Efficient Predictive Monitoring of Message Passing Interface Programs

JIAQIANG YAO, National University of Defense Technology, China
HAOCHENG GENG, National University of Defense Technology, China
ZHENBANG CHEN*, National University of Defense Technology, China

The Message Passing Interface (MPI) is the standard programming model for high-performance computing, yet nondeterministic scheduling in concurrent executions makes reliability assurance highly challenging. Beyond deadlocks, property-related bugs such as modifying a send buffer before a nonblocking send completes or accessing a freed RMA window are often hard to reproduce and validate, as they typically manifest only under rare event interleavings and can be both latent and catastrophic. Existing dynamic checkers usually cover only the observed schedule of a single execution; dynamic verification largely focuses on deadlocks; and static techniques scale poorly to medium-to-large programs, often timing out or running out of memory. Overall, existing techniques do not support scalable analysis of temporal properties in large MPI programs.

To address these limitations, we propose the first efficient predictive monitoring approach for temporal properties in MPI programs. From a single execution, we collect an event trace and use trace equivalence to predict the set of legal equivalent executions under the same input. We then check whether any predicted execution satisfies the target property. To make this process sound, we formalize three correctness criteria for MPI trace reordering and enforce them through MPI-semantics-driven dependence extraction, tracked with vector clocks. To improve efficiency, we exploit the bounded length of a pattern language and reduce long-trace reordering to reordering short patterns. We implement our approach in MPI-PRV, instantiate ten representative MPI bug properties, and evaluate it on 13 real-world MPI applications with 38 configurations. MPI-PRV successfully and correctly analyzes all 38 tasks, whereas MPI-SV and MUST analyze only 14 and 11 tasks, respectively. MPI-PRV also achieves orders-of-magnitude reductions in runtime and memory usage, demonstrating strong efficiency and scalability for large MPI programs.

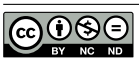
CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Formal software verification**; • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: MPI programs, predictive monitoring, trace equivalence

ACM Reference Format:
Jiaqiang Yao, Haocheng Geng, and Zhenbang Chen. 2026. Efficient Predictive Monitoring of Message Passing Interface Programs. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA187 (October 2026), 22 pages. https://doi.org/10.1145/3832278

*Corresponding author

Authors' Contact Information: Jiaqiang Yao, jiaqiangyao@nudt.edu.cn, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, Changsha, China; Haocheng Geng, genghaocheng23@nudt.edu.cn, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, Changsha, China; Zhenbang Chen, zbchen@nudt.edu.cn, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, Changsha, China.



ACM 2994-970X/2026/10-ARTISSTA187
https://doi.org/10.1145/3832278

1 INTRODUCTION

As data-intensive workloads and computational demands continue to grow, high-performance computing (HPC) [3] has become a cornerstone of modern scientific research and industrial applications. The Message Passing Interface (MPI) [11, 35] is the de facto standard programming model for HPC and is widely used in parallel applications. However, inherent nondeterminism in process scheduling makes it challenging to develop reliable MPI programs [8–10, 29].

Deadlocks [15] are among the most common bugs in MPI programs. Beyond deadlocks, MPI applications also suffer from many severe bugs, such as modifying a send buffer before a nonblocking send completes and accessing an already-freed RMA window. These issues can be naturally expressed as temporal properties. Importantly, temporal properties can also capture application-specific erroneous behaviors and help identify the root causes of bugs. However, due to nondeterministic process scheduling, they often remain latent and appear benign in most executions, yet can be catastrophic once triggered.

Existing approaches to MPI correctness can be broadly categorized into *dynamic* and *static* techniques. Dynamic techniques include (i) *runtime correctness checking*, which validates a concrete execution by instrumenting MPI calls, such as MARMOT [18], Umpire [39], and MUST [12, 13]. While these tools can detect certain runtime errors, they typically cover only a single schedule under a given input and therefore rarely expose nondeterministic bugs that depend on subtle event interleavings. Dynamic techniques also include (ii) *dynamic verification*, which predicts alternative process schedules for a given input; however, existing methods largely focus on deadlocks, ignore schedules irrelevant to deadlock matching, and provide limited support for additional temporal properties, such as ISP [37, 38] and DAMPI [40]. **In contrast, static techniques aim to verify temporal properties via explicit behavioral modeling but scale poorly in practice**, as exemplified by MPI-SPIN [30, 31] and MPI-SV [5]: MPI-SPIN requires manual modeling effort, and MPI-SV often times out or exceeds memory limits on medium-to-large applications even under a single input; moreover, some bugs manifest only at larger process scales, further challenging these approaches. Overall, none of the existing techniques effectively supports scalable verification or checking of temporal properties for large-scale MPI programs.

Accordingly, we propose an efficient predictive monitoring approach for temporal properties in MPI programs. Given a temporal property, our method maximizes the exploration of all legal process schedules induced by a single input, enabling predictive analysis of large MPI applications with low overhead and short runtime. *To the best of our knowledge, MPI-PRV is the first predictive monitoring technique that supports temporal properties for MPI programs.*

Specifically, given an MPI program $\mathcal{MP}$ and a temporal property ψ , where ψ consists of an abstract event definition D and an event-occurrence specification L, we instrument $\mathcal{MP}$ according to D to collect an execution trace σ and analyze it online. Based on Mazurkiewicz trace equivalence [23], we soundly reorder events in σ to obtain an equivalent trace σ' (denoted $\sigma' \equiv \sigma$), thereby constructing the set of equivalent executions $EQ(\sigma)$. We then check whether there exists a trace in $EQ(\sigma)$ that satisfies the specification. **To ensure the correctness of reordering**, we formalize three criteria for correct reorderings of MPI event traces and provide corresponding enforcement mechanisms for each. In particular, we ensure MPI-semantic soundness via dynamic dependence extraction, maintained with vector clocks.

To improve the efficiency of predictive monitoring, we introduce a pattern language [1] to specify event-occurrence constraints L. Exploiting the fact that patterns have bounded length, we reduce reordering the full event trace σ to reordering the pattern language, and then check whether a reordered pattern L' is a subsequence of σ while satisfying the correctness constraints of trace reordering. Based on this approach, we implement MPI-PRV, a predictive monitoring tool for

C/C++ MPI programs. We instantiate more than ten representative temporal bug properties and evaluate MPI-PRV on 13 real-world MPI applications with 38 configurations, including scalability experiments. The results demonstrate our method's effectiveness, efficiency, and strong scalability.

The main contributions of this paper are as follows.

- We propose a predictive monitoring framework for temporal properties in MPI programs. From a single execution, our framework predicts the set of legal equivalent executions under the same input, enabling effective predictive monitoring of temporal properties.
- We formalize three criteria for correct reordering of MPI event traces and provide an enforcement mechanism for each, ensuring the soundness of trace reordering.
- We design an efficient predictive monitoring algorithm for MPI programs. It analyzes large MPI applications with near-linear time and memory overhead, ensuring scalability.
- We implement MPI-PRV and instantiate a set of representative bug properties commonly seen in MPI applications. We evaluate MPI-PRV on real-world MPI programs. The results show that our approach successfully and correctly analyzes all 38 tasks, whereas MPI-SV and MUST analyze only 14 and 11 tasks, respectively. Compared with MPI-SV, MPI-PRV reduces runtime and memory consumption by 71.5× and 678.8×, respectively; compared with MUST, MPI-PRV reduces memory consumption by 88.2×.

2 ILLUSTRATION

In this section, we first briefly introduce MPI programs, then present the relevant concepts of event sequences and trace equivalence that form the foundation of our predictive scheduling. We also define the pattern language to characterize the temporal properties targeted for monitoring, and finally use a motivation example to demonstrate the effectiveness of our approach.

2.1 MPI Communication Mechanism

MPI programs consist of multiple processes that exchange information via MPI communication calls. These calls are broadly classified as blocking and nonblocking: blocking operations proceed only after the communication completes, whereas nonblocking operations return immediately and allow computation to overlap communication, improving efficiency but increasing analysis complexity. To focus on the core issues, we ignore MPI buffering and assume zero-sized send/receive buffers; hence we omit communication buffers throughout the rest of the paper.

We use a small set of common MPI primitives for exposition:

- Send(d, id, des, tag): a blocking send issued by process id to destination des with payload d; tag distinguishes messages and participates in matching.
- Recv(d, id, src, tag): a blocking receive issued by process id to obtain payload d; when $src{=}*$, it may match a send from any source process; tag participates in matching.
- Isend(d, id, des, tag, req): a nonblocking send that returns immediately; the handle req (a pointer to an MPI_Request object) uniquely identifies the communication request.
- Irecv(d, id, src, tag, req): a nonblocking receive that returns immediately; when $src{=}*$, it may match a send from any source; req uniquely identifies the communication request.
- Wait(req): waits for completion of the nonblocking communication request identified by req.
- Barrier(): a barrier synchronization that blocks the calling process until all processes in the communicator arrive; our approach also applies to other collective primitives.

2.2 Event Sequence and Trace Equivalence

An event e is the basic unit of an event trace σ, which models a single program execution. We abstract events from program actions, such as procedure calls, variable accesses, and assignments.

In MPI programs, multiple processes execute concurrently, therefore, the same operation occurring in different processes must be distinguished. We represent an event as a pair $e = \langle p, op \rangle$, where p is the process identifier and op is the program operation. For notational convenience, we write p_op for $\langle p, op \rangle$. Thus, events with the same op are distinct if they originate from different processes. Accordingly, the event alphabet Σ denotes the set of all events appearing in σ.

