Reduced Execution Semantics of MPI: From theory to practice ⋆

Sarvani Vakkalanka, Anh Vo, Ganesh Gopalakrishnan, and Robert M. Kirby

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

Abstract: There is growing need to develop formal verification tools for Message Passing Interface (MPI) programs, to eliminate bugs such as deadlocks and local assertion violations. Of all approaches, dynamic verification is most practical for MPI. Since the number of interleavings of concurrent programs grow exponentially, we devise a dynamic interleaving reduction algorithm (*dynamic partial order reduction*, DPOR) tailor-made for MPI, called POE. The key contributions of this paper are: (i) a formal semantics that elucidates the complex dynamic semantics of MPI, and played an essential role in the design of the POE algorithm and the construction of the ISP tool. (ii) a formal specification of our POE algorithm. We discuss how these ideas may help us build dynamic verifier for other APIs, and summarize a dynamic verifier being designed for applications written using a recently proposed API for multi-core communication.

1 Introduction

It has been widely observed that exploiting concurrency *efficiently* and *correctly* is an issue of growing importance. Practical concurrent programs are written using various shared memory and message passing application programming interfaces (APIs). The execution semantics of these programs are governed largely by the API semantics, the compiler induced semantics, and the runtime (*e.g.*, scheduling, memory allocation) semantics. Therefore, it has been widely recognized that two popular forms of formal verification – namely *model based verification* and *static analysis* – can play only a supportive role, with dynamic verification [1] being the most practical *primary* approach for real-world concurrent programs. Except for debugging concurrent algorithms, model based verification would be prohibitively expensive – especially if it were to be employed to directly model the API semantics, the C semantics, or the runtime semantics. Similarly, static analysis methods cannot accurately predict the synchronizations and communications in programs where locks or message communicators are determined through complex synchronizations.

Dynamic verification tools take as their input the user code and a user provided test harness. Then, using customized scheduling algorithms, they enforce specific classes of concurrent schedules to occur. Such schedules are effective in hunting down bugs, and often are sufficient to provide important formal coverage guarantees. Dynamic verification tools almost always employ techniques such as dynamic partial order reduction (DPOR) [2,3], bounded preemption searching [4,5], or combinations of DPOR and symmetry [6] reduction to prevent redundant state/interleaving explorations. While many such tools exists for verifying shared memory concurrent programs, there is a

⋆ Supported in part by Microsoft and NSF award CNS00509379

noticeable dearth of dynamic verification tools supporting the *scientific programming* community that employs the Message Passing Interface (MPI, [7]) API as their *lingua franca*. The importance of MPI is well known: it is employed in virtually all scientific explorations requiring parallelism, such as weather simulation, medical imaging, and earthquake modeling that are run on *expensive* supercomputers. Our group has developed the only formal dynamic verification tool for MPI programs currently in existence, called ISP [8,9,10,11,12]. As complete verification of MPI programs is infeasible, ISP focuses on detecting common errors in MPI programs such as deadlocks, assertion violations, and resource leaks. For these bug classes, ISP guarantees completeness of coverage under reasonable assumptions [8,9,10] that are almost always met in practice.

The main contributions of this paper are: (i) a simple and intuitive formal semantics for MPI that was developed hand-in-hand with the construction of ISP, and (ii) a rigorous description of the primary algorithm underlying ISP called "POE" (Partial Order avoiding Elusive Interleavings), which was sketched in [8]. This paper covers four MPI constructs in detail, highlighting the primary differences between our previous work [13,14] in which we wrote a reasonably comprehensive higher level reference semantics for about 150 MPI constructs in TLA+ [15]. We now highlight *why* dynamic verification of MPI programs is different from previous dynamic verification methods for shared memory programs. Then, beginning § 2, we elaborate on our formal semantics and formally describe the POE algorithm in § 3. In § 5, we sketch ongoing work, as well as our concluding remarks.

Challenges of Designing a Dynamic Verifier for MPI: MPI is inherently a low level notation, and the process of optimizing an MPI program increases the use of non-deterministic as well as asynchronous (non-blocking) MPI calls. In such programs, many MPI bugs remain latent, and surface when ported to a new MPI platform where the buffer allocations, node-to-node speeds, or MPI runtime scheduling policies may be different. ISP has been used with great success on many such programs of several thousands of lines of code such as ParMETIS [16] and MADRE [17], in addition to analyzing [18] almost all the examples from popular MPI textbooks such as [19], and has found numerous issues in these programs (*e.g.* [20]). We now list the challenges in developing a dynamic verifier that can handle such a range of programs, and is capable of running on ordinary laptop computers with acceptable runtimes.

```
1  P1                        P2                        P3
2  ---                       ---                       ---
3  Isend(to P3,d1,&h1);      Isend(to P3,d2,&h2);      Irecv(from ANY_SRC,x,&h3);
4  ..delay slot..            ..delay slot..            ..delay slot..
5  Wait(h1);                 Wait(h2);                 Wait(h3);
6                                                      if(p(x)) then OK else BUG
```

Fig. 1. MPI Example Showing Need for Formalization

Consider Figure 1 in which we employ *non-blocking sends* (`Isend`) and *non-blocking receives* (`Irecv`) – both typically used for efficiency. After starting an `Isend`, the process can continue issuing subsequent statements that are in the delay slot (the region from `Isend` till the following `Wait` or `Test`). Statements in the delay slot must not access the send/receive buffer. The same assumptions also apply to the `Irecv` call. The `Wait` detects whether the non-blocking operation has finished. Some MPI sys-

```
1  P1                      P2                      P3
2  ---                     ---                     ---
3  Isend(to P3,d1,&h1);    Barrier();              Irecv(from ANY_SRC,x,&h3);
4  Barrier();              Isend(to P3,d2,&h2);    Barrier();
5  Wait(h1);               Wait(h2);               Wait(h3);
6                                                  if(p(x)) then OK else BUG
```

Fig. 2. Augmented MPI Example Showing Need for MPI-specific DPOR

tems have enough memory that they can serve as temporary storage, absorbing the sent data and causing the `Wait` to return instantaneously; others will wait for the receiving process to come along and convey the data directly. Now in the first interleaving, it is possible that the MPI wildcard receive (receive from any sender, written '`MPI_Irecv(from MPI_ANY_SRC)`') of P3 matches P1. ISP must then replay the MPI program, ensuring that P2's send will match P3's receive (else, the data dependent bug may be missed). Now, in a dynamic verification tool for shared memory concurrent programs such as [5,3,2], one can force desired schedules simply by controlling the *issue order* of API calls. However, with MPI, if we *issued* the `Isend` of P1 followed by the wildcard `Irecv` of P3 and then issue the `Isend` of P2, we may still have P2's `Isend` race ahead (inside the MPI runtime) and *match* the `Irecv`. Thus, building a dynamic verifier for a complex API such as MPI requires a deep understanding of *the evolution states of each individual MPI call*. This understanding is the focus of the formal model proposed in this paper, where we show that each MPI call goes through four states: *issued* (notated as $\triangleright$), *returned* ($\triangleleft$), *matched* ($\Diamond$), and *completed* ($\bullet$).

