A Sound Reduction of Persistent-sets for Deadlock Detection in MPI Applications ⋆

Subodh Sharma[1] ⋆⋆, Ganesh Gopalakrishnan[2], and Greg Bronevetsky[3]

[1] University of Oxford, subodh.sharma@cs.ox.ac.uk
[2] University of Utah, ganesh@cs.utah.edu
[3] Lawrence Livermore National Laboratory, bronevetsky1@llnl.gov

Abstract. Formal dynamic analysis of Message Passing Interface (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, [9]) 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 [6]. There are also embedded system communication standards built around message passing, such as MCAPI [8]. In this paper, we study the problem of adequately testing message passing programs using formal techniques for the purpose of deadlock detection. 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 receives*) or the *non-deterministic* (or "wildcard") receives that can receive from any sender that targets the process issuing such a receive. The sends and receives issued by an MPI process that target the same destination or source from the same process are required to match in program order (the "non-overtaking rule of MPI", Section 3.5 in [9]). The MPI runtime computes the eligible matches for each receive operation. The matching operations are called *match pairs*. At any runtime state of an MPI program, a deterministic receive will always have a single matching send, thus,

⋆ This research has been supported by NSF OCI 1148127 and EPSRC project EP/G026254/1.
⋆⋆ The work was performed when the author was in University of Utah

all concurrent match pairs consisting of deterministic receives and matching sends can *commute* (i.e. match pairs can interleave resulting in the same program state). This is because all such match pairs have non-overlapping destinations/sources. However, non-deterministic receive matches do not, in general, commute. A non-deterministic receive $R(*)$ can have multiple eligible matching senders; an $R(*)$ matching a send S_i results in a system state different from when another send S_j matches the same receive where S_i and S_j are issued from different processes. This is not good news for dynamic partial order reduction (DPOR [3]) methods because in many MPI programs, $R(*)$ calls occur in sequence (typically in a loop). Thus, it seems that any DPOR technique is doomed to examine an exponential number of interleavings—something that does not bode well for our *Exascale* computing aspirations (exascale roadmap [10]) 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 the afore-mentioned schedule 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 [18]) do not guarantee coverage, and are also highly redundant because they will, in practice, generate many equivalent schedules (e.g., permuting deterministic message match pairs). 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 [12]) 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 (meaning no false alarms), (2) they can be made complete with respect to non-determinism coverage (meaning no omissions w.r.t. a safety property). Formal dynamic verifiers such as ISP [14, 17] and DAMPI [15, 16] 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). The MPI semantics-aware algorithms of these tools guarantee non-determinism coverage (e.g., all the potential match pairs w.r.t. a non-deterministic receive) while not examining the schedule space with respect to commuting deterministic receive/send match pairs. 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. *However, these dynamic verification tools suffer from the aforesaid exponential schedule explosion when a sequence of $R(*)$ commands are issued.* A practical dynamic verification tool that avoids this schedule explosion and provides reasonable coverage is, to the best of our knowledge, currently unavailable. This paper describes such a tool.

Contributions: The specific contributions of this paper are as following:

- Our general focus is on the problem of detecting deadlocks in MPI programs. We define a notion of *orphaning deadlocks* in which an MPI receive is left without a

matching send in some MPI program execution state. We modify ISP's dynamic partial order reduction algorithm called POE (standing for Partial Order avoiding Elusive interleavings, [14]) to result in a new algorithm called MSPOE (*Macro-Scopic* 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. It is a reasonable assumption since a large class of SPMD programs are coded in a manner that is consistent with our simplification.

- 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. To this end, we modify and re-state the definition of *independent transitions* in the context of MPI programs.
- 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 deadlocks that ISP has discovered on the set of selected realistic benchmarks. A study of any successful large-scale formal software testing or analysis approach (e.g. [5]) shows that rather than aiming for a theoretically complete algorithm, one almost always has to aim for "completeness in practice."[4]