Trace equivalence [23] provides a principled framework for reasoning about concurrent computations. The key idea is that swapping two adjacent independent events does not change the concurrent semantics of an execution. Given an event alphabet Σ, the independence relation $\mathcal{I}$ is a reflexive and symmetric relation on Σ, i.e., $\mathcal{I} \subseteq \Sigma \times \Sigma$, which specifies the set of event pairs that can be considered independent. The corresponding dependence relation is defined as

$$\mathcal{D} = (\Sigma \times \Sigma) \setminus \mathcal{I}.$$

For example, let $\Sigma = \{a, b, c\}$ and $\mathcal{I} = \{\langle a, b \rangle, \langle b, a \rangle\}$. Then $\mathcal{D} = \{\langle a, c \rangle, \langle c, a \rangle, \langle b, c \rangle, \langle c, b \rangle\}$. Since a and b are independent, $a \cdot b \cdot c$ is trace-equivalent to $b \cdot a \cdot c$, denoted by $a \cdot b \cdot c \equiv b \cdot a \cdot c$. Importantly, trace equivalence is generated by swapping adjacent independent events; non-adjacent events cannot be swapped directly. The relation $\equiv$ is transitive: if $\sigma \equiv \sigma_a$ and $\sigma_a \equiv \sigma_b$, then $\sigma \equiv \sigma_b$. Therefore, by repeatedly swapping adjacent independent events, one can derive the entire equivalence class of a trace σ.

2.3 Pattern Language

Pattern languages [1] form a subclass of regular languages. They constrain only a finite sequence of key events in a trace and are therefore less expressive than general regular languages. Given an event alphabet Σ, a language L is a pattern language over Σ^* if it can be written as

$$L = \Sigma^* e_1 \Sigma^* \cdots \Sigma^* e_d \Sigma^*,$$

where $d \in \mathbb{N}$ and $e_1, \ldots, e_d \in \Sigma$. Intuitively, L specifies only the ordered occurrence of the key events $e_1, \ldots, e_d$ and abstracts away the events between them. Accordingly, we denote the pattern language compactly as

$$L = \text{PATT} : e_1, e_2, \ldots, e_d.$$

We use this compact notation for pattern languages throughout the paper. Although pattern languages are less expressive than general specification formalisms, they are sufficient to capture many practically important properties. For example, to specify the use-after-free (UAF) property, we abstract `new`, `free`, and `deref` as allocation, deallocation, and dereference events, respectively. For any pointer referring to a specific heap object, the pattern $L = \text{PATT} : \text{new}, \text{free}, \text{deref}$ captures all UAF violations on that object. Using pattern languages also aligns with the *small bug depth* assumption: many defects depend only on the relative ordering of a small number of key events [2, 6]. Consequently, monitoring only these key events is sufficient in practice and is crucial for reducing the complexity of predictive analysis.

To accommodate a wider variety of specifications, we also support generalized pattern languages. Given an event alphabet Σ, a language L is a generalized pattern language if it can be expressed as a finite union

$$L = L_1 \cup L_2 \cup \cdots \cup L_m,$$

where $m \in \mathbb{N}$ is finite and, for each L_i, either $L_i = \{\varepsilon\}$ or L_i is a pattern language. By allowing unions of multiple patterns, generalized pattern languages are strictly more expressive than a single pattern language; a pattern language is the special case with $m = 1$. For simplicity, we use the term "pattern language" to refer to both in this paper.

```
      // ... (previous code omitted for brevity)
      if (world_rank < world_size-1 && world_rank > 0) {
        MPI_Isend((void*)(a0+(size*size)), (size*size), MPI_DOUBLE,world_rank+1, 0, MPI_COMM_WORLD, &req1);
        MPI_Irecv((void*)a0, (size*size), MPI_DOUBLE,world_rank-1, 0, MPI_COMM_WORLD, &req2);

        // a0 is being used before the receive completes
        for (i = 2; i < local_n; i++)
        for (j = 1; j < n+1; j++)
        for (k = 1; k < n+1; k++)
        a1[i*size*size+j*size+k] = ( /* ... uses a0[...] ... */ ) * fac;

        /* wait for incoming comms to complete */
        MPI_Wait(&req2, MPI_STATUS_IGNORE);
      }
      // ... (subsequent code omitted for brevity)

```

Fig. 1. Adept Kernel MPI Error Code Example.

P_0	P_1	P_2	P_3
...	...	...	...
Isend(a1,0,1,0,r1)			
	Isend(a1,1,2,0,r1)		
		Isend(a1,2,3,0,r1)	
			Irecv(a2,3,2,0,r2)
		Irecv(a2,2,1,0,r2)	
	Irecv(a2,1,0,0,r2)		
		use(a2)	
	use(a2)		
	...		
	wait(r2)	...	
		wait(r2)	
...	...	...	...

Fig. 2. Event subsequence of Adept Kernel MPI

2.4 Motivating Example

We motivate this work with a real bug that MPI-PRV discovered in the open-source project Adept Kernel MPI. The project provides standardized computational kernels for linear algebra, sparse computation, and stencil-based workloads. In **stencil.c**, there is a typical risk of prematurely using a nonblocking receive buffer: as shown in the source-code excerpt in Figure 1, the program issues `MPI_Irecv` to receive data into `a0` and then immediately reads and computes on `a0` in subsequent loops, while the completion wait (`MPI_Wait`) is performed only after the computation finishes. This ordering may access the receive buffer before the communication completes, leading to undefined behavior. Figure 2 illustrates an abstract event subsequence that may arise in a normal 4-process execution. It indicates that, under some legal schedules, the execution can contain the fragment $p2\_Irecv \cdot p2\_Use \cdot p1\_Isend$. Hereafter, we write p_i_Op to denote an abstract event of kind Op issued by process p_i (e.g., $p2\_Irecv$ is a nonblocking receive on process 2), and we use $\cdot$ for sequential composition. Accordingly, $p2\_Irecv$ denotes a nonblocking receive issued by process 2, $p2\_Use$ denotes the use of the received data, and $p1\_Isend$ denotes the matching send issued by process 1. This fragment implies that process 2 uses the receive buffer before the matching send starts or completes, potentially compromising safety and yielding nondeterministic outcomes.

Existing techniques have difficulty exposing this bug. MPI-SPIN requires manual modeling and does not scale well to real-world software. MPI-SV can verify temporal specifications, but even modeling a single execution path of Adept Kernel MPI with four processes involves 2,608 MPI communication operations; MPI-SV runs out of memory and fails to detect the bug under the given input. Dynamic verifiers such as ISP mainly target deadlocks and local assertions, and thus cannot

express this property. Finally, we ran the runtime correctness checker MUST five times, and none of the runs reported an error. In most executions, the trace contains only "benign" subsequences such as $p2\_Irecv \cdot p1\_Isend \cdot p2\_Use$ or $p1\_Isend \cdot p2\_Irecv \cdot p2\_Use$, rather than the risky ordering $p2\_Irecv \cdot p2\_Use \cdot p1\_Isend$.

In contrast, MPI-PRV does not rely solely on the trace observed in a single run. Instead, it predicts a set of legal equivalent reorderings under the same input. Thus, even if the execution exposes only $p2\_Irecv \cdot p1\_Isend \cdot p2\_Use$ or $p1\_Isend \cdot p2\_Irecv \cdot p2\_Use$, MPI-PRV can still derive the correctly reordered fragment $p2\_Irecv \cdot p2\_Use \cdot p1\_Isend$ via sound reordering, thereby reliably exposing the bug.

Concretely, we define abstract events and instrument the program, and specify the bug with the pattern language $L_{\text{error}} = \text{PATT} : p2\_Irecv, p2\_Use, p1\_Isend$. Under the given input, we execute the instrumented program and collect an event trace σ (Figure 2), where no failure is manifested in this run. During execution, MPI-PRV incrementally extracts the dependence relation $\mathcal{D}$ to ensure sound reordering. It first adds intra-process constraints; for example, in process P1, it orders $p1\_Isend$ before $p1\_Irecv$ to prevent their relative order from being swapped. It then adds inter-process constraints derived from MPI semantics; for instance, leveraging the blocking "wait-for-completion" semantics of $p2\_Wait$, it introduces a dependence from $p1\_Isend$ to the first event after $p2\_Wait$ (e.g., $p2\_Barrier$), ensuring that the relative order between $p1\_Isend$ and $p2\_Barrier$ is preserved in any reordering.

After constructing $\mathcal{D}$, we reduce trace reordering to reordering the pattern L_{error}. We show that

$$\text{Trace}_{L_{\text{error}}} = p1\_Isend \cdot p2\_Irecv \cdot p2\_Use$$

is a legal reordering of L_{error}, i.e., it can be derived from L_{error} via trace equivalence without violating any dependence constraints. Moreover, $\text{Trace}_{L_{\text{error}}}$ is a subsequence of σ. Therefore, σ admits a legal equivalent reordering that satisfies L_{error}, and MPI-PRV reports the bug. To improve efficiency, MPI-PRV uses vector clocks to maintain dependences and incrementally updates the feasibility of pattern reorderings, enabling low-overhead online prediction. In our experiments, **MPI-PRV detects this bug within 0.4 s.**

3 EFFICIENT PREDICTIVE MONITORING FOR MPI PROGRAMS

This section presents our efficient predictive monitoring framework for MPI programs and the key technical details.

3.1 Framework

To efficiently detect property-related defects in large-scale MPI programs, this paper proposes an efficient predictive monitoring framework for MPI programs, as illustrated in Figure 3. Given an MPI program $\mathcal{MP}$ and a temporal property ψ, we first analyze ψ in the parsing phase to derive abstract event definitions D and an event occurrence specification L. Here, D contains the definitions of abstract events and the instrumentation information of the program. All abstract events form an event alphabet Σ, and the event occurrence specification L is a pattern language over Σ, which defines the occurrence logic of events to be predictively monitored.

Based on the instrumentation information in the abstract event definition D, we instrument $\mathcal{MP}$ to establish the mapping relationship between abstract events and actual program behaviors (e.g., function calls, variable assignments), thereby obtaining the instrumented MPI program $\mathcal{MP}_1$. Given the program input, running $\mathcal{MP}_1$ yields the event sequence σ of the program's current execution. We then extract the dependence relation $\mathcal{D}$ (Section 3.3) in accordance with the definition of correct reordering (Section 3.2) and manage it using vector clocks (Section 3.4), which ensures the correctness of event sequence reordering based on trace equivalence.