Figure 2 provides additional insights into the need for a focused formal semantics that guides implementations. All MPI processes must issue ($\triangleright$) the Barrier call before it can be crossed; also no instruction after a barrier can be performed until all prior instructions have been issued (*but perhaps not matched,* $\Diamond$) in every process. Now suppose we want to replay the execution of this program so as to cause `Isend(to P3,d2,&h2)` in P2 to match P3's wildcard receive. Because of the MPI barrier semantics, we must issue `Isend(to P3,d1,&h1)` in P1 before we can issue `Isend(to P3,d2,&h2)` of P2. But now, all bets are off: the MPI runtime may again match `P1`'s call with `P3`. POE handles Figure 2 by first collecting, but deliberately not issuing the `Isend` and `Irecv` into the MPI runtime. Then ISP issues all the barrier calls into the MPI runtime. Thus, notice that we have issued the `Barrier` *before* issuing a statement prior to it! Our formal semantics helps us justify that this *out of order issue* of MPI commands is sound, as these commands are not related by the MPI *intra happens-before* [21] relation that is formally defined here.

After determining which `Isend`s can match a wildcard receive, ISP dynamically rewrites `Recv(ANY_SRC)` to `Recv(from P1)` in one replay and `Recv(from P2)` in the other replay. This is to ensure that both these matches happen, *regardless of the speed of the cluster machine or the MPI library* on which the dynamic verification is occurring. In other words, the scheduler can "fire and forget" *dynamically determinized* MPI commands into the MPI runtime, knowing that they will match eventually. Such practical details of ISP are well described in [9]; this paper's focus is on a suitable formalization of MPI.

The formal semantics presented in this paper not only explicate the salient events (namely, $\triangleright$, $\triangleleft$, $\Diamond$, and $\bullet$) marking the progress of an MPI API call, but also formally

define the intra process *happens-before* order that then allows us to schedule MPI actions in ISP. As opposed to our earlier formal specifications [13,14], our emphasis in this formalization has been to decide what to leave out, *i.e.*, focus on a few core MPI constructs, their constituent events, and the relationships between the events of various calls. Even so, we leave out many details of these constructs such as communicators, tags, etc., as our objective is to understand the happens-before relation *under a given set of communication matches*. Doing it all – *i.e.*, describing hundreds (of the total of over 300) MPI calls as well as all their event details and arguments – would be impractical and/or pointless. For example, even without revealing the event details, the work of [14] that covers about 150 MPI calls in detail occupies 192 printed (11-point font formatted) pages of TLA+. Another formal semantics for a small subset of MPI has recently been provided in [22]. It employs a fairly elaborate state machine to describe how MPI commands execute and interact with each other. One of the earliest MPI formal specifications was in [23]. These works do not meet our objectives of guiding the implementation of a dynamic verification tool.

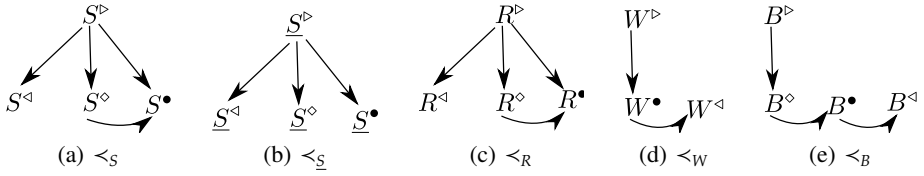
Summary of Features of ISP: Since the rest of this paper is on the aforesaid formal semantics of MPI, we close off this section with a summary of ISP's practical nature. ISP is a practical push-button dynamic verifier for MPI programs that is available for download and study [24]. It has three GUIs that help debug large programs, one written in Java, the other using Visual Studio, and the third using Eclipse. While we characterize only four MPI constructs in-depth in this paper, we believe that we can similarly model the remaining 60 or so frequently used MPI calls currently supported by ISP. For example, the widely used `MPI_Send` can be regarded as `atomic{MPI_Isend;MPI_Wait}`. As reported earlier, ISP has handled many large real-world case studies with success, including ParMETIS [16], a parallel genome assembler mpiBLAST [25], and a large benchmark from Livermore called IRS [26]. On ParMETIS (a 14K LOC example), ISP's efficient dynamic partial order reduction algorithm POE produced only *one* interleaving, finishing in seconds on a laptop [9]. ISP handles all practical aspects of MPI including `ANY_SOURCE`, `ANY_TAG`, `WAIT_ANY`, `PROBE` and `IPROBE`, almost all collectives, as well as communicators. The engineering details of the dynamic scheduling control methods of ISP are reported in [10], and are not included here.

2 MPI Introduction

This section provides an overview of the four MPI functions `Isend` (S), `Irecv` (R), `Wait` (W), and `Barrier` (B), based on our understanding of the MPI standard [7], reading the MPI sources, and from past experiences [13,14]. Most MPI programs have two or more processes communicating through MPI functions. The MPI library is part of the **MPI runtime** and can be thought of as another process. All the processes have MPI process ids called ranks $\in Nat = \{0, 1, \ldots\}$ that range from $0 \ldots n-1$ for n processes. Every MPI function is in one of the following states:
issued ($\triangleright$): The MPI function has been issued into the MPI runtime.
returned ($\triangleleft$): The MPI function has returned and the process that *issued* this function can continue executing.
matched ($\Diamond$): Since most MPI functions usually work in a group (for example, a S from one process will be matched with a corresponding R from another process), an MPI

function is considered *matched* when the MPI runtime is able to match the various MPI functions into a group which we call a *match-set*. Another example of a *match-set* is the one contributed to by a Barrier (B). All the function calls in the *match-set* will be considered as having attained the *matched* state.
complete ($\bullet$): An MPI function can be considered to be complete according to the MPI process that issues the MPI function when all visible memory effects have occurred (*e.g.*, in case the MPI runtime has sufficient buffering, we can consider an `Isend` to complete when it has copied out the memory buffer into the runtime bufer). The completion condition is different for different MPI functions (*e.g.*, the `Irecv` matching the `Isend` may not have seen the data yet, but still `Isend` can complete on the send side).

An MPI function $F \in \{S, R, W, B\}$ in state $s \in \{\triangleright, \triangleleft, \diamond, \bullet\}$ will be denoted F^s. We use F_i to denote that F is invoked by process with rank i. Two distinct MPI functions $M \in \{S, R, W, B\}$ and $N \in \{S, R, W, B\}$ from the same process i will be denoted as $M_{i,j}$ and $N_{i,k}$ where j and $k \in Nat$ and $j \neq k$.