Detailed look at an example: Let a call denoted by $S_{i,j}(k)$ be a asynchronous send call from process i sending to process k with the local process program counter (PC) at j. Similarly an asynchronous 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 asynchronous receive is represented by $R_{i,j}(*)$. We will use this notation through the rest of the paper. Note that in the notation, the arguments of the call can be suppressed for brevity since each call can also be uniquely identified by the process ID and the PC value. For instance, $R_{i,j}(*)$ can be uniquely identified by $R_{i,j}$. We would use the actual and suppressed notation interchangeably in the paper. Let us examine the example shown in Figure 1. Assume that all the asynchronous calls have their associated *wait calls* (wait is a blocking call to ensure the successful completion of the associated non-blocking send/receive request, Section 3.7.3 in [9]) posted which are not shown in the example for brevity. A scheduler such as ISP will explore 24 interleavings for this example. This is because, the first wildcard receive will have 4 eligible matching sends and the subsequent receive will have 3 eligible matching sends and so forth, leading to a total of 4! schedules. As long as all sends *commute*, such examples cannot have deadlocks and there is no necessity to examine other schedules. In the example under discussion, observe that all sends commute. MSPOE will analyze the program in Figure 1 in the following way: MSPOE will explore the first interleaving and will subsequently discover that the program does not issue any deterministic receive calls and all sends commute. Thus, it will conclude that every receive must find a match in each interleaving. It will terminate the exploration right after the first run.

Now consider the same example of Figure 1, however, replace $R_{4,2}(*)$ by $R_{4,2}(3)$. The code now has a deadlock. Figure 2 illustrates the various match-pairs possible for the

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

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

Fig. 1. Example illustrating explosion in schedule space

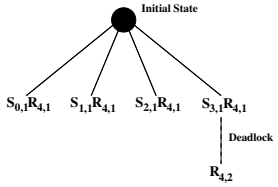

Fig. 2. Deadlocking match for $R_{4,1}(*)$

receive $R_{4,1}$. If $R_{4,1}$ were to match $S_{3,1}$ (right-most transition from the initial node), the subsequent deterministic call ($R_{4,2}$) will be orphaned, thus creating a refusal deadlock. ISP and other verification schedulers like DAMPI explore all the matches starting from leftmost choice shown in Figure 2 and then moving right with every new run, generating four interleavings before finding the deadlock. MSPOE, on the other hand, discovers that since there is a deterministic receive $R_{4,2}$, its matching send can get consumed by a prior wildcard receive. Thus, MSPOE prioritizes the schedule in which $S_{3,1}$ is matched with $R_{4,1}$ thus forcing the program to take a schedule where $R_{4,2}$ is orphaned. MSPOE detects the deadlock in 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 $j = l_i$.

The work presented in this paper can be understood with only a subset of MPI calls which comprises of: non-blocking send, non-blocking receive, wait, and the barrier call. Since providing the whole overview of MPI is beyond the scope of this paper, we will restrict the presentation to the afore-mentioned subset of MPI calls. We have already presented the notations for representing non-blocking send and receive calls. A non-blocking send or receive call returns a "handle" that is waited upon by a later issued *wait* (W) operation. For instance, the wait call for the corresponding $S_{i,j}(k)$ would be represented as $W_{i,l_i}(h_{i,j})$. In our illustrations of examples, we suppress showing the W calls explicitly. We replace them by suitably adding the program order edges. Note that our implementation handles them correctly. A *blocking* send call's effect is obtained by placing wait call immediately after the non-blocking send call. A blocking receive can be obtained in a similar fashion. 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 $S_{i,j}(k)$, $R_{i,j}(k)$, $R_{i,j}(*)$, $W_{i,j'}(h_{i,j})$ and $B_{i,j}$, for all possible i, j, 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 σ where $\sigma = \langle I, M, C, l \rangle$ that consists of *issued* ($I \subseteq Op$) instructions, *persistent-set* (M) set, *completed* set of instructions ($C \subseteq I$), 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). Let $\mathcal{S}$ denote the set of all states of an MPI program. Persistent-set M at a state $\sigma \in \mathcal{S}$ (denoted by M_σ) is a set of *match-set* moves. A match-set at a state is either a set of matching send and deterministic receive or a set of matching sends and a wildcard receive. 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 under the execution of a verification scheduler like ISP. 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 $t \in M_\sigma$ is represented by $\langle S_{i,l_i}(k),\, R_{k,l_k}(i)\rangle$ or just $\langle S_{i,l_i},\, R_{k,l_k}\rangle$. We denote the issue set and the completed set at σ by I_σ and C_σ respectively. Let $\mathcal{T}$ denote the set of all transitions of the system. A $t \in \mathcal{T}$ enabled at state σ which when executed results in a unique successor state σ', written as $\sigma \xrightarrow{t} \sigma'$. The successor state is also represented by the following: $\sigma' = t(\sigma)$. We define the whole MPI program as a state transition system $A_G = (\mathcal{S}, \mathcal{T}, \sigma_0)$. where σ_0 is the starting state of the system. We now define the transition rules to model MPI program execution as governed by ISP.

