Modeling Wildcard-Free MPI Programs for Verification

Stephen F. Siegel
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
siegel@cs.umass.edu

George S. Avrunin
Department of Mathematics
University of Massachusetts
Amherst, MA 01003
avrunin@math.umass.edu

ABSTRACT

We give several theorems that can be used to substantially reduce the state space that must be considered in applying finite-state verification techniques, such as model checking, to parallel programs written using a subset of MPI. We illustrate the utility of these theorems by applying them to a small but realistic example.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking, validation*

General Terms

Verification

Keywords

MPI, Message Passing Interface, parallel computation, formal methods, analysis, finite-state verification, model checking, deadlock, concurrent systems, SPIN

1. INTRODUCTION

Scientific computing provides predictions that are increasingly important not just for research but also for decision-making on issues of great significance to society, including economic policy, environmental regulation, and the safety and performance of such things as cars, airplanes, and buildings. Yet the parallelism that makes much of this computation practical makes it difficult to build correct programs. Parallel programs can behave non-deterministically, in the sense that they can produce different results when run on different platforms, and sometimes even when run twice on the same platform. Large parallel programs often display deadlocks that are difficult to reproduce, especially when ported to new platforms, and produce results that are not independent of the number of processors when they should be. Experience has shown that just to detect or reproduce these problems (let alone to pinpoint their causes and correct them) can be extremely time-consuming and labor-intensive. Moreover, the measures introduced to avoid these problems, such as extra barriers or redundant inter-processor communication, can severely hamper performance.

Finite-state verification (FSV) techniques, such as model checking [3], provide methods for determining whether a parallel program satisfies particular requirements, such as freedom from deadlock or the absence of specified race conditions. These techniques construct a finite model that represents all possible executions of a given program and use various algorithmic methods for determining whether the requirements hold in the model. While less powerful than approaches based on theorem-proving [6, 11], these techniques are highly automated and, unlike testing or run-time monitoring approaches [8, 15], can give results about all possible executions of the program. When FSV techniques show that a requirement may be violated (e.g., that the program may deadlock), they typically provide a *counterexample* tracing an execution that violates the property. For this reason, FSV techniques are useful for detecting and explaining bugs during development as well as for verifying that a finished program satisfies its requirements.

The main drawback to FSV techniques is the state space explosion problem: the number of states a concurrent program can reach is, in general, exponential in the number of processes in the program, and all these states must be represented in the model in some way. Indeed, almost all the questions one would want to answer about a concurrent program (e.g., does it deadlock, does a particular communication ever occur, etc.) are known to be at least *NP*-hard, and the number of reachable states for even small parallel scientific programs is enormous. Useful application of FSV techniques to parallel scientific programs thus depends on the development of methods for reducing the size of the state spaces that must be considered.

In this paper, we consider the widely-used Message Passing Interface (MPI) [9,10], which presents some special challenges for FSV techniques. For instance, the memory available for buffering messages between two processes, and thus the number of messages that can be buffered, can change dynamically and unpredictably during execution, and the models used for verification must take this into account. We describe a class of models for MPI programs written using a subset of the MPI communication constructs, and give several theorems that allow substantial reduction in the size of the state space that must be considered to verify some important classes of requirements for these programs.

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

In the next section, we describe our approach to modeling MPI programs. In §3, we state the theorems and discuss their significance. In §4, we discuss some future research directions and some conclusions about this approach.

2. MODELS

Our goal is to construct models that *conservatively* represent the behavior of MPI programs in the sense that each possible execution of the program is represented in the corresponding model, although the model may also represent executions that are forbidden by the MPI semantics. (Allowing such additional, "infeasible" executions in the model often makes for a much simpler, more compact model.) If we can prove, for example, that none of the executions represented in a conservative model deadlocks, then we can conclude that none of the possible executions of the corresponding program can deadlock. Our models focus primarily on the communication behavior and abstract away much of the detailed state and computation of the programs (e.g., we typically ignore the precise values of floating point variables). In this section, we give a brief description of our models and how they represent some of the special semantics of the MPI communication constructs. Full details may be found in [13].