(a) $\prec_S$ (b) $\prec_{\underline{S}}$ (c) $\prec_R$ (d) $\prec_W$ (e) $\prec_B$

Fig. 3. Partial Orders of MPI function states

2.1 MPI Functions

MPI_Isend (S) is a non-blocking send that has the following simplified prototype:

```
Isend (int dest, Datatype buffer, Request &handle);
```

where `dest` is the destination process rank where the message is to be sent, `buffer` is the actual data payload that must be sent and `handle` is set by the MPI runtime and uniquely identifies the S in the MPI runtime. The function call *returns* immediately (non-blocking) while the actual send can happen at a later time. An S is considered *complete* by the process issuing it if the data from the buffer is copied out. The buffer can be either copied out into the MPI runtime provided buffer or to the buffer space of the MPI process receiving this message. Hence, if the MPI runtime has buffer available, the S can be *completed* immediately. Otherwise, the S can be completed only after it is *matched* with a matching R. It is illegal for the MPI process to re-use the send buffer before the send is *completed*. The completion of a send is detected by the process issuing it using `Wait` (W). We use $\underline{S}$ to denote a buffered send and S to denote a send with no runtime buffering. For a given send, we have a partial order ($\prec_{\underline{S}}$) of its state transition as follows:

- $\{S^{\triangleright} \prec S^{\triangleleft},\ S^{\triangleright} \prec S^{\Diamond},\ S^{\triangleright} \prec S^{\bullet}\}$.
- When the send cannot be buffered by the runtime we have $\prec_S = \prec_{\underline{S}} \cup \{S^{\Diamond} \prec S^{\bullet}\}$.

Figure 3(a) shows the partial order when the send is not buffered $\prec_S$ while Figure 3(b) shows the partial order when the send is buffered $\prec_{\underline{S}}$. The evolution of the state of each

MPI operation according to these partial orders is caused by the processes issuing their operations and the MPI runtime advancing the state of the operations.
MPI_Irecv (R) is a non-blocking receive with the following prototype:

```
Irecv (int src, Datatype buffer, Request &handle);
```

where `src` is the rank of the process from where the message is to be received. The data is received into `buffer` and `handle` is returned by the MPI runtime which uniquely identifies the receive in the MPI runtime. The function call *returns* immediately and is considered *complete* when all the data is copied into `buffer`. It is illegal to re-use `buffer` before the receive completes. The completion of a receive is detected by the process using `Wait` (W). For a given R, we have a partial order ($\prec_R$) of its states as follows: $\{R^{\triangleright} \prec R^{\triangleleft},\ R^{\triangleright} \prec R^{\Diamond},\ R^{\triangleright} \prec R^{\bullet},\ R^{\Diamond} \prec R^{\bullet}\}$ shown in Figure 3(c).
MPI_Wait (W) is a blocking call and is used to detect the completion of a send or a receive and has the following prototype: `Wait (Request *handle)`, where `handle` is returned in a S or a R. The MPI runtime blocks the call to W until the send or receive is *complete*. The MPI runtime resources associated with the `handle` are freed when a W is invoked and `handle` is set to a special field called `REQUEST_NULL`. A W call with `handle` set to `REQUEST_NULL` is ignored by the MPI runtime. An S or R without an eventual W is considered as a resource leak. For a given W, we have the following partial order ($\prec_W$): $\{W^{\triangleright} \prec W^{\bullet},\ W^{\bullet} \prec W^{\triangleleft}\}$ as shown in Figure 3(d).
MPI_Barrier (B) is a blocking function and is used to synchronize MPI processes and has the following prototype `Barrier ()`. A process blocks after issuing the barrier until all the participating processes also issue their respective barriers. Note that unlike the traditional barriers used in threads where all the instructions before the thread barrier must also be complete when the barrier returns, the MPI B does not provide any such guarantees. An MPI B can be considered as a *weak fence* instruction. It is this behavior of MPI barriers that makes the traditional DPOR unsuitable for MPI (Figure 2). Given a B of a process, we have the following partial order ($\prec_B$): $\{B_{\triangleright} \prec B_{\Diamond} \prec B_{\bullet} \prec B_{\triangleleft}\}$ (shown in Figure 3(e)).

In addition to the above partial orders, we also have partial order rules for $S;W$ and $R;W$ (S may or may not be buffered) defined by providing the extra *coupling edges* between $\prec_S$ and $\prec_W$: $\prec_{SW} = \{S^{\bullet} \prec W^{\bullet}\}$, $\prec_{RW} = \{R^{\bullet} \prec W^{\bullet}\}$ As we show soon, this level of elucidation of MPI function states gives rise to an elegant formal semantics for it.

2.2 MPI Ordering Guarantees

We now describe various ordering guarantees for various MPI functions. These ordering guarantees provided by the MPI runtime according to the MPI standard define the order in which MPI program execution proceeds. We have already seen $\prec_S, \prec_{\underline{S}}, \prec_R \prec_W, \prec_B, \prec_{SW}, \prec_{RW}$ orders in Section 2.1 MPI also provides the following FIFO guarantees:

- For any two sends $S_{i,j}(l, \&h1),\ S_{i,k}(l, \&h2)$, $j < k$ from the same process i targeting the same destination (l), the first send $S_{i,j}$ is *matched-before* the second send $S_{i,k}$. Note that this order is irrespective of the buffering status of the sends.
- For any two receives $R_{i,j}(l, \&h1),\ R_{i,k}(l, \&h2)$ $j < k$ from the same process i receiving from the same source (l), the first receive is *matched-before* the second.

– For any two receives $R_{i,j}(*, \&h1)$, $R_{i,k}(l, \&h2)$, $j < k$ from the same process i, when the first receive $R_{i,j}$ can receive from any source (called wildcard receive and denoted as '$*$' henceforth), the first receive is always *matched-before* the second.

We call the above guarantees the *matches-before* ($\prec_{mb}$) ordering where $\prec_{mb} = \{S^{\Diamond}_{i,j}(l, \&h1) \prec S^{\Diamond}_{i,k}(l, \&h2), R^{\Diamond}_{i,j}(l, \&h1) \prec R^{\Diamond}_{i,k}(l, \&h2), R^{\Diamond}_{i,j}(*, \&h1) \prec R^{\Diamond}_{i,k}(l, \&h2)\}$. Matches-before is a fundamental ordering relation in MPI. The MPI standard requires that two messages sent from process i towards process j must arrive in the same order (called *non-overtaking* in MPI). The role of matches-before is to constrain the order in which sends and receives match so as to guarantee non-overtaking. Notice that the sends and receives need only to be matched in order. However, since the completion is only detected by a W, the MPI standard does not enforce any order on the completion of the sends and receives leaving the choice to the MPI library implementation.

Finally, for blocking MPI functions like W and B, any following MPI functions can be issued only after the blocking call is complete. We call this the *fence-order* ($\prec_{fo}$). Given an MPI process barrier $B_{i,j}$ or $W_{i,j}$ and a following MPI function $F_{i,k}$ where $F \in \{S, R, W, B\}$ and $(j < k)$, $\prec_{fo} = \{B^{\triangleleft}_{i,j} \prec F^{\triangleright}_{i,k}, W^{\triangleleft}_{i,j} \prec F^{\triangleright}_{i,k}\}$. In addition to the above orders we have the following issue order: For any two MPI function $M_{i,j}$ and $N_{i,k}$ where $M, N \in \{SR, B, W\}$ and $j < k$, we have $\prec_{io} = \{M^{\triangleright}_{i,j} \prec N^{\triangleright}_{i,j}\}$. The issue order precisely captures why B and W can block the issue of subsequent instructions.

