A Sound Reduction of Persistent-sets for deadlock detection in MPI Applications

Subodh Sharma[1], Ganesh Gopalakrishnan[1], and Greg Bronevetsky[2]

[1] School of Computing University of Utah, Salt Lake City, Utah 84112
{svs,ganesh}@cs.utah.edu
[2] Lawrence Livermore National Laboratory, Livermore, CA 94551 greg@bronevetsky.com

Abstract. Formal dynamic analysis of MPI programs is crucially important in the context of developing HPC applications. Existing dynamic verification tools for MPI programs suffer from exponential schedule explosion, especially when multiple non-deterministic receive statements are issued by a process. In this paper, we focus on detecting message-orphaning deadlocks within MPI programs. For this analysis target, we describe a sound heuristic that helps avoid schedule explosion in most practical cases while not missing deadlocks in practice. Our method hinges on initially computing the potential non-deterministic matches as conventional dynamic analyzers do, but then including only the entries which are found *relevant* to cause a refusal deadlock (essentially a macroscopic-view persistent-set reduction technique). Experimental results are encouraging.

1 Introduction

The Message Passing Interface (MPI, [16]) is one of the central APIs used in large-scale high performance computing (HPC) simulations. Most of today's supercomputers and high performance clusters are programmed using MPI, and this trend is expected to continue [12]. There are also embedded system communication standards built around message passing, such as MCAPI [15]. In this paper, we study the problem of adequately testing message passing programs using formal techniques. While our research is conducted with MPI-specific details, with relatively minor modifications our results also apply to other message passing paradigms.

In MPI, message send commands directly address the destination process while message receives are of two types: either directly address the source process (called *deterministic* or *specific receives*) or may be *non-deterministic* (or "wildcard") receives that can receive from any process. The sends and receives issued by MPI processes that target the same destination or source from the same process are required to match in issue order (the "non-overtaking rule of MPI"). At any runtime state of an MPI program, all eligible matches between deterministic receives and sends—specifically such *match pairs*—commute. This is because all such match pairs have non-overlapping destinations/sources. That is, for deterministic receives R_i and R_j, one can match (S_i, R_i) and (S_j, R_j) in either order. However, non-deterministic receive matches do not, in general, commute: at the very least, a non-deterministic receive $R(*)$ matching a send S_i results in a system state different from when another send S_j matches the same receive.

This is not good news for dynamic partial order reduction (DPOR [8]) methods because in many MPI programs, $R(*)$ calls occur in sequence (typically in a loop). Thus, it seems as if any DPOR technique is doomed to examine an exponential number of interleavings—something that does not bode well for our *Exascale* computing aspirations in which several message passing APIs (including MPI) are expected to play an important role. This paper develops a simple but very effective (in practice) heuristic that avoids this explosion in many cases.

Background and Related Work: It is important to have a balanced portfolio of verification tools in any area—including for MPI. Informal testing approaches for MPI (e.g., based on schedule perturbation [25]) do not guarantee coverage, and are also highly redundant because they will, in practice, generate many equivalent schedules (e.g., permuting deterministic message matches). While static analyzers for MPI exist (e.g., [1]), they are known to be unsound (can generate too many false alarms) when used for bug-hunting, due to their overapproximation of possible message matches. Model-checking based methods (e.g., MPI-SPIN [19]) can guarantee coverage, but on *models* of MPI programs; such models are very difficult to create, and become obsolete with each design change.

From a designer's perspective, dynamic formal testing tools are attractive in many ways: (1) they are sound, (2) they can be made complete with respect to non-determinism coverage. Formal dynamic verifiers such as ISP [21, 24] and DAMPI [22, 23] take an approach that integrates the best features of testing tools (ability to run on user applications) and model checking (message match non-determinism coverage guarantees). They run the MPI program under the control of *verification-oriented scheduling mechanisms* (a central scheduler for ISP and logical clocks for DAMPI). Thanks to their MPI semantics-aware algorithms, these tools guarantee non-determinism coverage (e.g., all the potential matches of a non-deterministic receive) while not bloating the schedule space with respect to deterministic receive/send matches. They have been shown to scale up to 1000 MPI processes for many MPI programs (in the case of DAMPI). The scheduling mechanisms in these tools are robust across all MPI-compliant platforms and computational delays between communication calls. To the best of our knowledge, there are no other dynamic analysis tools similar to ISP and DAMPI. *However, these dynamic verification tools suffer from the aforesaid exponential schedule explosion when a sequence of $R(*)$ commands are issued.* Therefore, a practical dynamic verification tool that avoids this schedule explosion and provides reasonable coverage is currently unavailable. This paper describes such a tool.

