Scientific Programming 21 (2013) 109–121
DOI 10.3233/SPR-130368
IOS Press

MPI runtime error detection with MUST: Advances in deadlock detection [1]

Tobias Hilbrich [a,*], Joachim Protze [a,c,d], Martin Schulz [b], Bronis R. de Supinski [b] and Matthias S. Müller [a,c,d]
[a] *Technische Universität Dresden, Dresden, Germany*
E-mail: tobias.hilbrich@tu-dresden.de
[b] *Lawrence Livermore National Laboratory, Livermore, CA, USA*
E-mails: {bronis, schulzm}@llnl.gov
[c] *RWTH Aachen University, Aachen, Germany*
[d] *JARA – High Performance Computing, Aachen, Germany*
E-mails: {mueller, protze}@rz.rwth-aachen.de

Abstract. The widely used Message Passing Interface (MPI) is complex and rich. As a result, application developers require automated tools to avoid and to detect MPI programming errors. We present the Marmot Umpire Scalable Tool (MUST) that detects such errors with significantly increased scalability. We present improvements to our graph-based deadlock detection approach for MPI, which cover future MPI extensions. Our enhancements also check complex MPI constructs that no previous graph-based detection approach handled correctly. Finally, we present optimizations for the processing of MPI operations that reduce runtime deadlock detection overheads. Existing approaches often require $\mathcal{O}(p)$ analysis time per MPI operation, for p processes. We empirically observe that our improvements lead to sub-linear or better analysis time per operation for a wide range of real world applications.

Keywords: Deadlock detection, message passing interface, correctness checking

1. Introduction

The Message Passing Interface (MPI) [10] is a de facto standard for parallel programming. It provides a comprehensive API that enables users to exchange messages between processes efficiently and portably. The standard's design targets and enables high performance through low latency communication and high scalability, but provides few syntactic or semantic extensions to enforce its correct use.

MPI applications can exhibit a wide range of error classes. Simple errors result from invalid arguments such as an invalid array length specification, while other errors involve MPI resources such as communicators or requests, e.g., the user starts a nonblocking communication but does not complete it before calling `MPI_Finalize`. Such errors are often hard to detect, since root-cause and symptoms are far apart. Further, some MPI usage errors may only occur for particular interleavings, for some MPI implementations, or for some systems. These errors arise from the flexibility in the MPI standard with respect to buffering of point-to-point communications or to making collective calls synchronizing.

Various runtime tools can detect MPI errors. They can detect some errors, e.g., the use of an invalid MPI datatype, locally on the application processes. Other errors, such as messaging deadlocks or type mismatches in messages, require information about more than one process and, thus, need a non-local approach. These runtime tools must communicate information from the application processes to a process or thread that runs non-local correctness checks, which complicates their design and scalability. Current tools either have incomplete functionality or scale poorly, which renders them insufficient for reliably detecting errors in large scale applications.

[1] This paper received a nomination for the Best Paper Award at the SC2012 conference and is published here with permission from IEEE.

[*] Corresponding author: Tobias Hilbrich, Technische Universität Dresden, D-01062 Dresden, Germany. E-mail: tobias.hilbrich@tu-dresden.de.

This paper presents MUST (Marmot Umpire Scalable Tool, named after its predecessors), a runtime tool that overcomes the shortfalls of current tools by providing a scalable solution for efficient runtime MPI error checking. MUST detects many classes of MPI correctness errors and provides a flexible plugin concept that allows users to customize the tool to the error classes of interest and to extend its functionality to cover new classes of errors. Although MUST covers various process-local correctness checks, this paper focuses on its non-local checks, primarily deadlock detection. Specifically, we describe how to use MUST to detect, to analyze, and to guide the removal of deadlocks in MPI applications easily.

Figure 1 sketches correct and incorrect MPI communications. We use the notation "Recv(from:x)" for an `MPI_Recv` call that uses `MPI_COMM_WORLD`, where x specifies the source rank and omits the remaining arguments for simplicity in the example. Similarly, we use "Send(to:x)" for `MPI_Send` and "Barrier()" for `MPI_Barrier`, respectively. Figure 1(a) shows a correct communication between two processes, while examples Fig. 1(b) and (c) are erroneous variations. In Fig. 1(b) both processes will block in the receive call and wait for each other to issue a matching send call, which will never happen (assuming no additional application threads execute MPI calls). This example will always deadlock, while Fig. 1(c) shows an implementation dependent deadlock. In MPI, calls to `MPI_Send` are usually buffered for small messages, which would allow both tasks to invoke their send calls and then their receive calls. This example deadlocks, however, if the send calls are not buffered and is therefore incorrect.

Figure 1(d) presents a potential deadlock that only manifests itself in some runs. The *Recv(from:ANY)* call of process 1 uses a wildcard source (`MPI_ANY_SOURCE` in MPI) that allows the receive operation to match a send from any process. If this call receives the message from process 0, the second receive of process 1 can receive the message from process 2. All three processes can then complete the *Barrier* call and continue execution. Alternatively, if process 1 first receives the message from process 2, then its second receive cannot complete, as process 2 does not issue another send before the *Barrier* call. Since process 1 cannot issue the *Barrier* until process 2 sends the message, both processes block indefinitely. These wildcard receives, as well as other MPI constructs, can lead to interleaving dependent MPI deadlocks, which only occur in some application runs.

Our graph-based approach uses the AND⊕OR model [4] to represent wait-for dependencies of active MPI calls. This model simplifies the AND–OR model, which can model wait-for conditions of the most general type. We analyze a graph of the AND⊕OR model to recognize and to visualize deadlocks. In this paper, we generalize this model and improve its application for runtime error detection tools. Specifically, our contributions include:

- Generalization of the AND⊕OR model to handle the full range of MPI usage scenarios previously only covered by the more complex AND–OR model;
- Optimizations in runtime deadlock detection that we empirically observe usually allow MUST to analyze MPI operations with sub-linear complexity;
- Extension to detect deadlocks that involve non-blocking wildcard receives robustly in complex applications; and
- A comprehensive application study that highlights the benefits of our optimizations.

Process 0	Process 1
Send(to:1)	Recv(from:0)
Recv(from:1)	Send(to:0)

(a)

Process 0	Process 1
Recv(from:1)	Recv(from:0)
Send(to:1)	Send(to:0)

(b)

Process 0	Process 1
Send(to:1)	Send(to:0)
Recv(from:1)	Recv(from:0)

(c)

Process 0	Process 1	Process 2
Send(to:1)	Recv(from:ANY)	Send(to:1)
	Recv(from:2)	
Barrier()	Barrier()	Barrier()

(d)

Process 0 (Master)	Processes 1 to (size $-$ 1)
for ($i = 0$; $i <$ size; i++)	Send(to:0)
Recv(from:ANY)	

(e)

Fig. 1. MPI communication examples. (a) Send–recv communication. (b) Recv–recv deadlock. (c) Send–send deadlock. (d) Schedule–dependent deadlock. (e) Master–slave communication.

