	**0.932**
	64	No	TO	**4.112**	TO	7.065
Floyd	8	No	3.986	3.715	**0.232**	0.249
vs	16	No	7.171	22.413	**1.582**	2.108
Processes	32	No	112.793	157.188	**4.665**	5.619
	64	No	763.232	TO	**10.941**	32.261
GaussElim	8	No	0.223	0.235	**0.177**	0.185
vs	16	No	0.599	0.618	0.773	**0.481**
Processes	32	No	4.398	**1.297**	4.097	4.155
	64	No	5.748	6.085	5.612	**5.429**
Heat	8	No	0.323	0.325	0.241	**0.225**
vs	16	No	**0.726**	0.789	1.646	1.549
Processes	32	No	4.515	2.497	4.119	**1.831**
	64	No	4.139	**3.706**	5.341	9.121
HeatErrors	8	Yes	0.408	0.323	**0.221**	0.229
vs	16	Yes	1.226	1.772	0.587	**0.461**
Processes	32	Yes	4.351	2.515	4.568	**1.987**
	64	Yes	9.492	6.634	6.548	**2.932**
Integrate	8	No	0.281	**0.042**	0.251	0.079
vs	16	No	TO	**0.223**	TO	0.279
Processes	32	No	TO	**3.592**	TO	3.961
	64	No	TO	**4.237**	TO	9.119
Diffusion	4	Yes	**0.029**	0.032	0.115	0.137
(Timesteps=1)	8	Yes	0.037	**0.035**	0.132	0.125
vs	16	Yes	0.229	**0.197**	0.295	0.337
Processes	24	Yes	1.457	1.434	**0.548**	1.482
Diffusion	2	Yes	**0.028**	0.034	0.131	0.127
(Grid=2x2)	4	Yes	0.031	**0.028**	0.132	0.127
vs	8	Yes	**0.029**	0.035	0.121	0.112
Timesteps	16	Yes	**0.032**	0.038	0.124	0.139
MatMul	8	Yes	0.129	**0.069**	0.087	0.157
(N=L=M=8)	16	No	4.088	1.269	1.339	**0.242**
vs	32	No	5.097	**1.488**	3.714	3.582
Processes	64	No	7.544	4.065	**3.835**	4.209
MatMul	8	Yes	0.134	**0.073**	0.087	0.157
(N=L=M=p)	16	Yes	0.869	**0.281**	0.714	0.356
vs	32	Yes	4.649	**0.633**	9.545	1.169
Processes	64	Yes	24.643	**4.586**	98.515	9.554
MatMul	4	No	0.059	**0.046**	0.068	0.075
(p=8)	6	No	0.092	**0.048**	0.104	0.067
vs	8	Yes	0.127	**0.071**	0.098	0.157
Size (N=L=M)	12	Yes	0.189	**0.099**	0.121	0.187

(4) MatMul with processes and matrix size varying with the number of processes (N=L=M=p).
(5) MatMul with fixed number of processes, but varying matrix size (but with N=L=M).

Finally, we run all benchmarks under both ∞- and 0-buffering settings. For our purposes, the buffering mode only affects the matches-before order. In particular, under ∞-buffering the matches-before ordering between sends and their corresponding waits are relaxed. See [14] for further details.

6.5 Runtime

Tables 1 and 2 show the total times for the various tools. In each row the smallest runtime is shown in bold. We keep a timeout of 20 minutes (denoted by TO). The experiments were run on a machine with 16 "Intel(R) Xeon(R) W-1270 CPU @ 3.40GHz" cores.

∞-buffering: SIMIAN easily dominates Mopper and Hermes in all benchmarks. This clearly demonstrates the gains offered by epochs and symmetry. The comparison with MOPPER-Opt is more interesting. MOPPER-Opt proposes an alternative encoding optimized for sequences of wildcard receives, which is one source of local symmetry in MPI programs. This allows MOPPER-Opt to match the performance of SIMIAN in some cases. However, we observe that MOPPER-Opt fails to scale as well as SIMIAN in benchmarks with richer communication structures, such as Floyd and variants of Diffusion and MatMul. Even in relatively simpler benchmarks, we find that SIMIAN is highly competitive with MOPPER-Opt, often outperforming it despite the overheads due to epoch decomposition and symmetry detection.

