Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings [⋆]

Sarvani Vakkalanka Ganesh Gopalakrishnan Robert M. Kirby

School of Computing, University of Utah, Salt Lake City UT 84112, USA,
`http://www.cs.utah.edu/formal_verification/cav08`

Abstract. Dynamic verification methods are the natural choice for debugging real world programs when model extraction and maintenance are expensive. Message passing programs written using the MPI library fall under this category. Partial order reduction can be very effective for MPI programs because for each process, all its local computational steps, as well as many of its MPI calls, commute with the corresponding steps of all other processes. However, when dependencies arise among MPI calls, they are often a function of the runtime state. While this suggests the use of dynamic partial order reduction (DPOR), three aspects of MPI make previous DPOR algorithms inapplicable: (i) many MPI calls are allowed to complete out of program order; (ii) MPI has global synchronization operations (e.g., *barrier*) that have a special weak semantics; and (iii) the runtime of MPI cannot, without intrusive modifications, be forced to pursue a specific interleaving because of MPI's liberal message matching rules, especially pertaining to 'wildcard receives'. We describe our new dynamic verification algorithm 'POE' that exploits the out of order completion semantics of MPI by delaying the issuance of MPI calls, issuing them only according to the formation of *match-sets*, which are ample 'big-step' moves. POE guarantees to manifest any feasible interleaving by dynamically rewriting wildcard receives by specific-source receives. This is the first dynamic model-checking algorithm with reductions for (a large subset of) MPI that guarantees to catch all deadlocks and local assertion violations, and is found to work well in practice.

1 Introduction

MPI [1] programs are an important class of *concurrent* programs used for the distributed programming of virtually all high performance computing clusters in the world. MPI will also be widely used for programming peta-scale supercomputers under construction [2]. Typical MPI programs are C programs (or C++/Fortran programs) that create a fixed number of processes at inception. These processes then perform computations in their private stores, invoking various flavors of send and receive API functions in the MPI library to exchange data, and also invoke global synchronization operations in the MPI library. Most MPI programs create processes that eventually terminate.

[⋆] Supported in part by NSF award CNS00509379, Microsoft HPC Institute Program, and SRC Contract 2005-TJ-1318.

MPI programs can contain many types of errors, including deadlocks, local assertion violations, resource leaks, and numerical inaccuracies. The primary goal of our work is to develop efficient methods to detect deadlocks and local assertion violations in MPI programs. Dynamic verification methods are the natural choice for verifying MPI programs because model extraction and model maintenance of MPI programs can be very expensive. This paper presents the first dynamic verification algorithm called POE (Partial Order reduction avoiding Elusive interleavings) for MPI that guarantees soundness (within the practical limits of runtime verification) and employs an effective partial order reduction algorithm. Of the many features of POE, the manner in which it guarantees coverage and implements reduction during dynamic verification are our main contributions. A good partial order reduction approach is crucial for verifying MPI programs because these programs mostly perform their computations in private stores, invoking MPI operations for message exchanges, where most (but not all) of these operations commute. Also, MPI calls occur with a high static and dynamic frequency, thanks to the many `for` loops in which MPI calls occur.

In this context, our verification tool ISP that uses the POE algorithm detects deadlocks missed by existing state-of-the-art tools. While the MPI 2.0 library itself supports over 300 MPI functions, ISP can handle 24 of the most commonly used MPI functions. In this paper, we describe the handling of five of these functions, namely `MPI_Isend`, `MPI_Irecv`, `MPI_Barrier`, `MPI_Wait`, and `MPI_Test`, and refer to them as 'send, receive, barrier, wait, and test.' Send and receive are, respectively, *non-blocking* operations, meaning that the issuing process can *start* the activity and proceed to execute later instructions while the send/receive proceeds in the background. The primary arguments of send are the destination process (this may not be a compile-time constant), the data being shipped, and a 'handle.' (Note: We do not detail some of the function arguments allowed by MPI calls, such as MPI 'tags' that affect message matching. Our implementation handles all allowed MPI arguments.) The issuing process may wait on the handle or test the handle. A wait blocks till the send operation finishes, while test returns false unless the send has finished, at which time it returns true. A send is deemed to have finished when the background process of copying the message out of the memory space of the sending process has finished. The arguments of receive are the source process ID (not necessarily a compile-time constant), the data receipt buffer, and a handle (with a semantics similar to the send handle). Instead of specifying a specific source process, receive can also mention '*', which is a *wildcard* receive that is open for receipt from any send that targets the receiving process. In effect, send and receive are *split operations*.

When an MPI process invokes a sequence of MPI calls, some of the calls may complete out of program order. For instance, if a process P0 invokes two consecutive non-blocking send operations targeting P1 and P2 respectively, the second send is allowed to finish before the first one (especially if the second send is shipping a much smaller amount of data). However, if both sends target the same process (say P1), then FIFO message ordering is required. This relaxed program order of MPI facilitates higher performance.

```
P0:  MPI_ISend(to P1, data = 22); ...rest of P0...
P1:  MPI_Irecv(*, x); if (x==22) then error1 else ...rest of P1...;
P2:  MPI_ISend(to P1, data = 33); ...rest of P2...
```

Fig. 1. Simple MPI Example Illustrating Wildcard Receives

Dynamic verification with persistent set based reductions was introduced in [5]. The dynamic partial order reduction algorithm (DPOR) [6] allows these dependencies to be accurately computed based on runtime state. This algorithm works by generating one interleaving of the program (maintained as a stack trace) and generating its interleaving variants. It ensures that the set of transitions explored from a state s forms a persistent set as follows. Consider the transitions t_i and t_j of processes p_i and p_j respectively such that $i < j$ in the current interleaving (this means that in the current interleaving t_i is executed before transition t_j). If t_i and t_j are dependent (i.e., t_i can either enable or disable t_j and vice versa), and t_i and t_j are co-enabled, then p_j is added to the pre-state of t_i hoping to eventually execute $t_j \dots t_i$. This approach does not work with MPI, as explained with the help of a short example (Figure 1).

