Dynamic Reduction based Verification of Slack Inelastic Message Passing Systems*

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

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

Abstract. In message passing distributed systems, *blocking send* commands can acquire a *non-blocking* behavior if the underlying runtime provides sufficient *slack*, *i.e.*, buffer space to fully absorb messages being sent. Programs in many concurrent languages are *slack elastic*, *i.e.*, adding slack does not introduce bugs. However, programs using MPI – the most widely used API for parallel programming – are slack *inelastic*. We provide the first approach we know to verify slack inelastic programs, while avoiding the naïve approach of considering every send with/without slack. In MPI programs, added slack causes new behaviors by unblocking *send*s, causing *later send*s to issue. The new sends can compete for non-deterministic (wildcard) receive matches, adding new behaviors. We characterize the complexity of Minimal Slack Enumeration (MSE). We then provide a practical algorithm called $\mathrm{POE}_{\mathrm{MSE}}$ that first considers a slack-free execution, performs MSE to discover where slack matters, and integrates dynamic partial order reduction and MSE. Experimental results show that our new algorithm is efficient on real examples, while handling slack inelasticity soundly.

1 Introduction

A majority of large-scale parallel programs used in science and engineering research are written using the Message Passing Interface (MPI [8]) application program interface (API), and are run on expensive supercomputers consisting of thousands of CPUs. An MPI program, typically written in languages such as C, represents a large-scale distributed computation, employing communication and synchronization calls to other MPI processes through MPI/user library functions. The MPI library is large, consisting of over 300 functions; user libraries are also large, containing legacy units. Many aspects of an MPI program are unknown statically (*e.g.*, the process targeted by an MPI send can be described through expressions). All this makes dynamic methods based on direct execution of the code [11,12,13] a practical approach for verification. Since MPI programs are highly optimized for performance, they are prone to many bugs such as deadlocks, MPI object leaks, assertion violations, and unintended communication matches (also known as communication races). Conventional MPI program testing methods are inadequate for tracking down these bugs mainly because they do not have schedule (interleaving) control methods such as partial order reduction (POR) that minimize wasted interleavings. Partial order methods are especially important for MPI, as MPI processes compute in separate memory spaces, and also most MPI calls commute.

* Supported in part by Microsoft and NSF award CNS00509379

Putting all these facts together, dynamic partial order reduction (DPOR) methods are essential to handle large MPI programs.

One hopes that message passing distributed systems are *slack elastic*. That is, in the absence of slack (system-provided buffering), sends degenerate into rendezvous sends, and with slack, they *safely* turn into non-blocking sends (conducive to higher overall program performance) – *i.e., no new deadlocks or safety violations are introduced by adding slack.* Unfortunately, MPI programs are *slack inelastic*: increased system-provided buffering can introduce these bugs. Our contribution here is a dynamic model checking algorithm that correctly and efficiently handles the slack inelasticity of MPI.

Background: Our previous work [4,6] contributed to dynamic stateless verification of MPI programs by offering a tool called ISP that has been used to check many practical MPI programs. One recent result [6] pertaining to ISP is the model checking of a 14KLOC MPI program with multiple processes in about five seconds.[1] The DPOR algorithm used in ISP is called POE (Partial Order considering Elusive interleavings). In addition to reducing interleavings, POE also ensures that the required interleavings are actually exercised (§ 2). However, POE does not handle MPI's slack inelasticity.

Related Work: In [2], language restrictions that disallowed non-deterministic send/receive message matches were proposed by Manohar and Martin as a means of guaranteeing slack *elasticity* (they coined this term). Siegel and Avrunin [7] made a very similar observation – that MPI programs without *wildcard receives* are slack elastic (adding slack may eliminate deadlocks, but never introduce them). *Wildcard receives* are non-deterministic receive statements that can match one of many *competing* sends targeting this receive. [7] proposes the following verification approach: verify an MPI program in their subset by allocating *zero* (we strictly mean *insufficient*) slack to the sends. If no deadlocks exist, it is possible to conclude that slack allocation will not introduce deadlocks. It is helpful to keep in mind that programs written within slack elastic subsets of MPI may still have deadlocks, *e.g.*, when two constituent processes first post message sends targeting each other, and subsequently post receives matching the sends (the so called *head-to-head* deadlocks). Such deadlocks vanish when slack is allocated.

Importance of handling slack inelasticity: Past research has studied how lossy buffers affect model checking; we are unaware of any work on how slack addition affects model checking. There is an explosion of software libraries and APIs under consideration for programming parallel systems (e.g., the Multicore Communications API [14]): the presence of non-deterministic receives renders any such API slack inelastic. The user community of MPI is very large (*e.g.*, every proposed Petascale supercomputer will primarily run MPI) and very diverse, making it mandatory to have the ability to verify MPI programs outside of the aforesaid syntactic restrictions. In addition, MPI libraries often employ wildcard receives.

Contributions, Roadmap: Given an MPI program, $\mathrm{POE}_{\mathrm{MSE}}$ can determine all the sends in the program that can match each instance of a wildcard receive. It does this by *exactly* computing the set of sends that can be co-enabled with the receive. This is a non-trivial calculation in MPI as explained momentarily. While co-enabledness over-approximation is easier (and may serve other purposes, such as computing persistent

[1] Without any POR, just five MPI processes with only five instructions each can generate in excess of 10^{10} interleavings – impractical for a dynamic model checker.

sets as in [1]), it does not serve our purposes. For us, an over-approximated set of co-enabled sends will contain those that cannot match a receive. If a dynamic model checker's scheduler is asked to force these matches, it will deadlock.

Computing co-enabledness is tricky in MPI because MPI function invocations can *complete out of program order – e.g.*, if a process P1 sends a megabyte to P2 and then a byte to P3, there is no reason why the second send should not be allowed to finish first. However, if both sends are destined to P2, the MPI runtime must finish them in issue order, in order to guarantee *non-overtaking* [8] (between any pair of processes, messages are to be delivered in FIFO order). In § 3, we formally define the *completes-before* relation to capture MPI's required completion ordering. Using completes-before, we can accurately determine whether a send S and a potentially matching wildcard receive R can be co-enabled.