0-buffering: In general, we observe that verification under 0-buffering is relatively faster than under ∞-buffering. Nevertheless, SIMIAN maintains its competitiveness vis-à-vis other baselines. Interestingly, many of these benchmarks deadlock under 0-buffering. Often such deadlocks are encountered during the first run of the program, which removes the need for further analysis.

6.6 Communication Structure

In Table 3, we explore the communication structure of the considered benchmarks in terms of epochs and symmetry. In the interest

Table 3: Communication Structure Summaries

Name vs X	X	Trace Size	Epochs				Symmetry
			(Size, Symmetry, Repeats) ...	Total	Unique	Repeated	Total
Adder	8	28	(14,6,1) (2,1,7)	8	2	6	7
vs	16	60	(30,14,1) (2,1,15)	16	2	14	15
Processes	32	124	(62,30,1) (2,1,31)	32	2	30	31
	64	252	(126,62,1) (2,1,63)	64	2	62	63
Floyd	8	176	(20,6,1) (147,0,1) (1,0,9)	11	3	8	6
vs	16	368	(20,6,1) (23,7,8) (147,0,1) (1,0,17)	27	4	23	13
Processes	32	752	(20,6,1) (23,7,24) (147,0,1) (1,0,33)	59	4	55	13
	64	1520	(20,6,1) (23,7,56) (147,0,1) (1,0,65)	123	4	119	13
GaussElim	8	84	(4,1,6) (2,1,2) (1,0,14)	22	3	19	2
vs	16	172	(4,1,14) (2,1,2) (1,0,22)	38	3	35	2
Processes	32	348	(4,1,30) (2,1,2) (1,0,38)	70	3	67	2
	64	700	(4,1,62) (2,1,2) (1,0,70)	134	3	131	2
Heat	8	144	(2,1,60) (1,0,24)	84	2	82	1
vs	16	296	(2,1,124) (1,0,48)	172	2	170	1
Processes	32	600	(2,1,252) (1,0,96)	348	2	346	1
	64	1208	(2,1,508) (1,0,192)	700	2	698	1
HeatErrors	8	144	(17,0,1) (2,1,31) (1,0,15)	47	3	44	1
vs	16	296	(33,0,1) (2,1,63) (1,0,31)	95	3	92	1
Processes	32	600	(65,0,1) (2,1,127) (1,0,63)	191	3	188	1
	64	1208	(129,0,1) (2,1,255) (1,0,127)	383	3	380	1
Integrate	8	28	(14,6,1) (2,1,7)	8	2	6	7
vs	16	60	(30,14,1) (2,1,15)	16	2	14	15
Processes	32	124	(62,30,1) (2,1,31)	32	2	30	31
	64	252	(126,62,1) (2,1,63)	64	2	62	63
Diffusion	4	88	(2,1,16) (18,8,2) (1,0,5)	23	3	20	9
(Timesteps=1)	8	188	(2,1,32) (42,20,2) (1,0,5)	39	3	36	21
vs	16	388	(90,44,1) (2,1,64) (90,44,1) (1,0,5)	71	4	67	89
Processes	24	588	(138,68,1) (2,1,96) (138,68,1) (1,0,5)	103	4	99	137
Diffusion	2	150	(2,1,32) (18,8,3) (1,0,8)	43	3	40	9
(Grid=2x2)	4	274	(2,1,64) (18,8,5) (1,0,14)	83	3	80	9
vs	8	522	(2,1,128) (18,8,9) (1,0,26)	163	3	160	9
Timesteps	16	1018	(2,1,256) (18,8,17) (1,0,50)	323	3	320	9
MatMul	8	54	(46,5,1) (1,0,8)	9	2	7	5
(N=L=M=8)	16	78	(2,1,7) (48,7,1) (1,0,16)	24	3	21	8
vs	32	126	(2,1,23) (48,7,1) (1,0,32)	56	3	53	8
Processes	64	222	(2,1,55) (48,7,1) (1,0,64)	120	3	117	8
MatMul	8	54	(46,5,1) (1,0,8)	9	2	7	5
(N=L=M=p)	16	110	(94,13,1) (1,0,16)	17	2	15	13
vs	32	222	(190,29,1) (1,0,32)	33	2	31	29
Processes	64	446	(382,61,1) (1,0,64)	65	2	63	61
MatMul	4	38	(2,1,3) (24,3,1) (1,0,8)	12	3	9	4
(p=8)	6	46	(2,1,1) (36,5,1) (1,0,8)	10	3	7	6
vs	8	54	(46,5,1) (1,0,8)	9	2	7	5
Size (N=L=M)	12	70	(62,5,1) (1,0,8)	9	2	7	5