In this example, MPI processes `P0` and `P2` are targeting `P1` which entertains a 'wildcard match,' *i.e.*, can receive from *any* process that has a concurrently enabled `ISend` targeting `P1`. As soon as one such send is chosen (say `P0`'s), the other send is not eligible to match with this receive of `P1` (it has to match another receive of `P1` coming later). This disabling behavior of the sends induces a dependency between them, as can be seen from the fact that the particular send that matches may or may not cause `error1` to be triggered. Consider some $i < j < k$, and a trace t where the ith action of t, namely t_i, is P2's send, and similarly t_j is P1's receive, and t_k is P0's send. In this trace, it is not necessary that P0's receive is matched with P2's send just because t_i is executed before t_k. MPI implements its own buffering mechanism that can cause one send to race ahead of the other send. Formally, unlike in DPOR, MPI's program order does not imply *happens-before* [7] in an MPI program's execution. Hence, it is possible that t_j is matched with t_k. There is no way in an MPI run-time (short of making intrusive modifications to the MPI library, which is often impossible because of the proprietary nature of the libraries) to force a match either way (both sends matching the receive in turn) by just changing the order of executing sends from P2 and P0.

Roadmap: Section 2 presents an overview of POE and discusses related work. Section 3 presents POE formally. Section 4 provides a summary of experimental results. Section 5 concludes the paper.

2 Overview of POE, and Related Work

Section 2.1 presents the barrier semantics of MPI, followed by the POE algorithm. Section 2.2 presents related work.

```
P0:  S0(to P1, h0) ; B0 ;                W(h0) ;
P1:  R (*, h1)     ; B1 ;                W(h1) ;
P2:                  B2 ; S2(to P1, h2); W(h2) ;
```

Fig. 2. Illustration of Barrier Semantics and the POE Algorithm

2.1 Barrier Semantics and Overview of the POE Algorithm

Barrier Semantics: No MPI process can issue an instruction past its barrier unless all other processes have issued their barrier calls. Therefore, an MPI program must be designed in such a way that when an MPI process reaches a barrier call, all other MPI processes also reach their barrier calls (in the MPI parlance, these are *collective operations*); a failure to do so deadlocks the execution. While these rules match the rules followed by other languages and libraries in supporting their barrier operations, in case of MPI, it is possible for a process P_i to have an operation OP_i before its barrier call, for another process P_j to have an operation OP_j after P_j's matching barrier call, and where OP_i can observe OP_j's execution. This means that OP_i can, in effect, complete after P_i's barrier has been invoked. This shows that the program ordering from an operation to a following barrier operation need not be obeyed during execution. This is allowed in MPI (to ensure higher performance), as shown by the example in Figure 2, and requires special considerations in the design of POE. In this example, one `MPI_Isend` issued by `P0`, shown as `S0`, and another issued by `P2`, shown as `S2`, target a wildcard receive issued by `P1`[1]. The following execution is possible: (i) `S0(to P1, h0)` is issued, (ii) R(*, h1) is issued, (iii) each process *fully* executes its own barrier, (`B0`, `B1`, or `B2`), and this "collective operation" finishes (all the `B`'s indeed form an atomic set of events), (iv) `S2(to P1, h2)` is issued, (v) now both sends and the receive are alive, and hence `S0` and `S2` become dependent, requiring a dynamic algorithm to pursue both matches. Notice that `S0` can finish after `B0` and `R` can finish after `B1`. (Note: Because of the placement of this barrier that is after `P0`'s send and `P1`'s receive, but *before* `P2`'s send, we sometimes refer to such barriers as 'crooked barriers.')

To recapitulate, MPI respects program ordering between any MPI operation $x \in \{\text{barrier}, \text{wait}, \text{test}\}$ and the MPI operation immediately following x in program order. A dynamic verification algorithm for MPI must therefore maintain a *completes-before* relation $\prec$ (defined in Section 3.2), and use it to determine, at runtime, all senders that can match a wildcard receive.[2]

POE Algorithm: We now present an overview of POE, as implemented by our verification scheduler (called the POE scheduler) that can intercept MPI calls and send them into the MPI run-time as and when needed:

[1] While not central to our current example, we also take the opportunity to illustrate how the handles `h0` through `h2`, and `MPI_Wait` (`W`) are used.

[2] Section 3 presents another detail of MPI which we refer to as 'trumping,' captured by another relation $\prec_c$.