Contributions: In this paper, we focus on the problem of detecting *orphaning deadlocks* in which an MPI receive is left without a matching send in some MPI program execution state. Our solution is to modify ISP's dynamic partial order reduction algorithm called POE (standing for Partial Order avoiding Elusive interleavings, [21]) to result in a new algorithm called MSPOE (*MacroScopic* POE). MSPOE applies to MPI programs that "do not decode data," i.e., do not employ data dependent control flows, and do not alter their control flows based on *which* sends a non-deterministic receive matches with. The formulation of MSPOE relies on a notion of *commuting sends*; this notion results from a macroscopic re-interpretation of the basic tenets of partial order reduction. We measure the efficacy of MSPOE on *real* examples, and show that

MSPOE can dramatically reduce the number of interleavings examined. *MSPOE is, by design, incomplete.* In practice, MSPOE has caught all the deadlock bugs that ISP has. A study of any successful large-scale formal software testing or analysis approach (e.g. [11]) shows that rather than aiming for a theoretically complete algorithm, one almost always has to aim for "completeness in practice."[3]

As an example (Figure 1), consider an MPI program with $n+1$ processes where each of the n processes sends a message to the $(n+1)^{th}$ process. The $(n+1)^{th}$ process posts n wildcard receive calls (say in a loop). If we want to completely cover all possible non-deterministic message matches, we will be forced to examine $n!$ execution schedules.

Observation: Let a call denoted by $S_{i,j}(k)$ is a send call from process i sending to process k with the local process program counter (PC) at j. Similarly a receive call sourcing from process k which is issued by process i indexed at j is denoted by $R_{i,j}(k)$. A non-deterministic receive is represented by $R_{i,j}(*)$. We will use this notation through the rest of the paper. For an MPI program that does not decode data and has an orphaned deterministic receive causing a deadlock, either there must be an unequal number of sends and receives in some execution path, or the following two conditions must be satisfied: (1) It employs a process that issues a wildcard receive $R_{k,l}(*)$ followed by a specific receive $R_{k,l'}(j)$; (2) In the execution order, two other processes supply sends $S_{i,n}(k)$ and $S_{j,m}(k)$. $R_{k,l}(*)$ consumes $S_{j,m}(k)$,"the send that was meant for the later appearing $R_{k,l'}(j)$". This will lead to orphaning of $R_{k,l'}(j)$ since there are no more sends that can match this receive. MSPOE works as follows. It first uses POE to compute the potential send matches for each MPI non-deterministic receive. It then chooses to include only *some of these sends* (called *relevant sends*—relevant for creating orphaning deadlocks).

Additional Related Work: One may initially think that our problem is one of symmetry detection, which has been extensively researched [2, 13, 6, 3]. Symmetry detection is based on constructing a smaller *quotient structure* of the system by exploiting the *automorphism* in the system's state space. These are computationally hard problems [2] which are impractical during dynamic verification of MPI programs. The work in [5] computes symmetries in communicating programs based on channel graphs and not directly applicable for our purposes. We have not come across any other effort where a simple (but highly effective in practice) approach such as MSPOE is studied.

Detailed look at some examples: In Figure 1, the ISP scheduler will explore six interleavings for this example. The six interleavings are illustrated in Figure 2. Note that solid circles are the states and the directed edges are the match-sets (sets of matching operations at a state) signaled to the runtime at that state.

The dotted arrow edges is the first interleaving that ISP explores. However, observe that the example code has only wildcard receive calls. Thus, as long as all sends *commute*, such examples cannot have deadlocks and there is no necessity to examine other schedules. MSPOE will analyze the program in Figure 1 in the following way. MSPOE will explore the first interleaving as shown by dotted arrows in Figure 2. It will discover that it did not encounter any deterministic receive calls. Thus, MSPOE will reduce the persistent-set (refer [9]) of each non-deterministic receive to a singleton set (containing

[3] In practice, it seems one can obtain at most two of the following three attributes: *sound, complete, scalable.*

P_0	P_1	P_2
$S_{0,1}(2)$;	$S_{1,1}(2)$;	$for(i = 1 \text{ to } 4)$
		$R_{2,i}(*)$;
$S_{0,2}(2)$;	$S_{1,2}(2)$;	end for;

Fig. 1. Deadlock free example

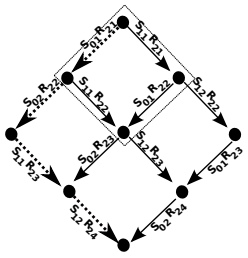

Fig. 2. State graph for Figure 1

P_0	P_1	P_2	P_3	P_4
$S_{0,1}(4)$;	$S_{1,1}(4)$;	$S_{2,1}(4)$;	$S_{3,1}(4)$;	$R_{4,1}(*)$;
				$R_{4,2}(3)$;
				$R_{4,3}(*)$;
				$R_{4,4}(*)$;

Fig. 3. Deadlocking example

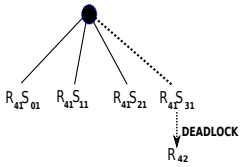

Fig. 4. Situation after first R(*) match

the entry that was explored in the current run of the program). (Note that in the Figure 2, the states that are within the dotted box will witness their persistent-set reduced. For the rest of the states, the persistent-set is a singleton-set to begin with.) Since, each persistent-set has already been reduced to singleton set by MSPOE, the ISP scheduler's search will terminate with only one interleaving.