Table 4: Component-wise Times (in s)

Name vs X	X	Time		
		Scheduler	Analyzer	Solver
Adder	8	**0.031**	0.014	0.013
vs	16	**1.084**	0.035	0.036
Processes	32	**1.381**	0.206	0.238
	64	2.358	1.957	**2.938**
Floyd	8	0.042	0.042	**0.543**
vs	16	**1.290**	0.094	0.553
Processes	32	**1.708**	0.239	0.541
	64	**4.764**	0.753	0.558
GaussElim	8	**0.151**	0.022	0.014
vs	16	**0.223**	0.032	0.014
Processes	32	**2.273**	0.028	0.011
	64	**3.165**	0.045	0.011
Heat	8	**0.264**	0.050	0.009
vs	16	**1.468**	0.031	0.006
Processes	32	**4.195**	0.052	0.008
	64	**3.119**	0.099	0.007
HeatErrors	8	**0.353**	0	0
vs	16	**0.565**	0	0
Processes	32	**2.408**	0.073	0.016
	64	**3.407**	0.183	0.019
Integrate	8	**0.032**	0.015	0.013
vs	16	**0.058**	0.035	0.037
Processes	32	**1.372**	0.198	0.235
	64	2.682	1.972	**2.925**
Diffusion	4	0.039	**0.061**	0.022
(Timesteps=1)	8	0.038	**0.585**	0.091
vs	16	0.135	**6.635**	1.783
Processes	24	1.021	**27.953**	7.294
Diffusion	2	0.022	**0.185**	0.021
(Grid=2x2)	4	0.027	**0.968**	0.021
vs	8	0.106	**6.952**	0.021
Timesteps	16	0.982	**65.333**	0.021
MatMul	8	**0.032**	0.018	0.026
(N=L=M=8)	16	**0.065**	0.024	0.023
vs	32	**1.389**	0.026	0.024
Processes	64	**2.618**	0.032	0.022
MatMul	8	**0.032**	0.017	0.026
(N=L=M=p)	16	0.066	0.043	**0.112**
vs	32	**1.699**	0.232	0.834
Processes	64	2.331	2.109	**11.731**
MatMul	4	**0.032**	0.017	0.015
(p=8)	6	**0.031**	0.018	0.021
vs	8	**0.032**	0.017	0.026
Size (N=L=M)	12	0.034	0.028	**4.875**

of space, we only report for the ∞-buffering mode. The observations in the 0-buffering mode are not particularly interesting or instructive over and above these. We report symmetry as the number of non-identity generators returned by BLISS. The richness of the epoch decomposition and symmetry extraction, as evident from the table, validates the effectiveness of our characterizations, while also substantiating our claim that real-world MPI programs tend to have highly repetitive and symmetric communication structures.