2.1 State transition rules (MPI/ISP)

Before we present the state transition rules, it is important to understand the *Matches-Before* (MB) ordering among MPI instructions. We define MB ordering among two operations issued from the same process by the operator $\prec_{lp}$. MPI standard requires instruction pairs $S_{i,j}(k) \prec_{lp} S_{i,j'}(k)$, $R_{i,j}(k) \prec_{lp} R_{i,j'}(k)$, and $R_{i,j}(*) \prec_{lp} R_{i,j'}(k/*)$ where $j < j'$. The standard also enforce the following ordering: $W_{i,j}(-) \prec_{lp} any_{i,j'}$ and $B_{i,j} \prec_{lp} any_{i,j'}$ where $any \in Op$ and the symbol "$-$" denotes don't care condition. MPI instructions are allowed to re-order and violate the program order, however, they must always obey the MB ordering.

Our state transition rules employ a precondition *Ready* to model MB ordering restriction based matching of transitions.

$$Ready(\sigma) = \{x \in I_\sigma | \forall y : (y \prec_{lp} x \Rightarrow \exists \sigma' \in Prev(\sigma) : y \in C_{\sigma'}\}$$

An instruction would be *ready* to be matched in a certain state only when all prior MB ordered operations have matched. $Prev(\sigma)$ returns the set of preceding states where each state upon firing a unique transition (enabled at that state) leads to σ. Function *isW* tests whether the instruction that is passed as an argument is a wait call. Similarly functions such as *isS* and *isR* test whether a given instruction is a send or a receive respectively. The rules called R_S and R_R can be used to model how the state advances upon instruction issue of send and receive calls from the process P_i.

$$R_S, R_R : \frac{\Sigma(\sigma\ as\ \langle I, M, C, l\rangle), \quad (isS(x_{i,-}) \vee isR(x_{i,-}))}{\Sigma\langle I \cup \{x_{i,-}\},\ M,\ C,\ l[i \leftarrow (l_i + 1)]\rangle}$$

Here, Σ is the predicate for the set of reachable states from the start state. Similarly, we define the rules for instruction issue of wait and barrier calls:

$$R_W, R_B : \frac{\Sigma(\sigma\ as\ \langle I, M, C, l\rangle), \quad (isW(x_{i,-}) \vee isB(x_{i,-}))}{\Sigma\langle I \cup \{x_{i,-}\},\ M,\ C,\ l\rangle}$$

The successful return of wait and barrier calls are modeled by the following rules:

$$R_{Wret}: \ \frac{\Sigma(\sigma \ as \ \langle I, M, C, l\rangle), \quad isW(x_{i,j}(h_{i,j'})), \quad \exists y_{i,j'} : y_{i,j'} \in C}{\Sigma\langle I, \ M, \ C \cup \{x_{i,j}\}, \ l[i \leftarrow l_i + 1]\rangle}$$

$$R_{Bret}: \ \frac{\begin{array}{l}\Sigma(\sigma \ as \ \langle I, M, C, l\rangle), \ Bars = \{x_{i,-} \in Ready(\sigma) | isB(x_{i,-}), i \in PID\}, \\ |Bars| = PID\end{array}}{\Sigma\langle I, \ M, \ C \cup Bars, \ (k \in PID, l[k \leftarrow l_k + 1])\rangle}$$

We now define the transition rule for the completion of send and receive instructions:
R_{SRM}:

$$\frac{\Sigma(\sigma \ as \ \langle I, M, C, l\rangle), \quad \{S_{i,j}(k), R_{k,n}(i/*)\} \subseteq Ready(\sigma)}{\Sigma\langle I, \ M \cup \{S_{i,j}(k), R_{k,n}(i)\}, C\}, \ l\rangle}$$