We structure the rest of the paper as follows. Section 2 compares our graph-based approach to related work. Section 3 introduces the AND⊕OR model and details MUST, our runtime error detection tool. We present our generalization of the AND⊕OR model in Section 4 and highlight the architectural and operational changes to the existing deadlock detection approach that support a far more efficient analysis of MPI operations in Section 5. Section 6 presents the challenges and technology that surround the handling of nonblocking wildcard receives; and we demonstrate the results of our improvements with two benchmark suites in Section 7.

2. Related work

Our work is closely related to MPI runtime error detection tools such as ISP [12], MPI-Check [9], MPICH extension [3], Marmot [7], and Umpire [13]. Figure 2 compares these approaches for the deadlock examples in Fig. 1(b), (c) and (d). We assume that the MPI implementation buffers the *Send* calls in Fig. 1(c) so that the example runs without producing a deadlock. We omit the MPICH extension since it ignores deadlocks and focuses on other correctness checks.

Marmot and MPI-Check implement a timeout-based approach that detects the recv–recv deadlock (Fig. 1(b)) but not the send–send deadlock (Fig. 1(c)), and only detects the schedule–dependent deadlock (Fig. 1(d)) if the error manifests (denoted by *Run*). Further, a timeout approach can lead to false positives and cannot represent the source of the deadlock graphically.

ISP investigates all interleavings of send/recv pairs to verify deadlock freedom of nondeterministic MPI programs. ISP analyzes both execution paths of the schedule–dependent deadlock example so it detects this error. ISP uses a centralized scheduler that executes the application multiple times to check all possible interleavings. Although this approach provides complete coverage, communication patterns often produce an exponential number of interleavings. Figure 1(e) shows an example in which the cost is prohibitive for ISP to validate all possible interleavings, even for a few tasks. Thus, although ISP provides good coverage its applicability is limited.

	Recv–recv	*Send–send*	*Schedule–dependent*
Marmot and MPI-check	Yes	No	Run
ISP	Yes	Yes	Yes
DAMPI	Yes	No	Yes
AND⊕OR	Yes	Yes	Run

Fig. 2. Runtime approach comparison.

DAMPI's [14] distributed detection of alternative interlavings overcomes ISP's limitations. To explore different interleavings, DAMPI rewrites MPI calls based on an existing enumeration of explorations to cover, which eliminates ISP's centralized scheduler. For each interleaving, DAMPI executes the application and detects deadlocks with a timeout. In summary, this approach can detect the recv–recv and the schedule–dependent deadlock. However, DAMPI cannot detect the send–send deadlock or visualize deadlocks graphically and can give false positives due to the timeout-based deadlock detection.

Umpire uses the AND⊕OR model with a graph-based deadlock detection that can detect both the recv–recv and the send–send deadlock but only detects the schedule–dependent deadlock if the error is manifested. This approach cannot lead to false positives and provides a graphical representation of the source of the deadlock. We build on this model and add support for MPI scenarios that the Umpire deadlock conditions do not cover and implement several significant optimizations to that approach. Further, this approach could be used to replace the timeout approach in DAMPI to overcome the drawbacks of timeout-based deadlock detection.

Our generalized AND⊕OR model [4] extends work on the AND–OR graph-theoretic deadlock model [2, 8,15]. The AND–OR model, in which processes can specify a Boolean equation as their wait-for condition, is difficult to visualize. Our work demonstrates that the AND⊕OR model is equivalent and supports intuitive visualization. Our transformation of the AND–OR model suits the limited AND–OR semantics of MPI but might lead to many additional nodes for other uses of AND–OR dependencies. Thus, its efficacy for general deadlock scenarios (i.e., beyond MPI) remains an open question.

3. MUST and the AND⊕OR model

Our Marmot Umpire Scalable Tool (MUST) extends and scales the functionality of Marmot [7] and Umpire [13]. End users require a deadlock detection that provides no false positives and enables a comprehensive understanding of the source of the deadlock. The solution must apply to all MPI applications with an acceptable overhead. Experience with ISP shows that exploration of all alternative interleavings can be impractical at scale. Thus, we restrict our approach to consider only deadlocks that manifest themselves in a given run. We base this runtime approach on the AND⊕OR model [4] that enables graph-based deadlock detection.

3.1. *Runtime deadlock detection with MUST*

The MUST library intercepts all MPI calls of all application processes at runtime. The tool then checks the correctness of the calls on the application processes or on additional tool nodes of a tree-based overlay network. The tree network allows scalable data aggregation and can run distributed or centralized correctness checks. We currently use a centralized *deadlock detector* that runs the graph-based deadlock detection on the root of the tree. Each application process forwards information about all communication calls to the deadlock detector.

The detector tracks the state of collective operations, as well as queues for outstanding point-to-point communications. Thus, MUST can determine if each MPI call can complete or must wait for another communication call, e.g., a matching point-to-point communication. We use this information to capture all wait-for dependencies between the processes. We represent these wait-for conditions as a graph and use a graph analysis to determine whether a deadlock exists at a certain execution step of the application.

3.2. *The AND⊕OR model*

MUST's detector evaluates each MPI operation and for each operation that can not complete it determines the MPI processes for which it waits. In the simplest case, an MPI process waits for exactly one other process, e.g., a blocked `MPI_Ssend` call that has no matching receive. More complex dependencies wait for multiple processes, e.g., a process issues an `MPI_Barrier` and waits for all processes in the communicator that have not yet issued a matching call. A different behavior holds for a process that issues an `MPI_Recv` with source equal to `MPI_ANY_SOURCE`. The process waits for all processes in the communicator until any *one* of them issues a matching send. In summary, MPI processes can wait for one, all, or any process in a subset of `MPI_COMM_WORLD`.