Note that all the orders are defined on MPI functions within a process. We call the union of all the above orders the *mpi-intra-happens-before-order*, denoted as $\prec_{mhb}$.

Definition 1. MPI-Intra-Happens-Before-Order $\prec_{mhb} = \prec_S \cup \prec_{\underline{S}} \cup \prec_R \cup \prec_W \cup \prec_B \cup \prec_{SW} \cup \prec_{RW} \cup \prec_{mb} \cup \prec_{fo} \cup \prec_{io}$.

MPI-Intra-Happens-Before-Order defines the partial order of the salient MPI events observable at each MPI process.

2.3 Commit States

A commit state is a state that decides the fate of an MPI function. Since this happens to be when the MPI functions are matched, we call the $\Diamond$ state of each MPI function its commit state. Hence, we consider the following states as commit states: $commit = \{S^{\Diamond}, R^{\Diamond}, B^{\Diamond}\}$. Since W does not have a $\Diamond$ state (this is because W is a local process operation), we do not consider W to have a commit state.

In this paper, we do not define the full semantics of how the local memory of an MPI program gets updated; should such a definition be desired, we would take the commit events, consider them occurring according to $\prec_{mhb}$, and define the state transition semantics for the process memories. These notions will help define the formal semantics of MPI in subsequent sections.

3 MPI Formal Semantics for Verification

In Section 3.1, *communication records* that model the MPI runtime state of each MPI operation are introduced. In Section 3.2, we show how the operations of the MPI runtime cause an *IntraHB* ordering among the communication records. This relation is

obtained from the *mhb* relation defined earlier, except it is constructed over communication records and not MPI operations. We will then give transition rules that describe how MPI processes and their runtimes advance in their execution states. All this defines the *full* execution semantics of MPI. In Section 4.1, we present the POE algorithm elegantly by constraining the transition rules to fire in a certain priority order. It is well known that partial order reduction algorithms can be obtained by prioritizing transition systems in such a manner that a proper subset of executions (interleavings) is obtained. Section 4.2 provides soundness arguments.

3.1 Communication Records and Global State

Formalizing MPI requires the formalization of the processes and the runtime. We consider the MPI function invocations as *visible operations* [27] or simply *operations*, and any other operations performed by the processes as *invisible*. The MPI runtime helps advance the state of the process operations. We also assume that all `Isend`s (S) considered are non-buffered (ones for which the system does not provide adequate buffering, to cause their matching W to return immediately). Section 4.4 deals with buffering.

The PID (MPI rank) of each of P processes ranges over $\{0, \ldots, P-1\}$. The MPI runtime is assigned PID $R \notin \{0, \ldots, P-1\}$. A process i's operations are are numbered $\{1, \ldots n_i\}$, where the first operation is `Init` and the n_i^{th} operation is `Finalize`, as required by the MPI standard (we suppress them). The j^{th} operation executed by the i^{th} MPI process is $p_{i,j}$, where operation $p \in \{S, R, B, W\}$. A state s of an MPI operation $p_{i,j}$ is denoted as $p^s_{i,j}$ where $s \in \{\triangleright, \triangleleft, \Diamond, \bullet\}$. We denote a send and its arguments as $S_{i,j}(l, h_{ij})$ where i is the rank of the processes issuing the send and j is the dynamic operation count of process i. l is the destination where the message is to be sent, *i.e.*, $l \in PID$ and h_{ij} is the send's handle. The wait corresponding to the send is $W_{i,k}(h_{ij})$ where $k > j$. `Irecv` (R) also follows similar notations, except that the source argument of R $l \in PID \cup \{*\}$, where $*$ is a wildcard receive. The set of all possible handles $H = \cup_{i \in PID}\ ^h\{i\} \times \{1 \ldots n_i\}$. A process that executes an MPI operation creates a single communication record in the MPI runtime, denoted by the **eight tuple** $cr = \langle pid, pc, op, src, dest, handle, match, state \rangle$, where $pid \in PID$, $\forall i \in PID, pc \in \{1, \ldots, n_i\}$, $op \in \{S, R, W, B\}$, $src \in PID \cup \{*\}$, $dest \in PID$, $handle \in H$, $match$ is a set of communication records that forms a match set with cr, and $state \in \{\triangleright, \triangleleft, \Diamond, \bullet\}$. We use $\perp$ for a communication record field when its value is undefined. For example, src is undefined for sends, while the $dest$ field is undefined for recvs.

The communication record generated by $p_{i,j}$ is denoted by $c_{i,j}$, with its field f denoted $c_{i,j}.f$. Often, a communication record $c_{i,j}$ is considered synonymous with $p_{i,j}(\ldots)$ where $p \in \{S, B, W, R\}$. In fact, this connection is more than superficial: after it issues, the state of an MPI operation, as defined in Section 2, is defined by the state of the associated communication record. The dynamic state of the MPI program consists of the current set of communication records in the MPI runtime and the current PC (program counter) of the processes. Let C be the set of all possible communication records. The total state space of the system is: $\{1, \ldots, n_0\} \times \{1, \ldots, n_1\} \times \ldots \times \{1, \ldots, n_{P-1}\} \times 2^C$. The current state s is denoted as $\langle l, C \rangle$ where l is the tuple of all the current program counters. $l = \langle PC_0, PC_1, \ldots, PC_{P-1} \rangle$ where $PC_i \in \{1, \ldots n_i\}$ and C is the current set of communication records. l_i is used to access the program counter of process with rank i.

$$\mathbf{PS}^{\triangleright}: \frac{s:\langle l, C\rangle,\ \ p_{i,l_i} = S_{i,l_i}(j,\ h_{i,l_i})}{s':\langle l, C \cup \{\ \langle i, l_i, S, \perp, j, h_{i,l_i}, \perp, \triangleright\rangle\ \}\ \rangle} \qquad \mathbf{PR}^{\triangleright}: \frac{s:\langle l, C\rangle,\ \ p_{i,l_i} = R_{i,l_i}(j,\ h_{i,l_i})}{s':\langle l, C \cup \{\ \langle i, l_i, R, j, \perp, h_{i,l_i}, \perp, \triangleright\rangle\ \}\ \rangle}$$

$$\mathbf{PW}^{\triangleright}: \frac{s:\langle l, C\rangle,\ \ p_{i,l_i} = W_{i,l_i}(h_{i,j})}{s':\langle l, C \cup \{\ \langle i, l_i, W, \perp, \perp, h_{i,j}, \perp, \triangleright\rangle\ \}\ \rangle} \qquad \mathbf{PB}^{\triangleright}: \frac{s:\langle l, C\rangle,\ \ p_{i,l_i} = B_{i,l_i}}{\langle l, C \cup \{\ \langle i, l_i, B, \perp, \perp, \perp, \perp, \triangleright\rangle\ \}\ \rangle}$$

Fig. 4. MPI Process Transitions

$s.l$ is used to access the program counter tuple and $s.C$ is used to access the communication records in the current state.

3.2 Transition Systems

In this section we describe the process transitions and Runtime transitions that constitute our formal model for MPI.