- The POE scheduler executes C program statements along each process. All C statements are executed in program order. When the scheduler encounters an MPI operation, it simply records this operation, but does not execute it. This process continues till the scheduler arrives, within each process, at an MPI operation that is program ordered with respect to some previously collected (but not issued) MPI operation (we call these points *fences*).
- While at a fence point for all processes, since all senders that match a wildcard receive are known, *rewrite* the receives into specific receives. In our example, `R(*)` is rewritten into `R(from P0)` and `R(from P2)`.
- Form *match-sets*. Each match-set is either a single big-step move (as in operational semantics) or a set of big-step moves. Each big-step move is a set of actions that are issued collectively into the MPI run-time by the POE scheduler (we enclose them in $\langle\langle \ldots \rangle\rangle$). In our example, the match-sets are:
  - { ⟨⟨ `S0(to P1)`, `R(from P0)` ⟩⟩, ⟨⟨ `S2(to P1)`, `R(from P2)` ⟩⟩ }
  - ⟨⟨ `B0`, `B1`, `B2`⟩⟩
- Execute the match-sets in priority order, with all big-step moves executed first. The execution of a big-step move consists of executing all its constituent MPI operations. When no more big-step moves are left, then for each remaining set of big-step moves, recursively explore (according to depth-first search) all the big-step moves contained in it. In our example, this results in the big-step move ⟨⟨ `B0`, `B1`, `B2` ⟩⟩ from being performed first. Subsequently, both the big-step moves in
{ ⟨⟨ `S0(to P1)`, `R(from P0)` ⟩⟩, ⟨⟨ `S2(to P1)`, `R(from P2)` ⟩⟩ }
are pursued.

Thus, one can notice that POE never actually issues into the MPI run-time any wildcard receive operations it encounters. It always dynamically rewrites these operations into receives with specific sources, and pursues each specific receive paired with the corresponding matching send as a match-set in a depth-first manner.

Additional Points About Barriers: It must be observed that the code snippet in Figure 1 can be verified with DPOR if the technique of dynamic rewriting of the wildcard receives is employed. However, the code snippet in Figure 2 cannot be verified with DPOR even with dynamic rewriting of wildcard receives employed. Due to the presence of the barrier, the send `S2` can never be executed *before* the send `S0`, whereas in DPOR, we will need dependent actions to be replayable in both orders. In any interleaving of this example, however, `S0` will always be issued before `S2`. The POE algorithm overcomes this problem by executing the big-step move ⟨⟨ `B0`, `B1`, `B2` ⟩⟩, and *then* forming the match-set
{ ⟨⟨ `S0(to P1)`, `R(from P0)` ⟩⟩, ⟨⟨ `S2(to P1)`, `R(from P2)` ⟩⟩ }.

2.2 Related Work

In [8], it was observed that DPOR may offer a way to determine, at runtime, which sends and receives can match in MPI programs. However, since no dynamic verification tool was built, the issues discussed in Section 1 pertaining to the difficulties of forcing specific send/receive matches were not faced. In [9], nothing more than the standard DPOR of [6] was needed, as we handled only some of the shared memory features of MPI for which a DPOR-like approach works. In our 2-page tools paper [10], we actually implemented DPOR for many of MPI's communication commands, and in the process observed the unsoundness resulting from our inability to force specific send/receive matches. The POE algorithm takes advantage of our formal understanding of MPI (as captured in an extensive TLA+ model for MPI we are building [11]), precisely builds the *completes-before* relation $\prec$, uses it to discover potential send/receive matches precisely, and employs dynamic rewriting to force desired matches.

While MPI-SPIN [12,13,14], which is based on SPIN [15], can detect the kinds of errors that POE can detect, this approach inherently requires major effort on the part of users in building, by hand, verification models of their MPI programs in Promela [15]. Given the extensive number of C constructs and user-level library calls used in writing many MPI programs, this effort is impractical in those cases. MPI-SPIN does provide a reduction algorithm called the *Urgent Algorithm* that allows all MPI send/receive channels to be treated as rendezvous channels. However, this algorithm applies only to programs that do not use wildcard receives (which are extensively used by many MPI program types). In general, MPI-SPIN relies on SPIN's POR algorithm which, unfortunately, does not "understand" the commuting properties of MPI calls. In its favor, MPI-SPIN supports a symbolic execution facility to compare a sequential algorithm against an MPI implementation of the algorithm to detect numerical inaccuracies - a feature not supported by ISP.

Other works [16,17,18,19] do not seem to run into the problems we run into with MPI, including out-of-order completion, barriers, split operations, or run-time scheduling realities.

The plethora of concurrency libraries catering to 'multicore programming' suggests that dealing with complex APIs will become important. Yet, most tools in this area are based on the conventional 'testing' approach. ISP can now handle 24 MPI function types (detailed on our website). We have successfully handled all 69 examples in the Umpire [4] tool distribution. These are examples for which Umpire itself, and approaches such as Jitterbug [20] do not offer coverage guarantees (conventional verification tools for MPI that we surveyed [21] are unsound). Inserting randomized 'padding' delays to potentially perturb MPI's internal schedules (as done in ConTest [22], Jitterbug, Marmot [3], and Umpire) is highly unreliable, and slows down testing by adding delays into computational paths. For instance, for many of our examples containing wildcard receives provided on our website, Marmot missed generating many feasible schedules that actually contain deadlocks.