In the example of Figure 3, there is a deadlock introduced by the use of the deterministic receive call. as shown in Figure 4. In Figure 4 if $R_{4,1}$ were to match $S_{3,1}$ (rightmost transition from the initial node), the subsequent deterministic call ($R_{4,2}$) will be orphaned, thus creating a refusal deadlock. ISP would explore all the matches starting from leftmost choice shown in Figure 4 and then moving right with every new run, generating four interleavings before finding the deadlock. MSPOE will, on the other hand, choose $S_{3,1}$ as the next relevant send to explore after any initial run. In this example, the deadlock will be detected within two interleavings.

2 Preliminaries

Let P be a concurrent MPI program and P_i is the i^{th} sequential process executing P where $i \in PID$ and $PID = \{0, 1, ..., n\}$. We assume the program is executed with finite many processes. Each P_i is L_i instructions long. Let l denote the program counter(PC) array; thus, $l_i \in l$ denotes the PC value for the i^{th} process. The jth MPI command in the ith process is denoted $p_{i,j}$ where $j \in L_i$.

A *non-blocking send* call issued by the process P_i with a program counter j with a destination as P_k is denoted as $S_{i,l_i}(k)$. A non-blocking send returns a "handle" that is waited upon by a later issued *wait* (W) operation. The send event occurs somewhere between $S_{i,l_i}(k)$ and W; the "occurrence" of a send is really the *match* event between a

send and its receive. In our presentation, we suppress the W calls (but our implementation handles them). A *blocking* send call's effect is obtained by placing W immediately after the send. Similarly, a non-blocking receive call is written as $R_{i,l_i}(k)$. If the receive is a wildcard then its denoted as $R_{i,l_i}(*)$. An MPI Barrier operation by process i is represented as $B_{i,j}$ where j is the l_i for that process. Let Op be the set of MPI operations comprised of $S_{i,j}(k)$, $R_{i,j}(k)$, $R_{i,j}(*)$, W and $B_{i,j}$, for all possible i, j, k. Note that an operation belonging to Op is a *visible* operation and all other operations (non MPI) are *invisible*. A visible operation is one that is intercepted by the ISP scheduler.

The execution state of an MPI program together with the MPI runtime is modeled using $\sigma \in \mathcal{S}$ where $\sigma = \langle I, P, M, l \rangle$ that consists of *issued* ($I \subseteq Op$) instructions, *persistent-set* (P) set, *matched* ($M \subseteq I$) instructions, and the PC array l. This is also the state that the ISP scheduler goes by (probing the internal state of the MPI processes and runtime is impractical). The set of instructions in $M \setminus I$ are the *enabled* ones. Persistent-set P at a state $\sigma \in \mathcal{S}$ (denoted by P_σ) is a set of *match-set* moves. A match-set at a state can either be (i) A matching send and receive or (ii) Matching barriers. Since match-set transitions the system from one state to a subsequent state, we view match-set moves as the *transitions* of the MPI program. The terms match-sets and transitions in this paper would be used interchangeably. Thus, when a send call $S_{i,l_i}(k)$ matches a receive call $R_{k,l_k}(i)$ at σ, the associated transition $t \in P_\sigma$ is represented by $\langle S_{i,l_i}(k), R_{k,l_k}(i) \rangle$. Let $\mathcal{T}$ denote the set of all transitions of the system. A $t \in \mathcal{T}$ enabled at state s which when executed results in a unique successor state s', written as $s \xrightarrow{t} s'$. The successor state is also represented by the following: $s' = t(s)$. We define the whole MPI program as a state transition system $A_G = (\mathcal{S}, \mathcal{T}, s_0)$. where s_0 is the starting state of the system. Further details are available at [17] (not necessary to follow our main ideas in this paper).

2.1 Nature of transitions in a Persistent-set

A persistent-set at a state can contain multiple transitions. Persistent-sets are constructed in a prioritized manner as discussed in our previous work [20] (appropriately summarized, as needed, in this paper). The only possibility of a persistent-set containing multiple transitions is when there is a wildcard receive involved. When all the potential senders to a wildcard receive $R(*)$ are determined at an execution state, ISP forms a transition involving $R(*)$ and each of the sends. The work in [20] views all resulting transitions as *dependent* and designates the collection of such transitions as *dependence transition group* (**DTG**). For instance, in Figure 2 the *DTG* with respect to the receive $R_{2,1}$ has the following transitions: $t_1 = \langle S_{0,1}, R_{2,1} \rangle$ and $t_2 = \langle S_{1,1}, R_{2,1} \rangle$. We define a function $Dtg(s) \uparrow_{R_{i,l}}$ that returns a set of transitions that belong to the *DTG* w.r.t. to the non-deterministic receive $R_{i,l}$ that are enabled at a state s.

Notice, however, multiple DTGs can co-exist at a state, *and they can influence each other.* The example shown in Figure 5 illustrates such a scenario. This figure shows one trace of the program. Here, the solid un-directed arrows represent the match-sets along which the execution proceeded. The dotted un-directed arrow represents another

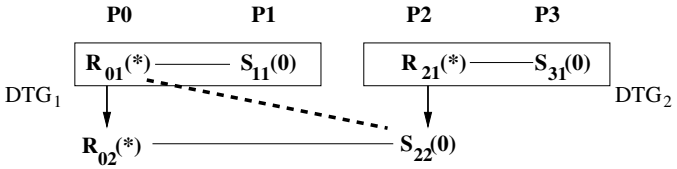

Fig. 5. Dependence among DTG transitions

possible match-set (not realized in the present execution). The solid directed arrows capture the IntraMB ("Intra process matches-before ordering) relation[4].

