Semantics Driven Dynamic Partial-order Reduction of MPI-based Parallel Programs

Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby
School of Computing, University of Utah, Salt Lake City, UT 84112
{rpalmer, ganesh, kirby}@cs.utah.edu

ABSTRACT

Most distributed parallel programs in the high performance computing (HPC) arena are written using the MPI library. There is growing interest in using model checking for debugging these MPI programs. In this context, partial-order reduction has considerable potential for containing state explosion, given the distributed memory nature of MPI programs. This potential is largely unmet. In this paper, we first define the formal semantics for a non-trivial subset of MPI. We then prove independence theorems based on the formal semantics, paving the way to a semantically clear and general partial-order reduction approach for MPI. Our work describes, for the first time, the exact dependencies between MPI non-blocking send operations and their tests for completion, namely wait and test. We also offer a cleaner solution than in previous works for MPI wildcard receives, a proper handling of which requires knowledge of the future course of computations. We show that Flanagan and Godefroid’s dynamic partial-order reduction algorithm offers a natural way to handle the need for future information. Our initial experimental results are encouraging.

Categories and Subject Descriptors:
D.2.4 [Software/Program Verification]: Model checking,
D.1.3 [Concurrent Programming]: Distributed programming,
D.3.1 [Formal Definitions and Theory]: Semantics.

General Terms: Verification.

Keywords: Partial-order Reduction, Concurrent Program Semantics, Transition Independence, Model Checking, MPI.

1. INTRODUCTION

Virtually all supercomputers and computational cluster machines are programmed in MPI [12], where the embedding language (C, C++, FORTRAN, etc.) specifies the intraprocess computations, and MPI function calls (over 130 in MPI 1.1 [12], and over 300 in MPI 2) provide a plethora of communication and synchronization commands. MPI is, in fact, considered the de facto standard for distributed programming in the high performance computing (HPC) arena. Writing MPI programs is known to be error-prone in practice ([5] provides a detailed experience report), even though MPI programs do not have the same degree of global interactions that thread programs have. Initial experience shows that model checking can be very effective in finding deep-seated errors in MPI programs [16, 20]. It is important that effective partial order reduction techniques [4] be employed in model checking MPI programs, as otherwise the relatively independent MPI processes actions will interleave and cause state explosion. However, without a precise formal semantic definition for MPI, it is impossible to characterize which MPI process interactions are non-commuting (and hence need to be interleaved in all possible ways) and which are indeed commuting (so that interleaving may be avoided).

Previously, there have only been sporadic attempts at characterizing the formal semantics of MPI. In [7], a LOTOS specification for a small subset of MPI is provided. This LOTOS description is too low level to serve as the basis for sound reduction approaches. In [21], a few MPI operations including some forms of sends and receives are modeled through a customized communicating automaton model. In [19], based on this semantics, a simple interleaving reduction algorithm called the Urgent Algorithm (discussed later) is proposed. This algorithm considers only a limited set of MPI process interactions, and is formulated in an MPI-specific manner, unlike our formulation here which is along the lines of existing partial order reduction approaches. Our contribution covers a larger subset of MPI, is directly built upon a high level formal semantic definition of MPI that we provide (full version in [14]), and capitalizes on dynamic information (as in [6]), thus overcoming many of the limitations of static partial order reduction approaches noted in [6].

Past work on applying formal methods to MPI includes work by Siegel and Avrunin [21, 23, 22, 19, 20] and our group [2, 13, 16, 14]. To contrast past work with this paper, consider the two main commands supported by MPI, namely send and receive. MPI communication commands include dozens of flavors of sends and receives. The send command is required to specify the receiver, but the receive command need not specify the sender. A receive command not mentioning its sender is called a wildcard receive. It receives data associated with any send that targets this receive (barring tag-matching, a detail we suppress in this paper). Sends

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2007 ACM 978-1-59593-748-3/07/0007 ...$5.00.
and receives may be of the non-blocking flavor (called the immediate mode in MPI). The completion of non-blocking sends and receives is awaited through the wait command, or instead tested through the test command. In addition, MPI has many collective operations such as barriers and reduces. In this context, our contributions are to answer the following questions with respect to MPI programs: (i) What is the formal notion of dependence (‘non-commuting actions’) in the standard sense, e.g., as defined in [4]) among MPI program operations? We provide a rigorous formal semantics for an interesting subset of MPI, and define dependence formally. For instance, we show that MPI’s non-blocking sends and receives are actually dependent on the wait and test operations that are used to test their completion. (ii) How do we handle, during model checking, possible matches between a wildcard receive and a send that will only be discovered in future, during state exploration? We show that the dynamic partial order reduction algorithm (DPOR) of [6] (with suitable adaptations for MPI that we describe in this paper) can elegantly handle this issue.