In this paper, we will assume that an MPI program consists of a fixed number of concurrent processes, each executing its own code with no shared variables. We assume that each process is single-threaded and that communication between processes takes place only through the blocking standard-mode functions MPI_SEND, MPI_RECV, MPI_SENDRECV, and MPI_SENDRECV_REPLACE, and the 16 collective functions, such as MPI_BARRIER and MPI_GATHER. We will mention some aspects of the semantics of these functions in the discussion that follows, but we refer the reader to the MPI Standard [9, 10] for complete details. As discussed in §4, we hope to be able to extend our results to other MPI functions in future work.

2.1 Process Automata and Channels

Our basic idea is to represent each process in a program $\mathcal{P}$ in a fairly standard way as a finite state automaton (FSA). The transitions of these automata are labeled by *local events*, representing actions involving only the corresponding process, and *send* and *receive* events, representing calls to the various MPI communication functions.

The arguments to the functions MPI_SEND, MPI_RECV, MPI_SENDRECV, and MPI_SENDRECV_REPLACE include locations in memory for the start of the sequence of data to be sent or received, the type and number of the elements of that sequence of data, a *tag* that may be used to distinguish particular classes of messages, and integers that describe the sender or receiver of the message with respect to a set of processes called a *communicator*. A receive statement can only receive messages that match its sender, receiver, tag, and communicator. *Wildcard receive* statements may specify MPI_ANY_SOURCE rather than a single sending process or MPI_ANY_TAG rather than a single tag. For simplicity, we will assume that the only communicator used is the default one, MPI_COMM_WORLD, which represents the set of all processes that exist at system initialization. Our model for $\mathcal{P}$ therefore includes a set of *channels*, each associated with a fixed sending and receiving process. (We allow multiple channels with the same sender and receiver to allow for the different possible tags.) We abstract away the details of the buffers and data sequences, and associate with each channel a (possibly infinite) set of *messages* that may be sent over that channel.

We impose some restrictions on the automaton M_p representing the process p. For a *model with no wildcard receives*, the requirements are as follows: at each state u other than the unique final state (which has no outgoing transitions), exactly one of the following must hold:

1. u is a *local-event state*: there is at least one transition departing from u, and all the transitions from u are labeled with local events,
2. u is a *sending state*: there is exactly one transition departing from u, labeled by a send event $c!x$, where c is a channel with sender p and x is a message that can be sent over c,
3. u is a *receiving state*: there is a channel d whose receiving process is p such that the transitions departing u are labeled by the receive events $d?y$, as y runs over the possible messages that can be sent over d, or
4. u is a *send-receive state*: there are channels c with sender p and d with receiver p, a message x that can be sent over c, a state u', and states v_y, and v'_y, for all messages y that can be sent over channel d, such that the following all hold:
   (a) the set of transitions departing from u consists of one transition to u' whose label is $c!x$, and, for each y, one transition labeled $d?y$ to v_y,
   (b) for each y, there is precisely one transition departing from v_y, it is labeled $c!x$, and it terminates in v'_y, and
   (c) for each y, there is a transition from u' to v'_y, it is labeled $d?y$, and these make up all the transitions departing from u'.

The general definition (for a model that may have wildcard receives) allows multiple receiving channels for the receiving and send-receive states, but we will not consider such models in this paper.

2.2 Execution Semantics

We represent the executions of a model of an MPI program as sequences of transitions. For simplicity, our representation will not distinguish between the case where a send and receive happen synchronously, and the case where the receive happens immediately after the send, with no intervening events. It is clear that in the latter case, the send and receive *could* have happened synchronously, as long as the sending and receiving processes are distinct.

For a sequence $S = (t_1, t_2, \dots)$ to represent an execution of the program $\mathcal{P}$, we require that, for each process p, the subsequence of S consisting of transitions in the automaton corresponding to p form a path through that automaton starting at its initial state, so that the execution of the program represents an interleaving of the executions of the processes. We also require that each channel behave like a queue. More formally, given a prefix $S^n = (t_1, \dots, t_n)$ of S and a channel c, we let $(c!x_1, c!x_2, \dots)$ denote the projection of the labels of $(t_1, \dots, t_n)$ onto the set of events that are sends on c. Then define $\mathsf{Sent}_c(S^n) = (x_1, x_2, \dots)$. This is the