Observe that if DTG_2 is fired before the transitions in DTG_1, then $S_{2,2}$ would be co-enabled with $S_{1,1}$, and both these sends can match $R_{0,1}$. In this case, DTG_1 must be augmented (exactly as per the "C1" condition described in [4]). This is what POE_{OPT} (optimized POE) described in [20] does.

In our example, DTG_1 is augmented—from containing the transition $\langle S_{1,1}, R_{0,1}\rangle$ to containing two transitions $\langle S_{1,1}, R_{0,1}\rangle$ and $\langle S_{2,2}, R_{0,1}\rangle$. *This is the main source of the exponential explosion alluded to in this paper*.

MSPOE seeks to ameliorate this explosion. It takes the following departure from the workings of POE_{OPT} The whole exercise of MSPOE is to optimistically treat transitions within a DTG in σ as *independent*. *This observation is true of MPI programs where application state is independent of the sender that matched the wildcard receive*. Instead, MSPOE takes a lazy approach to augmenting DTGs. As mentioned under "Observation" on Page 3, as far as orphaning deadlocks are concerned, it is the competition between a wildcard and a specific receive that must be regarded as the dependency relation *that truly matters.* We shall see that DTG augmentation done precisely at these moments leads to an exploration technique (MSPOE) that *often generates a single interleaving* (implying the absence of deterministic receives). In contrast, POE and POE_{opt} generate an interleaving blowup. Our results show that orphaning deadlocks are detected by MSPOE in all practical cases, avoiding this explosion.

3 Formal definition of independent transitions

In order to first define *independent transitions*, we first introduce the notion of *commuting sends* that are part of the transitions within a single DTG.

Definition 1 (Commuting Sends). *: Sends $S_{i,l}(k)$ and $S_{j,m}(k)$ are commuting sends iff the following conditions hold at a state s:*

- *Let $t_1 = \langle S_{i,l}(k), R_{k,n}(*)\rangle$ and $t_2 = \langle S_{j,m}(k), R_{k,n}(*)\rangle$ such that $t_1, t_2 \in P(s)$.*
- *$S_{j,m}(k) \in t'_2$ and $S_{i,l}(k) \in t'_1$ where $t'_2 \in P(t_1(s))$ and $t'_1 \in P(t_2(s))$.*[5]

[4] The edge between $R_{2,1}$ and $S_{2,2}$ indicates that their must be a wait operation W bound to $R_{2,1}$ lying in-between. This W has been suppressed but the effects are appropriately captured in the IntraMB edge shown.

[5] Here, we treat t'_1 and t'_2 as sets; they really are send-receive pairs which model transitions.

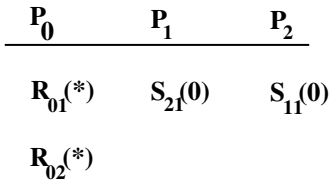

Fig. 6. Commuting example

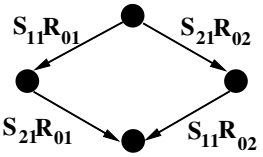

Fig. 7. Transition independence

Observe that in Definition 1, two sends, S_i and S_j can commute only when they are enabled and part of transitions t_1 and t_2 in a state s and firing one send at s should not leave the other send *disabled or unmatched* in the subsequent state. The C be the set of pairs of commuting sends ("commutes" predicate). We now define the independence relation used by MSPOE as:

Definition 2 (Independent Relation). *$I \subseteq \mathcal{T} \times \mathcal{T}$ is an* independence relation *iff for each $\langle t_1, t_2 \rangle \in I$ following conditions hold:*

1. **Enabledness:** *t_1 and $t_2 \in P(s)$ and there exists a $R_{k,n}(*)$ such that $t_1, t_2 \in Dtg(s) \restriction_{R_{k,n}}$.*
2. **Commutativity:** *If $S_{i,l}(k) \in t_1$ and $S_{j,m}(k) \in t_2$ then $(S_{i,l}, S_{j,m}) \in C$.*

Thus, with the independent relation, we now can say two transitions t_1 and t_2 are dependent when the send operations in t_1 and t_2 do not commute. Consider the example and its corresponding state graph shown in Figure 6 and Figure 7. The initial state s_0 has two enabled transitions, namely: $t_1 = \langle S_{1,1}, R_{0,1} \rangle$ and $t_2 = \langle S_{2,1}, R_{0,1} \rangle$. Note that transitions commute since they lead to the same final state. Firing t_1 disables t_2 in the next state, however, the transition enabled at $t_1(s)$ is $t_2' = \langle S_{2,1}, R_{0,2} \rangle$ and $t_2 \equiv_c t_2'$. Thus, t_1 and t_2 are independent. If the send calls in t_1 and t_2 do not commute (assuming t_1 was fired from s) then:

- The send from t_2 is disabled at $t_1(s)$.
- The operation available at $t_1(s)$ is not a receive t_2's send can match with. If the operation enabled at $t_1(s)$ is a receive, then it must be a deterministic receive which is sourcing from a process other than the process that issued t_2's send.