We now describe the closest related work to ours. In [21, 23], Siegel and Avrunin study ‘ordinary’ flavors of (i.e., blocking versions of) MPI send and receive (in reality known as MPI_Send and MPI_Recv, but we often suppress such details in this paper). The MPI run-time system is allowed (but not required) to provide buffering for a send. With buffering, the sender can post the message and proceed; without buffering, a rendezvous-based message exchange happens. In [21, 23], Siegel and Avrunin show that wildcard-free MPI programs may be analyzed for the absence of deadlocks by merely considering rendezvous-based executions, i.e., without having to model buffering. This avoids the state explosion caused by buffering.1 In [19], the authors propose the following extension to their earlier result: in an MPI program with wildcards, if all senders that can potentially match a wildcard receive are known, then the same rendezvous-style communication can be forced. The knowledge about the senders comes in two ways: (i) all these sends are found as moves out of the current global state s being examined during depth-first search based explicit state enumeration model checking, or (ii) only some of the sends are found as moves out of s; in this case it is assumed that the remaining sends will never be found ordered in all future computations from s (thus, what is offered can be considered the ample set [4]). The authors propose a so-called Urgent Algorithm based on these ideas.

While shown effective on many case studies, the Urgent Algorithm only provides a partial answer to how partial-order reduction algorithms may be designed for MPI. First of all, the Urgent Algorithm is defined with respect to a formal model of MPI communication (introduced in [21]) that models only a few MPI functions. Many functions, especially the deterministic flavors of MPI send and receive, namely MPI_Isend and MPI_Irecv (which are also called “non-blocking” or “asynchronous” or immediate mode) sends and receives, and frequently used during MPI program optimization) are not modeled. These commands behave in a deterministic manner because the associated communication buffers are explicitly provided by the user. They are termed non-blocking MPI calls, because after posting the send/receive, the computation advances without blocking.

1A very similar result was obtained in another setting by Manohar [11] and captured as the Slack Elasticity theorem. relying on a subsequent wait/test to check for message transmission. Although a recent paper [20] considers non-blocking MPI commands, no partial-order reduction method is proposed for these non-blocking operations. In contrast, in this paper we formalize the communication semantics of MPI, and then state and prove the classical notion of independence [4] among MPI program commands with respect to the formal semantics.

The behavior of wildcard receives (that future sends may induce a dependency on wildcard receives) is emblematic of a much more general issue with respect to MPI (and concurrent program analysis in general). It is well known that dependencies between operations will be precisely known only at runtime. While the specific examples that motivated Flanagan and Godefroid pertain to unknown aliasing relationships and array range overlaps, we observe that the same thinking can be applied to the statically unknown (but dynamically known) dependency information in a communication oriented language such as MPI. We already foresee the possibility of handling many more dynamic features of MPI – such as MPI_Cancel – which allows a pending MPI operation to be canceled.

In the context of DPOR itself, our algorithm has two primary differences: (i) our DPOR algorithm is tailored for MPI operations in some natural ways, and (ii) there is no place in our DPOR algorithm where we need to perform full expansion of a state to detect unsoundness. Regarding point (i), the MPI standard requires that correct MPI programs terminate at an MPI_Finalize call. Hence, the acyclic state-space requirement of MPI is checked by default in our algorithm, which keeps fingerprints of visited states in a hash table. For point (ii) we discuss this in detail with respect to the example shown in Figure 11 in Section 6.

Roadmap: Section 2 contains examples illustrating MPI and our contributions. Section 3 presents a simplified goto based modeling language for MPI, the semantics of transitions in that language, along with a number of theorems regarding the independence of transitions under the intended semantics of MPI. Section 4 gives the dynamic partial-order reduction algorithm that we have developed for use with MPI based programs, and provides a proof sketch that a number of interesting properties are preserved. Section 5 takes aim at the 2D diffusion example of [22]. We adapt this example to our MPI communication primitives, and are able to demonstrate the advantages of DPOR. We discuss preliminary results and integration into a practical analysis framework. Section 6 compares some interesting facets of our work with existing related work. Section 7 gives some future directions and concludes.

2In our approach, the more familiar operations such as MPI_Isend are modeled in terms of these more primitive operations.
1 if(rank != 0){
2    h = Issend 0 (addrof x)
3    Wait h
4 } else {
5    for(int i = 0; i < N; ++i){
6        h = Irecv i (addrof x)
7        Wait h
8    }
9 }
10 }
11 }

Figure 2: An assertion violation caused by nondeterministic message order.

1 if(rank==0){
2    count = 1
3    while(count < N){
4        h = Irecv * (addrof x) // Receive rank of
5        Wait h // completed proc
6        count++
7 }
8    assert(x == 3);
9 } else {
10     // do some work
11     if(done){
12         h = Issend 0 (addrof rank) // Send rank to proc 0
13         Wait h
14     }
15 }

Figure 3: The while loop may never terminate.

1 if(rank%3==0){
2    h = Irecv * (addrof x);
3    i = Irecv * (addrof y);
4    flag = Test h;
5    Wait i;
6 } else {
7    if((rank+2)%3==0){
8        h = Issend (rank+2)%rank (addrof x);
9     } else {
10        h = Issend (rank+1)%rank (addrof x);
11     }
12    Wait h;
13 }

Figure 4: The role of dynamic information in POR for MPI.