sequence of messages that are sent on c in the prefix S^n. The sequence $\mathsf{Received}_c(S^n)$ is defined similarly as the sequence of messages that are received on c in S^n. We then require that, for all n, $\mathsf{Received}_c(S^n)$ is a prefix of $\mathsf{Sent}_c(S^n)$. We set $\mathsf{Pending}_c(S^n) = \mathsf{Sent}_c(S^n) \setminus \mathsf{Received}_c(S^n)$, so $\mathsf{Pending}_c(S^n)$ represents the messages remaining in the queue for channel c after the execution of S^n. A sequence S satisfying these requirements will be referred to as an *execution prefix*.

2.3 From Code to Models

We briefly describe how our model relates to actual program code.

A state in a process automaton M_p represents the local state of process p—the values of its variables and program counter, etc. A local-event transition represents a change in state in p that does not involve any communication with other processes—for example, the assignment of a new value to a variable. The labels on the local-event transitions do not play a significant role in this paper. One could, for example, just use a single label for all the local-event transitions in a process. On the other hand, if one wishes to reason about particular local events in a correctness property, one could use different labels for those transitions so that they could be referenced in the property specification.

A sending state represents a point in code just before a send operation. At that point, the local state of the process invoking the send contains all the information needed to specify the send exactly: the value to be sent, the process to which the message is to be sent, and the tag. After the send has completed, the state of this process is exactly as it was before, except for the program counter, which has now moved to the position just after the send statement. That is why there is precisely one transition departing from the sending state.

A receiving state represents a point in code just before a receive operation. Unlike the case for send, this process does not know, at that point, what value will be received. The state of the process after the receive completes may depend on the particular value received, since the variable into which the value is stored may take on a new value. Hence transitions (possibly to distinct states) must be included for every possible value that could be received.

A send-receive state represents a point in code just before a send-receive statement. According to the MPI Standard, the send and receive operations may be thought of as taking place in two concurrent threads; we model this by allowing the send and receive to happen in either order. If the send happens first, this process then moves to a receiving state, whereas if the receive happens first, the process moves to one of the sending states. After the second of these two operations occurs, the process moves to a state that represents the completion of the send-receive statement. Notice that there is always a "dual path" to this state, in which the same two operations occur in the opposite order.

We model the collective operations by introducing an auxiliary process to serve as a coordinator. For instance, for the MPI_BARRIER function, we create a new process corresponding to the barrier. For each of the original processes, we replace the call to MPI_BARRIER by statements that send a fixed message to the barrier process and then receive a fixed message from the barrier process. The barrier process itself simply receives the corresponding messages from each of the original processes and then sends the appropriate messages back to them. None of the original processes can proceed until the barrier process has received all of their messages. The other collective operations are modeled in similar fashion.

2.4 Relations on Execution Prefixes

The theorems we present in the next section depend on identifying certain relations between execution prefixes. For instance, we show that if any execution prefix for a particular model leads to deadlock, there must be an associated prefix, also leading to deadlock, in which each communication takes place synchronously. In this section we sketch some of the relations we need.

We say that an execution prefix $S = (t_1, t_2, \dots)$ is *synchronous* if, for each channel c, whenever the transition t_i is labeled by a send event $c!x$, the transition t_{i+1} is labeled by the receive event $c?x$. For technical reasons involving the MPI semantics, we also require that the sender and receiver on channel c be different unless the source of transition t_i is a send-receive state.

Consider a send-receive state in which either the send event $c!x$ or the receive event $d?y$ is possible. Note that our definition of process automata requires that the automaton reaches the same state whether the transitions occur in the order $c!x\, d?y$ or $d?y\, c!x$. Our model is intended to reflect the MPI semantics in which the send and receive take place as if they were executed by independent threads, so we want the process execution in which the $c!x$ is followed by the $d?y$ to be the same as the one in which the $d?y$ is followed by the $c!x$. To deal with this issue, we introduce a series of relations on paths through process automata and on execution prefixes.