We use the terms *AND* semantics for processes that wait for all processes of a process-set and *OR* semantics for processes that wait for any process of a process-set. Deadlock criteria exist for both *AND* semantics (a cycle) and *OR* semantics (a knot, i.e., a set of nodes X for which X is the descendant set of each node in X). MPI usage can mix both semantic types; hence we need a more general model. The most general deadlock model, the AND–OR model, which allows arbitrary combinations of *AND* and *OR* semantics, is sufficiently general but more so than necessary. This model's generality complicates analysis and graphical visualizations. Thus, we use the AND⊕OR model, which limits each node in the graph to use exclusively outgoing *AND* or *OR* semantic arcs. Its wait-for graph (WFG) [4] uses the following definition:

Definition 1 (AND⊕OR WFG). A tuple $(V, E_{\mathrm{AND}}, E_{\mathrm{OR}})$ forms an AND⊕OR WFG if $(V, E_{\mathrm{AND}} \cup E_{\mathrm{OR}})$ is a directed graph and the following restriction holds:

$$\begin{aligned}&\{v \in V \mid \exists x \in V\colon (v,x) \in E_{\mathrm{AND}}\}\\ &\quad\cap \{v \in V \mid \exists x \in V\colon (v,x) \in E_{\mathrm{OR}}\} = \emptyset. \quad (1)\end{aligned}$$

A node in V represents a process. Two types of arcs model wait-for dependencies between the processes. Each node can have outgoing arcs of either the *AND* semantics (arcs in E_{AND}) or the *OR* semantics (arcs in E_{OR}). Figure 3(a) shows the WFG for the deadlock from Fig. 1(b). This graph only uses the *AND* semantic that we illustrate with solid arcs. Figure 3(b) shows the WFG for the deadlock from Fig. 1(d) for the interleaving that leads to deadlock. We assume that process 0 is blocked in the `MPI_Send` call, as MUST treats standard mode sends as unbuffered. Finally Fig. 3(c) shows a WFG with the OR semantics, for which we

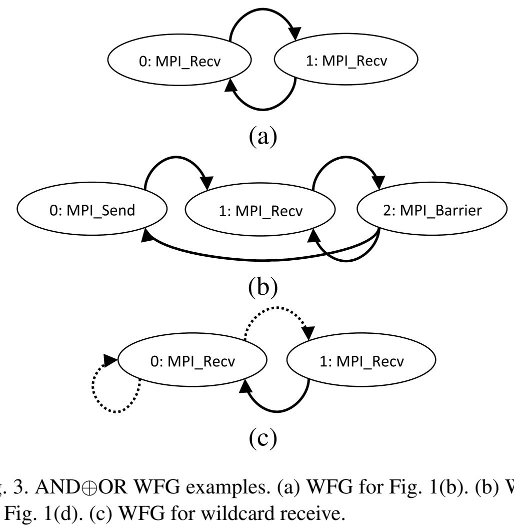


Fig. 3. AND⊕OR WFG examples. (a) WFG for Fig. 1(b). (b) WFG for Fig. 1(d). (c) WFG for wildcard receive.

use dashed arcs. The graph represents the wait-for conditions of a variation of example Fig. 1(b) in which process 0 uses a wildcard source.

The OR-knot [4], an AND$\oplus$OR WFG deadlock criterion, is a set of nodes X for which each node in X can reach all nodes in X and does not have an outgoing arc of the *OR* semantics that leads to a node not in X. MUST's graph search detects this criterion and supports visualization that distinguishes the root of a deadlock from processes that are waiting for it.

We provide new solutions for three fundamental limits in the use of the AND$\oplus$OR model for runtime deadlock detection:

- MPI constructs that are too general for the model;
- Wait-for analysis overhead; and
- Complexities in wildcard receive handling.

4. AND$\oplus$OR generalization

Some combinations of wait-for scenarios can lead to a blocked MPI process having outgoing dependencies of both *AND* and *OR* semantics. The example in Fig. 4 shows a case that uses `MPI_Waitall` and two user-defined communicators (sends intended to match the posted receives could occur later in the program). Communicator A consists of processes 0, 1, and 2 and communicator B consists of processes 0, 2, and 3. In the example, process 0 blocks until both nonblocking wildcard communication requests are completed. Thus, it waits for both requests to complete (*AND* semantics), which in turn wait for one process out of a set of processes (*OR* semantics). Thus, this example uses the *AND* and the *OR* semantics for a single process, which the AND$\oplus$OR model cannot handle. Some constructs in MPI can also lead to such cases, including `MPI_Sendrecv`, multithreaded MPI applications, and, in MPI-3, nonblocking collectives [6].

4.1. Transformation

The AND$\oplus$OR WFG lacks the generality to handle scenarios as in Fig. 4. We could use the AND–OR model for these cases but would lose our graphical deadlock criterion. Thus, we provide a novel transformation for wait-for dependencies of the AND–OR model that adds additional WFG nodes to translate them into the AND$\oplus$OR model. Intuitively, we add additional nodes that separate the wait-for conditions of nodes that use both *AND* and *OR* semantics such that each process (or node) only uses one of the two semantics in the resulting wait-for graph. We thus demonstrate that the AND$\oplus$OR model is equivalent to the AND–OR model while still providing the benefits of intuitive visualization for common MPI usage.

In general deadlock theory, wait-for dependencies link processes to resources. In MPI, processes wait for messages from processes so our definition of AND–OR wait-for dependencies formulates the dependencies as from a process to a process.

Definition 2 (AND–OR wait-for dependency). For a process set, V, and the associated set of valid wait-for dependencies, Λ, each dependency $w \in \Lambda$ equals one of the following:

- v with $v \in V$;
- $(w_1 \wedge w_2)$ with $w_1, w_2 \in \Lambda$;
- $(w_1 \vee w_2)$ with $w_1, w_2 \in \Lambda$.

Informally, Λ is the set of all Boolean equations with processes in V as atoms. Each process v can specify a wait-for dependency $R \in \Lambda$, which we denote (v, R). We do not require each process to have a dependency. We translate a set of general wait-for dependencies, $(v_1, R_1), (v_2, R_2), \ldots, (v_n, R_n)$, into the AND$\oplus$OR model as follows:

$$\text{translate}\big(\big((v_1, R_1), (v_2, R_2), \ldots, (v_n, R_n)\big)\big) = \bigcup_{i=1}^{n} t(v_i, R_i).$$

Figure 5 shows the function t that we use to translate the wait-for dependency of each process individually. We define the union of AND$\oplus$OR WFGs $(V^1, E^1_{\text{AND}}, E^1_{\text{OR}})$ and $(V^2, E^2_{\text{AND}}, E^2_{\text{OR}})$ as $(V^1 \cup V^2, E^1_{\text{AND}} \cup E^2_{\text{AND}}, E^1_{\text{OR}} \cup E^2_{\text{OR}})$. This union returns an

Process 0	Process 1	Process 2	Process 3
Irecv(comm:A, from:ANY, &reqs[0])	Recv(from:2)	Recv(from:0)	Recv(from:2)
Irecv(comm:B, from:ANY, &reqs[1])			
Waitall(2, reqs)			

Fig. 4. AND–OR semantics in MPI (Communicator A contains tasks 0, 1, and 2, communicator B contains tasks 0, 2, and 3).

$$t(v,R)=\begin{cases}\big(\{v,R\},\{(v,R)\},\emptyset\big), & \text{if } R\in V,\\ t(x,R_1)\cup t(y,R_2)\cup\big(\{v,x,y\},\{(v,x),(v,y)\},\emptyset\big), & \\ \qquad \text{if } R=(R_1\wedge R_2), x=\mathit{concat}(\alpha,v), y=\mathit{concat}(\beta,v), & \\ t(x,R_1)\cup t(y,R_2)\cup\big(\{v,x,y\},\emptyset,\{(v,x),(v,y)\}\big), & \\ \qquad \text{if } R=(R_1\vee R_2), x=\mathit{concat}(\alpha,v), y=\mathit{concat}(\beta,v). & \end{cases}$$

Fig. 5. Translation of AND–OR dependencies to AND⊕OR dependencies.