“MPI” in the remainder of this paper.

The N processes in an MPI distributed computation can be differentiated based on the unique process rank (integers in \([0,\ldots, N - 1]\)) that the MPI runtime system assigns to each MPI process. Messages are addressed by process rank (“rank”) or through wildcards (denoted by *) that stand for ANY_SOURCE. Consider the simple communication pattern where all processes send a message to the root process (rank 0) implemented in Figure 1. Here, Issend 0 means “send to 0” while Irecv i means “receive from i.” This code will deadlock for the following reasons. The process with rank 0 begins computing on Line 5. After posting an Irecv 0 on Line 5, this process will block on Wait h on Line 7. Since no Issend posted on Line 2 comes from a process with rank 0 (notice the test on Line 1), the Wait h on Line 7 will never unblock. If Line 5 iterates i from 1 instead of 0, the deadlock will disappear.

Figure 2 shows a second example where a process can receive from multiple potential sends in any order (notice the * signifying wildcard receive on Line 4, and that any non-zero rank process can perform an Issend on Line 12, conveying its rank as the message data). Let us assume that the non-zero ranks lie between 1 and 3. Here the developer expected the process with rank 0 to always finish last. However unless this is guaranteed using some other synchronization mechanism, it will be possible to violate this assertion. This example underscores the importance of model checking based verification of MPI programs, as opposed to random-testing which can miss such bugs.

A third example shown in Figure 3 contains an error specific to MPI – although it is tempting to program in this manner. MPI makes no guarantee about process fairness. In this example, while the user may have intended for a communication to eventually force the termination of the while loop, it may in fact continue forever. Liveness checking is future work for us. However, if this example creates a cycle in the state space, our algorithm will detect and report the cycle as an error (as discussed on Page ).

Coming to the notion of indepdence, consider Figure 1 again; which actions in this program are independent? It turns out that the Issend operations from line 2 are dependent only on the Wait corresponding to the Irecv operations on line 7 when i in the Irecv on line 6 is equal to the rank of the sending process! Similarly the Wait on line 3 is dependent only on the Irecv of line 6 when i is equal to the rank of the sending process. All other program actions are independent. All this information is a simple and direct consequence of our formal semantics, as we show later.

Our next example (Figure 4) demonstrates the need to use dynamic information in the computation of dependencies. Let this program be instantiated for N = 6 processes. This program causes processes with ranks 1 and 2 to send a message to the process with rank 3; in addition, processes with ranks 4 and 5 send a message to the process with rank 0. The messages can be received in either order – meaning that the value of x in the process with rank 0, after the Wait on Line 12 executes, could be either from rank 4 or rank 5. Moreover, since Test is used on line 4 it is possible that one of the messages is not received by this part of the program. Although dependencies exist, our dynamic partial-order reduction algorithm naturally forms clusters of independence e.g., as in [3]; however, unlike in their work, the clusters will not be statically and a priori determined.

3. LANGUAGE SEMANTICS

In this section we define the execution semantics for a simple goto-based program modeling language for MPI called MPIC (our much more detailed semantics of MPI appears in [14] that is cross-referenced with the natural language standard [12]). The grammar for this language is shown in
Figure 5: The Grammar of MPIC.

Figure 5. We have followed the C convention in modeling Boolean operations using Integers. Non-zero values are true; zero is false. For communication we have chosen a representative operation from each of the point-to-point operation groups.

The communication operations were chosen for their applicability in deterministic optimization. A program that implements message packaging and queuing would use Issend and Irecv to indicate to the MPI subsystem that additional buffering is unnecessary—thereby increasing performance. Wait and Test both complete communications. Wait blocks until the communication completes, whereas Test is always enabled and returns in a flag whether the communication has completed. Barrier is used to conservatively estimate the state of the collective operation and the processes that are currently participating. The parameter \( N \) is constant in our system and represents the number of processes participating in the distributed computation.

Each of the rules in Figures 7 and 8 manipulate the state tuple. Operations with multiple rules model the disjoint nature of the transition type. The Barrier operation is modeled using six rules implementing two transition types: an entrance and an exit. The reachable state space, viewed as a unary predicate \( \Sigma \), is recursively defined by the execution of these rules parameterized over the transition relation of a given program and some fixed number of processes \( N \). The transition relation is defined using two functions. The \( \text{proc} \) function returns the AST node for a given \( \text{pc} \) value. Sequential control flow is modeled by the \( \text{next} \) function. The \( \text{pc} \in \mathbb{N} \) is held in the local store of each process.

Although suppressed for simplicity in this presentation, data is transmitted between processes in the Wait (2) and Test (2) rules.