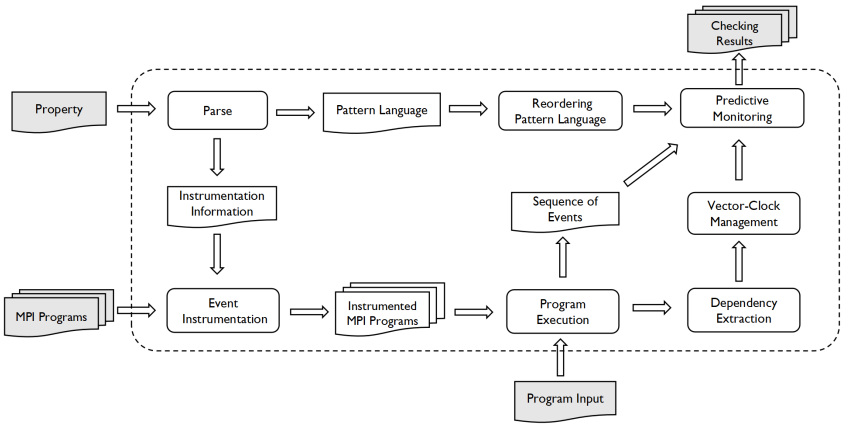


Fig. 3. The framework of MPI-PRV

To improve the efficiency of predictive monitoring, we leverage the finite-length property of pattern languages to transform the reordering of event sequences into that of pattern languages (Section 3.5). With event vector clock information, the event sequence σ, and the reordered pattern language, we perform incremental online checking on the reordered pattern language (Section 3.6) to determine whether there exists a reordered pattern language that satisfies vector clock constraints, and is a subsequence of σ. If such a reordered pattern language exists, it indicates that there exists a constraint-compliant reordered event sequence satisfying the target pattern language.

3.2 Correct Reordering

The core prerequisite of predictive monitoring is to ensure the correctness of generated event-trace reorderings, while aiming to cover, as much as possible, the event sequences induced by all legal process schedules under the given input. Accordingly, we introduce the following definitions of correct reordering for execution event traces in MPI programs.

Given an event trace $\sigma \in \Sigma^*$ over event alphabet Σ, let $\mathcal{R} : \Sigma^* \to \Sigma^*$ be a reordering operator and let $\sigma' = \mathcal{R}(\sigma) = [e'_1, e'_2, \ldots, e'_m]$. We call σ' a *valid reordering* of σ if:

$$\text{(a)} \quad \forall\, e_i, e_j \in \sigma :\ e_i \preceq_{proc} e_j \ \Rightarrow\ \sigma' \models (e_i \preceq_{proc} e_j),$$
$$\text{(b)} \quad \mathcal{R}_{cf}(\sigma') = \mathcal{R}_{cf}(\sigma),$$
$$\text{(c)} \quad \mathcal{R} \models C_{mpi}.$$

We denote this as $\sigma' \in \text{ValidReorder}(\sigma)$. Here, $\preceq_{proc}$ is the intra-process order: for $e_i, e_j \in \sigma$, if $proc(e_i) = proc(e_j) = p_k$ and e_i occurs before e_j in process p_k, then $e_i \preceq_{proc} e_j$. $\mathcal{R}_{cf}(\sigma)$ denotes the control-flow dependence relation induced by σ. C_{mpi} is the set of MPI semantic constraints, and $\mathcal{R} \models C_{mpi}$ requires that the reordering does not violate MPI operation semantics.

Condition (a) enforces intra-process soundness. Since each process executes sequentially, the relative order of events within a process must be preserved under any interleaving. Accordingly, we add order constraints between consecutive events in each process. Condition (b) preserves trace integrity after reordering. Changing control-flow dependencies may introduce new events or eliminate existing ones, undermining predictive monitoring; we therefore preserve, for every wildcard-source receive in the original trace σ, its concrete send–receive matching, so that control-flow dependencies remain consistent after reordering. Condition (c) captures MPI semantic

P_0	P_1
Event$_{01}$	
	Event$_{11}$
Send(d,0,1,1)	
	Recv(d,1,0,1)
Event$_{02}$	
	Event$_{12}$

(a)

P_0	P_1
Event$_{01}$	
	Event$_{11}$
Isend(d,0,1,1,r)	
	Irecv(d,1,0,1,r)
Event$_{02}$	
	Event$_{12}$

(b)

P_0	P_1
Event$_{01}$	
	Event$_{11}$
	Irecv(d,1,0,1,r)
	Event$_{12}$
Isend(d,0,1,1,r)	
Event$_{02}$	

(c)

Fig. 4. Correct Reordering Example

P_0	P_1
Event$_{01}$	
	Event$_{11}$
Send(d,0,1,1)	
	Irecv(d,1,0,1,r)
Event$_{02}$	
	Event$_{12}$
Event$_{03}$	
	Event$_{13}$

(a)

P_0	P_1
Event$_{01}$	
	Event$_{11}$
Isend(d,0,1,1,r)	
Event$_{02}$	
	Recv(d,1,0,1)
	Event$_{12}$
Event$_{03}$	
	Event$_{13}$

(b)

P_0	P_1
Event$_{01}$	
	Event$_{11}$
Isend(d,0,1,1,r1)	
	Irecv(d,1,0,1,r2)
Event$_{02}$	
	Wait(r2)
Event$_{03}$	
	Event$_{12}$

(c)

P_0	P_1
Event$_{01}$	
	Event$_{11}$
Isend(d,0,1,1,r1)	
	Irecv(d,1,0,1,r2)
barrier	
	barrier
Event$_{02}$	
	Event$_{12}$

(d)

Fig. 5. Dependence extraction example

constraints and is the most fundamental yet challenging requirement. MPI provides a rich set of operations with specialized semantics that substantially restrict feasible reorderings.

As shown in Figure 4a, with blocking send/receive operations, $p1\_Event_{12}$ cannot be moved before $p0\_\text{Send}(d, 0, 1, 1)$, and $p0\_Event_{02}$ cannot be moved before $p1\_\text{Recv}(d, 1, 0, 1)$ in any valid reordering. In contrast, when these calls are replaced with nonblocking ones (Figure 4b), $p1\_Event_{12}$ can be reordered to occur before $p0\_\text{Isend}(d, 0, 1, 1, r)$, and $p0\_Event_{02}$ can be reordered to occur before $p1\_\text{Irecv}(d, 1, 0, 1, r)$, as illustrated in Figure 4c. Hence, we dynamically extract MPI-semantics-driven dependences and incorporate them into $\mathcal{D}$ (Section 3.3).

3.3 Dependence Extraction Based on MPI Semantics

Constructing the dependence relation $\mathcal{D}$ is key to ensuring correct reordering. Given an event alphabet Σ and an execution trace σ, once $\mathcal{D}$ is fixed, the independence relation is determined as $\mathcal{I} = (\Sigma \times \Sigma) \setminus \mathcal{D}$. By trace-equivalence theory, only adjacent independent events can be swapped without changing the trace semantics. Hence, adding dependence constraints to $\mathcal{D}$ directly shrinks the space of feasible reorderings.

Algorithm 1: MPI-Semantics-Driven Dependence Update

```
   HandleDependency(e, 𝒟)
   Data: e = [p, op, targetId, tag, request]: MPI event; 𝒟: dependence relation
1  begin
2      Let Prev_p ← the last event of process p preceding e
3      𝒟 ← 𝒟 ∪ {⟨Prev_p, e⟩, ⟨e, Prev_p⟩}                    ▷ Preserve intra-process order
4      if op is a blocking Send or Recv then
5          𝒟 ← 𝒟 ∪ {⟨match_σ(e), next(e)⟩, ⟨next(e), match_σ(e)⟩}    ▷ completion boundary
6      else if op is Wait(req) then
7          for x ∈ Comp(req) do
8              𝒟 ← 𝒟 ∪ {⟨x, next(e)⟩, ⟨next(e), x⟩}
9      else if op is a blocking collective c_p on communicator C then
10         for q ∈ Part(C), q ≠ p do
11             𝒟 ← 𝒟 ∪ {⟨c_q, next(e)⟩, ⟨next(e), c_q⟩}
```

To satisfy condition (a) (Section 3.2), we add dependences between adjacent events in the same process (Lines 2–3). We then perform MPI-operation-specific matching and dependence handling to enforce conditions (b) and (c) (Section 3.2).

For a **blocking send or receive** event e (i.e., op = Send or op = Recv), we add bidirectional dependences between its matched event $\text{match}_\sigma(e)$ and the next event of e's process, $\text{next}(e)$ (where $\text{next}(e)$ denotes the event immediately following e in e's process, or a sentinel event if e is the last event of that process): $\langle \text{match}_\sigma(e), \text{next}(e)\rangle$ and $\langle \text{next}(e), \text{match}_\sigma(e)\rangle$ (Line 5). This rule captures the MPI completion boundary: a blocking send cannot return until its matching receive has been posted, and a blocking receive cannot return until the matching send has arrived, so the matched event must precede any subsequent event of e (and symmetrically) under any valid reordering. For example, in Figure 5a, when the execution encounters $p0\_\text{Send}(d, 0, 1, 1)$, its matched event is $p1\_\text{Irecv}(d, 1, 0, 1, r)$ and $\text{next}(e) = p0\_Event_{02}$; we therefore add $\langle p1\_\text{Irecv}(d, 1, 0, 1, r), p0\_Event_{02}\rangle$ (and its reverse), ensuring that $p1\_\text{Irecv}(d, 1, 0, 1, r)$ always occurs before $p0\_Event_{02}$ in any valid reordering. When the matched event is a blocking Recv instead, the same rule applied at the sender and at the receiver yields the two boundaries $\langle r, \text{next}(e)\rangle$ and $\langle e, \text{next}(r)\rangle$, preserving the completion order on both sides.