AND⊕OR WFG if and only if each node in $V^1 \cup V^2$ either only uses outgoing arcs of the *AND* type or of the *OR* type. The union operator combines the intermediate results that t returns into a final AND⊕OR WFG. The function t matches the Definition 2 of AND–OR wait-for dependencies and recursively translates the wait-for dependency of a node v:

$R \in V$**:** t adds the nodes v (the depending node) and R, to the set of nodes in the WFG and adds an *AND* semantic arc from v to R;

$R = (R_1 \wedge R_2)$**:** t introduces two additional nodes to the WFG, one prefixed with "α" and one prefixed with "β"; this case returns a WFG that results from the union of three WFGs: the WFGs obtained with recursive calls to $t(x, R_1)$, which translates the wait-for dependency R_1 applied to the first additional node, and to $t(y, R_2)$, which translates R_2 applied to the second additional node; and the WFG that contains the node v and the two additional nodes x and y along with *AND* arcs from v to x and y;

$R = (R_1 \vee R_2)$**:** Like the previous case, but the arcs from v to x and y have the *OR* semantic.

We assume that no node in V uses the "α" or "β" symbols. Thus, each step of t adds *AND* arcs or *OR* arcs. As t creates a valid AND⊕OR WFG in each step and each recursive call to t uses a different symbol for the first argument (the depending node), we conclude that t returns an AND⊕OR WFG.

4.2. Example

In the example that Fig. 4 shows, process 0 waits for both non-blocking receives to complete. The first receive waits for processes 0, 1, or 2 and the second waits for processes 0, 2 or 3. This corresponds to the AND–OR wait-for dependency: $R_0 = (((0 \vee 1) \vee 2) \wedge (2\vee(3\vee 0)))$. The remaining three processes are blocked in receive calls in which processes 1 and 3 wait for process 2, while process 2 waits for process 0.

Figure 6(a) shows the result of applying the *translate* function, which applies t to each wait-for condition, to this example. The first step of this computation for process 0 is:

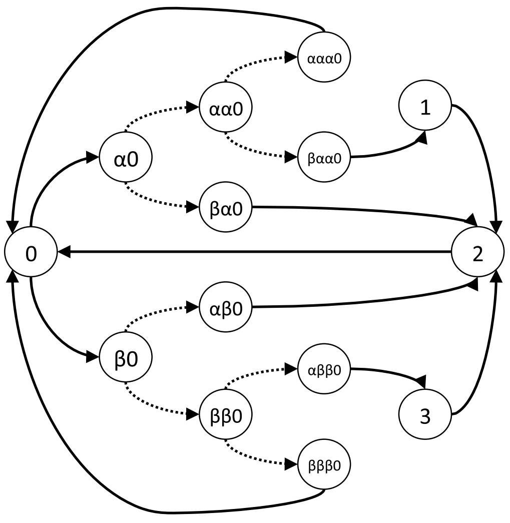
$$\begin{aligned} t(0,R_0) &= t\big(0,\big(\big((0\vee 1)\vee 2\big)\wedge\big(2\vee(3\vee 0)\big)\big)\big)\\ &= t\big(\alpha 0,\big((0\vee 1)\vee 2\big)\big)\\ &\quad\cup t\big(\beta 0,\big(2\vee(3\vee 0)\big)\big)\\ &\quad\cup\big(\{0,\alpha 0,\beta 0\},\{(0,\alpha 0),(0,\beta 0)\},\emptyset\big). \end{aligned}$$

The function introduces the additional nodes $\alpha 0$ and $\beta 0$. Afterwards, $\alpha 0$ represents the sub-wait-for condition $((0 \vee 1) \vee 2)$ of the wait-for condition of process 0 and $\beta 0$ represents $((2 \vee 3) \vee 0)$. Applying the recursive scheme of t repeatedly then leads to the WFG in Fig. 6(a).

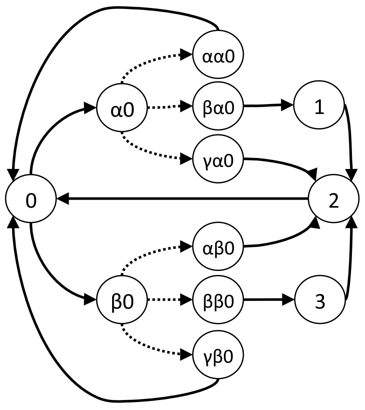
The WFG uses unnecessary intermediate nodes, e.g., $\alpha\alpha 0$ and $\beta\alpha 0$, which more elaborate transformation functions can avoid. Extensions of *translate* in MUST handle conjunctions and disjunctions of arbitrary order (Fig. 6(b)). Also, we remove intermediate nodes with exactly one incoming and one outgoing arc (Fig. 6(c)). Finally, our implementation in MUST replaces the additional node symbols of t with more meaningful labels. Thus, MUST provides the WFG in Fig. 7 for the example in Fig. 4. This more intuitive representation highlights that the `MPI_Waitall` call uses two requests, both resulting from calls to `MPI_Irecv`. We use a different node shape – a parallelogram – to indicate that the added nodes represent complex MPI call semantics instead of processes.

4.3. Deadlock criterion

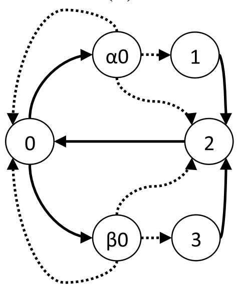
If *translate* returns a WFG with a deadlock, we can use the OR-knot [4] to represent it graphically. Fig. 7 shows an example: the nodes that are filled in gray form an OR-knot. Each node in this set can reach all other nodes in the set, while no node has an outgoing *OR* arc that leads to a node not in the set. Other OR-knots exist in this example, one includes all nodes and another contains processes 0, 2 and 3, and the `MPI_Irecv` node for "request[1]".



(a)



(b)



(c)

Fig. 6. AND⊕OR WFGs for the example of Fig. 4. (a) WFG that results from the *translate* function. (b) Result of *translate* extended to ternary (or higher order) Boolean operators. (c) Part (b) with unnecessary intermediate nodes removed.

