Distributed Wait State Tracking for Runtime MPI Deadlock Detection

Tobias Hilbrich
Technische Universität Dresden
D-01062 Dresden, Germany
tobias.hilbrich@tu-dresden.de

Bronis R. de Supinski
Lawrence Livermore National Laboratory
Livermore, CA 94551
bronis@llnl.gov

Wolfgang E. Nagel
Technische Universität Dresden
D-01062 Dresden, Germany
wolfgang.nagel@tu-dresden.de

Joachim Protze
RWTH Aachen University
D-52056 Aachen, Germany
JARA – High-Performance Computing
D-52062 Aachen, Germany
protze@rz.rwth-aachen.de

Christel Baier
Technische Universität Dresden
D-01062 Dresden, Germany
baier@tcs.inf.tu-dresden.de

Matthias S. Müller
RWTH Aachen University
D-52056 Aachen, Germany
JARA – High-Performance Computing
D-52062 Aachen, Germany
mueller@rz.rwth-aachen.de

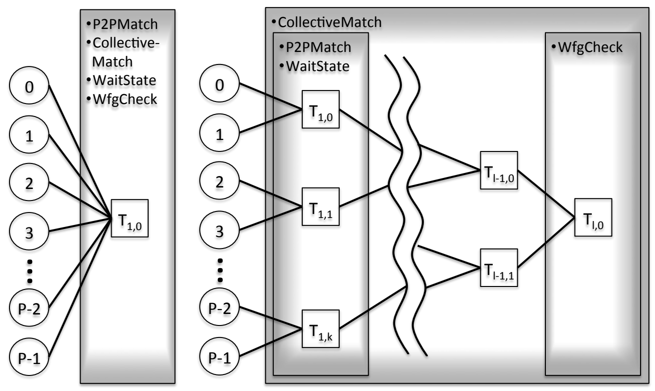
ABSTRACT

The widely used Message Passing Interface (MPI) with its multitude of communication functions is prone to usage errors. Runtime error detection tools aid in the removal of these errors. We develop MUST as one such tool that provides a wide variety of automatic correctness checks. Its correctness checks can be run in a distributed mode, except for its deadlock detection. This limitation applies to a wide range of tools that either use centralized detection algorithms or a timeout approach. In order to provide scalable and distributed deadlock detection with detailed insight into deadlock situations, we propose a model for MPI blocking conditions that we use to formulate a distributed algorithm. This algorithm implements scalable MPI deadlock detection in MUST. Stress tests at up to 4,096 processes demonstrate the scalability of our approach. Finally, overhead results for a complex benchmark suite demonstrate an average runtime increase of 34% at 2,048 processes.

1. INTRODUCTION

The Message Passing Interface (MPI) [20] is a de-facto standard for distributed memory programming. While the richness of the standard enables efficient and scalable programming across most platforms, it also allows for complex usage errors that are hard to identify and to resolve. Such errors may manifest as incorrect results, an application crash, as deadlock, or may silently be tolerated by the underlying MPI implementation.

SC13 November 17-21, 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503237



(a) Centralized. (b) Distributed.

Figure 1: Runtime MPI deadlock detection tool architectures.

Various tools to detect MPI usage errors at runtime exist, e.g., ISP [28], Marmot [16], Umpire [29], MPI-Check [17], and MUST [14]. While these tools differ in the error set that they detect, for deadlock detection, they either use an imprecise timeout mechanism or a precise but centralized algorithm that limits scalability. The former may provide false positives and does not analyze the problem in detail. Precise algorithms provide detailed insight, which helps application developers in the removal of these errors.

We extend runtime deadlock detection approaches for MPI with a new distributed analysis of MPI blocking semantics, which significantly increases scalability. Figure 1(a) sketches the architecture of centralized runtime deadlock detection tools such as MUST, where we represent application processes with circle shape nodes and the single tool process with a square shape node. MUST uses point-to-point and collective matching to determine which MPI calls match. This information forms the input for a *wait state* analysis

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

(a) Recv-recv deadlock.

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

(b) Send-send deadlock and wildcards.

Figure 2: MPI deadlock examples.

that simulates MPI blocking semantics. We detail and formalize this analysis in Section 3. In summary, we combine the following analyses for deadlock detection:

- Point-to-point matching (*P2PMatch* in Figure 1),
- Collective matching (*CollectiveMatch* in Figure 1),
- *Wait state* analysis (*WaitState* in Figure 1), and
- Graph-based deadlock detection (*WfgCheck* in Figure 1).

Recent extensions of MUST advance our original centralized implementations for point-to-point and collective matching towards distributed and scalable implementations [10, 13]. Thus, our wait state analysis is now the scalability bottleneck for our runtime deadlock detection approach. We propose and implement a distributed wait state analysis for scalable MPI runtime deadlock detection; Figure 1(b) sketches our new tool architecture. As a basis for our distributed tool we use a Tree-Based Overlay Network (TBON) as in tools like Periscope [6], TAUoverMRNet [22], LaunchMON [1], and STAT [2]. We use the complete tree to match MPI collective calls, while we use the first tool layer of the tree to match point-to-point calls. In Section 4, we detail how we use the TBON to implement a distributed wait state analysis. While we still use a centralized graph analysis, we conclude that its overhead is non-critical for application runs at less than 10,000 processes. Our contributions include:

- A transition system that formalizes wait state analysis;
- A distributed algorithm to analyze this system;
- An integration of our distributed wait state analysis with a synchronization algorithm and a centralized graph-based deadlock detection to derive a tool for scalable MPI deadlock detection; and
- Experimental results with synthetic stress tests at up to 4,096 processes and with SPEC MPI2007 at up to 2,048 processes.

We present related work in Section 2 and a formalization of wait state analysis in Section 3. We use this formalization to derive a distributed wait state algorithm in Section 4. Section 5 then describes how we execute deadlock detection. Finally, we present our application study in Section 6.

2. COMPARISON TO PRIOR WORK

Our approach relates to MPI runtime deadlock detection tools such as Umpire [29], MUST [14], MPIDD [8], MPI-Check [18], ISP [28], TAC [23], and DAMPI [30]. Umpire, MPIDD, and ISP use centralized approaches that limit scalability. MPI-Check uses a distributed handshake protocol that does not provide dependency analysis for detailed error reports. More importantly, it limits MPI usage, e.g., it does not support wildcard receives (i.e., `MPI_ANY_SOURCE`). DAMPI provides a distributed analysis that discovers alternative interleavings of an MPI application. However, it does not perform precise deadlock detection and instead relies on timeouts to conclude that deadlocks exist.

We only provide an overhead comparison to MUST's centralized implementation since for the two approaches Umpire and MPIDD, which perform a similar precise analysis of a single program execution, our approach already supersedes Umpire and, as best as we can tell, MPIDD is no longer maintained. The remaining approaches differ in their analysis type, where Marmot, DAMPI, and TAC use an imprecise timeout-based detection; and where ISP explores different program executions for its deadlock analysis.