P1	P2	P3
$B_{1,1}$	$B_{2,1}$	$B_{3,1}$
$R_{1,2}(*, \langle 1,2\rangle)$	$B_{2,2}$	$S_{3,2}(1, \langle 3,2\rangle)$
$B_{1,3}$	$S_{2,3}(1, \langle 2,3\rangle)$	$B_{3,3}$
$R_{1,4}(*, \langle 1,4\rangle)$	$W_{2,4}(\langle 2,3\rangle)$	$W_{3,4}(\langle 3,2\rangle)$
$W_{1,5}(\langle 1,2\rangle)$	$B_{2,5}$	$B_{3,5}$
$W_{1,6}(\langle 1,4\rangle)$		
$B_{1,7}$		

Fig. 3. An Example MPI Program

3 Formal Presentation of POE

3.1 Abstract Syntax

Let $Nat = \{0, 1, 2 \ldots\}$, $Bool = \{0, 1\}$, and $Bool_\perp = \{0, 1, \perp\}$. Given $P \in Nat$ MPI programs, their PID ("MPI rank" of each process) set is $\{1 \ldots P\}$, and PID_* is the set $\{1 \ldots P\} \cup \{*\}$ ($*$ is to model 'wildcard receives'; see below). Let $L \in PID \to Nat$ be the lengths of the given programs, each program being viewed as a sequence of instructions. For any function f, its application to any argument i, $f(i)$, is often written f_i for brevity; for example $L(1)$ (often written L_1) is the length of the first program. Also, a function f of two arguments can be applied to two arguments i and j, written $f_{i,j}$, or partially applied to one argument i, and that is written f_i (this partial application returns a function which later "expects" a j). Let $p \in PID \to Nat \to I$ (where I is the set of MPI instructions defined in this sequel) be the programs. Thus $p_1 \ldots p_P$ are the P programs, and the jth instruction of the ith program is $p_{i,j}$. Let $l \in PID \to Nat$ be the program counters (PC) $l_1 \ldots l_P$. Let $f[i \leftarrow e]$ be function update, i.e. $f[i \leftarrow e] = (f \setminus \{\langle i, f(i)\rangle\}) \cup \{\langle i, e\rangle\}$. Also, $map\ f\ lst = \{f(i) \mid i \in lst\}$. Let $\pi_1\langle a, b\rangle = a$. For a set of pairs S, let $f[S]$ denote function update performed for every pair in S, i.e., $f[S] = (f \setminus \{\langle i, f(i)\rangle \mid i \in (map\ \pi_1 S)\}) \cup S$.

Let $h \in PID \to Nat \to Bool_\perp$ be the handles $h_1 \ldots h_P$. In our formal model, *every* instruction has a handle; it is only the case that W and T (MPI wait and test instructions defined in this sequel) happen to use this handle in a specific way. Handle $h_{i,j}$ is initially $\perp$. In our description of POE, we use the setting of $h_{i,j}$ to 0 to model POE *encountering* (collecting) instruction $any_{i,j}(\ldots)$ in program order, and the setting to 1 to model POE *issuing* (executing) this instruction. POE will (i) set $h_{i,j}$ to 1 out of program order (but still correctly so according to $\prec$), and (ii) dynamically rewrite the wildcards before forming match-sets and executing them. The **total system state** is $\langle l, h\rangle$ (we keep track of the PC values and the handle array status).

The set of MPI instructions I is the smallest set that include the following: *Barrier*, written $B_{i,j}$, *Send*, written $S_{i,j}(k, \langle i, j\rangle)$, where $k \in PID$ is the process targeted, and $\langle i, j\rangle$ is the handle used to track the progress of this *Send*, *Receive*, written $R_{i,j}(k, \langle i, j\rangle)$ where $k \in PID_*$ is the process from which the message is sourced ($*$ means 'wildcard receive,' i.e., the message is sourced from any process), and $\langle i, j \rangle$ is the handle (as with send) to track the progress of this *Receive*. We do not show the data payloads for sends S and receives R; when needed in discussions, they will be shown as a third argument. For S (send) and R (receive), their handle $\langle i, j \rangle$ is used by a following W instruction, or tested by a following T instruction (not required to exist by the MPI standard, and we also do not require the W/T to exist). I also includes *Wait*, written $W_{i,j}(\langle m, n \rangle)$ where $\langle m, n \rangle$ refers to a handle. $W_{i,j}(\langle m, n \rangle)$ blocks till $h_{m,n}$ is set to 1. This event occurs when the instruction which set $h_{m,n}$ to 0 finishes. (This earlier instruction is an S or R.) I also includes *Test*, written $T_{i,j}(\langle m, n \rangle, l)$ where $\langle m, n \rangle$ refers to a handle and l is a PC. $T_{i,j}(\langle m, n \rangle, l)$ blocks till $h_{m,n}$ is set to 1, and this occurs when the instruction that set $h_{m,n}$ to 0 (an earlier S or R) finishes, in which case the control transfers to the new PC l. Finally, I includes a conditional goto to model loops (space prevents further discussion of goto and T).

Figure 3 illustrates our syntax. Process P1 has seven sequential commands, and P2 and P3 each have five each. All proper MPI programs start with `MPI_INIT`, and *terminate* with `MPI_FINALIZE`, and both these essentially have the semantics of a barrier. Thus, the set $B_{1,1}$, $B_{2,1}$, and $B_{3,1}$ models `MPI_INIT`. Likewise, the set $B_{1,7}$, $B_{2,5}$, and $B_{3,5}$ models `MPI_FINALIZE`. The set $B_{1,3}$, $B_{2,2}$, and $B_{3,3}$ is a 'crooked barrier'. Thus, notice that the two sends $S_{2,3}(1, \langle 2, 3 \rangle)$ and $S_{3,2}(1, \langle 3, 2 \rangle)$ both target P1, and they can *both* potentially match with $R_{1,2}(*, \langle 1, 2 \rangle)$.
Illustration: In this example, if $R_{1,4}(*, \langle 1, 4 \rangle)$ is changed to $R_{1,4}(2, \langle 1, 4 \rangle)$, it is possible that $R_{1,2}(*, \langle 1, 2 \rangle)$ matches $S_{2,3}(1, \langle 2, 3 \rangle)$, and then $S_{3,2}(1, \langle 3, 2 \rangle)$ cannot match $R_{1,4}(2, \langle 1, 4 \rangle)$ (this receive expects a message from P2, not P3). This results in a *deadlock*. Such deadlocks cannot be detected through static analysis alone, because in MPI, send targets (i.e., the 1 in $S_{3,2}(1, \langle 3, 2 \rangle)$) and receive sources can be computed at runtime.