Again consider the example of Figure 1. This program uses Assignment, Goto, Issend, Irecv, and Wait. We assume the existence of a suitable evaluator \( E \) for expressions in our language and that control flow is simplified into conditional goto statements. The rule for Assignment says that if there is a state \((c, p)\) such that \( \text{proc} \) maps to an assignment for the current \( \text{pc} \) of process \( i \), in the next state the system updates the value of the \( \text{pc} \) with \( \text{next}(\text{pc}) \), and assigns the evaluated value of expression \( e \) in the current state to the memory location referenced by \( x \). Goto is very similar except the expression evaluated is assigned to the \( \text{pc} \) in the next state. Issend and Irecv are again similar with the exception that \( x \) contains a numeric handle to the request that is created by the execution of this operation.

Before we elaborate on the semantics of Wait, we should mention a bit about requests. The natural language MPI standard [12] uses the term request extensively without giving a formal definition to it. Send and receive operations, regardless of their type (we show only Issend and Irecv in this paper), all produce or activate requests; Wait and Test operations consume or deactivate and possibly deallocate re-
Assignment:
\[
\Delta(c, p) \land p(i) = (l, g) \land proc(l(pc)) = (assign \; x; c)
\]
\[
\Delta(c, p[i \mapsto (l|pc \mapsto next(pc), E[\{addrof\; x, \; p(i)\mapsto E[e, p(i)]]], g)])
\]

Goto:
\[
\Delta(c, p) \land p(i) = (l, g) \land proc(l(pc)) = (goto \; e)
\]
\[
\Delta(c, p[i \mapsto (l|pc \mapsto E[e, p(i)], g)])
\]

Assert:
\[
\Delta(c, p) \land p(i) = (l, g) \land proc(l(pc)) = (assert \; e) \land E[e, p(i)] = true
\]
\[
\Delta(c, p[i \mapsto (l|pc \mapsto next(l(pc)], g)])
\]

Barrier\_init (1):
\[
\Delta((vacant, \emptyset), p) \land p(i) = (l, g) \land proc(l(pc)) = (\text{barrier\_init})
\]
\[
\Delta((in, \{i\}), p[i \mapsto (l|pc \mapsto next(l(pc)], g)])
\]

Barrier\_init (2):
\[
\Delta((in, s), p) \land p(i) = (l, g) \land i \notin s \land proc(l(pc)) = (\text{barrier\_init})
\]
\[
\Delta((in, s \cup \{i\}), p[i \mapsto (l|pc \mapsto next(l(pc)], g)])
\]

Barrier\_wait (1):
\[
\Delta((in, s), p) \land p(i) = (l, g) \land s = \{i : i \in 0..(N-1)\} \land s \neq \{i\} \land proc(l(pc)) = (\text{barrier\_wait})
\]
\[
\Delta((out, s \setminus i), p[i \mapsto (l|pc \mapsto next(l(pc)], g)])
\]

Barrier\_wait (2):
\[
\Delta((in, s), p) \land p(i) = (l, g) \land s = \{i : i \in 0..(N-1)\} \land s \neq \{i\} \land proc(l(pc)) = (\text{barrier\_wait})
\]
\[
\Delta((out, s \setminus i), p[i \mapsto (l|pc \mapsto next(l(pc)], g)])
\]

Barrier\_wait (3):
\[
\Delta((out, s), p) \land p(i) = (l, g) \land proc(l(pc)) = (\text{barrier\_wait}) \land i \in s \land s \setminus i \neq \emptyset
\]
\[
\Delta((out, s \setminus i), p[i \mapsto (l|pc \mapsto next(l(pc)], g)])
\]

Barrier\_wait (4):
\[
\Delta((out, s), p) \land p(i) = (l, g) \land proc(l(pc)) = (\text{barrier\_wait}) \land i \in s \land s \setminus i = \emptyset
\]
\[
\Delta((vacant, \emptyset), p[i \mapsto (l|pc \mapsto next(l(pc)], g)])
\]

Issend:
\[
\Delta(c, p) \land p(i) = (l, g) \land proc(l(pc)) = (\text{issend} \; x \; \text{dest addr})
\]
\[
\Delta(c, p[i \mapsto (l|pc \mapsto next(l(pc)], E[\{addrof\; x, \; p(i)\mapsto ([\text{Dom}(g)] + 1)], g([\text{Dom}(g)] + 1 \mapsto (i, E[\text{dest}, p(i)], E[\text{addr}, p(i)], \text{send, false})))))
\]

Irecv:
\[
\Delta(c, p) \land p(i) = (l, g) \land proc(l(pc)) = (\text{irecv} \; x \; \text{src addr})
\]
\[
\Delta(c, p[i \mapsto (l|pc \mapsto next(l(pc)], E[\{addrof\; x, \; p(i)\mapsto ([\text{Dom}(g)] + 1)], g([\text{Dom}(g)] + 1 \mapsto (E[\text{src}, p(i)], i, E[\text{addr}, p(i)], \text{recv, false})))))
\]

Figure 7: Semantic definitions of Assignment, Assert, Goto, Barrier\_init, Barrier\_wait, Issend, and Irecv.
Wait (1):

\[ \Sigma(c, p) \land p(i) = (l, g) \land \text{proc}((l, p(c))) = (\text{wait } e) \land \\
(E[e, p(i)] = 0 \lor (E[e, p(i)] \in \text{Dom}(g) \land \text{Completed}(g(E[e, p(i)]))) ) \\
\Sigma(c, p[i \mapsto (l|pc \mapsto \text{next}(l|pc)), E[\text{addrof } e], p(i) \mapsto 0], g)] \]