MUST provides these highlighted wait-for graphs to its users. If users are not interested in the details of the AND⊕OR model, we provide them a simpler output: A list of processes that form a deadlock. This list includes our intermediate nodes, so we

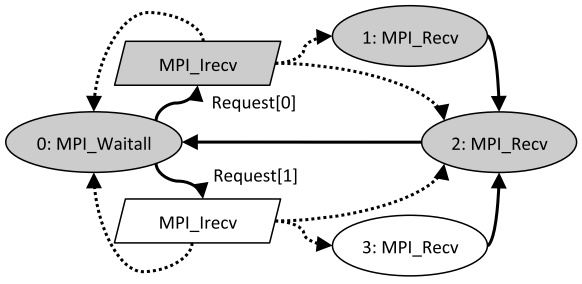


Fig. 7. MUST output for the example of Fig. 4.

not only point users to a certain process that issues an `MPI_Waitall` call, but also to the important request(s).

Deadlocks in the AND–OR model relate to deadlocks in the constructed AND⊕OR WFG. If *translate* returns an AND⊕OR WFG with an OR-knot, V_o, for each node $x \in V_o$ that t added, if x was added by processing a wait-for condition of a process v, then $v \in V_o$, since any path that leads to x from another process must pass through v. Further, no intermediate node can form a cycle or an OR-knot without using arcs of any node that represents a process. Thus, a process is deadlocked if the node that represents this process is part of an OR-knot. However, some of the additional nodes that are derived from a process v may not be part of an OR-knot, as Fig. 7 illustrates.

5. Optimized deadlock analysis

Our generalization of the AND⊕OR model allows us to model wait-for dependencies caused by any call in the current and upcoming MPI standard. However, this capability alone is insufficient. MUST also has to optimize the processing of MPI operations in order to reduce overhead and interference.

Runtime deadlock detectors intercept MPI calls and interpret them based on the MPI standard to determine which MPI calls are blocked at a certain point of execution. We need that information to compute the WFG, which we analyze to detect deadlocks. Current approaches run this analysis on a central process or thread [12,13]. For our long-term goal of deadlock detection that will scale to 10,000 or more processes, we must make the following processing steps scalable:

- Point-to-point matching;
- Collective matching;
- Wait-state analysis; and
- Graph-based deadlock detection.

The remainder of this section details optimizations of these steps that we implement in MUST.

5.1. *Runtime detection costs*

Graph-based deadlock detection has a complexity of $\mathcal{O}(p^2)$ for p processes [4] and, thus, is the most expensive processing step. However, MPI semantics allows a deadlock detector to analyze the MPI calls pessimistically. The detector does not need to analyze any MPI call of a process that is currently blocked in a preceding MPI call. Thus, even if a deadlock exists, the detector can process further MPI calls of other processes. As a result, we do not need to run the deadlock detection when we analyze an MPI event. We invoke the detection only if we suspect the presence of a deadlock:

- If the detector receives no additional MPI events within a configurable time period;
- If only some processes send MPI events; or
- When all processes report a call to `MPI_Finalize`.

Thus, we infrequently invoke the graph-based search, where a single search for about 10,000 processes completes within seconds. As a result, the major overhead for our runtime deadlock detection results from the other three processing steps rather than from the graph-based deadlock detection.

The detector must intercept and analyze more MPI calls at scale, which increases the overhead for message matching. These steps must eventually be distributed [5]. MUST still uses centralized components, but with an improved wait-state analysis. In Umpire [4], the detector always tracks the WFG. Calls such as wildcard receives or collective calls introduce $\mathcal{O}(p)$ arcs to the WFG for p processes ($(p-1)/2$ arcs on average for collective calls and p arcs on average for wildcard receives). Umpire's overhead to update the WFG for a single operation increases linearly with scale. Also, most applications increase their number of communication calls linearly with scale. Thus, the total overhead for runtime deadlock detection becomes $\mathcal{O}(p^2)$. Even with efficient distributed runtime deadlock detection, overhead would scale linearly, which would render a distributed approach impractical. Thus, we investigate the analysis time per MPI operation closely to provide a foundation for a distributed implementation.

5.2. *Delayed WFG construction*

MUST overcomes Umpire's limitation by constructing the WFG only on demand, i.e., if MUST invokes deadlock detection. Thus, with p processes, the detector analyzes the wait-for dependencies of up to p operations during a WFG construction. Each operation may require up to p arcs, thus, the WFG construction has a complexity of $\mathcal{O}(p^2)$, which matches the cost of the actual deadlock detection. For our goal of runtime deadlock detection with about 10,000 processes this infrequent overhead is acceptable. MUST's matching and wait-state analysis costs of different types of MPI operations are:

Send/Receive: $\mathcal{O}(1)$ if receives/sends use individual queues per communicator, send-receive rank pair, and tag;
Wildcard receive: $\mathcal{O}(p)$ to search all processes for a matching send call;
Single completion: $\mathcal{O}(\log k)$ to check the matching state of the nonblocking communication to which the request refers, which we find in $\mathcal{O}(\log k)$ for k requests;
Multi completion: $\mathcal{O}(n \log k)$ to check the matching state that is associated with each of the n requests in the completion;
Collective operation: $\mathcal{O}(1)$ to check whether all processes have issued the collective operation.

Analyzing send, receive (with specified source), and collective calls requires $\mathcal{O}(1)$, which allows our detector to handle them with low overhead. The detector can also analyze completion calls that use a single request efficiently with $\mathcal{O}(\log k)$. The challenging calls are wildcard receives, with $\mathcal{O}(p)$ complexity and multi-completions (i.e., `MPI_Waitall`) with $\mathcal{O}(n \log k)$ complexity. Multi-completions require n preceding calls to a send or receive initiator. Completions that complete all requests like `MPI_Waitall`, lead to an acceptable complexity of $\mathcal{O}(\log k)$ per operation on average (n times $\mathcal{O}(1)$ and once $\mathcal{O}(n \log k)$). The calls `MPI_Waitsome` and `MPI_Waitany` can impose higher cost as they may only complete one or a few requests. Section 7 presents an empirical study of these optimizations that shows sublinear growth in the analysis time per MPI operation for a wide variety of applications, which motivates future distributed implementations of runtime deadlock detection.

6. Wildcard receive handling

Wildcard receives complicate the correctness and efficiency of runtime deadlock detection. If the detector handles a wildcard call that has no known matching send call yet, it can treat the operation as blocked and uses the OR semantic to model wait-for dependencies. Handling wildcard receives for which multiple matching send calls are available is much more complex. MUST needs to adapt its matching decisions to the same decisions that the MPI implementation makes. Otherwise, the detector would not follow the same interleaving as the application run, which can lead to an erroneous analysis. Approaches such as ISP or DAMPI do not suffer from this property, as they rewrite nondeterministic MPI calls such that they enforce a known and controlled interleaving, which turns each execution of the application into a deterministic run.