Our model for MPI blocking semantics closely relates to approaches that use transition systems to model MPI applications for formal analysis [26, 27]. However, we follow a single recorded execution rather than all potential interleavings. We use return values of MPI calls to observe the interleaving that occurs at runtime. This approach ensures that we do not report false positives or need to impose assumptions on MPI usage. Our goal is scalable deadlock detection that allows developers to detect actual deadlocks at their target scale. Our previous centralized runtime deadlock detection approaches [9, 14] analyze the transition system that we propose in Section 3, but did not formalize it. This contribution serves to strengthen the theoretical foundation of our approach and to minimize the need for data distribution, which simplifies the specification of the distributed algorithm that extends our previous centralized algorithm.

We use logical timestamps of MPI events to create a consistent global state of an MPI application. This technique is closely related to approaches that capture a global state of a distributed system [4]. In particular, we represent states as logical timestamp vectors similar to vector clocks [5, 19]. We implement our distributed algorithm in a TBON through the GTI tool infrastructure [11]. Other infrastructures such as MRNet [25], SCI [15], and STCI [3] offer similar features. However, we rely on two extensions in GTI: order preserving event aggregation [12] and intralayer communication [13].

3. MPI WAIT STATE ANALYSIS

Figure 2 presents deadlock scenarios. We use *Recv* for `MPI_Recv`, *Send* for `MPI_Send`, and *Barrier* for `MPI_Barrier`. We only specify the source/target for send and receive calls and omit all other arguments. The example in Figure 2(a) presents a deadlock in which the receive call of process 0 blocks indefinitely since it waits for a send from process 1, which indefinitely waits for a send from process 0. This

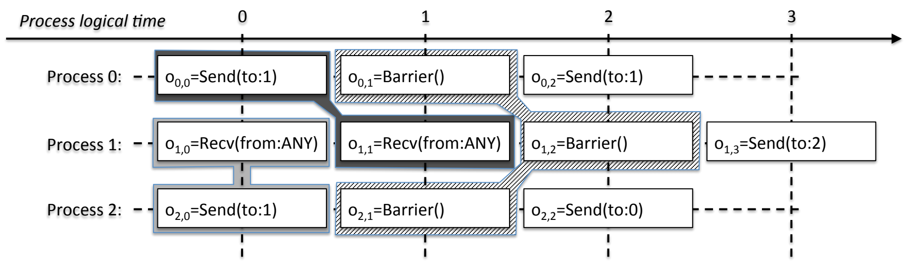


Figure 3: Illustration of a matched trace for the example in Figure 2(b).

deadlock will always manifest with any MPI implementation. Figure 2(b) presents a potential deadlock that manifests if send calls are not buffered. In the example, process 1 issues two *wildcard* receives that can receive a message from any process. Both of these wildcard receives can complete since processes 0 and 2 each send a message to process 1. Afterwards, all three processes enter the barrier and complete it. Finally, all processes issue a send call. Since no process issues a receive call, they will wait indefinitely. Note that the MPI standard allows implementations to buffer standard-mode send calls, in which case the deadlock is not manifested. However, the example remains unsafe.

We use graph-based deadlock detection for MPI applications [9]. Its required input includes information about currently active MPI calls of all processes and the current point-to-point and collective matching situation. From this information, we compute wait-for dependencies between the processes. To determine the current correct state of MPI, we simulate the blocking semantics of MPI calls, point-to-point matching, and collective matching. We intercept all MPI calls, match them, and then analyze their blocking semantics in a *wait state* analysis. This analysis considers MPI calls of each process in their execution order and models the conditions under which each call waits. Based on the MPI calls that could have already been processed, i.e., became *active*, we determine whether a call's wait conditions can be met. If so, we consider the next operation for this MPI process. Each state in the analysis provides all inputs that graph-based deadlock detection requires.

Where Umpire used an implicit graph search whenever it analysed a new MPI operation, wait state analysis can avoid repeated graph searches. While previous implementations of our tools [9, 14] use this property to execute graph-based deadlock detection only after a timeout, we have not formalized this property. This section introduces a transition system for wait state analysis based on which we formally demonstrate the soundness of this approach and derive and illustrate a distributed algorithm.

3.1 Transition System

The input of our wait state analysis is a matched trace that is derived from distributed point-to-point [13] and collective [10] matching. In the following, we refer to any MPI call as an *operation*. We formalize wait state analysis as a transition system, for which we first define the structure of the trace and a function that provides information on whether an operation is blocking.

Let P be a finite set of processes where we represent each process with its finite sequence of MPI operations. Say $P = \{0, 1, 2, \dots, p-1\}$ and let $t(i) = o_{i,0}, o_{i,1}, \dots, o_{i,m_i}$ denote the sequence of MPI operations of process $i \in P$. If no deadlock occurs, the last operation o_{i,m_i} is `MPI_Finalize` ($\forall i \in P$). We refer to the operations in the sequences $t(i)$ by means of pairs (i, j) where i is the process identifier (i.e., $i \in P$) and j the local and logical timestamp (i.e., $j \in \{0, \dots, m_i\}$). Let

$$Op \overset{def}{=} \{(i, j) | i \in P, j \in \{0, \dots, m_i\}\}$$

be the set of all MPI operations. In Figure 3 we illustrate the traces of the three processes from the example in Figure 2(b). We assume that the deadlock is manifest, i.e., the trace ends with the send calls that follow the barrier. We connect matching operations with shading, where we match the first receive of process 1 with the send of process 2, one of the possible executions of this example.

We must consider whether an operation is blocking, i.e., waits until some condition is met, or non-blocking, i.e., will always return after some finite time. We denote that with the Boolean-valued function $b : Op \to \{\perp, \top\}$, where $\top$ stands for "true" and $\perp$ for false. We define b as:

$$b(i,j) \overset{def}{=} \begin{cases} \top & : \begin{array}{l} o_{i,j} \text{ is an } \texttt{MPI\_Send}, \texttt{MPI\_Ssend}, \texttt{MPI\_Recv}, \\ \texttt{MPI\_Probe}, \text{a collective, or} \\ \texttt{MPI\_Wait[any,some,all]}, \end{array} \\ \\ \perp & : \begin{array}{l} o_{i,j} \text{ is an } \texttt{MPI\_Iprobe}, \texttt{MPI\_I[s,r,b]send}, \\ \texttt{MPI\_\{B,R\}send}, \texttt{MPI\_Test[any,some,all]}, \\ \text{or } \texttt{MPI\_Irecv} \end{array} \end{cases}$$

The definition of b includes no persistent communication operations since we can handle them like non-blocking point-to-point operations. For `MPI_Sendrecv` we use a series of calls such as the MPI standard suggests[1]. Also, MPI allows implementations to decide whether some operations, e.g., `MPI_Send`, use internal buffering. As a result, these operations may not be blocking. We consider these freedoms at the end of this section. The trace in Figure 3 only uses blocking operations according to our definition of b.