Nonblocking send/receive operations (Isend and Irecv) post their operations and return immediately, so they do not trigger additional dependence handling on their own. Their communication effects are enforced indirectly: when the matched operation is a blocking Send or Recv, the completion-boundary rule above already adds the required dependences; when both sides are nonblocking, completion is enforced later through the Wait that completes the request. For example, in Figure 5b, the execution encounters $p0\_\text{Isend}(d, 0, 1, 1, r)$, whose matching receive is $p1\_\text{Recv}(d, 1, 0, 1)$. When processing the matched $p1\_\text{Recv}(d, 1, 0, 1)$, the rule at Line 5 fires with $\text{match}_\sigma(p1\_\text{Recv}) = p0\_\text{Isend}(d, 0, 1, 1, r)$ and $\text{next}(p1\_\text{Recv}) = p1\_Event_{12}$, yielding the dependence $\langle p0\_\text{Isend}(d, 0, 1, 1, r), p1\_Event_{12}\rangle$ (and its reverse), which prevents the sender-side posting from being reordered after the receiver's subsequent event.

A blocking Recv is handled by the same completion-boundary rule as Send (Line 5), with $\text{match}_\sigma(e)$ being its matched send. An Irecv, being nonblocking, adds no dependence of its own; its completion is enforced either when the matched Send is processed (if blocking) or when a subsequent Wait completes the request. Notably, to satisfy condition (b) (Section 3.2), for any wildcard-source receive ($src = *$) we fix its resolved match in the original trace, so that the same

send–receive pairing is preserved under reordering. This may shrink the reordering space for programs involving wildcard receives, but it guarantees soundness.

In addition, we handle **Wait** and **blocking collective operations** via dedicated rules (Lines 6 and 9). For Wait(req), let Comp(req) denote the set of remote events whose occurrence is required to complete the nonblocking operation referenced by req — concretely, the matched send of the underlying Irecv (or the matched receive of the underlying Isend). For each $x \in \text{Comp}(req)$, we add bidirectional dependences between x and next(e) (Line 8). As shown in Figure 5c, when the execution encounters $p1\_\text{Wait}(r_2)$, the handle r_2 refers to $p1\_\text{Irecv}(d, 1, 0, 1, r_2)$, whose matched send is $p0\_\text{Isend}(d, 0, 1, 1, r_1)$; hence $\text{Comp}(r_2) = \{p0\_\text{Isend}(d, 0, 1, 1, r_1)\}$, and with $e_1 = \text{next}(e) = p1\_Event_{12}$ we add $\langle p0\_\text{Isend}(d, 0, 1, 1, r_1), p1\_Event_{12}\rangle$ (and its reverse), enforcing the wait-for-completion boundary. When the execution encounters a **blocking collective** (e.g., Barrier), let c_p denote the collective event of process p. For each other participant q of the same communicator, we add bidirectional dependences between c_q and next(e) (Line 11). For example, in Figure 5d, processing the barrier on $p0$ yields $\langle p1\_Barrier, p0\_Event_{02}\rangle$ (and its reverse), and processing the barrier on $p1$ yields $\langle p0\_Barrier, p1\_Event_{12}\rangle$ (and its reverse), so all participants' collective events are constrained to occur before any subsequent event in every other participant.

3.4 Vector Clock Management

To efficiently maintain the dependence relation $\mathcal{D}$ and thus constrain the reordering space, we manage $\mathcal{D}$ with vector clocks. Vector clocks [7] are a standard mechanism in distributed systems for tracking causal relations among events. A vector timestamp is a mapping that assigns each process a natural number, capturing the logical time of each process during execution:

$$\text{VC} : P \to \mathbb{N}.$$

Here, P is the set of processes and $\mathbb{N}$ is the set of natural numbers. A vector clock is a state variable composed of vector timestamps; it tracks each process's logical time as well as the relative progress among processes. Each process has a timestamp, which is updated as the process executes.

We associate each process $p \in P$ with a vector timestamp VC_p and label each event e with a vector timestamp VC_e. While scanning the event sequence, for the current event e executed by process p, we first advance p's local component, i.e., $\text{VC}_p[p]$ += 1 (Algorithm 3, Line 6), and identify the events $e' \in \Sigma$ such that $\langle e', e\rangle \in \mathcal{D}$. We propagate these dependences by merging vector timestamps. The merge operator $\sqcup$ is defined component-wise as

$$\text{VC}_1 \sqcup \text{VC}_2 \;=\; \left\langle \max\big(\text{VC}_1(p),\, \text{VC}_2(p)\big) \;\middle|\; p \in P \right\rangle.$$

Accordingly, we update $\text{VC}_p \leftarrow \text{VC}_p \sqcup \text{VC}_{e'}$ for each dependent event e', and finally set $\text{VC}_e \leftarrow \text{VC}_p$ (Algorithm 3, Lines 10–13).

For each event e executed by process p, we advance the local component $\text{VC}_p[p]$ += 1 and assign the updated process timestamp to the event, i.e., $\text{VC}_e \leftarrow \text{VC}_p$, which implicitly preserves the intra-process order under reordering. Accordingly, the explicit intra-process dependences added in Algorithm 1 (Lines 2–3) are redundant in the vector-clock implementation and are retained in the algorithm only for clarity of presentation. Moreover, event timestamps allow us to derive the required ordering constraints and check whether a reordering violates them. Let VC_e and VC_m be two vector timestamps; we define the partial order $\sqsubseteq$ as

$$\text{VC}_m \sqsubseteq \text{VC}_e \;\Leftrightarrow\; \forall p \in P,\ \text{VC}_m(p) \le \text{VC}_e(p).$$

Therefore, if $\text{VC}_m \sqsubseteq \text{VC}_e$, then m causally precedes e. Any valid reordering must preserve this order, i.e., m must occur before e. Conversely, if $\text{VC}_m \not\sqsubseteq \text{VC}_e$ (i.e., m and e are incomparable, or e

Algorithm 2: Reordering Pattern Language

```
   RecorderGeneralPATT(L)
   Data: L: A pattern language, consisting of multiple single patterns L_1, L_2, ..., L_m
1  begin
2      Let M be a list initially containing only the empty sequence ⟨⟩, i.e., M = [⟨⟩]
3      Let MapListM be a map, initially empty: MapListM ← ∅
4      Let m = |L|                                   ▷ Obtain the number of single patterns
5      for i = 1 to m do
6          Let d = Len(L_i)
7          subsequences ← GenerateSubsequences(L_i, d)   ▷ Generate the subsequences of L_i
8          for each sub in subsequences do
9              permutations ← GeneratePerms(sub)     ▷ Obtain the permutations of subsequence
10             for each perm in permutations do
11                 M.add(perm)
12         MapListM ← MapListM ∪ {(L_i, M)}
13         M ← [⟨⟩]                                  ▷ Reset M for the next single pattern
14     return MapListM
```

causally precedes m), then VC_m does not constrain m to precede e (Algorithm 3, Lines 23); hence m may occur before or after e in a reordering, provided that all required orderings are respected.

3.5 Reordering Pattern Language

Given a pattern language $L = L_1 \cup L_2 \cup \cdots \cup L_m$, each component L_i ($1 \le i \le m$) is a single pattern (i.e., contains no further union), written as $L_i = \mathrm{PATT} : e_1, e_2, \ldots, e_d$, where $m, d \in \mathbb{N}$ and both are finite. Empirically, most temporal properties depend on only a few key events and typically satisfy $\boldsymbol{d < 5}$. We therefore replace the costly reordering of a long trace σ with reordering and matching the short patterns L_i, substantially reducing prediction overhead. Because $L = \bigcup_{i=1}^{m} L_i$, it suffices to find an i such that some legal equivalent reordering of σ satisfies L_i, which implies that σ satisfies L. Compared with directly reordering traces with thousands of events, this shift from trace-level to pattern-level reordering is key to efficient predictive monitoring.

Algorithm 2 reorders a pattern language L by *precomputing* all reorderings required for monitoring. Given $L = L_1 \cup L_2 \cup \cdots \cup L_m$, it constructs a map MapListM that associates each single pattern L_i with its precomputed reordering information, enabling efficient predictive monitoring. Specifically, the algorithm processes each L_i in turn: it first enumerates all subsequences of L_i (Line 7), including L_i itself, and then generates all permutations for each subsequence sub (Lines 8–11), inserting them into a working list M. After all subsequence permutations of L_i are collected, the algorithm stores the key–value pair (L_i, M) into MapListM (Line 12) and resets M to initialize the processing of L_{i+1} (Line 13). The final output MapListM thus records, for every $L_i \in L$, the complete set of reorderings of all its subsequences. By shifting reordering generation from online monitoring to offline precomputation, MapListM enables efficient incremental lookup and update during predictive monitoring (Section 3.6).