It is interesting to note that different benchmarks scale in different ways. In Adder, Integrate, Diffusion, and MatMul (N=L=M=p) the symmetry scales with the size of the trace. In the other benchmarks, the scaling is primarily in the number of epoch repetitions. In Heat benchmark, epochs are only of sizes one and two (mainly arising from send-receive pairs and waits/barriers, respectively). This shows that for some benchmarks the matchset pruning heuristics are strong enough to give a deterministic set of potential matches. In Diffusion (Grid=2x2) and MatMul (N=L=M=8), we observe that the same epoch appears across different runs which suggests that maintaining a global cache of epochs across runs can help.

6.7 Component-wise Times

To better understand the practical working of our tool, in this section we compare the times spent in its various parts. Table 4 shows the time taken by the various components of SIMIAN, namely scheduler (dynamic execution), analyzer (epoch separation + symmetry detection + encoding), and solver (SMT solving). Similar to the previous section, we only show the component times for the ∞-buffering mode. We show the bottleneck times in bold.

In HeatErrors, for 8 and 16 processes, the scheduler itself detected a deadlock during dynamic execution, hence time spent in analyzer and solver is 0.

In general, the time taken by the scheduler to generate a program run is the one that takes most of the time across all benchmarks (with an exception of Diffusion). The solver times are negligible. Due to large trace sizes in Diffusion, the time spent in the analyzer is

significant. However, its impact can be seen in solving time, which is in comparison much less.

6.8 Representative Plots

Figures 3 and 4 show plots to visualize the trends for some representative benchmarks, namely Integrate, Diffusion (Timesteps=1) and MatMul (N=L=M=p). This allows us to show fine-grained information with data points from all the intermediate processes.

In Integrate and MatMul, we observe that Mopper and Hermes verification times scale exponentially and quickly become intractable, while SIMIAN and Mopper-Opt verification times scale much better. In Diffusion, we see that the verification time for Simian scales smoothly whereas Mopper-Opt verification times have high variance. This is because at different number of processes, the grid shapes are different (depending on the factorizations possible for a given number of processes). This leads to different communication structures, of which some are more and some are less amenable to the Mopper-Opt optimization (sequence of consecutive wildcard receives). On the other hand, SIMIAN uses a generic formulation of epochs and symmetry and is comparatively insensitive to grid shapes.

Figure 4 contains stacked area plots to visualise the various component times across all the processes. The variance is due to the scheduler, which is inherently non-deterministic. In Diffusion the variance is smoothed out due to averaging over different factorizations of the number of processes, corresponding to different grid shapes.

7 RELATED WORK

Analyses of message passing programs have been studied extensively in the past. We provide a brief review of prior work in this landscape.

Trace-based symbolic verifiers encode an execution of a program and reason over the alternate matching of actions different from the ones witnessed in the encoded execution. This is usually accomplished by SAT/SMT encoding of the semantics as described in Section 5.2. In [19] a trace verifier for MCAPI [16] programs, and in [13, 18, 20] trace verifiers for MPI programs were proposed. Of particular interest is the work in [20], in which a set of potentially deadlocking traces is computed from the actions observed in the witnessed execution. The approach prunes infeasible deadlocking traces by encoding the MPI runtime semantics through an SMT encoding. Our contributions of using symmetries and epochs are complementary and orthogonal to the contributions made in this work. Other trace verifiers (including MOPPER, MOPPER-Opt, and HERMES) generate a single SAT formula for the entire trace and invoke the solver on it. This does not scale well with larger programs, or larger number of processes.

Sherlock [12] is a trace verifier that detects deadlocks in concurrent Java programs. Sherlock is similar to the work in [20], however, its specification of a deadlock is unsound. In recent work [24], a sound tool to *predict* deadlocks in Java programs was proposed. REVELIO [26] is a tool that combines sequential tests to generate a concurrent test driver to detect wait-notify communication deadlocks in multi-threaded Java libraries. The encoding used to generate tests is similar to encodings seen in prior trace verification works for MPI programs.