R_{SR}:

$$\frac{\Sigma(\sigma \ as \ \langle I, M, C, l\rangle), \quad \{S_{i,j}(k), R_{k,n}(i)\} \subseteq Ready(\sigma)}{\Sigma\langle I, \ M \setminus \{S_{i,j}(k), R_{k,n}(i)\}, \ C \cup \{S_{i,j}(k), R_{k,n}(i)\}, \ l\rangle}$$

Rule R_{SRM} is responsible for constructing the persistent-set M at each state. In order to define the matching and completion of wildcard receive calls we first introduce the Fnc predicate. Let $\sigma\rightarrow$ denote σ has a next state, $\sigma\rightarrow_{R_{SR}}$ denote that σ has a next state through R_{SR} (i.e., R_{SR} can fire at σ), and $\sigma\nrightarrow_{R_{SR}}$ denote that R_{SR} cannot fire at σ. Similarly, $\sigma\nrightarrow_{R_{SR},R_B}$ denotes that neither R_{SR} nor R_B can fire at σ. Then we define the fence predicate as follows,

$$Fnc(\sigma) = \sigma\nrightarrow_{R_{SR},R_W,R_B,R_{Wret},R_{Bret},R_S,R_R}$$

When fence predicate is true then the only transition that is enabled at *sigma* is the wildcard transition. We can now define R_{SR*} to be:

$$\frac{\Sigma(\sigma \ as \ \langle I, M, C, l\rangle), \quad \{S_{i,j}(k), R_{k,n}(*)\} \subseteq M_\sigma, \quad Fnc(\sigma)}{\Sigma\langle I, \ M \setminus \{S_{i,j}(k), R_{k,n}(i)\}, \ C \cup \{S_{i,j}(k), R_{k,n}(i)\}, \ l\rangle}$$

In particular, we show the dynamic rewriting by changing '*' to i. Readers are encouraged to refer [14] for more details. We have presented only the required rules and details in order to make the paper self-contained.

We finally present the classical notion of *persistent sets* [4] which is crucial in understanding the match-set reductions presented later in the paper.

Definition 1 (Persistent in σ). *A set T of transitions enabled in a state σ is* persistent in σ *iff, for all non empty sequences of transitions from σ in A_G*

$$\sigma = \sigma_1 \xrightarrow{t_1} \sigma_2 \xrightarrow{t_2} \sigma_3 ... \xrightarrow{t_{n-1}} \sigma_n \xrightarrow{t_n} \sigma_{n+1}$$

and including only transitions $t_i \notin T$, $1 \leq i \leq n$, t_n is independent in σ_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 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 reduced state graph denoted as A_R.

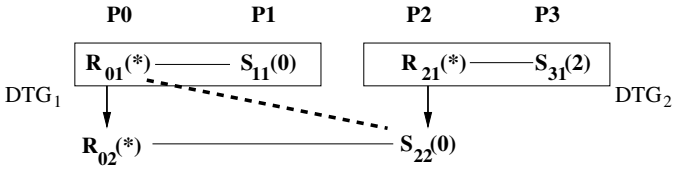

Fig. 3. Dependence among DTG transitions

2.2 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 [13] (appropriately summarized in the state transition rules, 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 [13] views all resulting entries in the persistent-set of a state as *dependent* and designates the collection of such transitions as *dependence transition group* (**DTG**). For instance, consider the example in Figure 3. 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 possible match-set (not realized in the present execution). The solid directed arrows capture the IntraMB ("Intra process matches-before ordering) relation[5]. The *DTG* with respect to the receive $R_{0,1}$ has the following transitions: $t_1 = \langle S_{1,1}, R_{0,1} \rangle$ and $t_2 = \langle S_{2,2}, R_{0,1} \rangle$. We define a function $Dtg(\sigma) \restriction_{R_{i,l}}$ that returns a set of transitions that are enabled at a state σ and belong to the *DTG* w.r.t. to the wildcard receive $R_{i,l}$.

Notice, however, multiple DTGs can co-exist at a state, *and they can influence each other.* The example shown in Figure 3 illustrates such a scenario. 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.

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. 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*. MSPOE takes a lazy approach to augmenting DTGs. As mentioned under example explanation on Page 3, as far as orphaning deadlocks are concerned, it is the competition between a wildcard and a deterministic receive for a particular send 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

[5] The edge between $R_{2,1}$ and $S_{2,2}$ indicates that there 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 MB edge shown.