This work shows that slack addition also affects co-enabledness fundamentally. Through $\mathrm{POE}_{\mathrm{MSE}}$, we can determine the minimal set of sends $S_1 \ldots S_n$ that must be provided slack in order to allow *some other* S and R to be co-enabled. We call this calculation the *minimum slack enumeration* (MSE) problem. MSE is a search problem over the completes-before graph of the execution whose complexity is shown to be #P-complete (§ 4). Finally, POE, MSE, and the DPOR algorithm of [1] are integrated to form our final $\mathrm{POE}_{\mathrm{MSE}}$ algorithm. In $\mathrm{POE}_{\mathrm{MSE}}$, we employ two alternating phases of state space reductions: interleaving reduction through POE based DPOR, and slack enumeration reduction through MSE. In summary, our *new* results in this paper are: (i) the formalization of MPI that captures non-overtaking as well as slack (Section 3), and the definition of completes-before based on this semantics, (ii) a new formulation of POE [4] through prioritized execution of the semantic rules (Figure 4(a)), (iii) the $\mathrm{POE}_{\mathrm{MSE}}$ algorithm, and (iv) experimental evaluation on realistic examples. In [3] we describe how we have validated our MPI semantics.

2 Illustration and Formal Modeling of MPI

We begin our presentation of MPI's formal semantics by noting that individual MPI processes compute in separate memory spaces, and interact *only* through MPI calls. MPI programs are required to terminate, with all processes calling `MPI_Finalize`. This meets the acyclic state space condition of [1]. This also permits us to model MPI processes as a straight-line abstract MPI call sequences defined by the control flow, and leave out all C constructs (see Figure 1; in [3], we present the C/MPI codes).

The MPI function calls that we consider in this paper are send (S), receive (R), wait (W), and barrier (B) (our implementation deals with well over 60 MPI functions, sufficient to write a large class of MPI programs). S stands for MPI's non-blocking send – called `MPI_Isend` in the MPI library. An invocation of S initiates sending in the background, and lets the invoking process proceed with its execution. Similarly, R stands for MPI's non-blocking receive, namely `MPI_Irecv`, which initiates receive in the background. W stands for MPI's `MPI_Wait` function. Such a call refers to a previously issued S or R via *handle*s. Frequently, programmers treat the interstices of an S and the following W as a delay slot, filling it with unrelated computations. We use S^b to denote an instance of S for which the MPI runtime provides adequate buffering.

An invocation of W that refers to an S operation must block until the S had a chance to copy its message out of the process memory space, and into either (i) the

P_0	P_1	P_2
$S_{0,1}(1)$	$S_{1,1}(2)$	$R_{2,1}(*)$
$W_{0,2}(\langle 0,1\rangle)$	$W_{1,2}(\langle 1,1\rangle)$	$W_{2,2}(\langle 2,1\rangle)$
$S_{0,3}(2)$	$R_{1,3}(0)$	$R_{2,3}(0)$
$W_{0,4}(\langle 0,3\rangle)$	$W_{1,4}(\langle 1,3\rangle)$	$W_{2,4}(\langle 2,3\rangle)$

(a) A Simple MPI Example

P_0	P_1	P_2
$S_{0,1}(1)$	$\mathbf{B}_{1,1}$	$R_{2,1}(*)$
$W_{0,2}(\langle 0,1\rangle)$	$S_{1,2}(2)$	$\mathbf{B}_{2,2}$
$S_{0,3}(2)$	$W_{1,3}(\langle 1,2\rangle)$	$W_{2,3}(\langle 2,1\rangle)$
$\mathbf{B}_{0,4}$	$R_{1,4}(0)$	$R_{2,4}(0)$
$W_{0,5}(\langle 0,3\rangle)$	$W_{1,5}(\langle 1,4\rangle)$	$W_{2,5}(\langle 2,4\rangle)$

(b) A Simple Example with Barriers

Fig. 1. Simple MPI programs to illustrate Slack Inelasticity

system memory space (in case of S^b) or (ii) into the receiver's space. In case (i), the W associated with the S^b can return immediately (in effect, W *is turned into a no-op*). In case (ii), W blocks till the entire message is received. A barrier B (`MPI_Barrier`) is a *collective* call, meaning it must be invoked by all the processes. No process may execute past its B call until all processes have issued their B calls. However, a send that is issued before a barrier may be alive at the time the instructions past B are executed.

Let $Nat = \{0, 1, 2 \ldots\}$, $Bool = \{F, T\}$, and $Bool_\perp = \{F, T, \perp\}$. Let P be the number of MPI processes in an MPI program, with their PIDs (a.k.a MPI ranks) be the set $PID = \{0 \ldots P-1\}$. Let $PID_* = \{0 \ldots P-1\} \cup \{*\}$. Each MPI process is viewed as a sequence of instructions, with lengths $L \in PID \to Nat$. For any function f, its application to argument i is written f_i; for example, L_1 is the length of the first process. The application of a two-ary function f to arguments i and j is written $f_{i,j}$, and its partial application to one argument i is written f_i. Let the program counter values be given by $l \in PC = PID \to Nat$ (l stands for "location"). We denote the individual program counters $l_0 \ldots l_{P-1}$. The MPI runtime is viewed as a special process with process ID P and always at the same PC. Let $p \in PID \to Nat \to OP$ be an MPI program, where $OP \in \{S_{i,j}(k), R_{i,j}(k), W_{i,j}(\langle m, n\rangle), B_{i,j}\}$ for $i, j, k, m, n \in Nat$; we often take $OP = \{S, R, W, B, \perp\}$. For example, $p_0 \ldots p_{P-1}$ are the P processes, and the jth instruction of the ith process is $p_{i,j}$. Associated with any instruction $p_{i,j}$ is a *handle* $\langle i, j\rangle$ uniquely identifying the instruction. The set of handles $H = PID \times Nat$. An MPI operation $op \in OP$ is the jth instruction of some process i. One can write such an operation $op_{i,j}(\ldots args \ldots)$ where $j = l_i$ and $op \in OP$. Let $f[i \leftarrow e]$ be function update, i.e. $f[i \leftarrow e] = (f \setminus \{\langle i, f(i)\rangle\}) \cup \{\langle i, e\rangle\}$.

Slack Inelasticity: Consider the example in Figure 1(a) in which the first send of P_0, *i.e.* $S_{0,1}(1)$, aims to send a data payload (not shown) to process P_1. Its corresponding wait is $W_{0,2}(\langle 0,1\rangle)$. The receive that this send will match with is $R_{1,3}(0)$ whose corresponding wait is $W_{1,4}(\langle 1,3\rangle)$. The rest of the code can be read similarly. Suppose none of the S are being treated as an S^b. This means that the W calls corresponding to the S will block. Therefore, the presence of $W_{1,2}(\langle 1,1\rangle)$ in P_1 **forces** $S_{1,1}(2)$ to match with $R_{2,1}(*)$. Finally, $S_{0,3}(2)$ matches $R_{2,3}(0)$. Since all the sends and receives are matched, all the wait (W) calls unblock and the program terminates.