3.2 Completes-before Relation of MPI

MPI guarantees process-pair-wise message delivery ordering with respect to the *issue* orders of sends and receives. To illustrate this idea, consider two sends that are issued by process i both targeting process j, and two matching receives that are issued by process j, hoping to source from i. These sends and receives must be carried out in program order. It is only when send operations target receive operations in *different* processes, or receive operations source from different processes, that program order can be relaxed.

Specifically, suppose process i has a send $S_{i,m}(j, \langle i, m \rangle, d_1)$, and another send $S_{i,n}(j, \langle i, j \rangle, d_2)$, for $n > m$. Here, d_1 and d_2 are the data payloads. Suppose process j has a receive $R_{j,u}(i, \langle j, u \rangle, x_1)$, and another receive $R_{j,v}(i, \langle j, v \rangle, x_2)$, for $v > u$. Here, x_1 and x_2 are j's receive buffers, MPI guarantees FIFO message ordering and ensure that x_1 is bound to d_1 and x_2 to d_2 during execution. The POE algorithm must never issue these sends and receives out of order. In fact, the POE algorithm can 'fire and forget' these operations in program order, and be guaranteed that the MPI runtime will *match them in this appropriate order.*

Now consider a slightly different example where there are three processes i, j, and k in the system. The receives are $R_{j,u}(k, \langle j, u\rangle, x_1)$ and $R_{j,v}(*, \langle j, v\rangle, x_2)$, where $k \neq i$, and furthermore, let process k never issue a send to process j. In this case, the first receive (which cannot match any of the offers made by i) will be *trumped* by the second receive, which now goes ahead; the result will be that x_2 is bound to d_1. The POE algorithm has to be aware of this 'trumping rule.'

A third variant of our example is one where the sends are as above, the receives are $R_{j,u}(k, \langle j, u\rangle, x_1)$ and $R_{j,v}(*, \langle j, v\rangle, x_2)$, where $k \neq i$, but now there is a third process k which issues a send, $S_{k,l}(j, \langle k, l\rangle, d_3)$. Now, $R_{j,v}(*, \langle j, v\rangle, x_2)$ does not trump. The receive $R_{j,u}(k, \langle j, u\rangle, x_1)$ can indeed match the new send $S_{k,l}(j, \langle k, l\rangle, d_3)$, thus binding x_1 to d_3, and x_2 to d_1. POE has to be aware of this lack of trumping, as well. Thus, we note that when the sequence
$R_{j,u}(k, \langle j, u\rangle, x_1);\ \ldots\ R_{j,v}(*, \langle j, v\rangle, x_2)$ appears in process j, the second receive can *conditionally complete* before the first one, in a manner that depends on the runtime state of the system.

We now define the **completes-before** relation, $\prec$. The POE algorithm presented in Section 3.4 will be based on $\prec$. A variant of $\prec$ called *conditionally completes* ($\prec_c$) is used to model the concept of trumping discussed earlier. We do not discuss $\prec_c$ any more in this paper, for the sake of simplicity (it is of course incorporated into our implementation of POE, in forming match-sets according to $\prec_c$).

$\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow S_{i,j_1}(k, \ldots)\ \prec S_{i,j_2}(k, \ldots)$
$\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow R_{i,j_1}(k, \ldots)\ \prec R_{i,j_2}(k, \ldots)$
$\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow R_{i,j_1}(*, \ldots)\ \prec R_{i,j_2}(k, \ldots)$
$\forall i, j_1, j_2 :\ j_1 < j_2 \Rightarrow R_{i,j_1}(*, \ldots)\ \prec R_{i,j_2}(*, \ldots)$
$\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow S_{i,j_1}(k, \langle i, j_1\rangle)\ \prec W_{i,j_2}(\langle i, j_1\rangle)$
$\forall i, j_1, j_2, k :\ j_1 < j_2 \Rightarrow R_{i,j_1}(k, \langle i, j_1\rangle)\ \prec W_{i,j_2}(\langle i, j_1\rangle)$
$\forall i, j_1, j_2 :\ j_1 < j_2 \Rightarrow B_{i,j_1}\ \prec any_{i,j_2}(\ldots)$
$\forall i, j_1, j_2 :\ j_1 < j_2 \Rightarrow W_{i,j_1}(\ldots)\ \prec any_{i,j_2}(\ldots)$

FIFO Lemma: Any MPI program execution respecting $\prec^*$, the transitive closure of $\prec$, guarantees the required FIFO message orderings between MPI processes.