We say that two paths π and ρ through a process automaton are *equivalent*, written $\pi \sim \rho$, if one can be obtained from the other by reversing the order of a (possibly infinite) set of send and receive transitions at send-receive states. This is an equivalence relation. We write $\pi \preceq \rho$ if π is a prefix of a path that is equivalent to ρ. Note that $\pi \sim \rho \Leftrightarrow \pi \preceq \rho \wedge \rho \preceq \pi$. We write $\pi \prec \rho$ if $\pi \preceq \rho$ and $\pi \not\sim \rho$. We say π and ρ are *comparable* if $\pi \preceq \rho$ or $\rho \preceq \pi$.

Given a sequence S of transitions and a process p, we write $S{\downarrow_p}$ for the projection of S onto the set of transitions of the process automaton M_p, i.e., the subsequence of S consisting of transitions from M_p.

The MPI Standard allows an MPI implementation to buffer an outgoing message, so that the send operation may complete before the receiving process has even initiated a receive operation. In such a situation, we think of the message as existing in a *system buffer*. The implementation may, however, block the sending process for an undetermined time. If the receiving process is in a matching receive and there is no pending message in the system buffer that also matches the message, the implementation must allow the communication to take place synchronously. If the implementation chooses to block the sender until these conditions are met, we say that it *forces the send to synchronize*. The possibility that an implementation may block some messages for undetermined amounts of time, possibly forcing some sends to synchronize, is a major source of the nondeterministic behavior of MPI programs, and thus a major problem in verifying their properties. To deal with this, we introduce the notion of a *universally permitted extension* of a finite execution prefix.

```
1   global_error = epsilon;
2   while (global_error >= epsilon) {
3     for (iter = 0; iter < niter; iter++) {
4       if (rank == 0) MPI_Send(grid[ny-2], nx, MPI_DOUBLE, 1, 0, comm);
5       else if (rank < nprocs-1)
6         MPI_Sendrecv(grid[ny-2], nx, MPI_DOUBLE, rank+1, 0, grid[0], nx,
              MPI_DOUBLE, rank-1, 0, comm, &status);
7       else MPI_Recv(grid[0], nx, MPI_DOUBLE, nprocs-2, 0, comm, &status);
8       if (rank == nprocs - 1) MPI_Send(grid[1], nx, MPI_DOUBLE, nprocs-2, 0, comm);
9       else if (rank > 0)
10        MPI_Sendrecv(grid[1], nx, MPI_DOUBLE, rank-1, 0, grid[ny-1], nx,
              MPI_DOUBLE, rank+1, 0, comm, &status);
11      else MPI_Recv(grid[ny-1], nx, MPI_DOUBLE, 1, 0, comm, &status);
12      /* local update of grid interior */
13    }
14    /* computation of local_error */
15    MPI_Allreduce(&local_error, &global_error, 1, MPI_DOUBLE, MPI_SUM, comm);
16  }
```

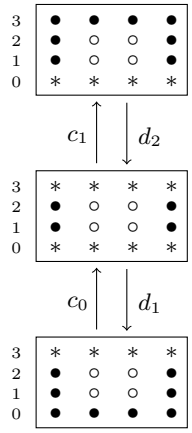

Figure 1: Jacobi iteration on a two-dimensional grid: $\bullet$ = boundary cell, $\circ$ = interior cell, $*$ = ghost cell

Let $S = (s_1, \ldots, s_m)$ be a finite execution prefix of a model $\mathcal{M}$ for an MPI program $\mathcal{P}$. A finite execution prefix $T = (s_1, \ldots, s_m, \ldots, s_n)$ extending S is *universally permitted* if, for all i such that $m < i \leq n$ and s_i is a send transition with, say, label $c!x$, then s_{i+1} is labeled by the event $c?x$ and $\mathsf{Pending}_c(s_1, \ldots, s_{i-i})$ is empty. In other words, after the last transition in S, no more buffering is allowed: only synchronous communication and the receipt of messages that were already buffered in the course of S are permitted. The idea is that the universally permitted extensions are precisely the ones that must be allowed by any legal MPI implementation, no matter how strict its buffering policy.

2.5 Example: Jacobi Iteration

In this section we will describe an example MPI program and model. Though simple, this example demonstrates some of the typical issues that arise in creating finite-state models from MPI codes. The same example will also be used throughout §3 to elucidate the general theorems presented there.