Based on the definitions of the traces t_i ($i \in P$) and the blocking property b we define wait state analysis as a transition system $\mathcal{T} = (\mathit{States}, \to_{\text{ws}}, L_0)$, where *States* constitutes the state space of p-tuples $(l_0, \dots, l_{p-1})$ where $l_i \in \{0, 1, \dots, m_i\}$ for $i \in P$. Each state represents the logical timestamps of the currently active MPI operations at an execution step of the application. We use the initial state $L_0 = (0, \dots, 0)$ and define the transition relation $\to_{\text{ws}} \subseteq \mathit{States} \times \mathit{States}$ as the smallest binary relation

[1]We treat `MPI_Sendrecv` as a single call in deadlock reports

on *States* that satisfies the following rules[2]:

(1) $o_{i,j}$ is a non-blocking operation:

$$\frac{b(i,j) = \perp \wedge l_i = j}{(l_0, \ldots, l_i, \ldots, l_{p-1}) \overset{\text{nb}}{\rightarrow}_{\text{ws}} (l_0, \ldots, l_i + 1, \ldots, l_{p-1})}$$

(2) $o_{i,j}$ is a send/receive/probe operation where $o_{k,n}$ is the matching receive/send/send operation:

$$\frac{l_i = j \wedge l_k \geq n}{(l_0, \ldots, l_i, \ldots, l_{p-1}) \overset{\text{p2p}}{\rightarrow}_{\text{ws}} (l_0, \ldots, l_i + 1, \ldots, l_{p-1})}$$

(3) $C = \{(i_0, j_0), (i_1, j_1), \ldots), (i_n, j_n)\}$ is a complete set of matching collective operations (i.e., each process in the process set associated with the collective is present):

$$\frac{(i,j) \in C \wedge l_i = j \wedge \forall (k,n) \in C : l_k \geq n}{(l_0, \ldots, l_i, \ldots, l_{p-1}) \overset{\text{coll}}{\rightarrow}_{\text{ws}} (l_0, \ldots, l_i + 1, \ldots, l_{p-1})}$$

(4) $o_{i,j}$ is a completion operation that uses MPI requests that are associated with non-blocking send/receive operations $o_{i,j_0}, o_{i,j_1} \ldots, o_{i,j_x}$ $(\forall k \in \{0, \ldots, x\} : j_k < j)$:

(I) $o_{i,j}$ is an `MPI_Waitany` or `MPI_Waitsome` and the send/receive operation o_{i,j_y} for some $y \in \{0, \ldots, x\}$ is matched with the receive/send operation $o_{k,n}$:

$$\frac{l_i = j \wedge l_k \geq n}{(l_0, \ldots, l_i, \ldots, l_{p-1}) \overset{\text{any}}{\rightarrow}_{\text{ws}} (l_0, \ldots, l_i + 1, \ldots, l_{p-1})}$$

(II) $o_{i,j}$ is an `MPI_Wait` or `MPI_Waitall` operation and each send/receive o_{i,j_y} $(y \in \{0, \ldots, x\})$ is matched with the receive/send operation o_{k_y,n_y}:

$$\frac{l_i = j \wedge \forall y \in \{0, \ldots, x\} : l_{k_y} \geq n_y}{(l_0, \ldots, l_i, \ldots, l_{p-1}) \overset{\text{all}}{\rightarrow}_{\text{ws}} (l_0, \ldots, l_i + 1, \ldots, l_{p-1})}$$

Rule (1) reflects that non-blocking operations always return after a finite amount of time, whereas the other rules define when a process is allowed to return from a blocking operation. For a process with an active point-to-point operation—Rule (2)—the process may advance to the next operation if the matching operation is also active. This rule also includes the use of `MPI_Probe`, which only differs from a blocking receive call in its consequences for point-to-point matching, since it does not receives a message. Similarly, for processes that are active in a collective operation—Rule (3)—a process may advance to the next operation if all processes that belong to the collectives process group activated their participating operation. Note that if a deadlock manifests, a matching operation may not exist, e.g., for $o_{0,2}$ in Figure 3. We consider the presence of a matching operation as a premise for Rules (2)-(4).

The rules do not enforce an order in which processes activate the next operation, which represents the semantics of MPI and allows us to require less synchronization when we derive a distributed algorithm in Section 4. In particular Rule (2) allows receiver processes in point-to-point communications to advance to the next operation while the sender is still active in the send. In such a case the sender would have communicated the complete message buffer to the receiver, while the sender still handles local computations, e.g., frees a temporary buffer. Rule (4) handles blocking completion operations, i.e., `MPI_Wait[any,some,all]`. For *any*/*some* completions it requires that a matching and active operation exists for at least one associated non-blocking communication. The two calls only differ in that MPI may complete multiple communications in `MPI_Waitsome`, which influences the trace of operations that we record, but not the transition system rules. For *all* completions, a matching and active operation must exist for all associated non-blocking operations.

The executions of the transition system $\mathcal{T}$ are generated by starting in the initial state L_0 and then repeatedly moving to a successor state according to the transition relation $\rightarrow_{\text{ws}}$ until no further rule can be applied. For the example in Figure 3 a possible execution of the wait state transition system is: $(0,0,0) \overset{\text{p2p}}{\rightarrow}_{\text{ws}} (0,0,1) \overset{\text{p2p}}{\rightarrow}_{\text{ws}} (0,1,1) \overset{\text{p2p}}{\rightarrow}_{\text{ws}} (0,2,1) \overset{\text{p2p}}{\rightarrow}_{\text{ws}} (1,2,1) \overset{\text{coll}}{\rightarrow}_{\text{ws}} (1,2,2) \overset{\text{coll}}{\rightarrow}_{\text{ws}} (2,2,2) \overset{\text{coll}}{\rightarrow}_{\text{ws}} (2,3,2)$. To illustrate the preconditions of the rules, consider the state $(0,0,1)$: In this state, Rule (2) cannot again be applied to $o_{2,0}$, since $l_2 \neq 0$, i.e., this operation is not the current operation of process 2. Also Rule (2) is not applicable to $o_{0,0}$, even though it is the active operation of process 0, since the matching operation $o_{1,1}$ is not active in this state (the state does not satisfy the second part of the precondition of Rule (2)). As a last example, Rule (3) is not applicable to $o_{2,1}$. While this operation is active on process 2, both matching operations $o_{0,1}$ and $o_{1,2}$ are not active in this state, i.e., the state does not satisfies the second part of the precondition of Rule (3).

We consider all MPI operations that are collective over some set of processes as collectives, e.g., `MPI_Comm_dup`, with the only exception of `MPI_Finalize`, for which we consider no rule applicable, such as to use it as a well defined terminal state (if reachable). The transition system always arrives in a terminal state $(l_0, \ldots, l_{p-1})$, where either $l_i = m_i$ for all $i \in P$ or $l_i < m_i$ for some i. In the former case, wait state analysis could analyze all operations in the trace and arrived at the `MPI_Finalize` operations for which no rule is applicable, i.e., no deadlock exists in the analyzed trace. In the latter case, where some $l_i < m_i$ exists, our simulation of MPI blocking semantics revealed a deadlock, which we can then analyze and visualize with our graph-based approach [9].

Only one final state exists even though the transition system is nondeterministic, e.g., for the example in Figure 3 both $(0,0,0) \rightarrow_{\text{ws}} (0,0,1)$ and $(0,0,0) \rightarrow_{\text{ws}} (0,1,0)$ are choices for the first transition. However, each rule that allows an application of $\rightarrow_{\text{ws}}$ never disables any transition that could have been applied at a previous state. That is, if a rule can increment l_k at state $(l_0, \ldots, l_{k-1}, l_k, l_{k+1}, \ldots, l_{p-1})$, then all states $(l'_0, \ldots, l'_{k-1}, l_k, l'_{k+1}, \ldots, l'_{p-1})$ with $l'_i \geq l_i$ $(i \in P - \{k\})$ still allow the application of this rule. Thus, a terminal state must exist since the traces have finite length.