We discuss in detail the ability of MSPOE to compute the independence of transitions in Section 6. We use the classical notion of *persistent sets* [10].

Definition 3 (Persistent in s). *A set T of transitions enabled in a state s is* persistent in *s iff, for all non empty sequences of transitions from s in A_G*

$$s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$$

and including only transitions $t_i \notin T$, $1 \leq i \leq n$, t_n is independent in s_n with all transitions in T.

Informally, this means that when a transition sequence is generated from a state s by choosing only transitions that are independent with transitions in T then the final state

Algorithm 1 MSPOE Algorithm

```
1:  Input:
2:      Stack of State: St                          ▷ St has s_0; initial state
3:      Vector of Set: P                            ▷ Persistent-set for each state
4:      Vector of Set: RP                           ▷ Reduced Persistent-set for each state

5:  s ← First(St)                                   ▷ Get bottom of Stack St
6:  St ← GenerateInterleaving(s)
7:  while ~ Empty(St) {                             ▷ continue until St becomes empty
8:      s ← Last(St)                                ▷ Get top of Stack St
9:      RP_s ← RP_s \ {Curr(s)} *                   ▷ Curr(s) returns the match-set chosen at state s
10:     P_s ← P_s \ {Curr(s)}
11:     if Empty(RP_s) { *                          ▷ RP_s was singleton and was explored in the interleaving
12:         St ← St − s                             ▷ Remove state s from St
13:     } else
14:         St ← GenerateInterleaving(s)
15:     }
16: }
```

reached cannot have a transition that is dependent with any of the transitions in T. The interleavings obtained by only executing the entries in the persistent-set at every state are the *representative* interleavings and result a quotient state graph denoted as A_R.

Let's revisit the state graph shown in Figure 2. Using Definition 2), we now can reason about the example. Notice that for the states shown in the dotted box, the DTGs at those states have only independent transitions. Thus, for the purpose of verification of safety properties (such as absence of deadlocks), examining only *one representative* interleaving would suffice.

4 Macroscopic Partial Order Elusive (MSPOE) Algorithm

Algorithm 1 presents the MSPOE algorithm in detail (statements tagged with $*$ are additions to POE which help transform POE into MSPOE). Since we elided the details of POE in this paper (see [20]), we prefer to present a high level view of MSPOE, placing details at [17]. In this algorithm, the match-set move (or the transition) selected at a particular state s in an interleaving is denoted by $Curr(s) \in P_s$ where P_s is the persistent-set at state s. RP_s is the *reduced* persistent-set at state s which is what MSPOE will accomplish (it trims down persistent-set sizes according to our macrosopic POR independence rules presented in § 3). We also maintain a stack St of states that have been visited but not completely explored. Algorithm 2 presents ISP scheduler's functioning to generate the interleaving of the program according to POE. The operator $\prec_{lp}$ captures the Matches-Before ordering among operations issued from a process (see [20] for details). Algorithm 3 depicts the prioritized match-set selection policy of POE which remains the same for MSPOE.

Algorithm 2 GenerateInterleaving from state s

```
1: Input:
2:     State: s
3:     Stack of State: St
4: Output:
5:     Stack of State: St

6: while s is not NULL {                          ▷ Continue until next state can't be found
7:     m ← Choose(P_s)                            ▷ Choose a match-set to explore from s
8:     RP_s ← RP_s ∪ {m} *
9:     if m = ⟨S_{i,l}(j), R_{j,m}(i)⟩ { *                          ▷ if m has det recv
10:        for all s' ← s − 1 until First(St) { *                    ▷ Update RP_{s'}
11:            if ∃B_{i,−} ∈ P_{s'} : B_{i,−} ≺_lp S_{i,l} { *
12:                goto Next_State *
13:            }
14:            if ∃m' ∈ P_{s'} : m' = ⟨S_{i,−}(j), R_{j,−}(∗)⟩ ∧ m' ∉ RP_{s'} { *
15:                RP_{s'} ← RP_{s'} ∪ {m'} *
16:            }
17:        }
18:    }
19:    Next_State: s ← Explore(s, m)              ▷ Get the next state by firing m from s
20:    St ← St + s                                ▷ Add s to the Stack
21: }
22: return St
```

Algorithm 3 Choose P_s

```
1: Input:
2:     State: s
3: Output:
4:     Match-set: m

5: if ∃m ∈ P_s : m contains barrier {
6:     return m
7: else if ∃m ∈ P_s : m contains wait {
8:     return m
9: else if ∃m ∈ P_s : m contains deterministic recv {
10:    return m
11: else if ∃m ∈ P_s : m contains non-deterministic recv {
12:    return m
13: }
```

MSPOE starts with the initial state s_0 in the stack. It generates a complete interleaving by calling the function *GenerateInterleaving* (line 6 in Algorithm 1) It repeats the following steps from this point forwards until the state stack (St) becomes empty:

– Select the last state s from the trace and remove the match-set entry explored in the trace from P_s and RP_s. If RP_s becomes empty then pop the state off from the state stack St.