3.3 Match-Set Formation

Fence Instructions: For an instruction $j \in I$, $fence(j)$ holds exactly when for all succeeding instructions $k \in I$ in program order, $j \prec^* k$. Notice that 'wait' and 'barrier' act as fences, and depending on the MPI program, other instructions may attain a fence status.
Ancestor Relation: The *ancestor* of an instruction i is some instruction j where $j \prec^* i$. The set $ancestors(i)$ is the set of *indices* of i's ancestors. To exploit the FIFO Lemma, POE issues instruction i to the MPI system only after all its ancestors j have been issued. POE can issue any instruction not connected by $\prec^*$ out of order, as the MPI system itself considers such instructions semantically unordered (and hence may reorder them).

Match-set definitions: We now define all match-set types. The match-set type MS^*_R will be a set of big-step moves. The match-set type MS_B will be one big-step move containing all the matching barriers. Match-set type MS_R will contain exactly one send $S_{i,u}(j,\ldots)$, and its matching non wild-card receive $S_{j,v}(i,\ldots))$. Match-set type MS_W will be a big-step move of exactly one wait, and MS_T will be a big-step move of exactly one test. Consider the big-step moves $\langle\langle\ldots\rangle\rangle$ themselves to be sets.

The main difficulty in forming match-sets is to determine which sends can match a wildcard receive. To compute MS^*_R, we start with a set containing just the wildcard receive in question. We then seek the maximal number of additional sends that we can add to this set, without hitting a fence. Finally we break $*$ into specific instances of PIDs. We also must make sure that for the members of any MS, all its *ancestors* have been *issued* into the MPI system. Modeling this requires the state of the h array.

Formal Definition of $MS(l,h)$: We define match-sets as a function of l (the array of PCs) and h (the array of handles). In our definitions, we often refer to a "band" of past PC values where the MS might lie; this is what the function ρ used below denotes:

$$\begin{aligned}
MS_B(l,h) = {} & if\ \exists \rho \in PID \rightarrow Nat\ :\ \forall x \in PID : 1 \le \rho_x \le l_x \\
& \wedge \forall k \in PID :\ p_{k,\rho_k} = B_{k,\rho_k}\ \wedge\ \forall u \in ancestors(p_{k,\rho_k}) : h_{k,u} = 1 \\
& \wedge h_{k,\rho_k} = 0\ \ then\ \langle\langle B_{k,\rho_k}\ \mid\ k \in PID\rangle\rangle\ \ else\ \emptyset\ .
\end{aligned}$$

$$\begin{aligned}
MS^*_R(l,h) = {} & if\ \exists \rho \in PID \rightarrow Nat\ s.t.\ \forall x \in PID : 1 \le \rho_x \le l_x \\
& \wedge \exists i \in PID : p_{i,\rho_i} = R_{i,\rho_i}(*,\ldots)\ \wedge\ \forall u \in ancestors(p_{i,\rho_i}) :\ h_{i,u} = 1 \\
& \wedge h_{i,\rho_i} = 0 \\
& \wedge \forall k \in PID \backslash \{i\} : p_{k,\rho_k} = S_{k,\rho_k}(i,\ldots) \wedge \forall u \in ancestors(p_{k,\rho_k}) : h_{k,u} = 1 \\
& \wedge h_{k,\rho_k} = 0 \\
& then\ \{\ \langle\langle R_{i,\rho_i}(k,\ldots), S_{k,\rho_k}(i,\ldots)\rangle\rangle\ \mid\ k \in PID \setminus \{i\}\}\ \ else\ \{\emptyset\}\ .
\end{aligned}$$

$$\begin{aligned}
MS_R(l,h) = {} & if\ \exists \rho \in PID \rightarrow Nat\ s.t.\ \forall x \in PID : 1 \le \rho_x \le l_x \\
& \wedge \exists i,j \in PID : p_{i,\rho_i} = R_{i,\rho_i}(j,\ldots)\ \wedge\ p_{j,\rho_j} = S_{j,\rho_j}(i,\ldots) \\
& \wedge \forall u \in ancestors(p_{i,\rho_i}) : h_{i,u} = 1\ \wedge\ \forall u \in ancestors(p_{j,\rho_j}) : h_{j,u} = 1 \\
& \wedge h_{i,\rho_i} = h_{j,\rho_j} = 0 \\
& then\ \langle\langle R_{i,\rho_i}(j,\ldots), S_{j,\rho_j}(i,\ldots)\rangle\rangle\ else\ \emptyset\ .
\end{aligned}$$

$$\begin{aligned}
MS_W(l,h) = {} & if\ \exists i \in PID, 1 \le j,k \le l_i, k < j : p_{i,j} = W_{i,j}(\langle i,k\rangle) \\
& \wedge h_{i,j} = 0 \wedge h_{i,k} = 1 \wedge \forall u \in ancestors(p_{i,j}) : h_{i,u} = 1 \\
& then\ \langle\langle W_{i,j}(\langle i,k\rangle)\rangle\rangle\ else\ \emptyset\ .
\end{aligned}$$

Priority Scheme: Let $MS(l,h)$ be an abbreviation for invoking $MS_B(l,h)$, $MS_R(l,h)$, and $MS_W(l,h)$ in some order. If this invocation returns $\emptyset$, we will explicitly invoke $MS^*_R(l,h)$ and pursue the contents of this set, if any. The above is the **priority search** scheme that POE uses (postpone wildcard receives until all senders are discovered).