This observation is also a consequence of known results on transition systems with an independence relation for its transitions [7, 24]. In these results, independent transitions α, β must not be able to inactivate each other and their effect must be independent of their order ($\alpha\beta$ or $\beta\alpha$). In our case, for each pair (i,k) of processes $i, k \in P$ the transitions of processes i and k are pairwise independent. Since the behavior of the processes is deterministic and finite, the com-

[2]In Rules (1)-(4) and in subsequent transition examples we use labels for the transitions to indicate the type of the executed operation, e.g., $\overset{\text{nb}}{\rightarrow}_{\text{ws}}$ for Rule (1).

Process 0	Process 1	Process 2
Send(to:1)	Recv(from:ANY)	
Reduce(root=1)	Reduce(root=1)	Reduce(root=1)
	Recv(from:ANY)	Send(to:1)

Figure 4: Unexpected matching possible.

mutativity of independent transitions yields the existence of a unique terminal state.

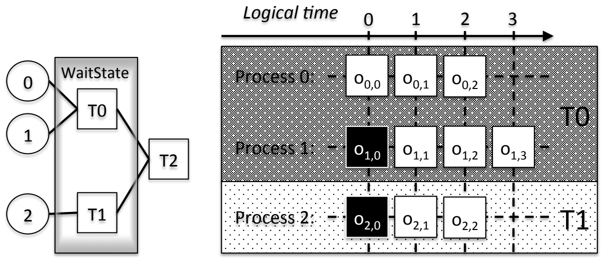
3.2 States and Deadlock

If the transition system arrives in a terminal state where we could not activate all `MPI_Finalize` operations, we can immediately apply our graph-based deadlock detection. However, in cases where some set of processes deadlocks while other processes may still advance, we should apply our deadlock detection to any reachable state $S = (l_0, \ldots, l_{p-1})$ of the transition system. We must not consider processes as blocked in their current operation if a transition exists that can advance the timestamp of this process. Thus, we consider a process $i \in P$ to be blocked in S if no $(S, S') \in \rightarrow_{\text{ws}}$ with $S' = (l_0, \ldots, l_i + 1, \ldots, l_{p-1})$ exists.

Consider for example the state $(2, 3, 1)$ in the example of Figure 3, where processes 0 and 1 have arrived at the send calls in which they deadlock, while process 2 has not returned from the barrier. In this state, no transition exists to advance processes 0 or 1, while one exists to advance process 2. Thus, we consider processes 0 and 1 as blocked and process 2 as unblocked in this state for graph-based deadlock detection. The dependency graph for this situation will not yet reveal a deadlock. For the terminal state $(2, 3, 2)$ we will consider all processes as blocked (since no transition allows any process to advance), where our deadlock detection then reveals a dependency cycle, which is a necessary and sufficient deadlock criterion in this situation.

3.3 Freedoms of MPI

Our transition system uses a fixed definition of b that is stricter than most MPI implementations. It considers all collectives as synchronizing and all standard-mode sends as blocking. Thus we can detect errors that do not manifest in a run with a certain MPI implementation, which is highly desirable. However, in combination with wildcard receives, these freedoms of MPI may lead to *unexpected* matches, such as in the example of Figure 4. If the reduce operation is non-synchronizing, as is likely for non-root processes, the send of process 2 may match the first wildcard receive of process 1, even though the send comes after the collective operation. As a result, for this match—since our transition system considers the reduce operation as blocking—it then cannot advance past its initial state. We define an unexpected match for a wildcard receive o_{i_1,j_1} in a terminal state S as a situation in which a send operation o_{i_2,j_2} exists that is active in S and that could match the active operation o_{i_1,j_1}, while point-to-point matching determines that it matches another send that is not active in S. As long as point-to-point matching provides no unexpected matches we can use our more conservative definition of b. If such an unexpected match occurs, we must adapt b to the choices of the MPI implementation, which we consider as a future extension. If this manifests as an issue with a target application, we can enforce standard-mode sends and collectives to be synchronizing to circumvent unexpected matches.



(a) TBON. (b) Data distribution between *T0* and *T1*.

Figure 5: Distribution of wait state for the example in Figure 2(b).

4. DISTRIBUTED WAIT STATE ANALYSIS

We can analyze MPI operations based on the formal transition system of the previous section. We can then apply graph-based deadlock detection to any of its intermediate states. If a deadlock exists, we can report the MPI operations that cause it along with their wait-for conditions to the user. In particular, if a deadlock exists in a state S, then it exists in all successive states S', which allows us to avoid expensive graph searches after analyzing each operation. In addition, the formalization highlights the data and the operations that an implementation of the transition system requires. We use this information to derive a distributed wait state analysis in the first layer of a Tree-Based Overlay Network (TBON), i.e., *WaitState* in Figure 1(b).

Figure 5(a) sketches a TBON for the three application processes in the example of Figure 2(b). In this example we use three tool processes *T0-T2* where we run our distributed wait state analysis on *T0* and *T1*. Since we do not assume shared memory between tool processes, the input data for wait state analysis is distributed, as Figure 5(b) sketches for the example in Figure 2(b). Here *T0* receives information on the operations of processes 0 and 1, while *T1* receives information on process 2. In cases such as the send $o_{2,0}$ and the receive $o_{1,0}$ (highlighted with white on black in Figure 5(b)) we must communicate between *T0* and *T1* to provide all input for wait state analysis. Our distributed point-to-point matching communicates within the first tool layer of the TBON, e.g., directly between *T0* and *T1*, to match point-to-point calls. It passes information on send operations to the tool node that receives information on the matching receive operation. As a result, in our example, *T0* can determine that the operations $o_{1,0}$ and $o_{2,0}$ match, but it lacks information on whether $o_{2,0}$ is active. Node *T1* neither knows which operation matches $o_{2,0}$ nor whether that matching operation is active.

4.1 Distribution

To derive a distributed algorithm for the wait state transition system, we distribute the state $L = (l_0, \ldots, l_{p-1})$ such that if processes $i, \ldots, i + k \in P$ send to a first tool layer node, then this node handles the components $l_i, \ldots, l_{i+k}$ of the global state $(l_0, \ldots, l_{p-1})$ in $\mathcal{T}$. In the example, *T0* handles the state of processes 0 and 1, while *T1* handles the state of process 2. As initially described, the execution of the wait state transitions system on each TBON node requires more input than point-to-point and collective matching provide.

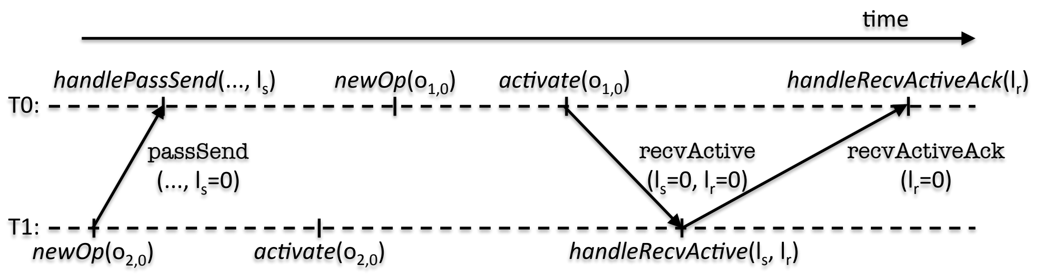


Figure 6: Data exchange for distributed wait state analysis of $o_{1,0}$ and $o_{2,0}$ in Figure 5(b).