Wait (2):

\[ \Sigma(c, p) \land p(i) = (l, g_i) \land \text{proc}(l_i(p(c))) = (\text{wait } e) \land \\
E[e, p(i)] \in \text{Dom}(g_i) \land p(j) = (l_j, g_j) \land \text{Match}(g_j(k), g_i(E[e, p(i)])) \land \\
m < k \Rightarrow \neg \text{Match}(g_j(m), g_i(E[e, p(i)])) \\
\Sigma(c, p[i \mapsto (l|pc \mapsto \text{next}(l|pc)), E[\text{addrof } e], p(i) \mapsto 0], \\
g_i[E[e, p(i)] \mapsto g_i(E[e, p(i)][false/true])], \\
j \mapsto (l_j, g_j[k \mapsto g_j(k)[false/true]])] \]

Test (1):

\[ \Sigma(c, p) \land p(i) = (l, g) \land \text{proc}(l(p(c))) = (\text{test } e_1 e_2) \land \\
(E[e_1, p(i)] = 0 \lor (E[e_1, p(i)] \in \text{Dom}(g) \land \text{Completed}(g(E[e_1, p(i)])))) \\
\Sigma(c, p[i \mapsto (l|pc \mapsto \text{next}(l|pc)), E[\text{addrof } e], p(i) \mapsto 0], E[\text{addrof } e_2], p(i) \mapsto \text{true}], g)] \]

Test (2):

\[ \Sigma(c, p) \land p(i) = (l, g_i) \land \text{proc}(l_i(p(c))) = (\text{test } e_1 e_2) \land E[e_1, p(i)] \in \text{Dom}(g_i) \land \\
\neg \text{Completed}(g_i(E[e_1, p_i])) \land p(j) = (l_j, g_j) \land \text{Match}(g_j(k), g_i(E[e_1, p_i])) \land \\
m < k \Rightarrow \neg \text{Match}(g_j(m), g_i(E[e, p(i)])) \\
\Sigma(c, p[i \mapsto (l|pc \mapsto \text{next}(l|pc)), E[e_1, p_i] \mapsto 0, E[e_2, p_i] \mapsto \text{true}], \\
g_i[E[e_1, p_i] \mapsto g_i(E[e_1, p_i][false/true])], j \mapsto (l_j, g_j[k \mapsto g_j(k)[false/true]])] \]

Test (3):

\[ \Sigma(c, p) \land p(i) = (l, g_i) \land \text{proc}(l_i(p(c))) = (\text{test } e_1 e_2) \land \\
E[e_1, p(i)] \in \text{Dom}(g_i) \land \neg \text{Completed}(g_i(E[e_1, p(i)])) \land \\
p(j) = (l_j, g_j) \land \neg \text{Match}(g_j(k), g_i(E[e_1, p(i)])) \\
\Sigma(c, p[i \mapsto (l|pc \mapsto \text{next}(l|pc)), E[e_2, p(i)] \mapsto \text{false}], g)] \]

Figure 8: Semantic definitions of Wait and Test.
quests depending on the flavor of operation and request. In our work we have defined a request to be a five-tuple that contains the rank of the sender and receiver, the address of memory to be read or written, the type of message being requested (either send or receive), and whether or not this particular request has been completed. Moreover, we ignore deallocation and deactivation and focus only on the producer/consumer relationship that exists between Issend, Irecv, Wait, and Test in this restricted context. A model that admits more of these operations (such as Ssend, Srecv, and Start) would have to take into account the additional complexity (cf. our TLA+ model of MPI [14]).

A process completes the communication initiated by an Issend or Irecv by executing a Wait or Test. The Wait operation is paired with an Issend or Irecv on a particular process via a request handle returned by the Issend or Irecv. The execution semantics are similar to Assignment as it requires that some process be positioned to execute a Wait operation in the current state. The additional restriction for Wait (1) is that either the request handle passed to the Wait evaluates to 0 (E[e, p(i)] = 0), a special value outside Dom(g) used to represent REQUEST·NULL, or if the handle is valid then the request has been completed by the Wait or Test of another process. In this case the pc of the process is updated to next(pc) and the handle is set to 0. The additional restriction for Wait (2) is that the request handle evaluate to a value in Dom(g) and there must be some other request on process j that will Match the request g(E[e, p(i)]). The final clause, in connection with the deterministic structure of a single process, enforces the program order matching requirement for requests imposed by MPI. In this case the pc and handle are updated as before. In addition, the global store is modified to reflect that the communication has completed. The data is also moved from sender to receiver in this step although tacit in our presentation.