Process Transitions: Figure 4 shows the process transitions for the four MPI functions that we have chosen to model. Since the processes *issue* the MPI functions into the MPI runtime, we annotate the name of each transition by $\triangleright$. $PS^{\triangleright}$ can be read as '**P**rocess-**S**end-Issue.' The state transition system shows the current state s on the top and the next state s' at the bottom. The initial state denoted by s_0 has l_i for every process i set to 1 and $C = \emptyset$. Every *issue* event results in the creation of a new communication record which is created and added to C in the next state s'. We describe the $PS^{\triangleright}$ transition in detail. For $PS^{\triangleright}$, the current state s is $\langle l, C\rangle$ and p_{i,l_i} denotes the MPI operation invoked by process i at the PC denoted by l_i. $S_{i,l_i}(j, h_{i,l_i})$ is the send being issued where j is the destination to which the message is to be sent. The transition creates a new communication record whose *pid* is i, *pc* is l_i, *op* is S, and *dest* is j. The *src* and *handle* fields are set to $\perp$. Finally, the state is set to $\triangleright$ denoting that the send is in the *issued* state. The rest of the transitions can be understood similarly.

MPI Runtime Transitions perform the actual message passing and synchronization operations. However, the transitions must obey the partial order rules described in Section 2. Given the set of communication records, we construct the *IntraHB* graph at every state s that follows from the $\prec_{mhb}$, as described now. The *IntraHB* relation helps define the MPI semantics, mainly by defining how MPI commands match.

Definition 2. *For a state $s = \langle l, C\rangle$, the graph $s.IntraHB = (V, E)$, where $V = s.C$. Further, for communication records $c_{i,j}$ and $c_{i,k}$, with $j < k$, $\langle c_{i,j}, c_{i,k}\rangle \in E$ iff one of these holds:*

- $c_{i,j}.op = c_{i,k}.op = S \wedge c_{i,j}.dest = c_{i,k}.dest$
- $c_{i,j}.op = c_{i,k}.op = R \wedge (c_{i,j}.src = * \vee c_{i,j}.src = c_{i,k}.src)$
- $c_{i,j}.op = S \wedge c_{i,k}.op = W \wedge c_{i,k}.handle = h_{ij}$
- $c_{i,j}.op = R \wedge c_{i,k}.op = W \wedge c_{i,k}.handle = h_{ij}$
- $c_{i,j}.op = B \vee c_{i,j}.op = W$

The edges between the communication records are added based on the partial orders defined in Section 2. It is called as an *IntraHB* (Intra Happens Before) graph since the edges are between the communication records within the process. $c_{i,j}$ and $c_{i,k}$ are two communication records of process i. The first condition in Definition 2 is to satisfy the ordering relation $S^{\Diamond}_{i,j}(l, h_{ij}) \prec S^{\Diamond}_{i,k}(l, h_{i,k})$. The second condition is to satisfy the ordering conditions $R^{\Diamond}_{i,j}(l, h_{i,j}) \prec R^{\Diamond}_{i,k}(l, h_{ik})$, and $R^{\Diamond}_{i,j}(*, h_{i,j}) \prec R^{\Diamond}_{i,k}(l, h_{ik})$. The

third and fourth conditions satisfy the ordering conditions $S^{\bullet}_{i,j}(l, h_{ij}) \prec W^{\bullet}i, k(h_{ij})$ and $R^{\bullet}_{i,j}(l, h_{ij}) \prec W^{\bullet}i, k(h_{ij})$ respectively. The last condition is to satisfy the $\prec_{fo}$ (*fence-order*).

Definition 3. *For communication records $c_{i,j}$ and $c_{i,k}$ and state s, if $\langle c_{i,j}, c_{i,k}\rangle \in E$ where E is the second component of $s.IntraHB$, we refer to $c_{i,j}$ as an* **ancestor** *of $c_{i,k}$.*

Figure 6 defines the POE algorithm in terms of execution steps termed as *execute*. For any state s in this execution beginning with the initial program state s_0, we define the set $s.C_R$ associated with s to be those communication records that have crossed through their commit points. The update rule to obtain $s_i.C_R$ (the C_R set of the current state s_i) from $s_{i-1}.C_R$ (the C_R set of the previous state) is the following:

Definition 4. $s_0.C_R = \emptyset$, *and for all* $i > 0$, $s_i.C_R = s_{i-1}.C_R \cup \{c_{i,j} \in s_i.C \mid c_{i,j}.state = \Diamond\}$.

Since W's commit point is the same as completion, we do not add W to C_R and instead add it to C_{cpl}, where $s.C_{cpl}$ are the set of communication records that are *completed* by the MPI runtime. The completion criteria is different for different MPI functions as described in Section 2.

Definition 5. $s_i.C_{cpl} = s_{i-1}.C_{cpl} \cup \{c_{i,j} \in s_i.C \mid c_{i,j}.state = \bullet\}$.

Definition 6. *For a given state $s = \langle l, c\rangle$ let $c_{i,k} \in s.C$ and $s.IntraCB = (V, E)$. Define $c_{i,k} \in C_{\downarrow}$ iff for every $c \in Ancestors(c_{i,k})$ one of the following is true:*

- $c.op = S \wedge c_{i,k}.op = W \wedge c \in s.C_{cpl}$
- $c.op = R \wedge c_{i,k}.op = W \wedge c \in s.C_{cpl}$
- $c.op = S \wedge c_{i,k}.op = S \wedge c \in s.C_R$
- $c.op = R \wedge c_{i,k}.op = R \wedge c \in s.C_R$
- $c.op = B \wedge c \in s.C_{cpl}$
- $c.op = W \wedge c \in s.C_{cpl}$

$s.C_{\downarrow}$ is the set of communication records where all the ancestors of every communication records are either completed or matched (depending on the MPI operations), *i.e.*, the ancestors have crossed their commit points. Hence, $s.C_{\downarrow}$ is the set of communication records that can now be matched or completed without violating $\prec_{mhb}$. It must be noted that each of the condition in Definition 6 satisfies $\prec_{mhb}$. Our semantics, in effect, defines the set of all allowed executions of MPI programs.

We now define the MPI runtime transitions in Figure 5. We employ a convenient notational abbreviation introduced through a simple example:

- For a set s and an item x, let $s + x$ denote $s \cup \{x\}$.
- Let $C : c_{x,i}[match \leftarrow @ + h_{y,j}]$ stand for "the set C except that the member $c_{x,i}$ in it has its match component updated by the addition of $h_{y,j}$" . Here, @ stands for $c_{x,i}.match$ (a notation inspired by TLA+).