To adapt to nondeterministic choices of the MPI implementation, we must monitor the field `MPI_Status.MPI_SOURCE` for wildcard receives. Blocking receives provide this information when they return, while the wait or test call that completes the receive provides it for nonblocking receives. Although the detector may determine that a nonblocking receive can match some send, the detector must still wait until the application completes the receive. The detector uses this information to choose the same match as the MPI implementation. Thus, the detector must queue all MPI operations that the application issues until the completion result arrives. Due to this queuing, existing tools, such as Umpire [13], can exhibit undesirable behavior in the following three scenarios:

- A deadlock occurs before or while a completion executes;
- The application never executes the completion; or
- The detector requires more memory than available to queue MPI operations.

These scenarios lead to unresponsiveness or incomplete analyses, which can make the tool incapable of detecting some deadlocks. MUST overcomes these limitations with advanced wildcard handling. We still pause the analysis of MPI operations when we wait for the completion of a nonblocking wildcard receive but we add two new analysis modes: *probing* and *deciding*. These modes handle the cases that could lead to tool unresponsiveness or incomplete analyses.

6.1. Probing in MUST

When a timeout invokes deadlock detection, we use *probing* if a wildcard receive with at least one known match has no available completion information. Figure 8 shows a situation that resembles the scenario in Fig. 1(d). If the first call to `MPI_Irecv` from process 1 matches the send of process 0, the application will complete. Otherwise, this example deadlocks with processes 0 and 2 stuck in the `MPI_Barrier` call (assuming that the MPI implementation buffers the `MPI_Send` call of process 0) and process 1 in the `MPI_Waitall` call. Since the completion call (in this case `MPI_Waitall`) hangs, the tool cannot obtain the matching decision from the MPI implementation. Thus, we must detect the deadlock without it.

Process 0	Process 1	Process 2
Send(to:1)	Irecv(from:ANY, &reqs[0])	Send(to:1)
	Irecv(from:2, &reqs[1])	
	Waitall(2, reqs)	
Barrier()	Barrier()	Barrier()

Fig. 8. Schedule–dependent MPI deadlock with an unavailable wildcard completion source.

The *probing* mode matches each wildcard receive with all matching send calls, in order to determine if a deadlock exists. If a specific matching decision leads to deadlock, then we report the error and abort any further analysis. Otherwise, we repeat our *probing* process until we have tested all possible matches. If no matching decision leads to deadlock then the detector waits for additional MPI operations.

Figure 8 shows an example that requires this approach. Process 2 encounters a wild card receive operation. MUST then tests for matches with the send in process 0, which does not lead to a deadlock, followed by a test with the send in process 2, which reveals the deadlock.

Several synthetic tests of Umpire and MUST require *probing*. Some stress tests require MUST to investigate several hundred interleavings in order to detect a deadlock. In theory, the number of interleavings can be exponential. However, we are not aware of any cases in which only one (or few) of the exponential number of interleavings leads to a deadlock.

We suspend the search of possible matches after a configurable time period and restart the search with an increased time period if no additional MPI operations arrive. If the search space exceeds the space that MUST can cover in a given amount of time, we might not be able to report a deadlock (false negative). MUST notifies users when probing starts and allows them to complete the wildcard receives in question at an early time, in order to avoid this rare situation altogether.

6.2. Deciding in MUST

When MUST waits for a wildcard source, the detector queues all other MPI operations until the source

Process 0	Process 1	Process 2
Irecv(from:ANY, &req)	Isend(to:0, &req)	Isend(to:0, &req)
//long communication	*//long communication*	*//long communication*
Wait(req)	Wait(req)	Wait(req)

Fig. 9. Late wildcard receive completion.

arrives. If these queues would exceed the available memory, we use the *deciding* mode. Figure 9 shows a scenario in which a nonblocking wildcard receive of process 0 has multiple available matches, while a completion only occurs after a long phase of additional communication. When *deciding*, MUST first performs a *probing* step to determine if a deadlock exists. If so, we report the error and abort. Otherwise, we enforce some matching decision to allow the execution to continue. In the example, MUST could decide to match the receive of process 0 with the send of process 1. Deciding can cause MUST to report false positives, so we note this behavior in our correctness log and allow users to issue a completion at an earlier time or to add a missing completion. Applications for which MUST can report false positives are rare, as their communication pattern must be dependent on wildcard matching decisions. Approaches such as DAMPI do not support such applications.

7. Application results

We use the NAS Parallel Benchmarks (NPB) [1] (v3.3) and SPEC MPI2007 [11] (v2.0) to evaluate MUST's runtime deadlock detection improvements. We use NPB problem size *D* and the *mref* size for SPEC MPI2007. We run these benchmarks on a Linux-based cluster with 1944 nodes of two 6 core Xeon 5660 processors. Each node has 24 GB of main memory and uses a QDR InfiniBand interconnect. We use a range of 16 to 512 cores to measure the behavior of MUST at increased scale and to validate the improvements of our optimized detector implementation. The NPB kernels *bt* and *sp* require square numbers of processes so we use 36 instead of 32, 121 instead of 128, and 529 instead of 512 processes for them. For simplicity, we do not highlight this difference in our graphs. Further, *126.lammps* contains a potential send–send deadlock that MUST detects. In this case we measure MUST's overheads and analysis time per MPI operation for all operations that MUST analyzes to detect the deadlock.

MUST currently offers three operation modes of communication between the MPI processes and the master. The first uses synchronous communication, the second uses immediate asynchronous communication, and the third uses aggregated asynchronous communication. The synchronous mode only issues an MPI call on the application processes after the centralized detector analyzes the respective event, which reflects Marmot's operation mode. The second mode immediately starts a non-blocking communication to notify the detector of a new communication event, whereas the third mode aggregates multiple events into a larger contiguous buffer for higher bandwidth communication. Umpire uses the last mode in combination with errorhandlers, signal handlers, and *atexit* handlers to guarantee that the centralized detector can process all important events even if an application process crashes or hangs. MUST also implements this communication mechanism, but our prototype crash handling requires additional investigation to guarantee that we catch all bugs on a wide range of platforms. We use the aggregated communication strategy for our measurements and provide the other communication strategies for cases where our crash handling might fail.

Figure 10(a) and (c) shows application slowdowns with MUST; a slowdown of 2 corresponds to a 100% increase in application run time. MUST overhead includes the run time for all correctness checks that we provide and run times that MUST's communication system consumes to forward MPI call information to the central deadlock detector. We run local correctness checks such as detection of invalid arguments or MPI resource leaks locally. An additional MPI process executes the central detector that checks type matching, verifies collective operations, and runs our deadlock detection.

We can analyze all kernels except *107.leslie3d* and *121.pop2* at 512 processes (Memory and runtime limitations). MUST can handle most kernels with a slowdown of about 2 or lower at 128 processes. The particularly challenging benchmark *121.pop2* invokes a large number of point-to-point calls (about 50,000 per process per second already at 32 processes).

The scalability limit of NPB problem size *D* and of the *mref* input set for SPEC MPI2007 is about 512 processes, which increases MUST's overhead due to a high communication to computation ratio. Figure 12 shows MUST's overhead for selected NPB kernels at problem size *E* and for the *lref* input set for SPEC MPI2007. The chart shows that MUST can handle applications at 1024 processes.