Now consider the case when $S_{0,1}(1)$ of P_0 is treated as an S^b. *In this case,* $W_{0,2}$ *is, in effect, turned into a no-op,* which enables P_0 to execute $S_{0,3}(2)$. This leads to P_0's $S_{0,3}(2)$ and P_1's $S_{1,1}(2)$ *to be co-enabled* with $R_{2,1}(*)$. Suppose the MPI runtime

non-deterministically chooses to match $S_{0,3}(2)$ with $R_{2,1}(*)$. This means that $R_{2,3}(0)$ of P_2 will no longer have a matching send – a deadlock!

In summary, if one adds buffering to some $S_{i,j}(k)$, the wait associated with this send, say $W_{i,k}(\langle i,j\rangle)$, is rendered a no-op, and this can *break some of the completes-before edges*. This may, in turn, co-enable some (seemingly unrelated) sends $S_{p,q}(r)$ with some wildcard receive, thus increasing non-determinism.

A Recap of POE**:** We use our current examples to illustrate two ideas – namely deliberate out of order execution, and dynamic rewriting – that already existed in POE [4], and are part of $\mathrm{POE_{MSE}}$ also. Consider Figure 1(b) (obtained by inserting a few barrier B calls into Figure 1(a)), where $S_{0,1}(1)$ is still treated as an S^b. Even with these Bs added, P_0's $S_{0,3}(2)$ and P_1's $S_{1,2}(2)$ can still compete for $R_{2,1}(*)$, because: (i) after issuing (but not completing) both $S_{0,3}(2)$ and $R_{2,1}(*)$, the Bs can be issued *and completed*, (ii) this now permits $S_{1,2}(2)$ to be also issued.

Suppose someone wants to build a dynamic model checker that explores the competing sends individually: in our example, they want to pursue the interleaving caused by matching $S_{1,2}(2)$, and then in another interleaving pursue $S_{0,3}(2)$. They will realize that the presence of MPI barriers does not allow this, in general; here, $S_{1,2}(2)$ cannot be issued unless $S_{0,3}(2)$ is also issued (due to the presence of the Bs). Unfortunately, if both sends are issued, the (unfair) MPI runtime may *always* pick $S_{0,3}(2)$. POE gets around this problems by intercepting the Ss and delaying their issue into the MPI runtime *as late as possible without breaking completes-before*. In our example, POE will collect (but not issue) $S_{0,3}(2)$, issue and complete the barrier B, and *then* issue either $S_{0,3}(2)$ or $S_{1,2}(2)$. Such dynamic reorderings are safe (guarantees non-overtaking).

Second, POE is capable of forcing specific matches to occur by rewriting $R_{2,1}(*)$ into $R_{2,1}(1)$ and $R_{2,1}(2)$, and issuing two "packets" (*matches* in § 3) into the MPI runtime, one containing $\{S_{1,2}(2), R_{2,1}(1)\}$ and the other containing $\{S_{0,3}(2), R_{2,1}(0)\}$. If we fire $R_{2,1}(*)$ into the MPI runtime, we lose external control over which send will match this wildcard receive.

3 MPI Semantics

The state of execution of an MPI program and its runtime is modeled using the triple $\langle p, l, C\rangle$. MPI program semantics is modeled through inference rules where the $\langle\langle p, l, C\rangle\rangle$ in the antecedent and consequent stands for $\langle p, l, C\rangle \in ReachSet$, where $ReachSet$ is the reached set of states. Changes to p models instructions being consumed. Changes to l models the PC advancing. C is a set of communication records modeling the MPI runtime state. For $c \in C$, $c = \langle pid, op, blocking, src, dest, handle, match, cpl, buff\rangle$ where $pid \in PID$, $op \in OP$, $blocking \in Bool$, $src \in PID_*$, $dest \in PID$, $handle \in H$, $match \subseteq H$, and $cpl, buff \in Bool$ (standing for "completed" and "buffered.") Figure 2 introduces four *process transitions* (denoted $\mathbf{P}-$) that model how MPI instructions are issued. Each $\mathbf{P}-$ generates and adds one communication record to C. Figure 3 introduces *runtime transitions* (denoted $\mathbf{R}-$). The $\mathbf{R}-$ help match and complete the communication records by updating the $match$ and cpl fields. A *run* of an MPI program is defined as any allowed sequence of $\mathbf{P}-$ and $\mathbf{R}-$ transitions. POE will be presented as an *a prioritized execution of these transitions*.

Definitions: Two transitions *may be co-enabled* if there is some state where both transitions are enabled. Two transitions are *independent* iff whenever they may be co-enabled

$$\mathbf{PS}: \frac{\langle\!\langle\ p, l, C\ \rangle\!\rangle,\ \ p_{i,l_i} = S_{i,l_i}(j)}{\langle\!\langle\ p_i[l_i \leftarrow \bot], l[i \leftarrow l_i + 1], C \cup \{\langle i, S_{i,l_i}(j), F, \bot, j, \bot, \emptyset, F, F\rangle\}\ \rangle\!\rangle}$$

$$\mathbf{PR}: \frac{\langle\!\langle\ p, l, C\ \rangle\!\rangle,\ \ p_{i,l_i} = R_{i,l_i}(j)}{\langle\!\langle\ p_i[l_i \leftarrow \bot], l[i \leftarrow l_i + 1], C \cup \{\langle i, R_{i,l_i}(j), F, j, \bot, \bot, \emptyset, F, F\rangle\}\ \rangle\!\rangle}$$

$$\mathbf{PW}: \frac{\langle\!\langle\ p, l, C\rangle,\ \ p_{i,l_i} = W_{i,l_i}(\langle i, j\ \rangle\!\rangle)}{\langle\!\langle\ p_i[l_i \leftarrow \bot], l, C \cup \{\langle i, W_{i,l_i}, T, \bot, \bot, \langle i, j\rangle, \emptyset, F, F\rangle\}\ \rangle\!\rangle}$$

$$\mathbf{PB}: \frac{\langle\!\langle\ p, l, C\ \rangle\!\rangle,\ \ p_{i,l_i} = B_{i,l_i}}{\langle\!\langle\ p_i[l_i \leftarrow \bot], l, C \cup \{\langle i, B_{i,l_i}, T, \bot, \bot, \bot, \emptyset, F, F\rangle\}\ \rangle\!\rangle}$$

Fig. 2. MPI Process Transitions

in a state, (i) the firing of one does not disable the other, and (ii) the same state is attained no matter which order they are fired in.