3.6 Efficient Incremental Predictive Monitoring

This section presents an efficient online incremental predictive monitoring algorithm. The algorithm adapts the PatternTrack procedure of Ang and Mathur [1] for monitoring pattern languages over concurrent traces, and is tailored to MPI by extracting the dependence relation $\mathcal{D}$ dynamically from MPI semantics, reducing long-trace reordering to short-pattern reordering, and integrating

Algorithm 3: Efficient Incremental Predictive Monitoring

PredictiveMonitor(σ, P, L, Σ, $\mathcal{D}$)
Data: $\sigma \in \Sigma^*$: event trace, P: number of parallel processes, L: pattern language, $(\Sigma, \mathcal{D})$: alphabet with dependence relation

```
1  begin
2      Let CacheMap be a Map, CacheMap(L_i) = [⟨⟩] for every L_i ∈ L
3      Let VC_e = ⊥ for every e ∈ Σ
4      Let VC_p = ⊥ for every p ∈ P
5      MapListM ← RecorderGeneralPATT(L)          ▷ Precompute all subsequence permutations
6      for each e ∈ σ do
7          Let e = [p, op, targetId, tag, request]
8          VC_p[p] ← VC_p[p] + 1                  ▷ Update the clock of the current process
9          HandleDependency(e, &D)                ▷ Update dependencies based on MPI semantics
10         for each ⟨e, e'⟩ ∈ D do
11             VC_p ← VC_p ⊔ VC_e'                ▷ Update vector clock according to D
12             D ← D \ {⟨e, e'⟩, ⟨e', e⟩}         ▷ Remove processed dependencies
13         VC_e ← VC_p
14         for each L_i ∈ L do
15             Let L_i = PATT : a_1, ..., a_d
16             if e ∈ ⟨a_1, ..., a_d⟩ then
17                 CacheList ← CacheMap(L_i)
18                 SubRecordlist ← MapListM(L_i)
19                 for each S ∈ SubRecordlist ending with e do
20                     Let S' ← S[: −1]
21                     if S' ∈ CacheList then
22                         Let S† ← Sort(S, PATT : a_1, ..., a_d)   ▷ Sort according to pattern L_i
23                         if ∀m ∈ S', (e occurs after m in S†) ∨ (VC_m ⋢ VC_e) then
24                             CacheList.add(S)   ▷ Pattern language (subsequence) reordering validity
25                 if ∃S ∈ CacheList with |S| = d then
26                     return YES                 ▷ Pattern language L_i is satisfied
27                 CacheMap ← CacheMap ∪ {(L_i, CacheList)}
28     return NO
```

vector-clock checks into the subsequence extension rule. It builds on three key ideas: (1) *Vector-clock–based dependence encoding*: we incrementally extract and maintain the dependence relation $\mathcal{D}$ and assign each event a vector timestamp, enabling low-overhead enforcement of required precedence constraints; (2) *From trace-level to pattern-level reordering*: we reduce the trace-equivalent reordering problem on a long execution σ to reordering and matching a short pattern language L, leveraging the bounded pattern length to sharply shrink the search space; (3) *Incremental caching of partial matches*: while scanning σ, we maintain per-L_i caches of feasible partial matches and validate only candidate extensions induced by newly observed events, avoiding repeated re-enumeration and rechecking from scratch.

As shown in Algorithm 3, the procedure starts with preprocessing: it initializes the cache CacheMap(L_i) for each single pattern L_i and initializes vector timestamps for every process and every event (Lines 2–4). It then invokes RecorderGeneralPATT(L) to construct MapListM (Line 5).

Here, MapListM(L_i) stores the full permutations of all subsequences of L_i, serving as a "pattern-reordering lookup table". This shifts the cost of generating subsequences and permutations from the online monitoring phase to an offline precomputation step.

During online monitoring, the algorithm performs a single pass over the event trace σ (Line 6). For each incoming event e, it first increments the clock component of e's process (Line 8) and then updates the dependence relation $\mathcal{D}$ on the fly according to MPI semantics (Line 9). For each dependence edge adjacent to e, i.e., $\langle e, e' \rangle \in \mathcal{D}$, it merges the vector timestamp of e' into the current process clock and removes the processed dependences (Lines 10–12). It finally sets $\mathrm{VC}_e = \mathrm{VC}_p$ (Line 13), enabling subsequent precedence checks via the partial order $\sqsubseteq$.

The algorithm then performs incremental matching for each single pattern $L_i = \mathrm{PATT} : a_1, \ldots, a_d$ (Line 14–27). Let CacheList = CacheMap(L_i) denote the set of validated legal *partial reorderings*, and let SubRecordlist = MapListM(L_i) be the precomputed candidate reorderings (Lines 17–18). When the current event e appears in L_i, the algorithm enumerates all candidates $\mathcal{S} \in$ SubRecordlist that end with e (Line 19) and extracts the prefix $\mathcal{S}' = \mathcal{S}[:-1]$. It proceeds to validate $\mathcal{S}$ only if $\mathcal{S}' \in$ CacheList (Line 21), thereby enabling incremental extension: it appends one new event to an already-validated prefix, rather than rechecking the sequence from scratch.

To check the validity of a candidate reordering, the algorithm first restores $\mathcal{S}$ to $\mathcal{S}^\dagger$ by following the canonical order of L_i (Line 22). The candidate $\mathcal{S}$ is admitted only if, for every event m in the prefix $\mathcal{S}'$, either (i) m still precedes e in $\mathcal{S}^\dagger$, or (ii) there is no causal precedence constraint from m to e (i.e., $\mathrm{VC}_m \not\sqsubseteq \mathrm{VC}_e$); if this holds, $\mathcal{S}$ is added to the cache (Lines 23–24). Intuitively, condition (i) covers events that the pattern itself orders before e, while condition (ii) covers events that are concurrent with or causally downstream of e and therefore need not precede it. Once a sequence of length d appears in the cache, L_i is satisfied and the algorithm immediately returns YES (Lines 25–26); otherwise, it updates CacheMap(L_i) and continues processing the subsequent events (Line 27).

Incrementality is demonstrated in the fact that the cache stores verified valid prefixes, and new candidates are generated by extending the "valid prefix + current event." At the same time, dependency checking relies on the $\sqsubseteq$ comparison of vector clocks, avoiding the need to explicitly maintain and repeatedly query a large number of dependency edges. *Efficiency* is reflected in the fact that the algorithm only traverses the sequence σ once, with most of the computation focused on candidate expansions for a small number of short patterns and vector clock comparisons. As a result, the overall overhead grows approximately linearly with $|\sigma|$, making it suitable for online predictive monitoring of large-scale MPI programs.

A worked example. Consider a trace $\sigma = p1\_a \cdot p0\_b$ and a pattern $L = \mathrm{PATT} : p0\_b, p1\_a$, with $P = \{p0, p1\}$ and $\Sigma = \{p1\_a, p0\_b\}$. For simplicity, assume no MPI-semantic dependencies are introduced. Algorithm 3 initializes CacheMap(L) = $[\langle\rangle]$, $VC_{p1\_a} = VC_{p0\_b} = \{0, 0\}$, and $VC_{p0} = VC_{p1} = \{0, 0\}$, and constructs MapListM(L) = $\{\langle\rangle, \langle p1\_a\rangle, \langle p0\_b\rangle, \langle p1\_a, p0\_b\rangle, \langle p0\_b, p1\_a\rangle\}$, where $\langle\rangle$ is the empty subsequence generated by Algorithm 2.

Processing the first event $p1\_a$: VC_{p1} becomes $\{0, 1\}$, and since $\mathcal{D} = \emptyset$, $VC_{p1\_a} = \{0, 1\}$. With CacheList = $[\langle\rangle]$ and SubRecordlist = MapListM(L), the candidates ending with $p1\_a$ are $\mathcal{S} = \langle p1\_a\rangle$ and $\mathcal{S} = \langle p0\_b, p1\_a\rangle$. For $\mathcal{S} = \langle p1\_a\rangle$, $\mathcal{S}' = \langle\rangle \in$ CacheList, so Line 21 admits it; $\mathcal{S}^\dagger = \langle p1\_a\rangle$, and since $\mathcal{S}' = \langle\rangle$, the condition in Line 23 holds, so $\langle p1\_a\rangle$ is added to CacheList. For $\mathcal{S} = \langle p0\_b, p1\_a\rangle$, $\mathcal{S}' = \langle p0\_b\rangle \notin$ CacheList, so it is discarded. After processing $p1\_a$, CacheMap(L) = $\{\langle\rangle, \langle p1\_a\rangle\}$.

Processing the second event $p0\_b$: $VC_{p0\_b} = VC_{p0} = \{1, 0\}$. Now CacheList = $[\langle\rangle, \langle p1\_a\rangle]$. The candidates ending with $p0\_b$ are $\mathcal{S} = \langle p0\_b\rangle$ and $\mathcal{S} = \langle p1\_a, p0\_b\rangle$. For the latter, $\mathcal{S}' = \langle p1\_a\rangle \in$ CacheList, and after sorting $\mathcal{S}^\dagger = \langle p0\_b, p1\_a\rangle$. For $m = p1\_a \in \mathcal{S}'$, the first disjunct of Line 23 fails,

but the second holds because $VC_{p1\_a} = \{0, 1\} \not\sqsubseteq VC_{p0\_b} = \{1, 0\}$. Hence $\langle p1\_a, p0\_b \rangle$ is admitted; its length equals $|L| = 2$, so Line 25 triggers and the algorithm returns YES.

4 IMPLEMENTATION AND EVALUATION

In this section, we describe the implementation of MPI-PRV, present our research questions and experimental setup, and report the evaluation results.

4.1 Implementation

We implemented our predictive monitoring approach for C/C++ MPI programs and developed a tool named MPI-PRV. Built on CCMOP's instrumentation technique [41], MPI-PRV identifies MPI operations and property-specified operations on the program's abstract syntax tree (AST). Instrumentation code is inserted at the corresponding locations to collect event sequences, enabling predictive analysis during execution. In addition, to ensure a fair experimental comparison, we extracted MPI-SV's [5] single-input verification mode. Note that the extracted MPI-SV no longer verifies the entire program space, but instead performs verification under a single input; for simplicity, we also refer to it as MPI-SV in the remainder of this paper.

MPI operation coverage. Beyond the point-to-point primitives introduced in Section 2, the current implementation supports commonly used blocking and nonblocking collectives (e.g., `MPI_Bcast`, `MPI_Gather`, `MPI_Scatter`, `MPI_Reduce`, `MPI_Allreduce`, and `MPI_Alltoall`), blocking completion and probe operations (e.g., `MPI_Waitall`, `MPI_Waitany`, and `MPI_Probe`), and RMA synchronization operations (e.g., `MPI_Win_fence` and `MPI_Win_lock`/`MPI_Win_unlock`). Less common operations, such as `MPI_Scan` and `MPI_Exscan`, are not yet explicitly modeled but can be supported by defining corresponding dependence-update rules. Operations without specialized communication semantics, such as `MPI_Finalize`, can be represented as user-defined events.