3.4 The POE Algorithm

We present the transition relation as an inference system which infers new states. Let $\langle l, h\rangle \in Rch$ mean that the state $\langle l, h\rangle$ has been reached. We invariantly maintain that $h_{i,l_i} = 0$. In the following, $h_{i,j}$ is set to 1 only by match-set moves. Non-MS moves are PC advances, and they result only in $h_{i,j}$ being set to 0 (the instruction is encountered but not issued). For a process i, a PC advance move is permitted if the instruction at its current PC is not a fence, or if the instruction has been issued (handle is set). The atomic transitions are the one of the $MS(l, h)$ moves, a PC move, or all the moves within $MS^*_R(l, h)$. Also $move(l, h, R)$ takes a system state $\langle l, h\rangle$, an atomic transition (set of instructions) R, sets the handle bits at the indices of the instruction. It does not advance the PC, as that will be done by the 'PC move' transition. Formally, let $\alpha \in I \rightarrow PID$ and $\beta \in I \rightarrow Nat$ be such that for instruction $r \in I$, $r = p_{\alpha(r),\beta(r)}$. Then, $move(l, h, R) = \langle l,\ h[\{\langle\langle \alpha(r), \beta(r)\rangle, 1\rangle \ \mid\ r \in R\}]\rangle$.

Init: $\langle l_0, h_0\rangle \in Rch$, where $l_0 = \lambda i.1$ and $h = (\lambda ij.if\ j = 1\ then\ 0\ else\ \perp)$.

```
Step: for ⟨l, h⟩ ∈ Rch
// All the deterministic singleton ample-set moves
if $MS(l,h) \neq \emptyset$ then $move(l,h,MS(l,h)) \in Rch$
// PC move which is also a singleton ample-set move
elseif $\exists i \in PID : \neg fence(p_{i,l_i}) \vee (h_{i,l_i} = 1)$
    then $\langle l[i \leftarrow l_i + 1], h_i[(l_i + 1) \leftarrow 0]\rangle \in Rch$
// Recursive exploration upon dependency. Ample = enabled.
elseif $MS^*_R(l,h) \neq \{\emptyset\}$ then $(map\ (\lambda r.move(l,h,r))\ (MS^*_R(l,h))) \subseteq Rch$
else Deadlocked.
```

Illustration of POE: POE will form match-sets (MS) from only those instructions that have a handle value of 0. In system state $\langle l, h\rangle$, if there exists a MS other than MS^*_R (will be a subset of I), POE picks any such set and invokes its operations (sets $h_{i,j}$ for that instruction to 1). MS^*_R is a set of subsets of I, and POE recursively invokes each member set in any order (in the implementation, these are backtrack points). If no MS can be built in the current system state, if possible, POE advances the PC l_i of some process i; else, the system is deadlocked. In our example (Figure 3), the first MS will be $\langle\langle B_{1,1}, B_{2,1}, B_{3,1}\rangle\rangle$, and these barrier calls are issued, setting $h_{1,1}, h_{2,1}$ and $h_{3,1}$ to 1. When $R_{1,2}(*, \langle 1, 2\rangle)$ from P1 is encountered, $h_{1,2}$ is set to 0 (instruction encountered, but recorded for future issue). Likewise, from P3, we encounter $S_{3,2}(1, \langle 3, 2\rangle)$, and set $h_{3,2} = 0$; we do not issue this send, as we have not carved out the maximal MS and we have not hit a fence. The system advances the PCs, finds the next MS $\langle\langle B_{1,3}, B_{2,2}, B_{3,3}\rangle\rangle$, and invokes it, setting the handle bits to 1. Following this, it will encounter $S_{2,3}(1, \langle 2, 3\rangle)$, setting $h_{2,3} = 0$. At this point, further PC advancement will place P1's PC facing $R_{1,4}(*, \ldots)$, which is $\prec$ ordered after $R_{1,2}(*, \ldots)$, and hence serves as a fence within P1. Now P2 encounters fence $W_{2,4}$, and P3 encounters fence $W_{3,4}(\langle 3, 2\rangle)$. At this point, the set $\langle\langle S_{2,3}(1, \langle 2, 3\rangle), S_{3,2}(1, \langle 3, 2\rangle), R_{1,2}(*, \langle 1, 2\rangle)\rangle\rangle$ is *promoted to an MS status.* The dynamic rewriting process produces two MSs (actually a set containing two MS sets) $\langle\langle R_{1,2}(2,\langle 1,2\rangle), S_{2,3}(1,\langle 2,3\rangle)\rangle\rangle$ and $\langle\langle R_{1,2}(2,\langle 1,2\rangle), S_{3,2}(1,\langle 3,2\rangle)\rangle\rangle$, and recursively invokes POE with these MSs. When the last MS-B is encountered, this corresponds to `MPI_FINALIZE`. At this time, if any handle is still a 0, and no more MS remains, an invalid end-state error is reported. In this example, no deadlock is encountered.

Correctness of POE: The correctness of POE consists of two steps. First, we must ensure that we abide by the FIFO Lemma in all scheduling decisions. This follows from POE never issuing actions contrary to $\prec$. However, whenever $\prec$ does not hold, POE may issue actions out of order. Second, we must ensure that we are executing according to conditions C0-C2 ([23]) of a correct partial order reduction algorithm (we do not require C3 owing to the acyclicity of MPI's state space). C2 is satisfied because local assertions only observe local process steps which are singleton ample. The priority scheme on Page 10 ensures that all singleton ample-sets contributed to by match-sets other than MS_R^* are exhausted. These preserve C1. Finally, the dependencies among the sends targeting a wildcard receive are correctly handled by doing a full recursive expansion of the constituents of MS_R^*, which also preserves C1.