RR (**R**untime-**R**eceive) transition matches a send and matching receive. In the current state $s = \langle l, C\rangle$, consider $c_{x,i}$ and $c_{y,j}$ such that both communication records are in $s.C_{\downarrow}$, *i.e.*, both the communication records are ready to be matched since all their ancestors

$$\textbf{RR}: \frac{s:\langle\ l, C\ \rangle,\ c_{x,i}, c_{y,j} \in s.C_{\downarrow},\ c_{x,i} = R_{x,i}(y, h_{x,i}),\ c_{y,j} = S_{y,j}(x, h_{y,j}),\ y \in PID}{s':\langle l, C : c_{x,i}[match \leftarrow @ + c_{y,j}, state \leftarrow \Diamond]\ \rangle}$$

$$\textbf{RR*}: \frac{s:\langle\ l, C\ \rangle,\ c_{x,i}, c_{y,j} \in s.C_{\downarrow}, c_{x,i} = R_{x,i}(*, h_{x,i}),\ c_{y,j}.op = S_{y,j}(x),\ y \in PID}{s':\langle l, C : c_{x,i}[match \leftarrow @ + c_{y,j}, src \leftarrow y, state \leftarrow \Diamond]\rangle}$$

$$\textbf{RS}: \frac{s:\langle\ l, C\ \rangle,\ c_{i,j} \in s.C_{\downarrow}, c_{k,l} \in s.C_{R},\ c_{i,j} = S_{i,j}(k, h_{i,j}),\ c_{k,l} = R_{k,l}(i, h_{k,l}),\ c_{k,l}.match = \{c_{i,j}\}}{s':\langle l, C : c_{i,j}[match \leftarrow @ + c_{k,l}, state \leftarrow \Diamond]\ \rangle}$$

$$\textbf{RB}: \frac{s:\langle\ l, C\ \rangle,\ C_1 \subseteq s.C_{\downarrow},\ |\ C_1\ |= P,\quad \forall c_{i,j} \in C_1 :\ c_{i,j} = B_{i,j}}{s':\langle\ l, C : \forall c_{i,j} \in C_1\ :\ c_{i,j}[match \leftarrow @ + C_1, state \leftarrow \Diamond]\ \rangle}$$

$$\textbf{RW}: \frac{s:\langle\ l, C\ \rangle,\ c_{i,k} \in s.C_{\downarrow},\ c_{i,k}.op = W}{s':\langle\ C : c_{i,k}[state \leftarrow \bullet]\ \rangle} \qquad \textbf{R}{-}^{\bullet}: \frac{s:\langle\ l, C\ \rangle,\ c_{i,k} \in s.C_{R}, c_{i,k}.op = R/S/B}{s':\langle\ l, C : c_{i,k}[state \leftarrow \bullet]\ \rangle}$$

$$\textbf{RR}^{\triangleleft}: \frac{s:\langle\ l, C\ \rangle,\ c_{i,k} \in C,\ c_{i,k}.op = R}{s':\langle\ l[l_i \leftarrow l_i + 1], C : c_{i,k}[state \leftarrow \triangleleft]\ \rangle} \qquad \textbf{RS}^{\triangleleft}: \frac{s:\langle\ l, C\ \rangle,\ c_{i,k} \in C,\ c_{i,k}.op = S}{s':\langle\ l[l_i \leftarrow l_i + 1], C : c_{i,k}[state \leftarrow \triangleleft]\ \rangle}$$

$$\textbf{RW}^{\triangleleft}: \frac{s:\langle\ l, C\ \rangle,\ c_{i,k} \in s.C_{cpl},\ c_{i,k}.op = W}{s':\langle\ l[l_i \leftarrow l_i + 1], C : c_{i,k}[state \leftarrow \triangleleft]\ \rangle} \qquad \textbf{RB}^{\triangleleft}: \frac{s:\langle\ l, C\ \rangle,\ c_{i,k} \in s.C_{cpl},\ c_{i,k}.op = B}{s':\langle\ l[l_i \leftarrow l_i + 1], C : c_{i,k}[state \leftarrow \triangleleft]\ \rangle}$$

Fig. 5. MPI Runtime Transitions

have either been matched or completed. If $c_{x,i}$ is $R_{x,i}(y, h_{x,i}$ *i.e.* it is a receive trying to receive a message from process y and $c_{y,j}$ is $S_{y,j}(x, h_{y,j})$ is a send that matches the receive, then the send is matched with the receive which is the new state s' where $c_{x,i}$'s *match* is updated. Note that it is acceptable to update either the send or receive match. We just chose to show the runtime transition with a receive match being set. We could also have set both the send and receive matches at the same time. However, our attempt is to keep the MPI runtime as generic as possible to allow all possible sets of interleavings. Note that the receive is set to the $\Diamond$ state but not to the $\bullet$ state. This is because the receive buffer is still not filled with the sent data. This also allows any following receive to be matched and completed before $c_{x,i}$ itself is completed. In other words, as soon as a receive 'chooses its partner,' the following receive can choose its partner.

RR* (**R**untime-**R**eceive-wildcard) transition is similar to RR transition except that the *src* field of the receive which was a wildcard receive previously is now replaced by y which is the matching send process rank. This models 'dynamic source rewriting for wildcard receives, as was illustrated in Section 1. Strictly speaking, **RR*** is a *family* of transitions involving one receiver and its matching senders; we however choose to view it as a single transition for convenience.

RS (**R**untime-**S**end) transition is similar to RR transition except that the receive's *match* set is already populated with the send's communication record. The matching receive is searched in the $s.C_R$ set because a previously fired RR/RR* transition has set the receive to the $\Diamond$ state which, in effect, moves the receive into the C_R set. The matching receive is found by comparing the receive's *match* to the send's communication record.

We only set the send's match field and move the send to the $\Diamond$ state to keep the runtime as generic as possible as described in RR transition.

RW (**R**untime-**W**ait) transition is in $s.C_{\downarrow}$ if its corresponding send or receive has been completed as described in Definition 6. Since the send (receive) is complete, the W can also be completed.

RB (**R**untime-**B**arrier) transition checks that the $C_1 \subseteq C_{\downarrow}$ where is C_1 has only barriers and the number of barriers is equal to P (number of MPI processes). That is, it checks that all the processes have issued their barriers. In that case, the barriers are moved to the $\Diamond$ state. The barriers are only moved to the $\Diamond$ state and not the $\bullet$ state even though there is no actual data to transfer. The reason for this is to seamlessly support other MPI collective functions like barriers that also have data to be transferred (e.g. `MPI_Bcast`). In such a case, the completion or the transfer of data can happen later. It can be seen that our runtime transitions can readily be applied to other MPI functions.

R-$^\bullet$ (**R**untime-any-help-complete) are the runtime transitions to complete any transitions that have been matched, *i.e.* are in C_R. The S, R and B that are matched can be moved to complete state. Note that there is a causal order between the RR(S/B)/RR* and RR(S/B)$^\bullet$ transitions for a specific send/receive/barrier.

The RS$^\bullet$ transition will be different from what is presented here *when the MPI runtime can buffer sends*. In that case, instead of checking that $c_{k,l} \in s.C_R$, we must check that $c_{k,l} \notin s.C_{cpl}$. Since the send is already completed (the send buffer is copied into the MPI runtime) when the send is buffered, it is already in the $s.C_{cpl}$ set. Because of this early copying of the send buffer, a buffered send need not be moved from state $\Diamond$ to $\bullet$. Hence, when trying to move a send from state $\Diamond$ to $\bullet$, we must first ensure that the send is not buffered *i.e.* the send is not yet completed.