Process Transitions: Process transitions are defined through the inference rules in Figure 2. **PS** models the issue of a S (send) operation of process p_i at PC l_i. The instruction is consumed (modeled by $p_i[l_i \leftarrow \bot]$), the PC advances ($l[i \leftarrow l_i + 1]$), and a communication record is added into the MPI runtime (no $match$ing receive is found yet, cpl is false, and $buff$ is F). Rule PR models the issuing of a *specific receive* (R). The issuing of a wildcard receive call can be similarly modeled (not shown). We can see from PW and PB that the W or B function calls do not advance the PC. This helps model the blocking behavior of these instructions. Since every process transition $p_{i,j}$ creates a communication record, we associate a communication record with $c_{i,j}$ with $p_{i,j}$. A communication record is discarded only when its $match \neq \emptyset$ and its $cpl = T$. Otherwise, the communication record is considered to be alive. For example, a receive must first be *matched* with a send and the receive operation is *completed* when the data is transferred from send buffer to the receive buffer.

Runtime Transitions: Runtime transitions help define how transitions help complete MPI operations. The relation *completes-before* mentioned on Page 5 will be defined as $CB = IntraCB \cup InterCB$, where $InterCB$ will be defined in Section 4. We now define $IntraCB$– the relation that defines how instructions within a process complete (hence formalizing non-overtaking).

Definition 1. *$IntraCB$ is the smallest transitively closed subset of $C \times C$ s.t. for $c_{i,j}$, $c_{i,k} \in C$ where $j < k$, $\langle c_{i,j}, c_{i,k}\rangle \in IntraCB$ whenever one of these hold:*

1. $c_{i,j}.op = S_{i,j}(l)$ **and** $c_{i,k}.op = S_{i,k}(l)$ *where* $l \in PID$
2. $c_{i,j}.op = R_{i,j}(l)$ **and** $c_{i,k}.op = R_{i,k}(l)$. *where* $l \in PID_*$
3. $c_{i,j}.op = R_{i,j}(*)$ **and** $c_{i,k}.op = R_{i,k}(l)$. *where* $l \in PID_*$[2]
4. $c_{i,j}.op = S_{i,j}(l)$ **and** $c_{i,k}.op = W_{i,k}(\langle i, j\rangle)$ *where* $l \in PID$
5. $c_{i,j}.op = R_{i,j}(l)$ **and** $c_{i,k}.op = W_{i,k}(\langle i, j\rangle)$ *where* $l \in PID_*$
6. $c_{i,j}.op = B_{i,j}$ **or** $c_{i,j}.op = W_{i,j}$

If $\langle c_{i,j}, c_{i,k}\rangle \in IntraCB$, we say that $c_{i,j}$ is $IntraCB$ predecessor of $c_{i,k}$. Let $s.C$ denote the the set of communication records that are alive in state s. We now introduce $s.C_R \subseteq s.C$ to denote the *ready set of communication records that can be matched/completed*:

[2] This condition covers the case where the first receive is from any source, while the previous case covers two receives from the same source.

Definition 2. *Given a state* $s = \langle p, l, C\rangle$, $c_{i,k} \in s.C$ *and* $\forall \langle c_{i,j}, c_{i,k}\rangle \in IntraCB$,
$\mathbf{s.C_R} = \{c_{i,k} \mid c_{i,k}.buff = T \ \vee \ (c_{i,k}.match = \emptyset \ \wedge \ c_{i,j}.match \neq \emptyset) \vee$
$(c_{i,k}.cpl = F \ \wedge \ c_{i,k}.match \neq \emptyset)\}$

Let $s.C_M \subseteq s.C_R$ be: $\mathbf{s.C_M} = \{c \in C_R \mid c.match = \emptyset \ \wedge \ c.buff = F\}$
$s.C_M$ is also called as co-enabled set of communication records. Basically, $s.C_M$ is the set of communication records that have not yet matched, and are eligible for matching. They eventually complete through $\mathbf{R}-$ transitions, provided that their $IntraCB$ predecessors have been matched. We can show that if $\langle c_{i,j}, c_{i,k}\rangle \in IntraCB$, then by the MPI semantics $c_{i,j} \in s.C_M \Leftrightarrow c_{i,k} \notin s.C_M$. That is, $c_{i,j}$ can never be co-enabled with $c_{i,k}$. We can now define MPI runtime transitions, as shown in Figure 3. We employ a convenient notational abbreviation introduced through a simple example:

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

$$\mathbf{RR}: \frac{\langle\!\langle\ p, l, C\ \rangle\!\rangle,\ c_{x,i}, c_{y,j} \in C_M,\ c_{x,i}.op = R_{x,i}(y),\ c_{y,j}.op = S_{y,j}(x),\ y \in PID}{\langle\!\langle\ p, l, C : c_{x,i}[match \leftarrow @ + \langle y, j\rangle]\ \rangle\!\rangle}$$

$$\mathbf{RR}*: \frac{\langle\!\langle\ p, l, C\ \rangle\!\rangle,\ c_{x,i}, c_{y,j} \in s.C_M, c_{x,i}.op = R_{x,i}(*),\ c_{y,j}.op = S_{y,j}(x),\ y \in PID}{\langle\!\langle\ p, l, C : c_{x,i}[match \leftarrow @ + \langle y, j\rangle, src \leftarrow y, op \leftarrow R(y)]\ \rangle\!\rangle}$$

$$\mathbf{RS}: \frac{\langle\!\langle\ p, l, C\ \rangle\!\rangle,\ c_{i,j} \in s.C_M, c_{k,l} \in s.C_R,\ c_{i,j}.op = S_{i,j}(k),\ c_{k,l}.op = R_{k,l}(i),\ c_{k,l}.match = \{\langle i, j\rangle\}}{\langle\!\langle\ p, l, C : c_{i,j}[match \leftarrow @ + \langle k, l\rangle, cpl \leftarrow T]\ \ c_{k,l}[cpl \leftarrow T]\ \rangle\!\rangle}$$

$$\mathbf{RW}: \frac{\langle\!\langle\ p, l, C\ \rangle\!\rangle,\ c_{i,k} \in s.C_M,\ c_{i,k}.op = W_{i,k}(\langle i, j\rangle)}{\langle\!\langle\ p, l[i \leftarrow l_{i,k} + 1], C : c_{i,k}[match \leftarrow @ + \langle i, k\rangle, cpl \leftarrow T]\ \rangle\!\rangle}$$

$$\mathbf{RB}: \frac{\langle\!\langle\ p, l, C\ \rangle\!\rangle,\ C_1 \subseteq s.C_M,\ \mid C_1 \mid = \mid PID \mid \forall c_{i,j} \in C_1 :\ c_{i,j}.op = B_{i,j}}{\langle\!\langle\ p, l[i \leftarrow l_{i,j} + 1], C : \forall c_{i,j} \in C_1\ :\ c_{i,j}[match \leftarrow @ + \langle i, j\rangle, cpl \leftarrow T]\ \rangle\!\rangle}$$