3.1 Assumptions and Properties of Interest

Some well-formedness assumptions are made of MPI programs that are handled by our DPOR algorithm. It is assumed that a process does not access the buffer passed to an Issend or Irecv until after the Wait or Test returns and if Test is used, the flag returned by Test is true. More formally, for any process i, if a is an address such that 9r : (Address(q, r) = a) ∧ ~Completed(q, r) then it assumed that no other action of process i will read or write t(a). This assumption is necessary for correctness according to the MPI standard [12]. An example that violates this assumption is: h = irecv 3 (addref h). A second assumption is that programs cannot explicitly reference the program counter of any process, i.e., statements of the form pc = exp and v = irecv 2 (addref pc) are a violation of this assumption. It possible to check these assumptions, with modest or no over-approximations, using existing static analysis methods.

Only certain properties are preserved by our reduction algorithm. In particular, under this execution semantics, the reduction algorithm proposed here preserves (i) deadlocks, (ii) cycles, and (iii) local assertions, more formally, assertions on Dom(li) for each process i that are invariant under stuttering.

3.2 Independence

We can now discuss the independence properties of the transition semantics presented in Figures 7 and 8. For completeness we restate the definition of independence from [8]. For any state σ ∈ Σ, the set of transitions enabled in σ, enabled(σ) is defined as enabled(σ) = { ti | ti(σ) ∈ Σ}. A transition t of process i and a transition t_j of process j where i ≠ j are independent (i.e., I(t_i, t_j)) if

\[ \forall \sigma \in \Sigma : t_i, t_j \in \text{enabled}(\sigma) \Rightarrow (t_i \in \text{enabled}(t_j(\sigma)) \land t_j(t_i(\sigma)) = t_j(t_i(\sigma))) \]

We say that two transitions t_i and t_j are dependent if ¬I(t_i, t_j).

With the transition semantics defined we are able to state and prove a number of theorems regarding the independence of the different transition types. For brevity we will only provide proof sketches, keeping the details for Appendix A.

**THEOREM 1.** If t_i is an Assignment, Goto, Assert, or Barrier transition for a given process i and t_j is any transition of process j where i ≠ j then I(t_i, t_j).

To prove this theorem for Assignment, Goto, and Assert it is sufficient to note that disjoint memory spaces are read and written. For each of the Barrier transitions, the key observation is that the execution of other processes cannot disable the enabled transition into or out of a Barrier. In both cases independence is with respect to observable behavior.

Corresponding theorems for Issend, Irecv, Wait, and Test require some additional proof machinery. Figure 9 gives the definition for the Complete predicate and pseudo-code for the Index operation. The Complete predicate returns true for two transitions when they could be used to cause a communication between two processes. In the pseudo-code, post(t) is the state generated by executing transition t, proc(t) is the process that executed t. The Index operation takes a transition t as an argument and returns the handle of the request referenced by t.

**THEOREM 2.** If t_i is an Issend, Irecv, Wait, or Test of process i and t_j is any transition of another process j where i ≠ j and ¬Completed(t_i, t_j) then I(t_i, t_j).

Intuitively, Complete indicates when two transitions could result in a communication under the right execution interleaving order. This theorem says that transitions that cannot result in a communication are independent with respect to the observable behavior. This is the case again because the transitions involved read from and write to disjoint memory spaces.

The consequence of the above two theorems is that all of the program’s non-communication actions are independent and many of the communication actions are also independent. Moreover, we have given a simple predicate (i.e., Complete) that can be evaluated during model checking to determine whether two MPI communication operations are in fact dependent. Using this predicate, the questions from Section 2 pertaining to the independence of actions in Example 1 can be answered.
Index(t) ≡
IF t is a Wait or Test
THEN
return the evaluated handle argument
ELSE
IF t is an Issend or Irecv
return the value written into the handle address
ELSE
return the null request value

Complete(a, β) ≡
Let (c1, p1) = post(a) in
Let (l1, g1) = p1(proc(a)) in
Let (c2, p2) = post(β) in
Let (l2, g2) = p2(proc(β)) in
Let i = Index(α) in
Let j = Index(β) in
IF not(i = null) ∧ not(j = null)
THEN
Match(g1(i), g2(j))
ELSE
false

Figure 9: The Complete operation.

4. PARTIAL-ORDER REDUCTION

As discussed in Section 2, traditional partial-order reduction techniques have prohibitive limitations in the setting of MPI for the following reasons: (i) Addressing is most likely computed at run time as a function of the pid or rank of a process. (ii) In the presence of wildcard receive operations it becomes difficult to know if all possible Issends will be interleaved without fully expanding the state. (iii) The dependencies are only between operations that Complete, therefore, one needs to compute the trajectory of the handle returned by the Issend or Irecv.