4.2 Research Questions

- **RQ1**: *Bug specification capability*: How effective is MPI-PRV at characterizing bugs? Can it detect latent bugs in programs effectively?
- **RQ2**: *Effectiveness*: Can MPI-PRV effectively analyze real-world MPI programs? How does its efficacy compare to state-of-the-art (SOTA) tools?
- **RQ3**: *Efficiency*: What is MPI-PRV's efficiency in analyzing real-world MPI programs? How does its efficiency compare to SOTA tools?
- **RQ4**: *Scalability*: As the length of analyzed event sequences increases, how do MPI-PRV's time and space overheads scale? Can it be effectively applied to large-scale MPI program analysis?

4.3 Experimental Setup

Benchmark. Table 1 summarizes the MPI programs in our benchmark suite. All subjects are real-world open-source MPI applications widely used in prior studies [5, 15, 17]. We additionally include a subset of buggy microbenchmarks from the MPI Bugs Initiative [19] to evaluate the effectiveness of our abstracted bug properties (Section 4.4.1). The "Any-Source Receive" column marks programs containing wildcard receives (e.g., `MPI_ANY_SOURCE`). For such programs, we conservatively fix the resolved matching observed in the original trace; hence the analysis may miss some feasible schedules (false negatives) but introduces no false positives. For programs without any-source receives, MPI-PRV predicts all legal schedules under the given input.

Baseline. We choose MPI-SV [5] as the static baseline and MUST [12, 13] as the runtime checking baseline. All methods analyze MPI programs under a single input. MPI-SV explores a large portion of the schedule space induced by that input, whereas MUST checks only the observed schedule.

Table 1. Information of MPI Programs in Our Benchmark.

Program Name	Code Size (LOC)	Any-Source Receive	Description
DTG	90	✓	Dependence transition group
integrate	181	✓	Integral computing
diffusion2d	197	×	Simulation of diffusion equation
Gauss_elim	341	✓	Gaussian elimination
image-manip	360	✓	Image manipulation
Congrad	361	×	Conjugate Gradient
poisson_mpi	564	×	Solving the Poisson equation
Heat	613	✓	Heat equation solver
adept-kernel-mpi	5461	×	Computations for particle transport
kf-ray	8435	✓	KF-Ray parallel raytracer
clustalW	23265	✓	Multiple sequence alignment
ParMETIS	49568	×	Load-balanced data partitioning
MiniAMR	712572	×	Adaptive Mesh Refine program

Properties. For the real-world evaluation, we monitor three bug classes commonly observed in MPI applications: (i) communication operations left incomplete at `MPI_Finalize`, (ii) modification of a send buffer before the corresponding nonblocking send completes, and (iii) access to a receive buffer before the corresponding nonblocking receive completes. Beyond these safety properties, the framework can detect functionality-related temporal bugs and support root-cause localization. Because some benchmarks, such as diffusion2d and Congrad, contain no known bugs, we use representative event sequences from benign executions as target properties to enable uniform evaluation and cross-tool comparison. The complete property suite is available in our artifact.[1]

Experimental configurations. For each real-world program, we set a time limit of 2 hours and a memory limit of 32 GB. We ran all experiments on a server with 128 cores at 3.1 GHz and 500 GB RAM, running Ubuntu 22.04. To reduce run-to-run variability, we repeated each experiment three times and report the average.

4.4 Experimental Results

4.4.1 Results of RQ1. Table 2 summarizes the ten pattern-language properties (*Prop1–Prop10*) used in our evaluation. Events in these properties can be distinguished by their process identifiers and parameters. It lists each property's identifier, description, corresponding pattern specification, and a dedicated buggy verification program, together with whether MPI-PRV detects the target bug. These properties cover major MPI bug classes considered by prior work, including synchronization errors, resource leaks, use-after-free, and RMA window lifecycle violations. Across the ten buggy programs, each property successfully exposes its corresponding defect under our predictive monitoring.

Many MPI bugs manifest only under rare schedules and are difficult to reproduce with schedule-observing checkers. For example, for *Prop6* (RMA access after window free), a concrete run may yield the benign sequence $Wincreate \cdot WinGet \cdot Winfree$, whereas predictive monitoring can still infer the feasible equivalent reordering $Wincreate \cdot Winfree \cdot WinGet$ that violates the property. To account for nondeterminism, we repeat predictive monitoring five times per program; **Yes** in Table 2 indicates that the bug is detected in all five runs.

[1] https://github.com/MPI-PRV/MPI-PRV

Table 2. Bug Detection Properties for MPI Programs

ID	Property Description	Pattern Language	Verification Program	Result
Prop1	Unfinished communication operations after `MPI_Finalize`	PATT: $px\_Finalize\ py\_Isend \cup px\_Finalize\ py\_Irecv$	`CallOrdering_Finalize_nok.c`	Yes
Prop2	Modifying the send buffer before a nonblocking send completes	PATT: $px\_Isend(x)\ px\_Write(x)\ py\_Irecv(x) \cup px\_Isend(x)\ px\_Write(x)\ py\_Recv(x)$	`Concurrency_Recv_Isend_nok.c`	Yes
Prop3	Using the receive buffer before a nonblocking receive completes	PATT: $px\_Irecv(x)\ px\_Use(x)\ py\_Isend(x) \cup px\_Irecv(x)\ px\_Use(x)\ py\_Send(x)$	`Concurrency_Irecv_Send_nok.c`	Yes
Prop4	Repeated release of a communicator or use after release	PATT: $CommCreate(c)\ CommFree(c)\ CommFree(c) \cup CommCreate(c)\ CommFree(c)\ CommUse(c)$	`Communicator_free_error.c`	Yes
Prop5	Use of a pointer after it has been freed	PATT: $CreatePtr(p)\ FreePtr(p)\ Use(p)$	`MPI_Ptr_uaferror.c`	Yes
Prop6	RMA access to a freed window	PATT: $WinCreate(win)\ WinFree(win)\ WinGet(win) \cup WinCreate(win)\ WinFree(win)\ WinPut(win)$	`MPIRMA_Use_error_nok.c`	Yes
Prop7	Multiple releases of an RMA window	PATT: $WinCreate(win)\ WinFree(win)\ WinFree(win)$	`MPIRMA_doublefree_nok.c`	Yes
Prop8	Modifying the persistent receive buffer before confirming completion of the receive operation	PATT: $RecvInit(x)\ Start(x)\ Write(x)\ Wait(x)$	`LocalConcurrency_Recv_init_Send_nok.c`	Yes
Prop9	Waiting for a nonblocking reduction only after reaching a subsequent barrier	PATT: $px\_Ireduce\ px\_Barrier\ px\_Wait$	`CallOrdering_Ireduce_Barrier_nok.c`	Yes
Prop10	Resource leakage due to unreleased communicators after creation or duplication	PATT: $CommCreate(c)\ NoFree(c)\ Finalize \cup CommDup(c)\ NoFree(c)\ Finalize$	`ResLeak_Comm_create_nok.c`	Yes

Furthermore, our property expressiveness extends beyond this. Users can abstract properties for specific bug event sequences or root causes (e.g., no dependent receive before broadcast) or extract function-related properties (e.g., a `MPI_Irecv` preceding another `MPI_Irecv` within the same process triggering functional errors).

Answer to RQ1: *Our approach can characterize the vast majority of MPI error types and supports temporal properties for functionality or bug root causes. Across the 10 buggy test programs, it effectively detects all target bugs.*

Table 3. Performance and Analysis Results of MPI-PRV, MPI-SV, and MUST

Program	#procs	#Events	MPI-PRV			MPI-SV			MUST		
			#T	#M	#R	#T	#M	#R	#T	#M	#R
DTG	5	15	0.007	4.00	✓	0.013	22.93	✓	0.513	476.98	✓
integrate	2	8	0.004	3.00	✓	0.336	21.06	✓	0.274	231.93	×
	8	50	0.023	4.00	✓	43.242	1088.00	✓	2.213	716.85	×
	10	64	0.026	4.12	✓	1171.041	23084.41	✓	3.139	870.40	×
diffusion2d	4	46	0.019	4.00	✓	0.171	32.53	✓	2.121	388.55	✓
Gauss_elim	2	18	0.007	4.00	✓	0.105	20.85	✓	0.536	228.63	✓
	4	40	0.011	4.00	✓	0.191	24.17	✓	0.822	388.00	✓
	8	84	0.023	4.00	✓	19.267	63.00	✓	0.961	708.73	✓
image-manip	4	16	0.006	3.00	✓	0.002	21.14	✓	0.495	391.29	×
	24	56	0.017	4.00	✓	43.860	1113.00	✓	0.976	1996.01	×
	127	264	0.109	4.00	✓	T/O	-	-	1.338	10303.00	×
Congrad	4	12	0.003	4.12	✓	0.001	21.00	✓	0.491	397.25	×
poisson_mpi	4	9196	1.898	6.00	✓	-	M/O	-	1.554	405.96	×
	8	22016	4.048	8.81	✓	-	M/O	-	2.116	733.53	×
	24	80156	16.827	21.31	✓	-	M/O	-	2.865	2008.11	×
	64	282808	71.868	66.54	✓	-	M/O	-	8.727	5367.00	×
	96	347100	114.130	80.30	✓	-	M/O	-	16.240	8182.00	×
	127	594140	193.995	135.57	✓	-	M/O	-	37.582	10612.00	×
Heat	4	64	0.024	4.00	✓	1287.000	9830.00	✓	0.621	395.42	✓
adept-kernel-mpi	4	2608	0.546	4.31	✓	-	M/O	-	1.178	389.84	✓
	8	5889	1.082	4.87	✓	-	M/O	-	1.253	706.21	✓
	24	16848	3.883	6.00	✓	-	M/O	-	1.431	1984.00	✓
	64	50728	12.585	9.85	✓	-	M/O	-	5.292	5216.00	✓
	127	101254	32.445	16.86	✓	-	M/O	-	6.366	10301.00	✓
kf-ray	11	140	0.035	4.12	✓	10.947	236.00	✓	0.213	962.50	×
clustalW	5	37	0.011	3.96	✓	0.197	24.00	✓	0.690	471.48	×
	20	172	0.044	4.12	✓	T/O	-	-	1.180	1673.00	×
	127	1135	0.474	4.36	✓	-	M/O	-	1.324	10283.00	×
ParMETIS	4	598176	158.000	100.06	✓	-	M/O	-	1.259	410.07	×
	6	1161960	469.713	200.55	✓	-	M/O	-	36.734	718.30	×
	8	1895692	987.426	316.65	✓	-	M/O	-	150.663	1235.77	×
MiniAMR	4	7608	1.441	4.00	✓	-	M/O	-	2.696	392.00	×
	8	22816	4.307	6.02	✓	-	M/O	-	3.125	708.00	×
	16	53232	10.235	8.15	✓	-	M/O	-	4.560	1359.62	×
	24	87448	26.807	12.19	✓	-	M/O	-	5.442	1972.04	×
	64	273728	70.517	33.24	✓	-	M/O	-	13.838	5264.00	×
	96	425792	122.273	49.00	✓	-	M/O	-	25.790	7876.00	×
	124	463848	147.399	54.00	✓	-	M/O	-	58.010	10150.00	×
Average	-	-	64.532	31.87	-	4615.168	21632.48	-	10.648	2812.49	-