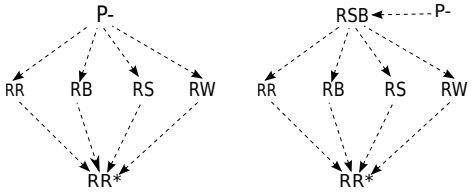
Fig. 3. MPI Runtime Transitions

Consider the $\mathbf{RR}$ transition as an example. It fires precisely when a specific receive finds a matching send. Consider $\mathbf{RR}*$: it fires when a send targeting the process of this receive is found. Notice that we set the src of this receive to y, thus modeling dynamic rewriting of wildcard receives.

POE: The Hasse diagram given in Figure 4(a) defines how POE works by firing transitions from higher toward lower priority order (the arrows point towards lower priorities). Giving $\mathbf{RR}*$ the lowest priority helps force the maximal set of sends to match a wildcard receive (otherwise, the **C1** condition [9] is violated). The proof of soundness for POE algorithm is given in [3].

4 The $\mathbf{POE_{MSE}}$ Algorithm

To model slack, we add the MPI runtime transition called **RSB** shown in Figure 4(c) when there is buffering available for a Send (an S^b transition). **RSB** simply sets the completion bit, as the system buffer can instantaneously absorb the message even before a match is found. When a send in buffered, the W operation corresponding to the send



(a) POE (b) $\mathrm{POE}_{\mathrm{MSE}}$

$$\frac{\langle\!\langle\ p, l, C\ \rangle\!\rangle, c_{i,j} \in s.C_R,\ c_{i,j}.buff}{\langle\!\langle\ p, l, C : c_{i,j}[cpl \leftarrow T, buff \leftarrow F]\ \rangle\!\rangle}$$

(c) RSB Transition

Fig. 4. Prioritized Execution Orders for POE and $\mathrm{POE}_{\mathrm{MSE}}$

turns into a no-op, and all communication records generated by it (including within $IntraCB$) are removed. Figure 4(b) gives the priority execution order with the **RSB** transition. In the MSE phase of POE_{MSE}, the sends endowed with slack are fired as per the **RSB** rule. To determine these sends, we need the notion of $InterCB$ which establishes orderings **across** processes, building over $IntraCB$, as now explained.

4.1 InterCB Edges and Path Properties

The theorems in this section are proven in our technical report [3].

Definition 3. $Intra(c_{i,j}) = \{c_{i,k} |\ \langle c_{i,j}, c_{i,k}\rangle \in IntraCB\}$

Theorem 1. *For $c_{i,j}$, $c_{k,l} \in s.C_M$ and $c_{i,j}.op = R_{i,j}(*)$ and $c_{k,l}.op = S_{k,l}(i)$, for any $c \in Intra(c_{k,l})$, and for any state s', $c_{i,j} \in s'.C_M \Leftrightarrow c \notin s'.C_M$. That is, if $R_{i,j}(*)$ is matchable with $S_{k,l}(i)$ and are co-enabled, then an $IntraCB$ successor of $S_{k,l}(i)$ can never be co-enabled with $R_{i,j}(*)$.*

Theorem 2. *For two communication records $c_{i,j}$, $c_{k,l}$ such that $c_{i,j}.op \neq R_{i,j}(*)$ and $c_{k,l}.op \neq R_{i,j}(*)$, if $\langle i, j\rangle \in c_{k,l}.matches$ then for all $c \in Intra(c_{i,j})$ and any state s', $c \in s'.C_M \Leftrightarrow c_{k,l} \notin s'.C_M$.*

The above theorems say that if $c_{i,j}, c_{k,l}$ are co-enabled in some state s, then $c_{i,j}$ and any communication record in $Intra(c_{k,l})$ can never be co-enabled. We define $InterCB$ based on the above theorems.

Definition 4. *$InterCB$ is the smallest transitively closed subset of $C \times C$ such that for some state s:*

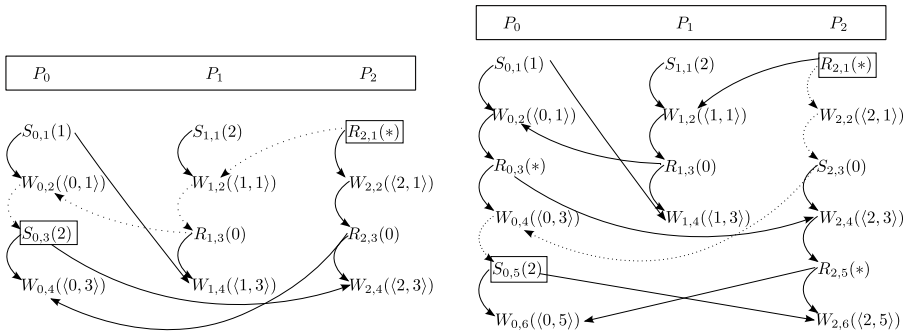
1. *If $c_{i,j}, c_{k,l} \in s.C_M$ are such that $c_{i,j}.op = R_{i,j}(*)$ and $c_{k,l}.op = S_{k,l}(i)$, then $\forall c \in Intra(c_{k,l})$, $\langle c_{i,j}, c\rangle \in InterCB$.*
2. *For a communication record $c_{i,j}$ where $c_{i,j}.op \neq W_{i,j}(\langle i, m\rangle)$, $\forall \langle k, l\rangle \in c_{i,j}.matches$, $\forall c \in Intra(c_{k,l})$, we have $\langle c_{i,j}, c\rangle \in InterCB$.*

The $InterCB$ is constructed at the end of an interleaving.