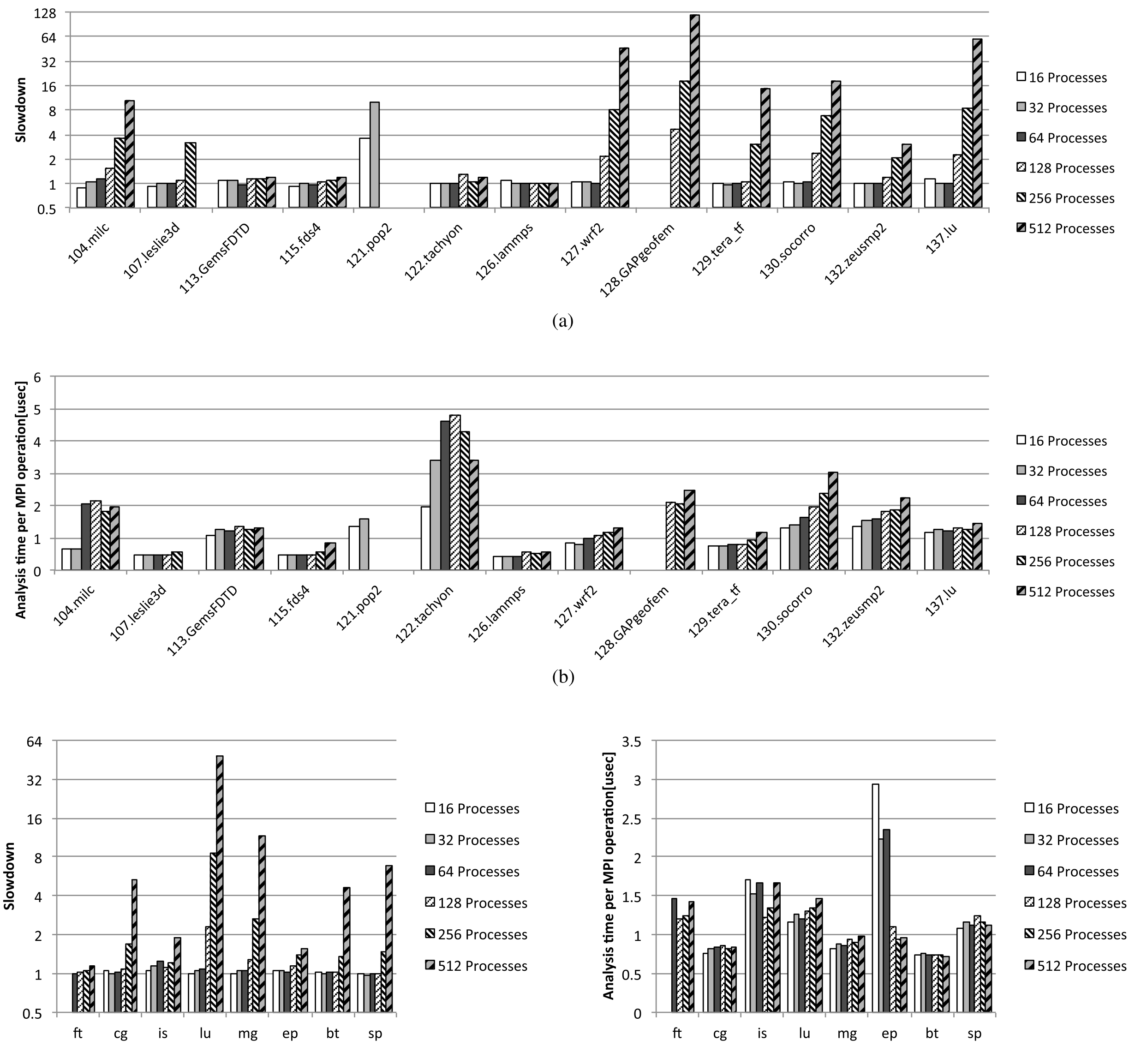


Fig. 10. MUST overheads and associated analysis time per MPI operation. (a) Slowdown for SPEC MPI2007. (b) Analysis time per MPI operation for SPEC MPI2007. (c) Slowdown for NPB. (d) Analysis time per MPI operation for NPB.

Figure 10(b) and (d) show the analysis time per MPI operation of our deadlock detector. This metric indicates that distributed runtime deadlock detection is feasible. If this time increases linearly with scale, distributed detection becomes impractical. Scaling behavior, e.g., a change in the ratio between collective and point-to-point calls, influences this metric. Buffering of operations based on the order in which they are processed can also impact it. Thus, our measurements exhibit some variability. Of the 21 benchmarks, *127.wrf2*, *132.zeusmp2*, *lu*, *137.lu*, *130.socorro* and *129.tera_tf* show repeated increases in the analysis time per MPI operation across scale, which Fig. 11 details. The charts indicate a sublinear increase for the benchmarks *127.wrf2*, *132.zeusmp2*, *lu*, and *137.lu*, for which we use logarithmic fits in Fig. 11. The benchmarks *130.socorro* and *129.tera_tf* show a more linear increase for this metric. The benchmark *130.soccorro* completes arrays of MPI requests with repeated calls to `MPI_Waitany`, which causes MUST's increased analysis time for this benchmark. For *129.tera_tf*, the amount of point-to-point communication calls de-