Thus, we introduce additional communication in the TBON to distribute this information:

- `passSend`: Passes information on a send operation o_{i_1,j_1} hosted on node T to the node T', which hosts the matching receive operation o_{i_2,j_2}, includes the timestamp of the send (provides T' with the information that o_{i_1,j_1} and o_{i_2,j_2} match);
- `recvActive`: Informs node T hosting a send operation o_{i_1,j_1} that the matching receive operation o_{i_2,j_2} is now active (provides T with the information that o_{i_1,j_1} and o_{i_2,j_2} match, as well as that the precondition of Rule (2) is satisfied for i_1, i.e., that $l_{i_2} \geq j_2$);
- `recvActiveAck`: If a send o_{i_1,j_1} on node T received the `recvActive` message and is active itself, then T notifies the node T' hosting the matching receive o_{i_2,j_2} that o_{i_1,j_1} is now active (notifies T' that the precondition of Rule (2) is satisfied for i_2, i.e., that $l_{i_1} \geq j_1$);
- `collectiveReady`: If all processes on a node that belong to the process group of a collective operation are active, the nodes sends this message towards the TBON root; and
- `collectiveAck`: If the root of the TBON determines that all processes $k_0, \ldots, k_n$ that belong to a collective have activated their participating operation, it broadcasts this message towards the first tool layer (notifies all nodes that processes $k_0, \ldots, k_n$ satisfy the precondition of Rule (3)).

Figure 6 details these communications for nodes *T0* and *T1*, when they handle operations $o_{1,0}$ and $o_{2,0}$. When *T1* receives the send operation $o_{2,0}$, it generates the `passSend` message and directs it to *T0*. This message includes point-to-point matching information, which we hide, and the timestamp of $o_{2,0}$, i.e., 0. We issue a handler whenever a tool node receives a message, e.g., *handlePassSend* when *T0* receives the `passSend` message. The `passSend` handler searches for a receive that matches the send that the message describes. If the handler finds such a receive operation, it updates the operation with the timestamp of the send. If the handler does not finds the receive operation, as in Figure 6, then our point-to-point matching implementation stores the send operation in its matching structures. When *T0* receives $o_{1,0}$ (*newOp*) it matches this operation with send $o_{1,0}$. Afterwards, when $o_{1,0}$ becomes active, node *T0* generates the `recvActive` message that it transfers to node *T1*. This message includes the timestamps of both $o_{1,0}$ and $o_{2,0}$. When *T1* handles the `recvActive` message, it searches the send operation with the timestamp that the message provides and then checks whether this operation is active. In the example, the send is already active (*activate* was already issued for $o_{2,0}$). As a result, *T1* generates the `recvActiveAck` message and sends it to *T0*. This message contains the timestamp of the receive that matches $o_{2,0}$, which *T0* uses to find the receive operation in its trace. Node *T1* can advance the current timestamp for process 2 when it handles the `recvActive` message, whereas *T0* can advance the timestamp for process 1 when it handles the `recvActiveAck` message. The handling of wildcard receive operations is similar, except that if a possibly matching send call exists, the node that hosts the receive waits for an additional status update that reveals the matching decision of the MPI implementation.

For collectives, we must determine whether all participating processes can activate their operation. For that, each node first waits until all participating processes in its subset of the overall state are active in the collective. For example, *T0* requires that both $o_{0,1}$ and $o_{1,2}$ are active for the barrier (operations $o_{0,1}$, $o_{1,2}$, and $o_{2,1}$) in Figure 6. Once all participating processes of a node are active in a collective, the node generates the `collectiveReady` message and sends it towards the TBON root. We propagate this message with an aggregation [11], i.e., each node in the tree only forwards the message if all its descendants are active in the collective. When the root determines that all processes of a collective's process group are active, it generates the `recvActiveAck` message and broadcasts it to the first tool layer of the TBON. In our example, *T0* waits until the `recvActiveAck` message arrives and may then advance the state for processes 0 and 1 to the next operation.

Our implementation represents each operation with an object that stores state information. We use o to refer to such objects, where we use $o.l$ as the operations timestamp, $o.l_s$ as the timestamp of a matching send operation, $o.l_r$ as the timestamp of a matching receive operation, $o.active$ to express whether the operation is active, and $o.gotRecvActive$ to store whether a send received a `recvActive` message. In particular, each object has an attribute *canAdvance* in our implementation that stores whether process $i \in P$ can take the transition from l_i to $l_i + 1$, i.e., whether any transition rule applies to the operation. We set this attribute to $\top$ for $o_{i,j}$ exactly if:

- $b(i,j) = \perp$,
- A `collectiveAck` has arrived for collective $o_{i,j}$,
- A `recvActive` has arrived for active send $o_{i,j}$,
- A `recvActiveAck` has arrived for receive $o_{i,j}$, or

```
Function newOp(o)
    o.l:=nextTimestamp()
    o.active:=⊥
    if o.isBlocking() then
        o.canAdvance:=⊥
    else
        o.canAdvance:=⊤
    if o.isSend() then
        communicate::passSend(..., o.l)
```

```
Function activate(o)
    o.active:=⊤
    if isLastInactiveCollectivOnNode(o) then
        communicate::collectiveReady(o.comm)
    if o.isRecv() ∧ o.hasMatchingSend() then
        communicate::recvActive(o.l_s, o.l)
    if o.isSend() ∧ o.gotRecvActive then
        communicate::recvActiveAck(o.l_r)
        send.canAdvance:=⊤
```

```
Function handleCollectiveAck(communicator)
    o_0,...,o_k:=findActiveOpsInTrace(communicator)
    for o_i in o_0,...,o_k do
        o_i.canAdvance:=⊤
```

```
Function handlePassSend(...,l_s)
    if hasMatchingRecv(...) then
        storeSendForMatching(..., l_s)
    else
        recv:=findMatchingRecv(...)
        recv.l_s:=l_s
        if recv.active then
            communicate::recvActive(recv.l_s, recv.l)
```

```
Function handleRecvActive(l_s,l_r)
    send:=findOpInTrace (l_s)
    send.l_r:=l_r
    send.gotRecvActive:=⊤
    if send.active then
        communicate::recvActiveAck(send.l_r)
        send.canAdvance:=⊤
```

```
Function handleRecvActiveAck(l_r)
    recv:=findOpInTrace(l_r)
    recv.canAdvance:=⊤
```

Figure 7: Handler functions for distributed wait state tracking.

- For the completion operation $o_{i,j}$ that is associated with the non-blocking send/receive operations o_{i,j_0}, o_{i,j_1},...,o_{i,j_x} ($\forall k \in \{0,\dots,x\} : j_k < j$):
  - $o_{i,j}$ is an `MPI_Waitany` or `MPI_Waitsome` and $\exists y \in \{0,\dots,x\}$ such that canAdvance(o_{i,k_y}) $= \top$, or
  - $o_{i,j}$ is an `MPI_Wait` or `MPI_Waitall` and $\forall y \in \{0,\dots,x\}$ canAdvance(o_{i,k_y}) $= \top$.