Definition 5. *For a set of communication records C, define $CB(C) = IntraCB(C) \cup InterCB(C)$. A completes before graph for a given interleaving (execution) $s_0, s_1, \ldots$ is $CB_G = (V, E)$ where $V = s_0.C \cup s_1.C \ldots \cup s_n.C$, and $E = CB(V)$.*

Co-enabledness: From Theorems 1, 2, we can show that if there is a path from $c_{i,j}$ to $c_{k,l}$ in CB_G, then $c_{i,j}$ and $c_{k,l}$ can never be co-enabled.

Slack Introduction Illustration: Figure 5(a) is the CB_G obtained for the MPI program of Figure 1(a) for the interleaving generated by the POE algorithm during the initial slack-free execution. Here, the arrows within a process denoting $IntraCB$ and those across denoting $InterCB$. Notice that **all** paths from $R_{2,1}(*)$ to $S_{0,3}(2)$ involve $W_{0,2}(\langle 0,1\rangle)$. Thus, if we render $W_{0,2}(\langle 0,1\rangle)$ into a no-op by firing $S_{0,1}(1)$ as an S^b, the dotted path is broken, making $S_{0,3}(2)$ co-enabled with $R_{2,1}(*)$, leading to a deadlock as discussed. However, if there is some path that does not contain any waits associated with a send, then introducing slack does not break such paths, as shown in Figure 5(b). $\mathrm{POE}_{\mathrm{MSE}}$ avoids pursuing such unproductive slack introductions.



(a) Path that can be broken (b) Path that cannot be broken

Fig. 5. Example showing IntraCB, InterCB and Path

Minimal Wait sets: In the following, we assume that $c_{i,j}$ corresponds to wildcard receive and $c_{k,l}$ corresponds to a potential matching send.

Definition 6. *For a given communication record c with $c.op = W$, $onpath(c)$ is the set of all simple paths of CB_G on which c occurs.*

Definition 7. *Let π be the set of all simple paths of CB_G on which $c_{i,j}$ and $c_{k,l}$ both appear. Let $W_{all} = \{c \mid c.op = W(\langle m,n\rangle) \wedge c_{m,n}.op = S_{m,n}(\ldots) \wedge c_{m,n}.buff = F \wedge \exists p \in \pi : p \in onpath(c)\}$. These are the Ws which can be turned into a nop, and they lie on some $p \in \pi$. A minimal wait set W_{min} is the set of waits such that (i) $W_{min} \subseteq W_{all}$, and (ii) for any path $p \in \pi$, and for any two waits w' and w'' in W_{min}, we have $w' \in p \Leftrightarrow w'' \notin p$, and furthermore (iii) $\bigcup_{x \in W_{min}} onpath(x) = \pi$.* That is, there is exactly one wait in W_{min} for any path between $c_{i,j}$ and $c_{k,l}$, but that all paths are covered by some (not necessarily distinct) W.

Theorem 3. *Given a set of paths π between $c_{i,j}$ and $c_{k,l}$, finding W_{min} is NP-Complete.*

Proof. (Sketch; discussions near Definition 8) The reduction is from the monotone-1-in-3 sat to our problem. The above problem is in NP. Given a certificate W_c, we can easily check that each path has exactly one wait in W_c. A monotone-1-in-3 SAT

formula f is 3-CNF that has no negations and must be satisfied by assigning exactly one literal in every clause to true. Given a formula f, let v represent the set of variables and c be the set of clauses. v represents W_{all}. We construct a completes-before graph $CB_G = (V, E)$ with $V = v$ and for every clause $c_i = (x_i \vee x_j \vee x_k)$, we add an edge between x_i, x_j and x_j, x_k. That is $x_i \rightarrow x_j \rightarrow x_k$ forms a path in the graph. We also mark x_i and source vertex and x_k as sink vertex. A path consists of exactly one source and one sink vertex. If W_{min} exists for each of the paths, then f can be made true by assigning true to the variables corresponding to vertices in W_{min} and vice versa [3].

Theorem 4. *Finding all the minimal wait sets is #P-Complete.*

Proof. From Theorem 4, we have a P-time reduction from monotone 1-in-3 SAT to minimal wait set problem. Also, the number of solutions to a formula f is the same as the number of minimal wait sets. So, finding all the minimal wait sets is #P-Complete.

Notes: In principle, $|\pi|$ can be exponential, but in practice, it is small. *Even so,* our complexity results show that we cannot avoid a cost that is exponential in $|\pi|$ when determining all minimal wait sets. This justifies the use of a subset construction method for finding all minimal wait sets, MWS, as captured in Definition 8. While exponential, it works well in practice. We also note that in some cases, we may be forced to pick more than one W per path p. These correspond to SAT instances that are not 1-in-3 SAT. Definition 8 handles these situations also correctly [3].

Definition 8. *Given a set of paths π between $c_{i,j}$ and $c_{k,l}$, let W_{all} be the set of waits which can be turned into a no-op. The minimal wait sets $MWS = \{W_{min} \subseteq W_{all}\}$ are the family of set of waits such that for each $W_{min} \in MWS$, turning all the waits within W_{min} disconnects $c_{i,j}$ and $c_{k,l}$, but no proper subset of W_{min} has this property.*

4.2 Minimal Slack Enumeration

We first provide an algorithm to compute MWS based on Definition 8. Given a set of paths π between $c_{i,j}$ and $c_{k,l}$, (i) take as input a set of paths π, (ii) determine W_{all}, (iii) build the powerset $\mathcal{P}(W_{all})$, (iv) sort it by ascending cardinality, and (v) eliminate from the powerset any s_i such that $s_i \supset s_j$, and s_j itself disconnects $c_{i,j}$ and $c_{k,l}$. The resulting powerset is the MWS (see [3] for a pseudo-code that defines MWS).

Recall that all these steps occur after the initial slack-free execution according to POE. Now we must re-execute the MPI program being verified by buffering the sends corresponding to MWS, and replaying the execution. We arrange for all this to occur by (i) modeling the buffered sends as S^b,[3](ii) modeling the firing of these S^b through RSB (Figure 4(b)), and (iii) marking when these RSB transitions must fire by maintaining *backtrack sets* (idea inspired by [1]) as described by our $\mathrm{POE_{MSE}}$ algorithm described in Figure 4.2. Figure 4.2 gives a full description of $\mathrm{POE_{MSE}}$ including the initial slack-free execution and the later backtrack/replay for MSE.
In addition to C_R and C_M defined in § 4, we add the following to every state s:

– $s.trans$: Set of $\mathbf{P}-$ or $\mathbf{R}-$ transitions.