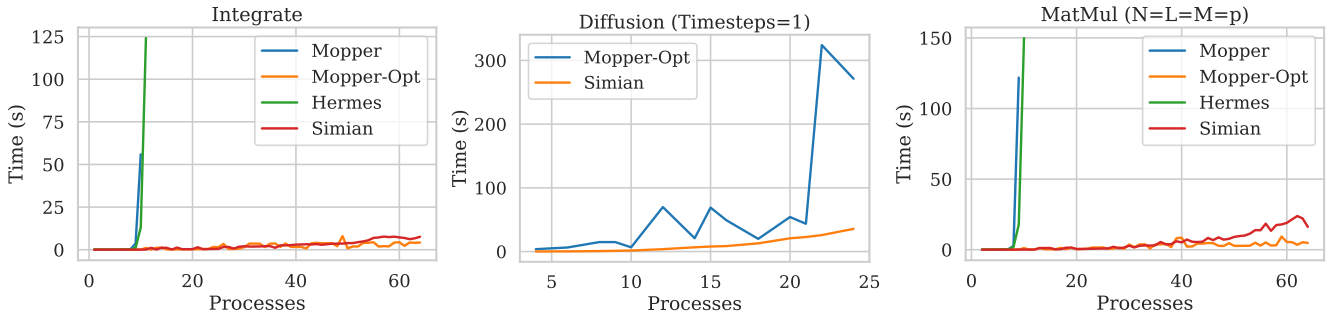
Static analyzers. ParTypes [31] is a type-based analyzer for MPI programs. Given the protocol specification of a program, it verifies the program using session-types. The tool can avoid the state-space explosion problem and scale to much larger MPI programs through its type-based analysis. However, the technique is limited to loop-free and deterministic programs. MPI-Checker [10] is another static analyzer, based on Clang-LLVM, to detect errors in MPI programs. However, its ability to detect communication deadlocks is limited to only certain scenarios. In particular, the nondeterminism in communication is not handled in MPI-Checker.

Model checkers. ISP [42], DAMPI [44], MPI-SPIN [37], and AISLINN [4] are explicit-state model checkers that verify MPI programs on a fixed input for reachability properties such as deadlocks and user-defined assertions. They re-run the program under analysis to explore different executions and are found to be slow in comparison to trace-based symbolic verifiers [20]. HERMES is a model checker which is built on trace verification technique of MOPPER. It has support for *multi-path* programs, however, is limited to detecting only deadlocks. CIVL [49] is a symbolic model checker for various concurrent programs, such as OpenMP, Pthreads, CUDA, and MPI. CIVL is outperformed by trace verification tools such as MOPPER [14]. MPI-SV [47] is a recent symbolic model checker that extracts *path-level* symbolic models and verify them individually. While it is shown to be more performant that CIVL, it is ultimately a full bounded symbolic verifier and will face known issues in addressing large symbolic inputs.

Debuggers. There are numerous debuggers for MPI programs, such as UMPIRE [43], MARMOT [28], MUST [15], which monitor deadlocks and other memory-based errors in the program run. While scalable for single program run, these approaches are neither sound nor complete.

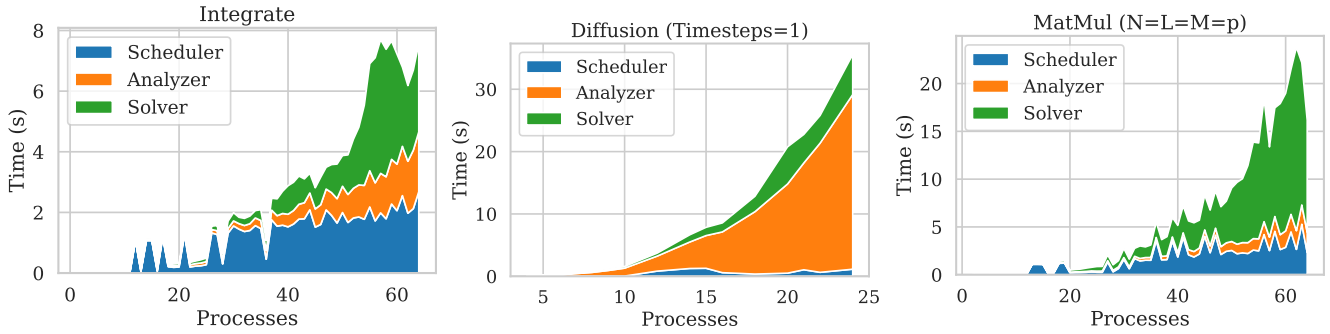
Symmetry in model checking. Expoiting structural symmetries in model checking is a well-studied area [5, 11, 21]. A reasonably recent survey [45] notes the fundamental and applied aspects of symmetry in automated formal verification. Popular explicit state model checkers such as MURφ [22], SPIN [17] and TOPSPIN [9] support symmetry reduction.