RR$^\triangleleft$ (**R**untime-**R**eceive-return) and **RS**$^\triangleleft$ (**R**untime-**S**end-return) transition increments the program counter and sets the state to $\triangleleft$. Note that the receive/send can return at any time irrespective of their current state because they are non-blocking operations.

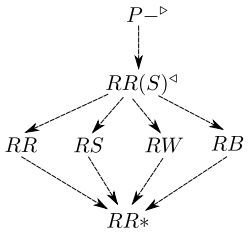
RW$^\triangleleft$ (**R**untime-**W**ait-return) and (**R**untime-**B**arrier-return) **RB**$^\triangleleft$ transitions also increment the PC and set the state to $\triangleleft$. However, note that there is a causal order from RW to RW$^\triangleleft$ and RB to RB$^\triangleleft$.

It can be easily verifed that the $\prec_{mhb}$ order is respected by all the transitions discussed in this section.

4 POE Algorithm

The reachable state space generated by firing the MPI transitions (both process and runtime) described in Section 3 describes the full reachable state space of a given MPI program. The POE algorithm is obtained by erecting a partial order (priority order) among the runtime transitions, as shown in Figure 6, where an arrow point from a higher priority transition to a lower priority transition.

In Figure 6, $P{-}^\triangleright$ stands for the process issue transitions (Notation: $P-$ abbreviates PS, PR, etc.) This means that all processes will first try to issue transitions through the $P-$ moves. When no such issue is possible, we can entertain the $R-$ moves (runtime transitions). The runtime transitions for send and receive return (namely $RR^\triangleleft$, $RS^\triangleleft$) are next in the priority order and have the same priority. This means that every send/receive issued immediately returns since the return transitions increment the program counter.




```
1:  POE(s) {
2:    if (s is a final state) return;
3:    t = getHighestPriorityTrans(s);
4:    if (t is not an R* transition) {
5:      s' = execute(s,t); POE(s');
6:    } else {
7:      for all (t in R*TransitionsAtState(s)) {
8:        s' = execute(s,t); POE(s'); }}}}
```

Fig. 6. The Priority Order for getHighestPriorityTrans, and the POE$_{MSE}$ algorithm

The above two priorities also means that the POE algorithm lets each of the processes execute until they reach the blocking call (W or B) and then executes any runtime transitions. This is because there are no more process transitions available until the blocking calls complete and return. POE combines the RB and the corresponding $RB^{\triangleleft}$ transitions into a single RB transition. Similarly, RW and corresponding $RW^{\triangleleft}$ transitions are shown in Figure 6 as RW. Once the blocking calls, return, the processes all execute until they again reach their blocking points. The RS and RR transitions are the combined RS, corresponding $RS^{\bullet}$ and RR together with the corresponding $RR^{\bullet}$. The RR, RW, RS and RB transitions have the same priority. Finally, the $RR*$ ($RR*$ together with corresponding $RR^{\bullet}$) transition has the lowest priority. This means that the $RR*$ transition is only executed when there are no other transitions left. This was illustrated in Section 1 by postponing the wildcard receives till all senders were discovered, so that one can perform the maximally diverse extent of rewrites. The **for** loop on Line 7 shows that we will process even (seemingly) unrelated **RR*** transition families together. This is important for soundness, to ensure the C1 condition of [27, Chapter 10], as follows: firing one of the **RR*** transitions may enable a send transition that now matches another **RR*** transition. We now illustrate the working of POE algorithm on a simple example.

4.1 Illustration of POE

Consider the MPI program shown in Figure 7. The two sends $S_{0,1}$ and $S_{1,2}$ can match the wildcard receive $R_{2,1}$. The POE algorithm first executes the process transitions corresponding to $S_{0,1}$, $B_{1,1}$ and $R_{2,1}$. At this point, process P_2 gets blocked on the barrier call. However, the send and receive both return to now execute their $B_{0,2}$ and $B_{2,2}$ calls. At this point all the processes are blocked on their barrier calls and no more process transitions are available to be executed. Either RR* transition that matches $S_{0,1}$ to $R_{2,1}$ or RB transition that matches and returns the three barriers can be executed. Using the priority order, since the RB transition has a higher priority, the barriers get matched and return. The RR* transition is still not executed by the runtime. Since all the processes now return from their barriers, the processes start issuing new instructions, since the process transitions have a higher priority than the RR* transition. The $S_{1,2}$ is issued and the W operations of all the processes are issued. Since W is a blocking call, all the processed block at their W operations. At this point, the only transitions are the two RR* transitions where one transition matches $R_{2,1}$ with $S_{0,1}$ and the other matches $R_{2,1}$ with $S_{1,2}$. The POE algorithm executes one interleaving by matching $R_{2,1}$ with $S_{0,1}$ by

re-writing the source of receive to 0. The interleaving also matches $R_{2,4}$ with $S_{1,2}$. It then re-starts the process and re-executes the interleaving this time by matching $R_{2,1}$ with $S_{1,2}$. But now, since $R_{2,1}$ is expecting a send from process 1 and there is no such send available, *the program deadlocks* which is detected by POE. The example illustrates how POE algorithm prioritizes the MPI transitions so that it is able to find all the matching transitions of a wildcard receive.

4.2 Proof of Correctness

Theorem 1. *Assuming non-buffered sends, POE finds all possible deadlocks.*

Proof. The only source of non-determinism is due to wildcard receives. To show that we detect all deadlocks, it is sufficient to prove that we detect all possible sends that can match a wildcard receive (*i.e.*, ample sets are correctly built according to C1, [27, Chapter 10]). We prove this by contradiction. Assume that there is a send S_i that can match the wildcard receive R_j under a native MPI program execution, but somehow, due to our priority ordering, is not matched by the POE algorithm. This means that when the RR* transition involving R_j is executed (matching another send, say S_i'), the S_i is not yet issued (if it were issued, then the $RR*$ rule would have picked it). Also, since $RR*$ transition is of the lowest priority, this means that there are no other higher priority transitions that could meanwhile have been executed. Thus, in particular, S_i is blocked at a B or a W instruction of the process of S_i. But then, even a native execution could also not get past this B or W and provide a match to $RR*$.

4.3 Implementing POE

POE can be supported by any MPI-standard compliant MPI library. This is because ISP forwards MPI calls to the MPI runtime only when they are ready to match. One can consider ISP as an auxiliary runtime that uses the MPI runtime just to transport messages, and ensure the progress of the MPI processes. The priority order of POE has the run-time effect of reshuffling the MPI commands – but in a manner that does not violate *IntraHB*. Clearly, no one can be sure of the exact *IntraHB* intended by the original MPI designers or (tacitly) assumed by the scores of MPI users merely by reading [7] or studying particular MPI library code bases. However, there is sufficient evidence that we have indeed unearthed this *IntraHB* relation by the fact that we could effortlessly port ISP to run on three operating systems (Linux, MAC OS/X, and Windows), and on four MPI libraries (MPICH2, OpenMPI, Microsoft MPI, and MVAPICH MPI). Further engineering details of ISP are described in [9,10].