[3] In implementing $\mathrm{POE_{MSE}}$, ISP implements S^b by providing ISP's own buffer resources to fake "ample buffering." Therefore, ISP can be run on any machine and still simulate slack.

– $s.backtrack$: The set of transitions that must be executed from state s in future interleavings (replays).
– $s.done$: The set of transitions that have already been executed from s (initially $\emptyset$).
– $s.curr$: The transition executed from this state in the current interleaving.
– Given a transition t, $Proc(t)$ gives the set of processes involved in the transition (*e.g.*, $\mathbf{R}-$ transitions occur by matching S and R of different processes).
– For transition t, $t.c$ is the set of all communication records involved in t.

```
1:  POE_MSE(s_0, statevec) {
2:    statevec.push(s_0);
3:    t = GetHighestPriorityTransition(s.trans − s.done);
4:    s_0.backtrack = s_0.backtrack ∪ {t};
5:    while (!empty(statevec)) {
6:      GenerateInterleaving(statevec);
7:      CB_G = GetCompletesBeforeGraph(statevec);
8:      for (i = statevec.size()−1; i ≥ 0; i−−) {
9:        s = statevec[i];
10:       UpdateBacktrackSet(s, CB_G);
11:     }
12:     for (i = statevec.size()−1; i ≥ 0; i−−) {
13:       if (statevec[i].backtrack = ∅) {
14:         statevec[i].pop();
15:       } else {
16:         break;
17:   }}}}
```

Fig. 6. Full POE_{MSE} algorithm

Figure 4.2 shows the full POE_{MSE} algorithm. The algorithm takes as input the initial state s_0 and $statevec$ which is a partial state list. The algorithm first generates an interleaving as shown in Figure 7. While generating the interleaving, the algorithm uses the Hasse diagram in Figure 4(b) to determine the priority order among the transitions. The function $Execute$ executes the transition and generates a next state as the result. Once the interleaving is generated, the algorithm generates the completes-before graph CB_G as shown in line 5 of Figure 4.2. The algorithm now invokes `UpdateBacktrackSet` for each state in the interleaving where the $s.curr$ is an $RR*$ transition.

`UpdateBacktrackSet` (Figure 8). takes the CB_G and the state s whose backtrack set must be updated. The only states considered are those involved in an $\mathbf{RR}*$ transition in the current interleaving with the receive $R_i(*)$. Line 3 checks if there is a matching send co-enabled with $c_{i,j}$ and adds it to $s.backtrack$. If there is some send $S_j(i)$ in CB_G such that $S_j(i)$ and $R_i(*)$ are not co-enabled in s (line 7), the algorithm (line $8-32$) finds all the paths π between $R_i(*)$ and $S_j(i)$ in CB_G. If there are no paths, then it finds the transition involving process j in $s.trans$ and adds it to $s.backtrack$. Otherwise, all the transitions in $s.trans$ are added to $s.backtrack$. However, if $\pi \neq \emptyset$ (lines $15 - 32$), the algorithm finds all the minimal wait sets $minwaitsets$ (line 15). If $minwaitsets = \emptyset$ then it is not possible for $c_{i,j}$ and $c_{k,l}$ to ever be co-enabled. The algorithm returns in this case. Otherwise, it looks for the sends corresponding to waits in $minwaitsets$ available in $s.C_R$ and creates new $\mathbf{RSB}$ transitions for each of these (line 24). One of these transitions are added to the $s.backtrack$ (line 29). (Note that

each $p \in minwaitset$ corresponds to set of all sends that must be buffered at the same time. In our implementation, the backtrack set is a set of sets and can execute multiple independent transitions from a state. The algorithm in Figure 8, though correct, may generate redundant interleavings). If s does not sends any sends in S_{min}, the backtrack set is updated with $s.trans$ (line 32). The soundness proof of POE$_{MSE}$ is given in [3].

5 Experimental Results, Concluding Remarks

Number of interleavings (notice the extra yet necessary interleavings of POE_{MSE})	POE_{MSE}	POE
sendbuff.c	5	1
sendbuff-1a.c	2 (deadlocked)	1
sendbuff2.c	1	1
sendbuff3.c	6	1
sendbuff4.c	3	1
sendbuff5.c	1(deadlocked)	1(deadlocked)
ParMETIS$_b$	2	1
Overhead of POE_{MSE} on ParMetis (runtime in seconds (x) denotes x interleavings)	POE_{MSE}	POE
ParMETIS (4procs)	20.9 (1)	20.5 (1)
ParMETIS (8procs)	93.4 (1)	92.6 (1)
ParMETIS$_*$	18.2 (2)	18.7(2)

Table 1. Table 1. Experiment Results: ParMETIS$_b$ is ParMETIS$_*$ with buffering. ParMETIS$_*$ is ParMETIS modified to use wildcard receives.

```
1: GENERATEINTERLEAVING(statevec) {
2:    s = statevec[0];
3:    for (i = 0; i < statevec.size()−1; i++) {
4:       s = Execute(statevec[i].curr);
5:    }
6:    s.curr = GetHighestPriorityTransition(s.backtrack);
7:    s.done = s.done ∪ {s.curr};
8:    s.backtrack = s.backtrack − {s.curr}
9:    while (s.trans − s.done ≠ ∅) {
10:      s = Execute(s.curr);
11:      s.curr = GetHighestPriorityTransition(s.trans − s.done);
12:      s.done = s.done ∪ {s.curr};
13:   }}
```