Symmetry in SAT-solving. Message passing programs are not the only domain where the resulting SAT encodings exhibit significant symmetry. Consequently, there is a body of research exploring how to best exploit symmetry while solving SAT instances [2, 3, 8, 25, 33, 40, 41]. BREAKID[8] and SHATTER[2] are static pre-processors which detect symmetry in the formula and add symmetry-breaking constraints, much like we do. To detect symmetry, they construct a graph with a node for each variable/literal and compute it's automorphisms. In contrast, our program graph has a node for each MPI action. The number of variables in our SAT encoding is quadratic in the number of MPI actions, making symmetry detection at the formula level largely inferior to our approach. With both BREAKID and SHATTER, we observed that symmetry breaking considerably slows down the overall runtime. [33, 40, 41] exploit symmetry dynamically within the solver. Such symmetry may not even be apparent in the SAT formula before solver execution. Since we treat the SAT solver as a black-box, these methods are orthogonal to ours.



(a) Integrate **(b) Diffusion (Timesteps=1)** **(c) MatMul (N=L=M=p)**

Figure 3: Total Times



(a) Integrate **(b) Diffusion (Timesteps=1)** **(c) MatMul (N=L=M=p)**

Figure 4: Component Times

8 CONCLUSION

This paper presents a technique to significantly reduce the search space for examining communication deadlocks in an MPI program. The technique exploits symmetries by way of specifying symmetry breaking predicates. Often the symmetries may not be present globally in a program. The approach decomposes a program into self-contained epochs that allow for composable detection of deadlocks. Our technique is sound and complete for single-path MPI programs.

Experimental results show that exploiting symmetries and epochs can give exponential savings in solving times. In some cases, these gains can also be realized by MOPPER-Opt, which includes a specific optimization for one source of local symmetry, namely consecutive wildcard receives. In contrast, our tool SIMIAN handles epochs and symmetries in a much more general fashion, allowing it to verify complex benchmarks with rich communication structures where MOPPER-Opt fails to scale, while also staying competitive on simpler benchmarks due to the light-weight nature of it's overheads. Thus, SIMIAN subsumes all prior state-of-the-art trace verifiers for MPI programs.

As a part of future work, we will consider an extension of symmetries and epochs to multi-path programs. Further, we would like to expand the scope of our tool to a larger set of MPI primitives and evaluate against recent benchmarks such as those discussed in [29]. It would also be interesting to explore the applicability of our techniques in concurrency frameworks other than MPI.

REFERENCES

[1] Fadi A Aloul, Arathi Ramani, Igor L Markov, and Karem A Sakallah. 2002. Solving difficult SAT instances in the presence of symmetry. In *Proceedings 2002 Design Automation Conference*. 731–736.

[2] Fadi A Aloul, Karem A Sakallah, and Igor L Markov. 2006. Efficient symmetry breaking for boolean satisfiability. *IEEE Trans. Comput.* 55, 5 (2006), 549–558.

[3] Markus Anders. 2022. SAT Preprocessors and Symmetry. *arXiv preprint arXiv:2205.12799* (2022).