4.4 Send Buffering Issues

The POE algorithm works only when the sends do not have adequate (system-provided or user-provided) buffering. However, if sends can be buffered, it can miss deadlocks present in a program. Consider the MPI example shown in Figure 8.

When none of the sends are buffered, only $S_{1,1}$ can match the wildcard receive $R_{2,1}$ and there is no deadlock. However, when $S_{1,1}$ is buffered, the $W_{1,2}$ can complete even before the send is matched. This enabled the the $S_{0,1}$ and $R_{1,3}$ to match and since $S_{0,1}$ is matched, it can complete unblocking the $W_{0,2}$. Now, $S_{0,3}$ is issued and since the wildcard receive is not yet matched, it can be matched with $S_{0,3}$ and result in a deadlock

P_0	P_1	P_2
$S_{0,1}(2, h_{01})$	$B_{1,1}$	$R_{2,1}(*, h_{21})$
$B_{0,2}$	$S_{1,2}(2, h_{12})$	$B_{2,2}$
$W_{0,3}(h_{01})$	$W_{1,3}(h_{12})$	$W_{2,3}(h_{21})$
		$R_{2,4}(1, h_{24})$
		$W_{2,5}(1, h_{25})$

Fig. 7. "Crooked" Barrier Example

P_0	P_1	P_2
$S_{0,1}(1, h_{01})$	$S_{1,1}(2, h_{11})$	$R_{2,1}(*, h_{21})$
$W_{0,2}(h_{01})$	$W_{1,2}(h_{11})$	$W_{2,2}(h_{21})$
$S_{0,3}(2, h_{03})$	$R_{1,3}(0, h_{13})$	$R_{2,3}(0, h_{23})$
$W_{0,4}(h_{03})$	$W_{1,4}(h_{13})$	$W_{2,4}(h_{23})$

Fig. 8. Buffering Sends and Deadlocks

since $R_{2,3}$ will not have a matching send. Note that this deadlock cannot happen when none of the sends are buffered. We call this the *slack inelastic property* [28] of MPI. One solution would be to buffer all the sends. However, this will mean that any deadlocks corresponding to non-buffered sends will not be detected by POE. Since buffer allocation is a dynamic property, our goal is to extend POE so that it can detect all forms of deadlocks. Currently, we are working on a slack independent error detection algorithm that improves upon POE.

5 Concluding Remarks

In this paper, we presented a formal semantics for a subset of four MPI operations. Our semantics first characterize the states through which each MPI function call go through. It then describes the MPI runtime transitions that help progress the issued MPI functions through their states. The crux of our definitions was the identification of the relations MPI Intra Happens Before ($\prec_{mhb}$) as well as the *IntraHB* graph. Basically these relations capture how MPI guarantees the non-overtaking property. Our definitions reveal the full generality of executions admitted by all MPI standard compliant libraries.

We take our formal semantic definitions and simply introduce a priority of firing of the MPI runtime rules. This gives our MPI-specific partial order reduction algorithm POE. We also have implemented our ISP tool by closely following our formal semantics.

The broader lessons from our work are that real world APIs such as MPI are really complex. Yet, by discovering the essential states through with each API call goes through, one can set up elegant state transition systems that then help guide reduction algorithms and implementations. Given the push towards multi-core CPUs and the general parallelism "feeding frenzy," many APIs are being defined by various groups. Our ideas may play a role in developing dynamic verification tools for these APIs also. This fact has been reaffirmed not only through ISP but through another line of recent work [29] on developing a dynamic formal verifier for applications written using the recently proposed Multi-core Communications API (MCAPI [30]).

References

1. Patrice Godefroid. Model Checking for Programming Languages using Verisoft. POPL, 1997, 174-186.
2. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121. ACM, 2005.

3. The Java Pathfinder. http://javapathfinder.sourceforge.net
4. Madan Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Pages 446–455, 2007.
5. CHESS: Find and Reproduce Heisenbugs in Concurrent Programs. http://research.microsoft.com/en-us/projects/chess/
6. Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Chao Wang. Automatic Discovery of Transition Symmetry in Multithreaded Programs using Dynamic Analysis. SPIN 2009, Grenoble, June 2009.
7. MPI Standard 2.1. http://www.mpi-forum.org/docs/.
8. S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. *CAV 2008.*
9. A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, , and R. Thakur. Formal verification of practical mpi programs. *PPoPP 2009.*
10. S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, and R. M. Kirby. Scheduling considerations for building dynamic verification tools for MPI. *PADTAD-VI 2008.*
11. Sarvani Vakkalanka et al. Static-analysis Assisted Dynamic Verification of MPI Waitany Programs (Poster Abstract). Accepted in EuroPVM/MPI, September 2009.
12. Anh Vo et al. Sound and Efficient Dynamic Verification of MPI Programs with Probe Non-Determinism. Accepted in EuroPVM/MPI, September 2009.
13. R. Palmer, M. DeLisi, G. Gopalakrishnan, R. M. Kirby. An Approach to Formalization and Analysis of Message Passing Libraries *FMICS 2008, 164–181*
14. Guodong Li et al. Formal Specification of the MPI 2.0 Standard in TLA+. Under Submission. http://www.cs.utah.edu/formal_verification/mpitla/.
15. Leslie Lamport. Specifying Systems: The TLA Language and Tools. Addison-Wesley, 2004.
16. G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *SuperComputing (SC)*, 1996.
17. S. F. Siegel and A. R. Siegel. MADRE: The Memory-Aware Data Redistribution Engine. *EuroPVM/MPI 2008.*
18. Workshop on Exploiting Concurrency Efficiently and Correctly, Grenoble, June 2009. Discussion of Challenge Problems. http://www.cs.utah.edu/ec2.
19. P. Pacheco Parallel Programming with MPI *Morgan Kaufmann, 1996, ISBN 1-55860-339-5*
20. Michael DeLisi. Umpire Test Suite Results using ISP. http://www.cs.utah.edu/formal_verification/ISP_Tests/
21. Leslie Lamport. Time Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM. 21(7):558-564, 1978.
22. S. F. Siegel. Efficient Verification of Halting Properties for MPI Programs with Wildcard Receives. *VMCAI 2006, 413–429*
23. P. Georgelin, L. Pierre, and T. Nguyen. A Formal Specification of the MPI Primitives and Communication Mechanisms. Rapport de Recherche LIM 1999-337, Marseille, Oct 1999.
24. http://www.cs.utah.edu/formal_verification/ISP-release/.
25. mpiBLAST: Open-Source Parallel BLAST. http://www.mpiblast.org/.
26. The IRS Benchmark Code. https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/.
27. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.
28. R. Martin and A. Manohar. Slack Elasticity in Concurrent Computing *n Intl Conf. on Mathematics of Program Construction. LNCS 1422*
29. Subodh Sharma, Ganesh Gopalakrishnan, Eric Mercer, and Jim Holt. MCC - A runtime verification tool for MCAPI user applications. Accepted in FMCAD 2009, Austin.
30. The Multicore Communications API (MCAPI). http://www.multicore-association.org