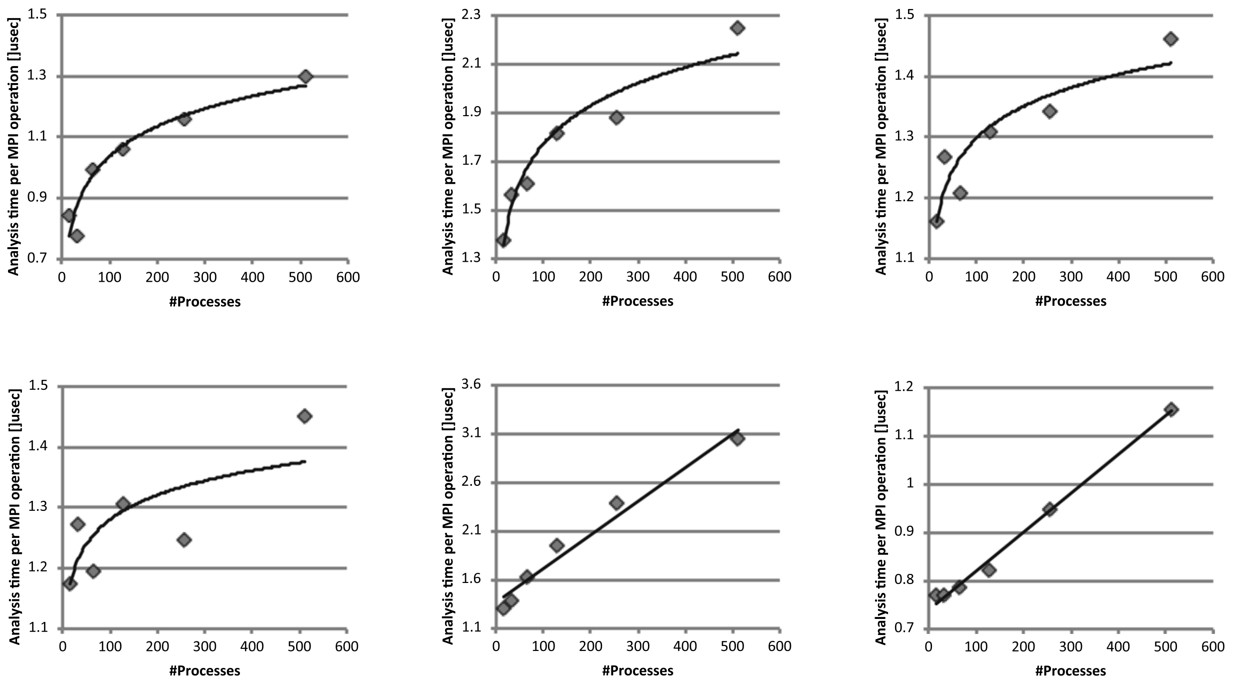


Fig. 11. Detailed analysis time per MPI operation plots. (a) MPI2007: *127.wrf2*. (b) MPI2007: *132.zeusmp2*. (c) NPB: *lu*. (d) MPI2007: *137.lu*. (e) MPI2007: *130.socorro*. (f) MPI2007: *129.tera_tf*.

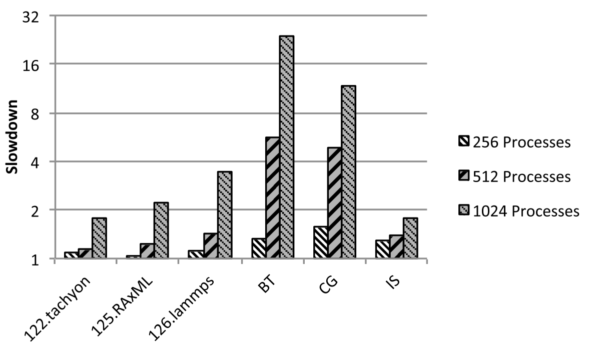


Fig. 12. MUST overheads with larger data sets.

creases while the amount of collective calls per process stays constant. MUST currently incurs higher analysis time for collective calls than for point-to-point calls, due to type matching.

8. Conclusions

We present MUST, a novel runtime error detection tool for MPI applications. Key features include type matching, collective verification, and deadlock detection. We contribute theoretic and processing extensions for the AND$\oplus$OR model based deadlock detection. With p processes, the existing approach based on this model required an analysis time of $\mathcal{O}(p)$ for each blocking MPI operation, which makes deadlock detection prohibitively expensive at scale. We overcome this limitation and achieve sublinear analysis time for a wide variety of applications. We demonstrate this result for two major benchmark suites for up to 1024 processes. Additionally, the generalization of our deadlock model allows us to handle complex wait-for semantics that arise with certain existing MPI constructs and will become more common with future MPI extensions such as nonblocking collectives.

Although our current approach can scale to at least 1024 processes for many applications and inputs, future use cases will need additional advances. Deadlock detection becomes more challenging as systems scale even further and some errors may only occur at higher scales. Our work provides a basis for scalable distributed MPI runtime deadlock detection, but we will need additional scalable techniques for point-to-point matching, collective matching, and wait-state analysis.

Acknowledgements

Part of this work was performed under the auspices of the US Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-555978). This work has been supported by the CRESTA project that has received funding from the European Community's Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703.

References

[1] D.H. Bailey, L. Dagum, E. Barszcz and H.D. Simon, NAS parallel benchmark results, IEEE Parallel and Distributed Technology, Technical Report, 1992.

[2] V.C. Barbosa and M.R.F. Benevides, A graph-theoretic characterization of AND–OR deadlocks, 1998.

[3] C. Falzone, A. Chan and E. Lusk, Collective error detection for MPI collective operations, in: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*, Springer, 2005, pp. 138–147.

[4] T. Hilbrich, B.R. de Supinski, M. Schulz and M.S. Müller, A graph based approach for MPI deadlock detection, in: *ICS'09: Proceedings of the 23rd International Conference on Supercomputing*, ACM, New York, NY, USA, 2009, pp. 296–305.

[5] T. Hilbrich, M.S. Müller, B.R. de Supinski, M. Schulz and W.E. Nagel, GTI: A generic tools infrastructure for event based tools in parallel systems, in: *IPDPS 2012: Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium*, 2012.

[6] T. Hoefler, A. Lumsdaine and W. Rehm, Implementation and performance analysis of non-blocking collective operations for MPI, in: *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*, IEEE Computer Society/ACM, November 2007.

[7] B. Krammer and M.S. Müller, MPI application development with MARMOT, in: *PARCO*, Vol. 33, Central Institute for Applied Mathematics, Jülich, Germany, 2005, pp. 893–900.

[8] S. Lee, Fast, centralized detection and resolution of distributed deadlocks in the generalized model, *IEEE Trans. Softw. Eng.* **30**(9) (2004), 561–573.

[9] G.R. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva and Y. Zou, MPI-CHECK: A tool for checking Fortran 90 MPI programs, *Concurrency and Computation: Practice and Experience* **15**(2) (2003), 93–100.

[10] Message Passing Interface Forum, MPI: A message-passing interface standard, Version 2.2, April 2009, available at: http://www.mpi-forum.org/docs/mpi22-report.pdf.

[11] M.S. Müller, M. van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W.C. Brantley, C. Parrott, T. Elken, H. Feng and C. Ponder, SPEC MPI2007 – an application benchmark suite for parallel systems using MPI, *Concurrency and Computation: Practice and Experience* **22**(2) (2010), 191–205.

[12] S.S. Vakkalanka, S. Sharma, G. Gopalakrishnan and R.M. Kirby, ISP: A tool for model checking MPI programs, in: *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 285–286.

[13] J.S. Vetter and B.R. de Supinski, Dynamic software testing of MPI applications with umpire, *Supercomputing, ACM/IEEE 2000 Conference*, 4–10 November 2000, pp. 51–51.

[14] A. Vo, Scalable formal dynamic verification of MPI programs through distributed causality tracking, PhD dissertation, University of Utah, School of Computing, March 2011.

[15] H.J. Yoon and D.Y. Lee, Deadlock-free scheduling of photolithography equipment in semiconductor fabrication, *IEEE Transactions on Semiconductor Manufacturing* **17**(1) (2004), 42–54.