The algorithm we propose and have implemented appears in Figure 10. The algorithm adds pids of processes with enabled transitions to the state set

IF t is a Wait or Test
THEN
return the evaluated handle argument
ELSE
IF t is an Issend or Irecv
return the value written into the handle address
ELSE
return the null request value

Complete(a, β) ≡
Let (c1, p1) = post(a) in
Let (l1, g1) = p1(proc(a)) in
Let (c2, p2) = post(β) in
Let (l2, g2) = p2(proc(β)) in
Let i = Index(α) in
Let j = Index(β) in
IF not(i = null) ∧ not(j = null)
THEN
Match(g1(i), g2(j))
ELSE
false

Figure 10: Dynamic partial-order reduction based depth first search.

empty set is therefore persistent trivially. If the enabled set is non-empty then any transition that is enabled and not independent of the transition selected but not executed in a given state will eventually be executed by the algorithm. This means that the stack will be searched when that transition is backed off of and the process executing the transition will be scheduled before the dependent transition.

The second part of the proof shows that a cycle may be delayed but will not be ignored. Since all cycles are considered errors when the cycle is eventually closed it is reported at line 18.

Appendix A provides the lemmas used in our proof. The complete proofs are contained in [15]

5. EXPERIMENTS USING DPOR

To demonstrate the effectiveness of the specialization of partial-order reduction for MPI primitives, consider the two dimensional diffusion simulation described in [22]. Here the authors model the MPI primitives using Promela and attempt to model check for a 4 x 4 grid (16 processes). They report that the model checker runs out of memory. It is not clear whether the authors attempt to use the partial-order reduction implemented in SPIN.

To handle this program we made a few modifications. First we changed the pseudo-code shown in their paper into C. We then transformed the program so that all of the send and recv operations were the corresponding Issend and Irecv operations followed by a Wait. Optimization came next—we moved the MPI operations such that setting up buffers for communication could overlap the sending and receiving of buffers. 4

4This program code can be downloaded from our web-site at
if(rank == 0){
    h = Irecv * (addrof x);
} else {
    h = Issend 0 (addrof x);
}
Wait h;

Figure 11: A non-deterministic receive operation.

From the program text we extract a model automatically using the Microsoft Phoenix compiler [17]. We then automatically simplified the model by inlining and slicing such that only the communication skeleton is preserved. We are then able to verify this example generating only 7 \times 10^5 states in about one minute on an ordinary laptop computer (2 GHz, 1GB memory). We could not handle this example states in about one minute on an ordinary laptop computer.

6. DISCUSSIONS

Our algorithm is similar in many respects to the algorithm proposed in [6], with one notable difference. In the Godefroid and Flanagan work, when a process p is added to the backtrack set backtrack(v) for some state v, the algorithm checks to insures that p is enabled. If p is disabled, a check is performed to find some transition \( t \in enabled(v) \) such that p becomes enabled in \( t(v) \). If no such \( t \) can be found \( v \) is fully expanded. Our algorithm does not require this check as a result of the semantics of MPI. In particular, only Wait and the Barrier operations can become disabled. We have already shown that Barrier transitions are independent of transitions of other processes.

For Wait, consider the execution sequence in Figure 11, instantiated for three processes. If the communication execution order is \( \text{Irecv; Issend}; \text{Wait}_{\text{Irecv}}; \text{Issend}; \text{Wait}_{\text{Irecv}} \); then the problem is that the Irecv operation may not form a communication with the Issend of process 2 because \( \text{Wait}_{\text{Irecv}} \) will force a match and disable the ability of \( \text{Wait}_{\text{Irecv}} \) to form a communication non-deterministically under our execution semantics. However after executing the algorithm, the following interleaved sets would be produced (shown between transitions):

\[
\begin{align*}
0, 1, 2 & \quad \text{In this state we force the Issend2 to happen first in the subsequent exploration sequence} \\
\text{Irecv} & \quad \text{The Wait}_{\text{Irecv}} \text{ is disabled} \\
\text{Issend1} & \quad \text{The Wait}_{\text{Irecv}} \text{ has no other choice and the Issend2 does not complete the Issend1; operation so process 2 is not scheduled} \\
\text{Wait}_{\text{Issend1}} & \quad \text{The Wait}_{\text{Irecv}} \text{ is still forced to match Issend1; Issend2} \\
\text{Wait}_{\text{Irecv}} & \quad \text{Since the Issend2 is eventually forced to happen first--meaning above the receive, then it remains to show that some interleaving will either have Wait}_{\text{Issend2}} \text{ or Wait}_{\text{Irecv}} \text{ before Wait}_{\text{Issend1}} . \text{ We know this is the case because Wait}_{\text{Issend1}} \text{ and Wait}_{\text{Issend2}} \text{ are guaranteed to happen in both orders whenever they appear along any path because Complete(Wait}_{\text{Irecv}}, \text{Wait}_{\text{Issend1}}).}
\end{align*}
\]

6.1 Other Related Work

This paper has focused on the presentation of a partial-order reduction algorithm that is enabled by the communication semantics of MPI. Other models of MPI exist including [21] where the authors build a model of MPI from first principles. They then propose an urgent scheduling for MPI operations included in their model that preserve a halting properties. They have implemented many of the MPI primitives—including a number of non-blocking operations in the SPIN [20] model checker. The approaches are difficult to compare because they work for different subsets of MPI. It is also not clear how or whether partial-order reduction is being used in their current implementation.