Note: #procs: Number of processes; #Events: Number of events; #T: Time cost (seconds); #M: Memory overhead (MB); #R: Analysis result (✓: Success, ×: Failure); -: Unknown result; M/O: Memory overflow; T/O: Timeout.

4.4.2 Results of RQ2 and RQ3. Table 3 reports the performance and analysis outcomes of MPI-PRV, MPI-SV, and MUST on 13 real-world MPI applications. For each application, we checked the same satisfiable property under identical inputs. The table varies the number of processes (#procs) and

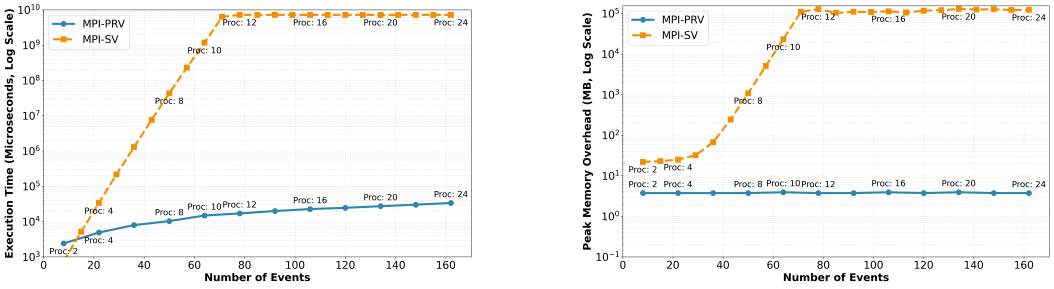
observed events (#Events), and reports runtime (#T), peak memory usage (#M), and the analysis result (#R), where ✓ indicates that the tool finds an event trace satisfying the property.

Overall, we constructed 38 test cases, ranging from 2 to 127 processes and from 8 to 1,895,692 events. MPI-PRV successfully completes all 38 cases within the time and memory limits, and all reported results are correct. MPI-SV completes 14 cases with correct results, but fails to finish the remaining cases due to timeouts or out-of-memory errors as the process count or trace length grows. MUST finishes all cases without timeouts or out-of-memory failures, yet produces correct results for only 11 cases. This is expected because MUST is a dynamic correctness checker: it analyzes only the observed schedule and cannot anticipate property-guided alternative schedules, which leads to many false negatives.

Answer to RQ2: *Across the 38 workloads, MPI-PRV, MPI-SV, and MUST produce correct results on 38, 14, and 11 cases, respectively; thus, MPI-PRV is substantially more effective for analyzing properties of real MPI programs.*

For efficiency, when computing averages, we assign 2 hours to each timeout (T/O) and 32 GB to each out-of-memory case (M/O). Under this metric, MPI-SV exhibits the highest overhead, timing out on 2 cases and exceeding the memory limit on 21 cases, with average runtime 4615.168 s and average memory usage 21632.48 MB. These results indicate that MPI-SV generally cannot produce results within the given budgets once the trace exceeds roughly 200 events. In contrast, MPI-PRV maintains low overhead across all workloads: runtime ranges from milliseconds to a few hundred seconds (e.g., 193.995 s on poisson_mpi with 594,140 events and 127 processes), with an average runtime of 64.532 s. MUST is faster on average (10.648 s) since it checks only the observed schedule and avoids schedule prediction. In terms of memory, MPI-PRV consistently stays within 3–316.65 MB and scales mildly with program size (e.g., on poisson_mpi, increasing processes from 4 to 127 raises memory from 6.00 MB to 135.57 MB), yielding an average of 31.87 MB. Compared with MPI-SV, MPI-PRV reduces runtime and memory by 71.5× and 678.8×, respectively; compared with MUST, it reduces memory by 88.2×.

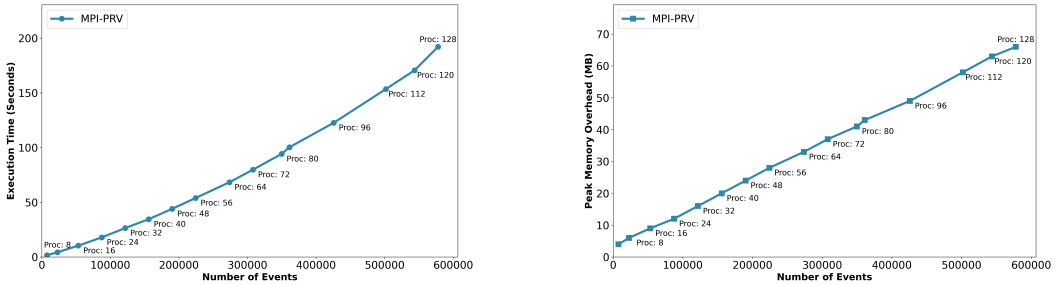
Answer to RQ3: *Across the 38 cases, MPI-PRV incurs average time and memory overheads of 64.532 s and 31.87 MB, outperforming the baselines with orders-of-magnitude lower overhead than MPI-SV and substantially lower memory consumption than MUST.*



(a) Execution Time Overhead (µs, Log Scale)

(b) Peak Memory Overhead (KB, Log Scale)

Fig. 6. Performance overhead of Integrate.



(a) Execution Time Overhead (Seconds)

(b) Peak Memory Overhead (MB)

Fig. 7. Performance overhead of MiniAMR.

4.4.3 Results of RQ4. To characterize how MPI-PRV's overhead evolves with the length of the analyzed trace, we conduct a sampling-based evaluation on Integrate and MiniAMR. We monitor *Prop1* under the same input, impose no memory limit, and use a 2-hour timeout. We further compare the time and memory overhead trends against MPI-SV. In the resulting plots, each data point is labeled with its MPI process count.

Figure 6 reports results on the small-scale benchmark Integrate (2–24 processes; up to 160 events). Both subfigures use a log-scale y-axis to highlight growth trends. MPI-PRV consistently incurs microsecond-level time overhead and only a small peak memory footprint, and both grow mildly as the number of events and processes increases. In contrast, MPI-SV's overhead increases sharply: its runtime and peak memory rise by multiple orders of magnitude as the trace grows, indicating poor scalability even on this small benchmark.

Figure 7 reports results on the large-scale benchmark MiniAMR (4–128 processes; up to 577,856 events). MPI-SV fails to complete once the trace exceeds 200 events, and thus we report only MPI-PRV. As the number of events increases from 0 to 577,856, MPI-PRV's time overhead increases from 0 to 192.08 seconds and its peak memory overhead increases from 4 MB to 66 MB, both following an approximately linear trend. Even at 128 processes and nearly 6×10^5 events, MPI-PRV stays within a few minutes and tens of MB, demonstrating stable scalability for large MPI traces.

> **Answer to RQ4:** *MPI-PRV incurs low time and memory overhead and scales approximately linearly with trace length, enabling efficient predictive monitoring of large-scale MPI programs.*

4.5 Threats to Validity

Internal validity. For programs that issue wildcard receives (e.g., `MPI_ANY_SOURCE`), we conservatively fix the resolved matching observed in the original trace to preserve control-flow consistency under reordering. Consequently, MPI-PRV may miss schedules that would arise under alternative wildcard matchings (i.e., false negatives for such programs), while never introducing false positives. For programs without any-source receives, the analysis is sound and complete with respect to the legal reorderings under the fixed input. The choice of the ten monitored properties also introduces a selection bias: while they cover representative MPI bug classes reported in prior work, other defect categories (e.g., liveness properties beyond reachability) are not yet covered by the pattern language used in this paper. Finally, predictive monitoring is performed on a single execution per input; although trace-equivalent reorderings expose bugs that the observed schedule does not, bugs that require distinct inputs are out of scope.

External validity. Our benchmark consists of 13 open-source MPI applications spanning 38 configurations, including both kernels widely used in prior MPI verification evaluations and large-scale real-world projects (ParMETIS, MiniAMR). Nonetheless, these subjects do not cover the entire MPI ecosystem; in particular, our current implementation targets C/C++ programs, and MPI applications written in Fortran (such as most NAS Parallel Benchmark kernels) are not supported by the current toolchain.