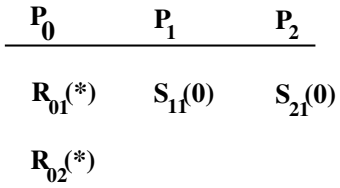

Fig. 4. Commuting example

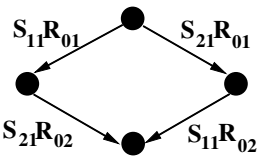

Fig. 5. Transition independence

(MSPOE) that *often generates a single interleaving* (implying the absence of deterministic receives). In contrast, POE generate an interleaving blowup. Our results show that orphaning deadlocks are detected by MSPOE in all practical cases, avoiding this explosion.

3 Independent transitions

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

Definition 2 (Commuting Sends). *: Sends $S_{i,l}(k)$ and $S_{j,m}(k)$ are commuting sends iff the following conditions hold at a state σ:*

- *Exists $R_{k,n}(*) \in Ready(\sigma)$ such that $t_1 = \langle S_{i,l}(k), R_{k,n}(*)\rangle$ and $t_2 = \langle S_{j,m}(k), R_{k,n}(*)\rangle$ and $t_1, t_2 \in P_\sigma$.*
- *$S_{j,m}(k) \in t'_2$ and $S_{i,l}(k) \in t'_1$ where $t'_2 \in P_{t_1(\sigma)}$ and $t'_1 \in P_{t_2(\sigma)}$.*[6]

Observe that in Definition 2, two sends, $S_{i,l}$ and $S_{j,m}$ can commute only when they are enabled and part of transitions t_1 and t_2 in a state σ and matching one send at σ should not leave the other send *disabled or unmatchable* in the resulting state. Let C be the set of pairs of such commuting sends ("commutes" predicate). Then, we have the following: that $t_1 \equiv_C t'_1$ and $t_2 \equiv_C t'_2$.

We now define the independence relation used by MSPOE as:

Definition 3 (Independent Relation). *Ind $\subseteq \mathcal{T} \times \mathcal{T}$ is an* independence relation *iff for each $\langle t_1, t_2\rangle \in$ Ind the following conditions hold:*

1. **Enabledness:** *t_1 and $t_2 \in P_\sigma$ and there exists a $R_{k,n}(*)$ such that $t_1, t_2 \in Dtg(\sigma) \upharpoonright_{R_{k,n}}$.*
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 4 and Figure 5. The initial state σ_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 the sends $S_{1,1}$ and $S_{2,1}$ commute. 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 were not commute (assuming t_1 was fired from σ) then:

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

Algorithm 1 MSPOE Algorithm

```
1: Input:
2:     Stack of State: St                          ▷ St has σ0; initial state
3:     Vector of Set: P                            ▷ Persistent-set for each state
4:     Vector of Set: RP                           ▷ Reduced Persistent-set for each state

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

– The send from t_2 is disabled at $t_1(\sigma)$.
– The operation available at $t_1(\sigma)$ is not a receive that t_2's send can match with. If the operation enabled at $t_1(\sigma)$ 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.

This explanation formulates a detailed summary of our initial observation for the refusal deadlocks. We discuss in detail the ability of MSPOE to compute the independence of transitions in Section 6.

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). In this algorithm, the match-set move (or the transition) selected at a particular state σ in an interleaving is denoted by $Curr(\sigma) \in P_\sigma$ where P_σ is the persistent-set at state σ. RP_σ is the *reduced* persistent-set at state σ which is what MSPOE will accomplish (it trims down persistent-set sizes according to our macroscopic 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. Algorithm 3 depicts the prioritized match-set selection policy of POE which remains the same for MSPOE.

MSPOE starts with the initial state σ_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:

Algorithm 2 GenerateInterleaving from state σ

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

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

Algorithm 3 Choose P_σ

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

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

– Select the last state σ from the trace and remove the match-set entry explored in the trace from P_σ and RP_σ (lines 8-10). If RP_σ becomes empty then pop the state off from the state stack St (lines 11-12).

– If after executing the step the last state has non-empty RP_σ then generate further interleaving from σ (line 14).

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. 6. MSPOE with redundant exploration

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

- From P_σ, choose a match-set m according to POE's prioritized match-set selection procedure (line 7).
- Add m to RP_σ (line 8).
- If m involves a deterministic receive, then search for each state σ' in the stack St and perform the following: (1) If $P_{\sigma'}$ contains a match-set m' involving a send from the same process whose send is a part of m at P_σ then add m' to $RP_{s'}$ (lines 10, 14-15). (2) However, if $P_{s'}$ contains a barrier operation MB ordered with the send that is part of m then terminate (lines 10-12) and move-on to explore the next state in the interleaving (line 19). (3) While generating the new state we fire the state transition rules described in Section 2.1. Consider the example shown in Figure 6. 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. This serves as a favorable optimization for MSPOE.
- 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 it did not complete within the time-limit. Example that were deadlock-free and did not finish in 2 hours were independently run on ISP and were verified to be deadlock-free. The results pertaining to the reductions obtained are documented in Table 5. 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.

Table 1. Interleaving results for deadlock detection

				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 √	0.013
	∞	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	20.19
		8	No	2401	2401	1806.738
Madre	0	4	Yes	1	1 √	0.05
	∞	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
Heat Diffusion	0	8	Yes	5041 √	23 √	12.033

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. 7. Communication in 2D-Diff

2D-Diffusion: We tested ISP's POE and MSPOE algorithm on *2D-Diffusion* [2] example. The code has a deadlock when evaluated in *zero buffering mode*. In this mode, the 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 7. 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` [2] is another benchmark that uses heavy non-determinism to compute an integral of sin function over the interval $[0, Pi]$. `Integrate` has a master-