[4] Stanislav Böhm, Ondřej Meca, and Petr Jančar. 2016. State-space reduction of non-deterministically synchronizing systems applicable to deadlock detection in MPI. In *International Symposium on Formal Methods*. Springer, 102–118.

[5] Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. 1993. Exploiting Symmetry In Temporal Logic Model Checking. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV '93)*. 450–462.

[6] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. 1996. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, Vol. 5. Morgan Kaufmann Pub, 148.

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[8] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. 2016. Improved static symmetry breaking for SAT. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 104–122.

[9] Alastair F. Donaldson and Alice Miller. 2006. A Computational Group Theoretic Symmetry Reduction Package for the Spin Model Checker. In *Algebraic Methodology and Software Technology, 11th International Conference, AMAST, Proceedings*

(Lecture Notes in Computer Science, Vol. 4019), Michael Johnson and Varmo Vene (Eds.). Springer, 374–380.

[10] Alexander Droste, Michael Kuhn, and Thomas Ludwig. 2015. MPI-checker: static analysis for MPI. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*, Hal Finkel (Ed.). ACM, 3:1–3:10.

[11] E. Allen Emerson and A. Prasad Sistla. 1993. Symmetry and Model Checking. In *Computer Aided Verification, 5th International Conference, CAV (Lecture Notes in Computer Science, Vol. 697)*. Springer, 463–478.

[12] Mahdi Eslamimehr and Jens Palsberg. 2014. Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 353–365.

[13] Vojtech Forejt, Saurabh Joshi, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. 2017. Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs. *ACM Trans. Program. Lang. Syst.* 39, 4 (2017), 15:1–15:27.

[14] Vojtěch Forejt, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. 2014. Precise predictive analysis for discovering communication deadlocks in MPI programs. In *International Symposium on Formal Methods*. Springer, 263–278.

[15] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. 2012. MPI runtime error detection with MUST: advances in deadlock detection. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, Jeffrey K. Hollingsworth (Ed.). IEEE/ACM, 30.

[16] Jim Holt, Anant Agarwal, Sven Brehmer, Max J. Domeika, Patrick Griffin, and Frank Schirrmeister. 2009. Software Standards for the Multicore Era. *IEEE Micro* 29, 3 (2009), 40–51.

[17] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295.

[18] Yu Huang and Eric Mercer. 2015. Detecting MPI Zero Buffer Incompatibility by SMT Encoding. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9058)*, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 219–233.

[19] Yu Huang, Eric Mercer, and Jay McCarthy. 2013. Proving MCAPI executions are correct using SMT. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 26–36.

[20] Yu Huang, Benjamin Ogles, and Eric Mercer. 2020. A predictive analysis for detecting deadlock in MPI programs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 18–28.

[21] C. Norris Ip and David L. Dill. 1993. Better Verification Through Symmetry. In *International Conference on Computer Hardware Description Languages and their Applications - CHDL (IFIP Transactions)*, David Agnew, Luc J. M. Claesen, and Raul Camposano (Eds.). 97–111.

[22] C. Norris Ip and David L. Dill. 1996. Verifying Systems with Replicated Components in Mur*phi*. In *Computer Aided Verification, 8th International Conference, CAV (Lecture Notes in Computer Science, Vol. 1102)*. Springer, 147–158.

[23] Tommi Junttila and Petteri Kaski. 2007. Engineering an efficient canonical labeling tool for large and sparse graphs. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 135–149.

[24] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound deadlock prediction. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 146:1–146:29.

[25] Hadi Katebi, Karem A Sakallah, and Igor L Markov. 2010. Symmetry and satisfiability: An update. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 113–127.

[26] Dhriti Khanna, Rahul Purandare, and Subodh Sharma. 2021. Synthesizing Multi-threaded Tests from Sequential Traces to Detect Communication Deadlocks. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. IEEE, 1–12.