P_0	P_1	P_2
$S_{0,1}(2)$	$S_{1,1}(2)$	$R_{2,1}(*)$
		$R_{2,2}(*)$
$B_{0,2}$	$B_{1,1}$	$B_{2,3}$
	$S_{1,3}(2)$	$R_{2,4}(2)$

Fig. 8. MSPOE with redundant exploration

– If the after executing the step the last state has non-empty RP_s then generate further interleaving from s.

Algorithm 2 takes as input a state and generates an interleaving from that state in the following manner:

– From P_s, choose a match-set m according to POE's prioritized match-set selection procedure.
– Add m to RP_s.
– If m involves a deterministic receive, then search for each state s' in the stack St and perform the following: (1) If $P_{s'}$ contains a match-set m' involving a send from the same process whose send is a part of m at P_s then add m' to $RP_{s'}$. (2) However, if $P_{s'}$ contains a barrier operation MB ordered with the send that is part of m then move terminate $RP_{s'}$ update and move-on to explore the next state in the interleaving. Consider the example shown in Figure 8. Notice that no matter which interleaving is explored, $S_{1,3}$ can never be enabled and be a potential match for receive calls $R_{2,1}$ and $R_{2,2}$ since such a match is restricted by the presence of barriers. We avoid such unnecessary augmentation of persistent states by adding the barrier check (lines 12-13) to the MSPOE algorithm.
– Repeat all the step until no more states can be explored.

Formal Details: MSPOE is sound, as it explores only feasible interleavings. It is *deliberately incomplete*: our aim is to have a practical alternative to ISP and DAMPI which guarantee completeness (in terms of non-determinism coverage), but suffer from an exponential schedule blow-up. §5 shows that MSPOE is a welcome addition to the practitioners' toolkit.

5 Experimental Results

All the experiments were run on Intel Core i7 quad-core 2.67 GHz with 8 GB of RAM. We set a time limit of 2 hours to verify the benchmarks. We abort the verification process if the it did not complete within the time-limit. The results pertaining to the reductions obtained are documented in Table 1. Summary of the tabulated results is that MSPOE explored only one interleaving for almost all benchmarks detecting the same deadlocks that ISP did. The sign $\surd$ in the MSPOE column next to the number of interleavings examined illustrates that MSPOE also caught the same deadlock as ISP did.
2D-Diffusion: We tested ISP's POE and MSPOE algorithm on *2D-Diffusion* [7] example. The code has a deadlock when evaluated in *zero buffering mode*. In this mode, the

				Interleavings		Time(sec)
Benchmark	Buffering	# of procs	Deadlocks?	ISP	MSPOE	MSPOE
Mat-Multiply	0	4	No	54	1	0.001
		8	No	120	1	0.002
	∞	4	No	54	1	0.3
		8	No	120	1	0.3
2D-Diffusion	0	4	Yes	1	1 √	-
	∞	4	No	90	1	0.314
		8	No	> 10,500	1	0.442
Pi- Monte-Carlo	0	4	No	36	1	0.002
		8	No	5040	1	0.003
	∞	4	No	36	1	0.24
		8	No	5040	1	0.3
Integrate_mw	0	4	No	81	81	-
		8	No	2401	2401	-
Madre	0	4	Yes	1	1 √	-
	∞	4	No	> 8000	1	1.48
		8	No	> 8000	1	3.09
Parmetis	0	4	No	1	1	128.933
Gaussian Elimination	0	4	No	1	1	0.24
		8	No	1	1	0.276
	∞	4	No	180	1	0.31
		8	No	> 20,000	1	0.324

Table 1. Interleaving results for deadlock detection

P_0	P_1	P_2
$R_{0,1}(*)$	$S_{1,1}(0)$	$S_{2,1}(0)$
$R_{0,2}(*)$	$S_{1,2}(0)$	$S_{2,2}(0)$
$R_{0,3}(*)$		
$R_{0,4}(*)$		
$B_{0,5}$	$B_{1,3}$	$B_{2,3}$
$S_{0,7}(1)$	$S_{1,4}(2)$	$S_{2,4}(0)$
...	...	

Fig. 9. Communication in 2D-Diff

send calls act as synchronous operations. The deadlock was caught by ISP and MSPOE right in the first interleaving. When the same code is run on *infinite buffering mode*, the code becomes deadlock free. The code was modified to run with a single time-step. Its communication pattern is shown in the Figure 9. Note that if sends were treated as synchronous then after barriers each process is blocked on their respective sends causing a deadlock.

Integrate: `Integrate_mw` [7] is another benchmark that uses heavy non-determinism to compute an integral of sin function over the interval $[0, Pi]$. `Integrate` has a master-slave pattern where the root process divides the interval in a certain number of tasks. The root process then delegates to each worker process a single task and then waits for results from them by posting wildcard receive calls. Workers that finish early with their

```
Worker i: while(1) {
            R(from 0, any-tag); // Recv task
            if(work-tag)
              S(master, result-tag);
            else break;
          }

Master: for(i = 1 to nprocs-1) {
          Send(i, work-tag); // send to each worker the task
          tasks++;
        }
        while(tasks <totalTasks){
           Recv(*, result-tag); // recv result
           S(S.S, work-tag);  // assign more task
           tasks++;
        }
        for(i = 1 to nprocs-1) {
           Recv(i, result-tag); // recv result
           S(i, terminate-tag);  // terminate signal to worker i
        }
```

Fig. 10. High-level Code Pattern of "Integrate"

work are provided with more tasks until all tasks are distributed (as detailed in the high level code in Figure 10).