```
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. 8. High-level Code Pattern of "Integrate"

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 work are provided with more tasks until all tasks are distributed (as detailed in the high level code in Figure 8).

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 [11], 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 5.

Parmetis: Parmetis [7] 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.

Heat Diffusion: This is the benchmark obtained from the Supecomputing 2011 tutorial presented by T. Hilbrich, G. Gopalakrishnan and others. The benchmark solves the heat equation on a 2-D grid. ISP discovered the deadlock in 5041^{st} interleaving after running for almost 2.5 hours, however, the same deadlock was discovered by MSPOE in mere 23 interleavings running for approximately 12 seconds.

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 [15]), 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 9, 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 9, 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 10 is another example where MSPOE will fail to detect a deadlock. In Figure 10, 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

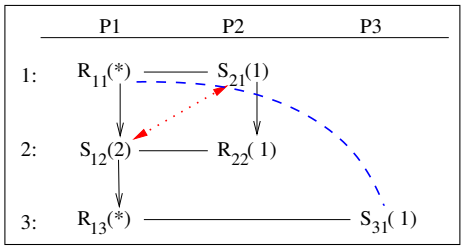

Fig. 9. Deadlock because cyclic dependency between $S_{1,2}$ and $S_{2,1}$

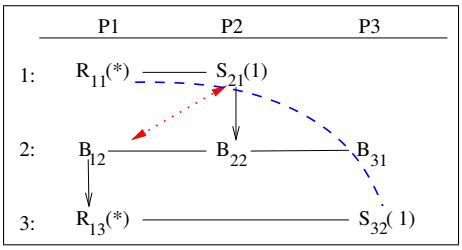

Fig. 10. Deadlock because barriers do not discharge

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 incomplete works extremely well (in other 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 an 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. FEVS Benchmark. http://vsl.cis.udel.edu/fevs/index.html.
3. 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.
4. 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.
5. P. Godefroid, M. Levin, and D. Molnar. Whitebox fuzzing for security testing. *Communications of the ACM*, Mar. 2012.
6. 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.
7. G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *SuperComputing (SC)*, 1996.
8. http://www.multicore-association.org.
9. Message Passing Interface. http://www.mpi-forum.org/docs/mpi-2.2.
10. V. Sarkar, W. Harrod, and A. Snavely. Software challenges in extreme scale systems. *SciDAC Review Special Issue on Advanced Computing: The Roadmap to Exascale*, pages 60–65, Jan. 2010.
11. S. F. Siegel. The MADRE web page. http://vsl.cis.udel.edu/madre, 2008.
12. S. F. Siegel. MPI-Spin web page. http://vsl.cis.udel.edu/mpi-spin, 2008.
13. S. Vakkalanka. *Efficient dynamic verification algorithms for MPI applications*. PhD thesis, University of Utah, Salt Lake City, Ut, USA", 2010.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.