5 RELATED WORK

Dynamic approaches analyze concrete executions and thus naturally account for nondeterminism, but are fundamentally limited by input and schedule coverage. **Runtime correctness checkers** such as MUST [12, 13], TAC [24], MARMOT [18], and Umpire [39] instrument MPI calls and analyze the resulting trace online; however, they typically validate only the observed schedule under a given input. **Dynamic verification** explores alternative schedules under a fixed input. Representative tools include ISP [37, 38] and its extensions [8, 17], DAMPI [40], and NINJA [28], which injects timing perturbations across repeated executions to surface latent message races. Despite these advances, existing dynamic verifiers mainly target deadlocks and limited assertions, offer restricted support for user-specified temporal properties, and may incur substantial overhead.

Static approaches aim to detect errors without executing the program via program analysis and model-based reasoning [31–33]. While they can provide stronger guarantees, they often suffer from over-approximation and scalability issues on real MPI code. MPI-SPIN [30, 31] relies on model checking with manual modeling effort, and MPI-SV [5] and McSimGrid [25] encode communication behaviors with partial-order reduction to verify temporal properties but scale poorly as the explored schedule space grows. Overall, static techniques can be costly in time and memory and may produce false positives.

This work is also related to **runtime predictive analysis** for concurrent programs, which infers bugs from alternative feasible reorderings beyond the observed trace [1, 4, 14, 20]. Prior work largely focuses on implicit concurrency constraints, such as data races, deadlocks, and atomicity violations [16, 27, 34]. While many approaches rely on expensive enumeration or SAT/SMT solving, more recent work improves scalability by exploiting partial-order structure and specialized algorithms [4, 21, 26], and extends prediction beyond races to other bug classes [1, 22, 36].

6 CONCLUSION

We presented MPI-PRV, a predictive monitoring approach for MPI programs that checks temporal properties from a single execution. MPI-PRV combines trace equivalence with MPI operational semantics to extract sound event dependences and ensure the correctness of schedule-induced reorderings. To scale to long executions, it encodes these dependencies with vector clocks and reduces trace-level reordering to reordering and matching short pattern languages, enabling efficient online prediction. Our implementation on C/C++ MPI programs demonstrates effective detection of multiple bug classes and near-linear growth in analysis time with respect to the trace length. We plan to extend MPI-PRV to support more expressive specifications, evaluate it on larger MPI applications, and integrate it with symbolic execution or fuzzing to broaden input coverage.

Data-Availability Statement

Our artifact is available at the following URL: https://github.com/MPI-PRV/MPI-PRV

Acknowledgments

This research was supported by National Key R&D Program of China (No. 2024YFF0908003) and the NSFC Program (No. 62172429). We thank the anonymous reviewers for their valuable feedback.

References

[1] Zhendong Ang and Umang Mathur. 2024. Predictive monitoring against pattern regular languages. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2191–2225. doi:10.1145/3632915

[2] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. *ACM SIGARCH Computer Architecture News* 38, 1 (2010), 167–178. doi:10.1145/1735970.1736040

[3] Rajkumar Buyya. 1999. *High performance cluster computing: programming and applications*. Prentice Hall PTR.

[4] Yan Cai, Hao Yun, Jinqiu Wang, Lei Qiao, and Jens Palsberg. 2021. Sound and efficient concurrency bug prediction. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 255–267. doi:10.1145/3468264.3468549

[5] Zhenbang Chen, Hengbiao Yu, Xianjin Fu, and Ji Wang. 2020. MPI-SV: a symbolic verifier for MPI programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 93–96. doi:10.1145/3377812.3382144

[6] Dmitry Chistikov, Rupak Majumdar, and Filip Niksic. 2016. Hitting families of schedules for asynchronous programs. In *International Conference on Computer Aided Verification*. Springer, 157–176. doi:10.1007/978-3-319-41540-6_9

[7] Colin Fidge. 2002. Logical time in distributed computing systems. *Computer* 24, 8 (2002), 28–33. doi:10.1109/2.84874

[8] 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. doi:10.1007/978-3-319-06410-9_19

[9] Ganesh Gopalakrishnan, Paul D Hovland, Costin Iancu, Sriram Krishnamoorthy, Ignacio Laguna, Richard A Lethin, Koushik Sen, Stephen F Siegel, and Armando Solar-Lezama. 2017. Report of the HPC Correctness Summit, Jan 25–26, 2017, Washington, DC. *arXiv preprint arXiv:1705.07478* (2017). doi:10.48550/arXiv.1705.07478

[10] Ganesh Gopalakrishnan, Robert M Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R De Supinski, Martin Schulz, and Greg Bronevetsky. 2011. Formal analysis of MPI-based parallel programs. *Commun. ACM* 54, 12 (2011), 82–91. doi:10.1145/2043174.2043194

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.

[12] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R de Supinski, and Matthias S Müller. 2013. MPI runtime error detection with MUST: advances in deadlock detection. *Scientific Programming* 21, 3-4 (2013), 109–121. doi:10.1155/2013/314971

[13] Tobias Hilbrich, Martin Schulz, Bronis R de Supinski, and Matthias S Müller. 2010. MUST: A scalable approach to runtime error detection in MPI programs. In *Tools for High Performance Computing 2009: Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*. Springer, 53–66. doi:10.1007/978-3-642-11261-4_5

[14] Jeff Huang, Qingzhou Luo, and Grigore Rosu. 2015. GPredict: Generic predictive concurrency analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 847–857. doi:10.1109/icse.2015.96

[15] 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. doi:10.1145/3324884.3416588

[16] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound deadlock prediction. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29. doi:10.1145/3276516

[17] 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. doi:10.1007/978-3-319-95582-7_28

[18] Bettina Krammer, Katrin Bidmon, Matthias S Müller, and Michael M Resch. 2004. MARMOT: An MPI analysis and checking tool. In *Advances in Parallel Computing*. Vol. 13. Elsevier, 493–500. doi:10.1016/s0927-5452(04)80063-7

[19] 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. doi:10.1109/correctness54621.2021.00008

[20] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. What happens-after the first race? enhancing the predictive power of happens-before based dynamic race detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29. doi:10.1145/3276515

[21] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal prediction of synchronization-preserving races. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29. doi:10.1145/3434317

[22] Umang Mathur and Mahesh Viswanathan. 2020. Atomicity checking in linear time using vector clocks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 183–199. doi:10.1145/3373376.3378475

[23] Antoni Mazurkiewicz. 1986. Trace theory. In *Advanced course on Petri nets*. Springer, 278–324. doi:10.1007/bfb0031416

[24] Patrick Ohly and Werner Krotz-Vogel. 2007. Automated MPI correctness checking what if there was a magic option?. In *Proceedings of the 8th LCI International Conference on High-Performance Clustered Computing*. 19–25.

[25] Anh Pham, Thierry Jéron, and Martin Quinson. 2017. Verifying MPI applications with SimGridMC. In *Proceedings of the First International Workshop on Software Correctness for HPC Applications*. Association for Computing Machinery, New York, NY, USA, 28–33. doi:10.1145/3145344.3145345

[26] Jake Roemer, Kaan Genç, and Michael D Bond. 2020. Smarttrack: efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 747–762. doi:10.1145/3385412.3385993

[27] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods Symposium*. Springer, 313–327. doi:10.1007/978-3-642-20398-5_23

[28] Kento Sato, Dong H Ahn, Ignacio Laguna, Gregory L Lee, Martin Schulz, and Christopher M Chambreau. 2017. Noise injection techniques to expose subtle and unintended message races. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, USA, 89–101. doi:10.1145/3018743.3018767

[29] Zheng Shi, Lasse Møldrup, Umang Mathur, and Andreas Pavlogiannis. 2026. The Complexity of Testing Message-Passing Concurrency. *Proceedings of the ACM on Programming Languages* 10, POPL (2026), 1–32. doi:10.1145/3776643

[30] Stephen F Siegel. 2006. Using mpi-spin to model check mpi programs with nonblocking communication. *Recent Advances in PVM/MPI* (2006).

[31] Stephen F Siegel. 2007. Verifying parallel programs with MPI-Spin. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 13–14. doi:10.1007/978-3-540-75416-9_8

[32] Stephen F Siegel and Timothy K Zirkel. 2011. Automatic formal verification of MPI-based parallel programs. *ACM Sigplan Notices* 46, 8 (2011), 309–310. doi:10.1145/2038037.1941603

[33] Stephen F Siegel and Timothy K Zirkel. 2011. TASS: The toolkit for accurate scientific software. *Mathematics in Computer Science* 5, 4 (2011), 395–426. doi:10.1007/s11786-011-0100-7

[34] Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. 2011. Predictive analysis for detecting serializability violations through trace segmentation. In *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*. IEEE, 99–108. doi:10.1109/memcod.2011.5970516

[35] Marc Snir. 1998. *MPI–the Complete Reference: the MPI core*. Vol. 1. MIT press.

[36] Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound dynamic deadlock prediction in linear time. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1733–1758. doi:10.1145/3591291

[37] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M Kirby. 2008. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *International Conference on Computer Aided Verification*. Springer, 66–79. doi:10.1007/978-3-540-70545-1_9

[38] 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. doi:10.1145/1345206.1345258

[39] Jeffrey S Vetter and Bronis R De Supinski. 2000. Dynamic software testing of MPI applications with Umpire. In *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 51–51. doi:10.1109/sc.2000.10055

[40] 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 *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10. doi:10.1109/sc.2010.7

[41] Yongchao Xing, Zhenbang Chen, Shibo Xu, and Yufeng Zhang. 2023. CCMOP: A Runtime Verification Tool for C/C++ Programs. In *International Conference on Runtime Verification*. Springer, 339–350. doi:10.1007/978-3-031-44267-4_18

Received 2026-01-30; accepted 2026-04-16