4 Summary of Experimental Results

We have implemented the POE algorithm in our ISP runtime model-checker for MPI that is downloadable, along with our examples, from our website. A summary of our results is as follows:

- In all the 69 examples from the Umpire test suite, ISP produces the same theoretical number of interleavings required by our formal algorithm. This number is far smaller than the number of interleavings without reduction.
- Existing MPI program testing approaches (e.g., Umpire, Marmot) cannot detect deadlocks with assurance on many simple examples. In all these cases, the POE algorithm detects the deadlocks (see our webpage for the results).
- For some examples with several hundreds of lines of code that have no wildcard receives (where the code checks for local assertions), POE requires exactly *one* interleaving. Existing testing tools will wastefully explore multiple interleavings where the MPI operations have no dependencies.
- POE's setting of handle bits turns into collecting MPI operations without issuing them. These book-keeping steps of ISP have negligible overheads. The main overhead of ISP is that of restarting MPI for each replay. In [10], we provide techniques that can dramatically reduce this overhead. This technique will be integrated into our current ISP version.
- ISP supports 24 MPI functions, including many collective operations, MPI communicators, and non-deterministic wait functions such as `MPI_WAIT_ANY`. However, in a significant number of cases, we can allow an MPI program to issue operations even outside of this set. These extra functions (such as `MPI_TYPE_CREATE`) can still be issued into the MPI run-time without being trapped by the verification scheduler of POE.

- POE's scheduler is designed to be parallelized using MPI in future versions of ISP. Also a static analysis package to remove computations that do not affect control flow has been prototyped and will be integrated into ISP.

5 Concluding Remarks

We have described an algorithm for handling out of order execution and barrier semantics in verifying MPI programs for deadlocks and local assertions. We emphasize that POE works on unaltered MPI source programs. The verification tool implementing POE works well in practice, and is sound within the practical limits of runtime verification. An example of such a limitation is captured by the `MPI_Test` function. The outcome of `MPI_Test` (true or false) depends on the speed of computation of MPI processes. It is possible for a given MPI runtime to always produce the `true` outcome, for example. We are investigating the modification of the open source MPICH 2.0 library to overcome such limitations.

We have a formal TLA+ model of MPI 2.0 [11] (and we even have an execution framework that takes short MPI programs and runs them against this semantics [24]), we are in a position to rigorously prove the MPI semantics described in this paper.

Acknowledgements: The authors wish to thank Rajeev Thakur of Argonne National Labs and Bill Gropp of UIUC for their ideas and encouragement.

References

1. Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core.* MIT Press, Cambridge, MA, USA, 1998.
2. Invited Talk by Al Geist at EuroPVM/MPI 2007, "Sustained Petascale: The Next MPI Challenge.
3. Bettina Krammer, Katrin Bidmon, Matthias S. Mller, and Michael M. Resch. Marmot: An MPI analysis and checking tool. In *Parallel Computing 2003*, September 2003.
4. Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing*, pages 70–79, 2000.
5. Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.
6. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 110–121. ACM, 2005.
7. Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
8. Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby. Semantics driven dynamic partial-order reduction of MPI-based parallel programs. In *Parallel and Distributed Systems - Testing and Debugging (PADTAD-V)*, July 2007.
9. Salman Pervez, Robert Palmer, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. Practical model checking method for verifying correctness of MPI programs. In *EuroPVM/MPI*, pages 344–353, 2007.

10. Sarvani Vakkalanka, Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A tool for model checking MPI programs. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 285–286, 2008.
11. Guodong Li, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Formal specification of the MPI-2.0 standard in TLA+. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 283–284, 2008.
12. Stephen F. Siegel. Efficient Verification of Halting Properties for MPI Programs with Wildcard Receives. In *Proceedings of Verificaiton, Model Checking,and Abstract Interpretation: 6th International Conference, VMCAI*, 2005.
13. Stephen F. Siegel and George S. Avrunin. Modeling Wildcard-free MPI Programs for Verification. In *to appear in Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 2005.
14. Stephen F. Siegel and George S. Avrunin. Verification of MPI-based software for scientific computation. In *In Proceedings of the 11th International SPIN Workshop on Model Checking Software*, pages 286–303, Barcelona, April 2004.
15. Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.
16. Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. UUCS-07-008:Runtime Model Checking of Multithreaded C/C++ Programs. Technical report, University of Utah, School of Computing, 2007. http://www.cs.utah.edu/research/techreports/2007/ps/UUCS-07-008.ps.
17. Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In Dragan Bosnacki and Stefan Edelkamp, editors, *SPIN*, volume 4595 of *Lecture Notes in Computer Science*, pages 58–75. Springer, 2007. Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings.
18. Madan Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 446–455, 2007.
19. http://research.microsoft.com/projects/CHESS/.
20. Richard Vuduc, Martin Schulz, Dan Quinlan, Bronis de Supinski, and Andreas Saebjornsen. Improved distributed memory applications testing by message perturbation. In *Parallel and Distributed Systems: Testing and Debugging (PADTAD - IV)*, 2006.
21. Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. A survey of MPI related debuggers and tools. Technical Report UUCS-07-015, University of Utah, School of Computing, 2007. http://www.cs.utah.edu/research/techreports.shtml.
22. Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.
23. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 2000.
24. Robert Palmer, Michael Delisi, Ganesh Gopalakrishnan, and Robert M. Kirby. An approach to formalization and analysis of message passing libraries. In *Formal Methods for Industry Critical Systems (FMICS)*, Berlin, 2007.
25. Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of MPI programs with intel message checker. In *SE-HPCS '05*, pages 78–82, 2005.