Fig. 7. Algorithm to generate an Interleaving

We provide three classes of experimental results (Table 1). First we report variants of a contrived example called `sendbuff` (see [3] for details) where we show that POE$_{\mathrm{MSE}}$ performs the minimal number of extra interleavings over POE to ensure soundness in the presence of slack. All of these examples explores POE$_{\mathrm{MSE}}$'s capabilities to detect the different matchings as well as deadlocked situations. For each of the sendbuff variant that we constructed, POE is only able to detect one possible matching, while POE$_{\mathrm{MSE}}$ allows several `Isend's` to buffer and thus discovers several more interleavings. We also reproduced our example in Figure 1(a) as sendbuff-1a.c where our algorithm indeed caught the deadlock at the second interleaving, where $S_{0,3}(2)$ is

```
1: UPDATEBACKTRACK(s, CB_G) {
2:    c_{i,j}.op = R_{i,j}(*), c_{k,l}.op = S_{k,l}(i)
3:   if (c_{i,j} ∈ s.curr.c ∧ c_{k,l} ∈ s.C_M ∧ c_{k,l} ∉ s.curr.c) {
4:      t = GetRR*Trans(c_{i,j}, c_{k,l}, s.trans);
5:      s.backtrack = s.backtrack ∪ {t}
6:   }
7:   if (∃c_{k,l} ∈ CB_G : c_{k,l} ∉ s.C_M) {
8:      π = FindAllPaths (c_{i,j}, c_{k,l}, CB_G);
9:      if (π = ∅) {
10:        if (∃t ∈ s.trans ∧ j ∈ Proc(trans))
11:           s.backtrack = s.backtrack ∪ {t};
12:        else
13:           s.backtrack = s.backtrack ∪ s.trans;
14:     } else {
15:        minwaitsets = MinimalWaitSets (π, W_all);
16:        if (minwaitsets = ∅)
17:           return;
18:        for each (p ∈ minwaitsets) {
19:           S_min =GetSendsOfWaits (p);
20:           rsbtrans = ∅;
21:           if (S_min ∩ s.C_R ≠ ∅) {
22:              for each (c ∈ S_min ∩ s.C_R) {
23:                 c.buff = T;
24:                 t = new RSBTrans(c);
25:                 s.trans = s.trans ∪ {t};
26:                 rsbtrans = rsbtrans ∪ {t};
27:              }
28:              if (rsbtrans ≠ ∅) {
29:                 t = RandomSelect (rsbtrans);
30:                 s.backtrack = s.backtrack ∪ {t};
31:              } else {
32:                 s.backtrack = s.backtrack ∪ s.trans;
33:   }}}}
```

Fig. 8. Algorithm to update backtrack sets

matched with $R_{2,1}(*)$. Next we study large realistic examples that show that $\mathrm{POE}_{\mathrm{MSE}}$ adds virtually no overheads. We used ParMETIS, a hypergraph partition library (14K LOC of MPI/C), as a benchmark for measuring the overhead of $\mathrm{POE}_{\mathrm{MSE}}$ (shown in Table 1 as ParMETIS (xprocs) where x is the number of processes that we ran the benchmarks with; $ParMETIS_*$ is a modified version where we rewrote a small part of the algorithm using wildcard `Irecv's`). In most of our benchmarks where no additional interleavings are needed, the overhead is less than 3%, even in the presence of wildcard receives, where the new algorithm has to run extra steps to make sure we have covered all possible matchings in the presence of slack. Finally we study large examples with contrived slack inelasticity situations inserted into them, which show that should slack inelastic behaviors arise in practice, POE would (silently) behave unsoundly, while $\mathrm{POE}_{\mathrm{MSE}}$ would (silently, and with minimal overheads) introduce the necessary re-executions, with certain (minimally chosen) sends endowed with slack that certainly introduce new behaviors, including potentially buggy ones. This is reflected in Table 1 as $ParMETIS\_b_*$ where we rewrote the algorithm of ParMETIS again, this time not only to introduce wildcard receives, but also to allow the possibility of a different or-

der of matching that can only be discovered by allowing some certain `Isend's` to be buffered. Our experiment shows that $\mathrm{POE}_{\mathrm{MSE}}$ successfully discovered the alternative matching during the second interleaving. So far, we have never encountered practical examples that are slack inelastic. However, the very purpose of a tool such as ISP incorporating $\mathrm{POE}_{\mathrm{MSE}}$ is to keep the user worry-free about soundness when verifying thousands of large programs.

In conclusion, we provide the first known approach to verify slack inelastic programs dynamically, while avoiding the naïve approach of considering every send with-/without slack. To model MPI's action co-enabledness relation that is governed by MPI's weak ordering semantics and slack, we define a formal semantics for MPI that delicately separating out the notions of *completion* and *matching* of MPI communications. We then define the notion of completes-before paths, and show that two MPI actions separated by such a path not co-enabled. Then we show that slack addition to MPI sends turn the associated wait operations into no ops. If these waits lie on completes-before paths, these paths are broken, thus adding more sends that can match a wildcard receive. We show the complexity of optimally determining minimal slack enumeration. Finally we present our $\mathrm{POE}_{\mathrm{MSE}}$ algorithm integrating MSE and POE.

We are releasing a full-fledged version of ISP implementing MSE that runs on Unix, Mac, and Windows VisualStudio, and comes with over 120 medium-to-large examples, and full install scripts at our web site [3]. In future, we will closely examine whether we can avoid the step in line 33 of Figure 8, and also whether static analysis methods may save some effort during MSE.

References

1. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In J. Palsberg and M. Abadi, editors, *POPL*, 110–121. ACM, 2005.
2. R. Manohar and A. Martin. lack Elasticity in Concurrent Computing. In Intl. Conf. on the Mathematics of Program Construction, *Lecture Notes in Computer Science* 1422.
3. S. Vakkalanka et al., Dynamic Reduction based Verification of Slack Inelastic Message Passing Systems, `http://www.cs.utah.edu/formal_verification/cav09-slack.html`
4. S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In A. Gupta and S. Malik, editors, *CAV*, Springer LNCS 5123, 66–79, 2008.
5. S. V. Sharma, G. Gopalakrishnan, and R. M. Kirby. A survey of MPI related debuggers and tools. UUCS-07-015, `http://www.cs.utah.edu/research/techreports.shtml`.
6. A. Vo, S. Vakkalanka, G. Gopalakrishnan, R.M. Kirby. Formal Verification of Practical MPI Programs. In: PPoPP (2009) to appear.
7. S. F. Siegel and G. S. Avrunin. Analysis of MPI Programs. Technical Report *UM-CS-2003-036*, Department of Computer Science, University of Massachusetts, 2003.
8. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.
9. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.
10. G. Karypis. METIS and ParMETIS. http://glaros.dtc.umn.edu/gkhome/views/metis.
11. M. Musuvathi and S. Qadeer. Fair stateless model checking. In PLDI '08 362–371, New York, NY, USA, 2008. ACM.

12. P. Godefroid, B. Hanmer, and L. Jagadeesan. Systematic software testing using VeriSoft: An analysis of the 4ess heart-beat monitor. *Bell Labs Technical Journal*, 3(2), April-June 1998.
13. Rupak Majumdar, Koushik Sen: Hybrid Concolic Testing. ICSE 2007: 416-426
14. `http://www.multicore-association.org`