Other previous work in formalizing MPI such as [7, 14, 21, 13, 2] do not implement the semantics proposed directly in a model checker. Rather these models serve to augment the program model in a library format.

There are several model checkers that have partial-order reduction such as SPIN [10], Zing [1], Verisoft [9], Bogor [18] and perhaps others. In each of these, the reduction is not tailored to MPI.

Partial-order reduction has been studied extensively—a survey of which is beyond the scope of this paper. This work is most closely related to the dynamic partial-order reduction algorithm of Godefroid and Flanagan [6]. The two primary differences being (i) our algorithm is tailored for MPI operations, and (ii) there is no place in our algorithm where full expansion is used to deflect unsoundness.

7. FUTURE DIRECTIONS

One problem that faces software model checking is getting a reasonable model to verify from some piece of source code. We have mentioned in Section 5 that we have a framework that helps in this regard although details are not included in this paper. Static analysis to help reduce the size of the models checked is an important area of future work.

There is immense potential for additional research in this area. There are many more MPI operations that require a formal semantic characterization, to show independence. We conjecture that these proofs are mechanizable. Once the independence theorems are known they can be used in model checking to eliminate unnecessary redundant exploration by the model checker.

More significantly, the approach of formalizing communication libraries and building formal semantics based partial-order reduction algorithms may emerge to be a viable approach in analyzing other library based parallel and distributed programs.

8. REFERENCES


APPENDIX

A. PROOF SUMMARY

A.1 Independence theorems

The full proofs are contained in [15]. For each of the proofs to follow we assume the requirements of Section 3.1 hold.

**Lemma 1.** If $t_i$ is an Assignment, Goto, or Assert transition for a given process $i$ and $t_j$ is any transition of process $j$ where $i \neq j$, then $I(t_i, t_j)$.

**Lemma 2.** If $t_i$ is a Barrier transition for a given process $i$ and $t_j$ is any transition of process $j$ where $i \neq j$ then $I(t_i, t_j)$.

One interesting consequence of Lemma 2 is that it is unnecessary to examine more than one execution interleaving order of processes entering and exiting barrier operations. This can greatly reduce the cost of analysis when barriers are heavily used to force lock-step execution.

**Definition 1.** Transitions $t_i$ is said to complete $t_j$ if $\text{Complete}(t_i, t_2)$ evaluates to true.

**Lemma 3.** If $t_i$ is an Issend, Irecv of process $i$ and $t_j$ is any transition of another process $j$ where $i \neq j$ and a $\text{Complete}(t_i, t_j)$ then $I(t_i, t_j)$.

**Lemma 4.** If $t_i$ is a Wait of process $i$ and $t_j$ is any transition of another process and a $\text{Complete}(t_i, t_j)$ then $I(t_i, t_j)$. 
Lemma 5. If $t_i$ is a Test of process $i$ and $t_j$ is any transition of another process $j$ where $i \neq j$ and $\neg \text{Complete}(t_i, t_j)$ then $I(t_i, t_j)$.

The final theorem of this section combines all of the lemmas as a convenience for the proofs in the next section.

Theorem 4. For all transitions $t_1$, $t_2$, and states $\sigma \in \Sigma$, $t_1, t_2 \in \text{enabled}(\sigma)$ and $\neg \text{Complete}(t_1, t_2) \Rightarrow I(t_1, t_2)$.

A.2 Correctness of the reduction algorithm

We begin with two definitions from [8].

Definition 2. The set of transitions $T$ enabled in state $q$ is persistent in state $q$ if and only if all nonempty sequences of transitions

$$q = q_1 \xrightarrow{t_1} q_2 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} q_n \xrightarrow{t_n} q_{n+1}$$

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

Definition 3. A set $T_s$ of transitions is a conditional stubborn set in state $s$ if $T_s$ contains at least one enabled transition, and if for all transitions $t \in T_s$, the following condition holds: for all sequences $q = q_1 \xrightarrow{t_1} q_2 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} q_n \xrightarrow{t_n} q_{n+1}$ of transitions such that $t$ and $t_n$ are dependent in $q_n$, at least one of the $t_1, \cdots, t_n$ is also in $T_s$.

The proof will show that for all states visited by the algorithm that if the set of enabled transitions is non-empty, then the set of transitions explored by the algorithm at that state is a conditional stubborn set[24].

Lemma 6. When backing off of a state created by a communication operation of process $p$, $p$ is added to the interleave set of the pre-state of the nearest dependent transition in the search stack.

Theorem 5. At line 26 the set of transitions $T$ explored by the algorithm of Figure 10 is persistent in $\text{top}(s)$.

The MPI standard requires that all processes eventually call MPI_Finalize. A cycle in the full state space will prohibit processes from reaching MPI_Finalize and therefore is an error. The algorithm shown in Figure 10 checks for cycles while performing the model checking. It is therefore desirable to show that the algorithm actually detects the presence of cycles.

Theorem 6. The algorithm of Figure 10 discovers a cycle in the reduced state space if and only if there is a cycle in $A_G$. 