We sketch the core of our distributed wait state tracking algorithm in Figure 7. It includes minimal versions of the functions *newOp*, *activate*, and the message handlers. For simplicity we do not describe the handling of completions that evaluate the state of their associated non-blocking operations based on the above rules. In addition, each node checks whether *canAdvance* is true for an active operation of process $i \in P$, if so, the node may advance the state of i, which we do not sketch here.

4.2 Implementation

With the algorithm in Figure 7, we implement distributed wait-state tracking in MUST. We use intralayer communication [13] for the `passSend`, `recvActive`, and `recvActiveAck` messages, while we use aggregations and broadcasts for the `collectiveReady` and `collectiveAck` messages. Our GTI infrastructure [11] automatically issues a handler when one of these messages arrives on a node.

We run distributed wait state analysis in parallel to point-to-point and collective matching. As a result, we may need to wait until a match is known before we can advance our analysis, as the handler *handlePassSend* in Figure 7 shows. We can remove operations from our trace data structure if we advance to a successive operation of the same process. As a result, a TBON node does not need to store complete traces for the processes that it handles. Instead, it stores a window of operations that starts at the current state of the transition system and ends with the last operation that it received from an application process. Thus, if our implementation processes operations faster than MPI processes issue new operations, we require a fixed amount of memory.

Importantly, our point-to-point and collective matching implementations can stream their communication, i.e., we can aggregate their communication buffers into a larger continuous buffer to allow for bandwidth efficient communication. However, our distributed wait state implementation cannot aggregate its messages since TBON nodes must wait for messages before they can continue their analysis. Thus, with our distributed wait state implementation we only use immediate communication, which impacts performance.

Situations exist that may prevent point-to-point matching from retrieving matching information for wildcard receives. In our centralized implementation we used a probing [14] technique to handle these situations. We currently do not extend this approach to our distributed implementation.

5. DEADLOCK DETECTION IN MUST

If the distributed implementation of the wait state transition system arrives in a terminal state where some process could not activate `MPI_Finalize`, we can immediately deduce that a deadlock exists. However, in order to provide more detail on the deadlock, as well as to issue deadlock detection for an intermediate state of the transition system, e.g., if a subset of the processes deadlocks early in the execution, we apply a graph-based deadlock detection [9].

We implement graph-based deadlock detection at the TBON root (*WfgCheck* in Figure 1(b)). Executing the graph detection after every transition of the wait state analysis un-

Function `handleRequestConsistentState`

```
stopProgress() /* Do not apply transitions */
$o_0,\ldots,o_n$:=getActiveOps()
$T_0,\ldots,T_k$:=findNodesToWhichWeSend($o_0,\ldots,o_n$)
outstandingPongs:=$k$
for Node $T$ in $T_0,\ldots,T_k$ do
    communicate::ping(1) /* sent to $T$, 1 counts
        the number of remaining pings for the
        double ping-pong */
if $k == 0$ then
    communicate::ackConsistentState
```

Figure 8: Handler function to create a consistent state.

necessarily incurs high overheads. Thus, we issue deadlock detection after a configurable timeout, without loss of precision. Since the TBON root does not have any information on the state of our distributed wait state implementation, we use messages to request and to provide this data. When the root starts deadlock detection, it first requests a consistent state (Section 3.2), for which it broadcasts the `requestConsistentState` message towards the first tool layer of the TBON. When a node in the first tool layer receives this message, it stops the wait state transition system. Afterwards, for each process we must determine whether a transition rule can be applied for that process. Since we use messages in the TBON to determine whether a rule can be applied, this action is equivalent to waiting for answers of all satisfiable requests. Consider for example the situation in Figure 6, if we assume that a request for a consistent state arrives after *T0* activated $o_{1,0}$ and after *T1* activated $0_{2,0}$, i.e., where the `recvActive` message of *T0* is sent but not yet received. In that situation we must wait until the `recvActive` message arrives, is handled, and the resulting `recvActiveAck` message also arrives on *T0*, which processes it.

Figure 8 shows the handler for `requestConsistentState`. Each active point-to-point message may require the arrival and processing of up to three intralayer messages (`passSend`, `recvActive`, and `recvActiveAck`). We use a double ping-pong synchronization between tool nodes in the first non-application layer of the TBON to guarantee that nodes can process and generate any of these messages. Each node T determines the set of node identifiers $T_0,\ldots,T_k$ that will host matching receives for active send operations on T. Afterwards, T uses a double ping-pong synchronization with each of the nodes $T_0,\ldots,T_k$ to ensure that all messages, which may still be in transit, arrive and are processed. Since messages in GTI are non-overtaking, our implementation processes and replies to these outstanding messages before the double ping-pong synchronization finishes. Once a node completes its ping-pong synchronizations, it sends the `ackConsistentState` message towards the TBON root.

When the root receives all `ackConsistentState` messages, it generates the `requestWaits` message to query for the actual wait-for conditions of all processes. The `ackConsistentState` message serves two purposes: it ensures that no node can send wait-for information before all ping-pong synchronizations are finished (nodes that host receives may not know which nodes will synchronize with them); and if a collective is complete, `collectiveReady` or `collectiveAck` messages may still be in transit when the root issues its `requestConsistentState` message. The `ackConsistentState` message ensures that all `collectiveAck` messages will arrive before the `requestWaits` message arrives, since both upwards (even with aggregations [12]) and downwards communication are non-overtaking in GTI.

When the nodes in the first tool layer receive the `requestWaits` message, they describe the wait-for conditions of all active operations for which a node cannot apply a transition. Afterwards, each node continues the execution of the wait state transition system. The root waits until it receives the wait-for information for all processes, builds a wait-for graph and checks for deadlock. If a deadlock exists, we log it in an HTML report and output a notification.

While the process of gathering all wait-for information, building the wait-for graph, and checking for deadlock is centralized, we only perform it after a timeout. We show in Section 6 that this rare analysis causes acceptable overhead for up to 4,096 processes.

6. APPLICATION STUDY

We use the Sierra cluster at the Lawrence Livermore National Laboratory to analyze the overheads associated with our distributed wait state analysis and our centralized graph detection. This system consists of 1,944 nodes of two 6 core Xeon 5660 processors and 24 GB of main memory each. The system uses a QDR InfiniBand interconnect.

We use a synthetic stress test to compare our previous centralized implementation of runtime graph-based deadlock detection [14] with our new distributed approach. The stress test executes multiple iterations of a cyclic exchange, where each process sends to its right neighbor and receives from its left neighbor. Each transfer sends/receives a single integer value and every 10th iteration issues an `MPI_Barrier` call to provide a mix of point-to-point and collective operations. This test is communication bound and latency sensitive, which stresses our runtime approach. The *fan-in* of a TBON node, i.e., the amount of lower level nodes it receives events from, impacts tool overhead. High fan-ins can cause higher tool overheads, while lower fan-ins decrease overhead at the cost of extra computing resources. Since we use a stress test, while we also cannot rely on our bandwidth efficient communication (see Section 4.2), we use low fan-ins of 2, 4, and 8. Our tool communication system is based on a communicator virtualization and can utilize MPI communication to transfer events between TBON nodes.