This benchmark does not have a deadlock. Notice that MSPOE does not demonstrate any savings over ISP while exploring the schedule space. This is because, the master process finally posts deterministic receive calls targeting each worker before it sends termination signals to each worker. This causes the MSPOE to fully expand the persistent-sets of each prior wildcard receive.

MADRE: MADRE [18], a memory aware data redistribution engine, is a library written in MPI which mainly performs load balancing tasks in an efficient manner. MADRE moves the data blocks across nodes in a distributed system within the bounds of memory available to each of the application's process. We tested MADRE with its *unitBred* algorithm on various data-sets. *unitBred* algorithm is of particular interest to us because it uses MPI_ANY_SOURCE and MPI_ANY_TAGS. MADRE has no bugs provided normal MPI send calls are not treated as blocking calls. We ran ISP's POE and then MSPOE algorithm with sbt9 dataset with unitBred algorithm and the results are documented in the Table 1.

Parmetis: Parmetis [14] is a parallel hypergraph partitioning code-base. Since, Parmetis only uses deterministic calls, ISP and MSPOE complete the verification process in a single interleaving. Parmetis was selected as a benchmark despite the absence of non-determinism because the application issues a lot of MPI calls which served as a basis to evaluate the scalability of the data-structures used in MSPOE. When run on 4 processes, Parmetis issues $\sim 55,000$ calls.

6 Discussion

As shown, in all our experiments, MSPOE has managed to detect deadlocks whenever POE (supported by the ISP tool) has; and managed to return (by generating) a small

number (typically 1) of interleavings in other cases. In the latter cases, MSPOE computes the full persistent sets, but trims it down based on our macroscopic reduction criterion. The real value to a designer is the following (take an example similar to 2D diffusion for discussion): if given 10^3 processes, POE will simply take forever while exploring the persistent sets computed from the initial trace. MSPOE will, on the other hand, examine the initial trace, and perform macroscopic commutation aware persistent set reductions. *This is a search bounding method substantially different from other obvious reduction approaches* (e.g., depth-bounding or bounded mixing [22]), and further this bounding heuristic is *tuned toward detecting orphaning deadlocks*. Further studies are underway to further characterize MSPOE.

An important question pertaining to the working of MSPOE is the following: Does MSPOE precisely compute all the dependent actions in an MPI program? Notice that MSPOE only augments the persistent-set of prior states (at which a wildcard move took place) only when a deterministic receive is witnessed later in the trace. It is by no means a complete criterion to discover all dependent transitions.

Consider, for instance, some patterns that MSPOE cannot handle. In the example shown in Figure 11, if $S_{3,1}$ matched $R_{1,1}$ then $S_{1,2}$ and $S_{2,1}$ would engage in a cyclic wait on each other causing a deadlock. Notice that $S_{1,2}$ can't match unless $S_{2,1}$ successfully completes since $R_{2,2}$ is the only match of $S_{2,1}$ and $S_{2,1}$ is an enabler operation for $R_{2,2}$. Notice that MSPOE will fail to discover such a deadlock. However, a pertinent question that will underscore the usability of MSPOE is the following: how often such coding patterns are employed in applications, if at all? In real MPI codes that we have assessed, we did not witness such a coding style. Typically, a deterministic communication from a process following a wildcard receive is accomplished by *reply channels*. Processes often employ reply channels to perform dynamic load balancing duties by sending data/task to the sender that matched the prior wildcard receive. Thus, in our opinion, it is rare (almost to none) to observe that applications issue hard-wired deterministic receives/sends following a wildcard receive operation. Notice that in Figure 11, if $S_{1,2}(2)$ is re-written as $S_{2,1}(status.Source)$ (indicating a reply-channel) then the deadlock in the code disappears.

Figure 12 is another example where MSPOE will fail to detect a deadlock. In Figure 12, note that the barriers would not discharge if $S_{3,2}$ were to match $R_{1,1}$ thereby causing the deadlock. Notice that $S_{3,2}$ is unordered w.r.t. $B_{3,1}$. This can happen only when $S_{3,2}$ is issued before $B_{3,1}$ however the wait associated with $S_{3,2}$ is issued after the barrier. Again, such a coding practice is flawed and we have not witnessed any real MPI program so far that employs such a coding style. Typically, global fence operations (such as barriers) are issued only *after* the local fence operations such as waits are successfully discharged. If such were to be the programming style then the wait calls for both $R_{1,3}$ and $S_{3,2}$ should have been issued before the respective process barriers. In which case, the match-set $\langle B_{1,2}, B_{2,2}, B_{3,1}\rangle$ would be issued only after the completion of $\langle S_{3,2}, R_{1,3}\rangle$. Even in alternate trace when $S_{3,2}$ pairs-up with $R_{1,1}$, notice that $S_{2,1}$ will now find a match in $R_{1,3}$. Hence, the deadlock will disappear.