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

[28] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. 2003. MARMOT: An MPI Analysis and Checking Tool. In *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, Dresden, Germany (Advances in Parallel Computing, Vol. 13)*, Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, and Wolfgang V. Walter (Eds.). Elsevier, 493–500.

[29] Mathieu Laurent, Emmanuelle Saillard, and Martin Quinson. 2021. The MPI BUGS INITIATIVE: a Framework for MPI Verification Tools Evaluation. In *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 1–9.

[30] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. 2012. Verifying GPU Kernels by Test Amplification. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, 383–394.

[31] Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2015. 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, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 280–298.

[32] Anna Lubiw. 1981. Some NP-complete problems similar to graph isomorphism. *SIAM J. Comput.* 10, 1 (1981), 11–21.

[33] Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. 2018. Cdclsym: Introducing effective symmetry breaking in sat solving. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 99–114.

[34] Matthias S. Mueller, Ganesh Gopalakrishnan, Bronis R. de Supinski, David Lecomber, and Tobias Hilbrich. 2011. Dealing with MPI Bugs at Scale: Best Practices. In *In Automatic Detection, Debugging, and Formal Verification*.

[35] Micha Sharir. 1981. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications* 7, 1 (1981), 67–72.

[36] Subodh Sharma. 2013. *Predictive Analysis Of Message Passing Applications*. Ph. D. Dissertation. University of Utah. https://collections.lib.utah.edu/ark:/87278/s6mk9n24

[37] Stephen F. Siegel. 2007. Verifying Parallel Programs with MPI-Spin. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4757)*, Franck Cappello, Thomas Hérault, and Jack J. Dongarra (Eds.). Springer, 13–14.

[38] Stephen F. Siegel and Timothy K. Zirkel. 2011. FEVS: A Functional Equivalence Verification Suite for High-Performance Scientific Computing. *Math. Comput. Sci.* 5, 4 (2011), 427–435.

[39] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[40] Rodrigue Konan Tchinda and Clémentin Tayou Djamegni. 2019. Enhancing static symmetry breaking with dynamic symmetry handling in CDCL SAT solvers. *International Journal on Artificial Intelligence Tools* 28, 03 (2019), 1950011.

[41] Tevich Treethanyaphong and Athasit Surarerks. 2018. Dynamic symmetry breaking in SAT using augmented clauses with a polynomial-time lexicographic pruning. In *2018 2nd European Conference on Electrical Engineering and Computer Science (EECS)*. IEEE, 242–247.

[42] Sarvani S Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M Kirby. 2008. ISP: a tool for model checking MPI programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 285–286.

[43] Jeffrey S. Vetter and Bronis R. de Supinski. 2000. Dynamic Software Testing of MPI Applications with Umpire. In *Proceedings Supercomputing 2000, November 4-10, 2000, Dallas, Texas, USA. IEEE Computer Society, CD-ROM*, Jed Donnelley (Ed.). IEEE Computer Society, 51.

[44] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. 2010. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*. IEEE, 1–10.

[45] Thomas Wahl and Alastair F. Donaldson. 2010. Replication and Abstraction: Symmetry in Automated Formal Verification. *Symmetry* 2, 2 (2010), 799–847.

[46] Ruini Xue, Xuezheng Liu, Ming Wu, Zhenyu Guo, Wenguang Chen, Weimin Zheng, Zheng Zhang, and Geoffrey M. Voelker. 2009. MPIWiz: subgroup reproducible replay of mpi applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*. 251–260.

[47] Hengbiao Yu, Zhenbang Chen, Xianjin Fu, Ji Wang, Zhendong Su, Jun Sun, Chun Huang, and Wei Dong. 2020. Symbolic verification of message passing interface programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1248–1260.

[48] Wenting Zhao. 2017. Encoding Lexicographical Ordering Constraints in SAT. (2017).

[49] Manchun Zheng, Michael S Rogers, Ziqing Luo, Matthew B Dwyer, and Stephen F Siegel. 2015. CIVL: formal verification of parallel programs. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 830–835.