Figure 9 presents slowdowns as ratios between communication times with either implementation and of a reference run. The existing implementation scaled to up to 512 processes. While the slowdown with our tool is high for this stress test—with a fan-in of 2 about a factor of 70 for 16 processes and about a factor of 45 for 4,096 processes—it does not increases with scale, unlike with the centralized implementation, for which we would project a slowdown of about 8,000 at 4,096 processes. At low scales, application processes primarily use intra-node (shared memory supported) communication, while most communication with our tool occurs across nodes (due to process placement). Thus, our tool overhead is higher at lower scales and then decreases when the amount of internode communication increases for reference runs at higher scales.

For the stress test measurement in Figure 9, no timeouts occur, since information on MPI calls constantly arrives. As

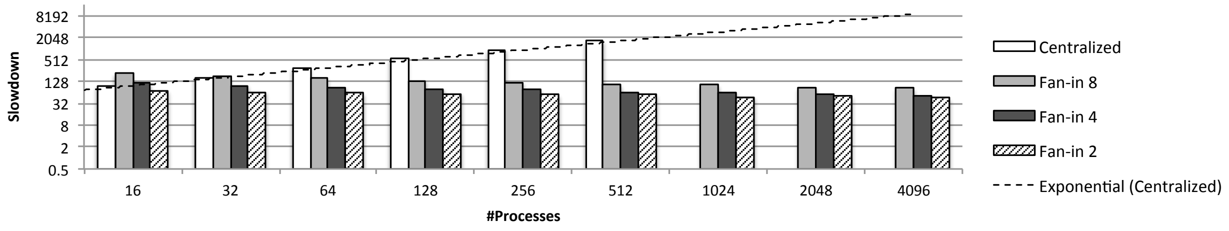
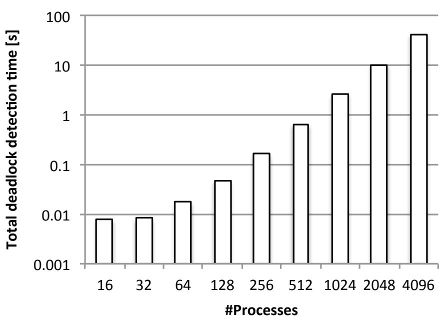
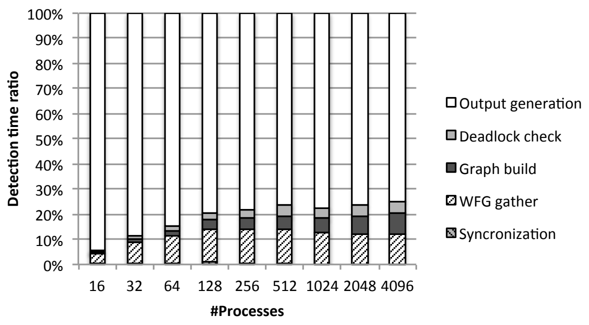


Figure 9: Distributed wait-state tracking slowdown with MUST for the synthetic stress test.

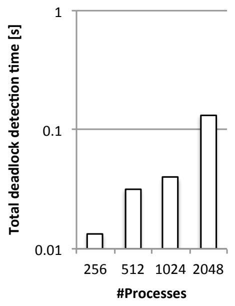


(a) Total deadlock detection time.

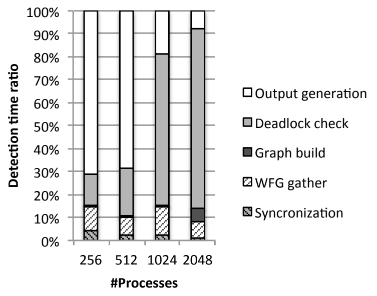


(b) Breakdown of deadlock detection time.

Figure 10: Deadlock detection time for wildcard receive example.



(a) Total deadlock detection time.



(b) Breakdown of deadlock detection time.

Figure 11: Deadlock detection time for 126.lammps.

a result, the TBON root issues no graph detections. We use a deadlock case to study the runtime of the centralized graph detection. Each process of this test case issues a wildcard receive without issuing any send operation beforehand. This causes a deadlock with a wait-for graph of maximal size (p^2 arcs, for p processes). Figure 10(a) presents the runtime of the graph detection, where we measure the time span that starts when the timeout occurs and that ends when the root reported the deadlock. While this timespan is noticeable at 4,096 processes, the centralized graph detection is still applicable for this challenging deadlock scenario. We split the overall detection time into the following groups:

- **Synchronization:** Synchronization time to compute a consistent state (see Section 5);
- **WFG gather:** Time spent to receive wait-for information for all processes;
- **Graph build:** Time spent to build the wait-for graph;
- **Deadlock check:** Runtime of the graph search for a necessary and sufficient deadlock criterion; and
- **Output generation:** Time to provide an HTML report, and a wait-for graph of the deadlocked processes in DOT[3].

Figure 10(b) presents the detection time ratios of these activity groups for the wildcard deadlock test case. It shows that synchronization time is low, while the generation of the DOT graph representation consumes about 75% of the overall detection time at higher scales. We can optimize the overheads for gathering WFG data and for building the graph, e.g., we could build the graph in parallel to receiving its arcs, while we could use more compact arc representations for our internal tool communication. However, graphs with p^2 arcs are not human readable for more than a few processes, which is an important direction for future work. A distributed graph search would still provide incomprehensible output. Thus, we plan to investigate graph transformations and simplifications, which could simplify wait-for information when we communicate it towards the root, e.g., in our wildcard stress test we would detect that all processes wait for all other processes with an *OR* semantic [9]. If we propagate aggregated and simplified wait-for information towards the root then we could reduce the graph search time and the complexity of the output graphs.

We use SPEC MPI2007 [21] (v2.0) to study our tool for a benchmark suite that is based on real-world applications. This benchmark operates at up to 2,048 processes[4]. We use a fan-in of 4 since the stress test measurements suggest that it provides low overhead while limiting extra tool resources. Figure 12 shows slowdowns for our distributed wait state tracking in MUST. Slowdowns are low for most

[3]http://www.graphviz.org/
[4]http://www.spec.org/mpi/docs/faq.html#DataSetL

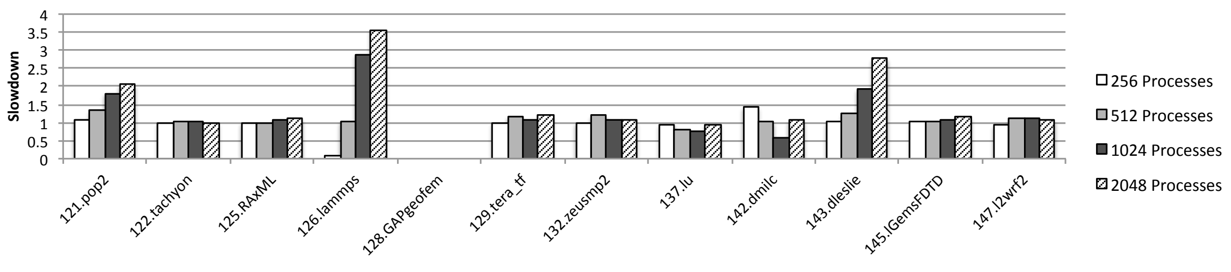


Figure 12: Distributed wait-state tracking slowdown with MUST on Sierra for SPEC MPI2007.

applications, where *121.pop2* and *143.dleslie* are more challenging due to high communication ratios. We currently do not report results for the application *128.GAPgeofem*. This long running application issues many communication calls and thus leads to long traces that can exceed the available main memory with MUST. In future work we will extend our GTI infrastructure such that we can *prefer* the processing of messages that are associated with our distributed wait state implementation over messages that are associated with point-to-point or collective matching, thus reducing the size of the trace window (see Section 4.2).