In all our benchmarks, none of above mentioned coding styles were employed except the deterministic receive calls following a wildcard receive. MSPOE, thus, as a result of such observations, despite being in-complete works extremely well (in other

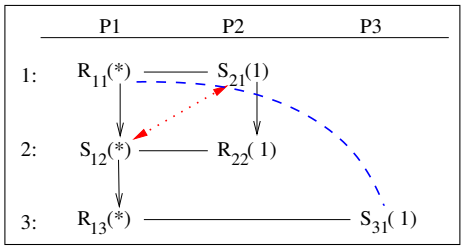

Fig. 11. Deadlock because cyclic dependency between $S_{1,2}$ and $S_{2,1}$

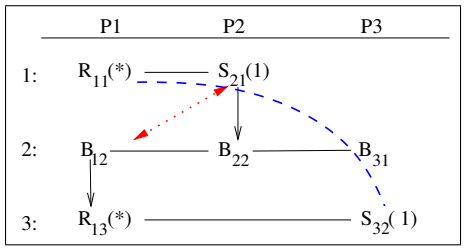

Fig. 12. Deadlock because barriers do not discharge

words, appears complete) in practice. Constructing a methodology that is complete forms the basis of our future work.

7 Conclusions

We have presented a novel algorithm MSPOE that demonstrates significant savings in the exploration space of programs for the purpose of communication deadlock detection. In many cases the reductions were from tens of thousands of interleavings to just one interleaving. We document the MSPOE reduction results observed over several benchmarks. We further present evidence on the criticality of the match-set selection in avoiding redundant explorations and for early detection of bugs.

Future work: Conditional communication flow pattern is sill not tackled by MSPOE. However, MSPOE algorithm can be notified of the causal receive calls whose buffers when decoded would result in a conditional communication flow. Such information can be statically mined and provided to the dynamic verification scheduler. To gather the afore-said information, we would require a MPI specific control flow graph (CFG). Work in [1] presents *pCFG* which is a CFG for MPI programs. Our future work would therefore lie in modifying the *pCFG* work to handle non-deterministic MPI operations. Furthermore, we will develop flow-sensitive static analysis methods on top of the improved *pCFG* to analyze conditional communication patterns.

References

1. G. Bronevetsky. Communication-Sensitive Static Dataflow for Parallel Message Passing Applications. In *CGO: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2009. ISBN: 978-0-7695-3576-0.
2. E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In *CAV*, pages 147–158, 1998.
3. E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des.*, 9:77–104, August 1996.
4. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.
5. A. F. Donaldson, A. Miller, and M. Calder. Finding symmetry in models of concurrent systems by static channel diagram analysis. *Electr. Notes Theor. Comput. Sci.*, 128(6):161–177, 2005.

6. E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Form. Methods Syst. Des.*, 9:105–131, August 1996.
7. FEVS Benchmark. http://vsl.cis.udel.edu/fevs/index.html.
8. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In J. Palsberg and M. Abadi, editors, *POPL*, pages 110–121. ACM, 2005.
9. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An approach to the State-Explosion Problem*. PhD thesis, Univerite De Liege, 1994–95.
10. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.
11. P. Godefroid, M. Levin, and D. Molnar. Whitebox fuzzing for security testing. *Communications of the ACM*, Mar. 2012.
12. G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. de Supinski, M. Schulz, , and G. Bronevetsky. Formal analysis of mpi-based parallel programs: Present and future. *Communications of the ACM*, Dec. 2011.
13. C. N. Ip and D. L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.*, 9:41–75, August 1996.
14. G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *SuperComputing (SC)*, 1996.
15. http://www.multicore-association.org.
16. Message Passing Forum. http://www.mpi-forum.org/docs.
17. S. Sharma. *Predictive Analysis of Message Passing Applications*. PhD thesis, University of Utah, 2012.
18. S. F. Siegel. The MADRE web page. http://vsl.cis.udel.edu/madre, 2008.
19. S. F. Siegel. MPI-Spin web page. http://vsl.cis.udel.edu/mpi-spin, 2008.
20. S. Vakkalanka. *Efficient dynamic verification algorithms for MPI applications*. PhD thesis, University of Utah, Salt Lake City, Ut, USA", 2010.
21. S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of mpi programs with reductions in presence of split operations and relaxed orderings. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 66–79, Berlin, Heidelberg, 2008. Springer-Verlag.
22. A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. de Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.
23. A. Vo, G. Gopalakrishnan, R. M. Kirby, B. R. de Supinski, M. Schulz, and G. Bronevetsky. Large scale verification of mpi programs using Lamport clocks with lazy update. In *PACT*, 2011.
24. A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal verification of practical MPI programs. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 261–270, New York, NY, USA, 2009. ACM.
25. R. Vuduc, M. Schulz, D. Quinlan, B. de Supinski, and A. Sornsen. Improving distributed memory applications testing by message perturbation. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, PADTAD '06, pages 27–36, New York, NY, USA, 2006. ACM.