The application *126.lammps* contains a potential send-send deadlock that is not manifested in application runs, since the MPI implementation buffers these calls. However, our conservative definition of b (see Section 3) allows us to detect it. Figure 12 reports the overall runtime consumed until the application is aborted, which is dominated by a timeout-based implementation of `MPI_Abort` at higher scales. Figure 11(a) presents total detection time to find the deadlock, which is low compared to our wildcard test case, since the resulting wait-for graph uses fewer arcs. Figure 11(b) presents the breakdown of the detection time, where the output generation time is low, since the deadlock can be expressed as a cycle between two processes.

Finally, Figure 12 shows reproducible performance "gains" with our tool for *137.lu* and *142.dmilc*. For *137.lu*, our tool communication reduces the number of outstanding buffered sends that the application uses. Due to MPI internal handling, high amounts of buffered sends can impact performance. We can reproduce this behavior with a wrapper for `MPI_Send`, which replaces every 50th call with `MPI_Ssend`. For *142.dmilc* we could not yet determine the source of the performance improvement. In summary, at 2,048 processes (excluding *126.lammps* and *128.GAPgeofem*) our tool causes an average runtime increase of 34% for SPEC MPI2007.

7. CONCLUSIONS

An important input for runtime MPI deadlock detection is information on whether MPI calls of application processes at a certain execution step are blocking, i.e., wait for an unsatisfied condition at this execution step. The MPI specification provides no vendor independent API for such information. We present a transition system that formalizes this analysis with state information that allows distribution. The definition of the transition system highlights freedoms of MPI implementations that allow us to interpret the MPI standard more strictly than most implementations. While this allows us to detect deadlocks that will not occur in an application run, it also leads to the notion of *unexpected* matches. For these matches we plan to extend our model such that it correctly adapts to point-to-point matches that we would otherwise not consider.

We use the transition system to derive a distributed algorithm and its implementation in the runtime correctness tool MUST. Thus, we can combine this implementation with distributed approaches for runtime point-to-point and collective matching to derive a scalable runtime deadlock detection tool. The actual graph search for a deadlock criterion, which we only issue after a configurable timeout, remains centralized. An application study with a strong scaling benchmark suite shows that our distributed wait state analysis causes an average overhead of 34% for 2,048 processes, while we demonstrate constant or even decreasing overheads across scales for a synthetic weak scaling benchmark. The overhead for our graph search is noticeable but not limiting at 4,096 processes. Experiments with the benchmark suite show that our approach is applicable to a wide range of applications. For readability of our output graphs and to reduce search and communication time, we will investigate graph simplifications to reduce wait-for graph size, while we will also extend our tool infrastructure to reduce the memory footprint of our distributed wait state analysis.

8. ACKNOWLEDGMENTS

We thank the ASC Tri-Labs and the Los Alamos National Laboratory for their friendly support. This work is partly supported by the CRC 912 HAEC. Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-641235). 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.

9. REFERENCES

[1] D. H. Ahn, D. C. Arnold, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Overcoming Scalability Challenges for Tool Daemon Launching. In *Proceedings of the 37th International Conference on Parallel Processing (ICPP '08)*, pages 578–585, 2008.

[2] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *International Parallel and Distributed Processing Symposium, 2007. IEEE Computer Society*, 2007.

[3] D. Buntinas, G. Bosilca, R. L. Graham, G. Vallée, and G. R. Watson. A Scalable Tools Communications Infrastructure. In *Proceedings of the 2008 22nd International Symposium on High Performance Computing Systems and Applications*, HPCS '08, pages 33–39, Washington, DC, USA, 2008. IEEE Computer Society.

[4] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[5] C. J. Fidge. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

[6] M. Gerndt, K. Fürlinger, and E. Kereku. Periscope: Advanced Techniques for Performance Analysis. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *John von Neumann Institute for Computing Series*. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[7] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1996.

[8] W. Haque. Concurrent Deadlock Detection in Parallel Programs. *International Journal on Computers and Applications*, 28(1):19–25, 2006.

[9] 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*, pages 296–305, New York, NY, USA, 2009. ACM.

[10] T. Hilbrich, F. Hänsel, M. Schulz, B. R. de Supinski, M. S. Müller, W. E. Nagel, and J. Protze. Runtime MPI Collective Checking with Tree-Based Overlay Networks. In *Recent Advances in the Message Passing Interface*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.

[11] 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 *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 1364–1375, Washington, DC, USA, 2012. IEEE Computer Society.

[12] T. Hilbrich, M. S. Müller, M. Schulz, and B. R. de Supinski. Order Preserving Event Aggregation in TBONs. In Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 19–28. Springer Berlin Heidelberg, 2011.

[13] T. Hilbrich, J. Protze, B. R. de Supinski, M. Schulz, M. S. Müller, and W. E. Nagel. Intralayer Communication for Tree-Based Overlay Networks. In *ICPP Workshops*, Fourth International Workshop on Parallel Software Tools and Tool Infrastructures, Los Alamitos, CA, USA, 2013. IEEE Computer Society Press.

[14] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 30:1–30:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[15] T. H. Jun and G. R. Watson. Scalable Communication Infrastructure. `http://wiki.eclipse.org/PTP/designs/SCI`.

[16] B. Krammer and M. S. Müller. MPI Application Development with MARMOT. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *John von Neumann Institute for Computing Series*, pages 893–900. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[17] 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):93–100, 2003.

[18] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock Detection in MPI Programs. *Concurrency and Computation: Practice and Experience*, 14:911–932, 2002.

[19] F. Mattern. Virtual Time and Global States of Distributed Systems. In M. Cosnard, P. Quinton, M. Raynal, and Y. Robert, editors, *Parallel and Distributed Algorithms Conference*, pages 215–226. North-Holland, 1989.

[20] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.2. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, 2009.

[21] 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):191–205, 2010.

[22] A. Nataraj, A. D. Malony, A. Morris, D. C. Arnold, and B. P. Miller. A Framework for Scalable, Parallel Performance Monitoring. *Concurrency and Computation: Practice and Experience*, 22(6):720–735, 2010.

[23] P. Ohly and W. Krotz-Vogel. Automated MPI Correctness Checking: What if there was a magic option? In *Proceedings of the 8th LCI International Conference on High-Performance Clustered Computing*, 2007.

[24] D. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In D. Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer Berlin Heidelberg, 1994.

[25] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, New York, NY,

USA, 2003. ACM.

[26] S. F. Siegel and G. S. Avrunin. Modeling Wildcard-Free MPI Programs for Verification. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 95–106, New York, NY, USA, 2005. ACM.

[27] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings. In A. Gupta and S. Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 66–79. Springer Berlin Heidelberg, 2008.

[28] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: A Tool for Model Checking MPI Programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 285–286, New York, NY, USA, 2008. ACM.

[29] J. S. Vetter and B. R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[30] A. Vo, G. Gopalakrishnan, R. M. Kirby, B. R. de Supinski, M. Schulz, and G. Bronevetsky. Large Scale Verification of MPI Programs Using Lamport Clocks with Lazy Update. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 330–339, Washington, DC, USA, 2011. IEEE Computer Society.