The program, whose code is outlined in Figure 1, computes a solution to the two-dimensional Laplace equation using Jacobi iteration [1]. Conceptually, there is a global rectangular grid in which the values on the boundary points are specified. The rows of this grid are distributed among the `nprocs` processes in such a way that each process has $\texttt{ny} - 2$ interior rows, stored in the `grid[`i`]`, for $1 \leq i \leq \texttt{ny}-2$. In the process of rank 0, `grid[0]` holds the bottom boundary row, and in the process of rank $\texttt{nprocs}-1$, `grid[ny`-1`]` holds the top boundary row. In all processes of positive rank, `grid[0]` is used to mirror the contents of the `grid[ny` $- 2$`]` on the process below; on all processes of rank less than $\texttt{nprocs} - 1$, `grid[ny` $- 1$`]` is used to mirror the contents of the `grid[1]` on the process above. The values of these *ghost cell* rows are updated on each iteration of the `for` loop through a series of point-to-point MPI functions. This grid structure is illustrated on the right side of Figure 1.

After updating the ghost cells, the process proceeds to update the interior values of `grid`, a purely local operation. This is repeated `niter` times, after which another local operation computes the local error for that process. A call to `MPI_Allreduce` adds the local error terms and returns the sum to each process, where it is stored in `global_error`. These steps are repeated until the global error falls below a threshold ϵ, at which point all processes exit the code fragment. The values of `nx`, `ny`, `niter` and ϵ are assumed to be constant and the same on every process, and `rank` is assumed to have been set to the rank of the process.

We construct a model $\mathcal{M}_0$ of the Jacobi code as follows. First, we let $\mathcal{M}_0$ contain one channel c_i, which sends from process i to $i+1$, for $0 \leq i \leq \texttt{nprocs}-2$, and one channel d_j, which sends from process j to $j-1$, for $1 \leq j \leq \texttt{nprocs}-1$. Next, we abstract away the grid values. Specifically, in $\mathcal{M}_0$, in place of each operation that sends an array of floating point numbers, there will be an operation that sends the single integer 1. A coordinator process and another set of channels (from each process to the coordinator and from the coordinator to each process) are added to model the reduction operation. The `local_error` variables are also abstracted away. So in $\mathcal{M}_0$, a process begins the reduction operation by simply sending a 1 to the coordinator. The coordinator receives all these messages in rank order.

We make a slightly more precise abstraction of the global error in $\mathcal{M}_0$. In particular, after receiving the 1 from all processes, the coordinator makes a non-deterministic choice between 0 and 1 and sends the chosen value to each of the processes, in rank order. The choice of 1 represents a value for `global_error` which is less than ϵ, while 0 represents a value which is not. We know that this abstraction results in a conservative model, since (i) the MPI Standard guarantees that the reduction operation return the same value to every process, and (ii) every process has the same value for ϵ, by assumption.

An execution prefix for $\mathcal{M}_0$, in the case $\texttt{nprocs} = 3 = \texttt{niter}$, might be synchronous, as in (leaving out local events)

$$c_0!1, c_0?1, c_1!1, c_1?1, d_2!1, d_2?1, d_1!1, d_1?1,$$

or it might proceed more in a lockstep pattern, as follows:

$$c_0!1, c_1!1, c_0?1, c_1?1, d_2!1, d_1!1, d_2?1, d_1?1.$$

What is less obvious is that there are execution prefixes in which the processes can move further apart from each other. In particular, a point can be reached at which the process of rank i is in the i^{th} iteration of the `for` loop, for all $0 \leq i < \texttt{nprocs}$. In the $\texttt{nprocs} = 3 = \texttt{niter}$ case this

global state is arrived at by the following prefix:

$$c_0!1, c_0?1, c_1!1, c_1?1, d_2!1, d_2?1, d_1!1, c_1!1, c_1?1, d_2!1. \quad (1)$$

It is this wide range of possible behaviors that makes it difficult to reason about the correctness of MPI programs, and also leads to the explosion of the state space that can make finite-state verification infeasible.

There are a number of correctness properties that we might like to establish for our Jacobi program. In addition to freedom from deadlock, we might want to check that all ghost cells are updated correctly. Specifically, just before the i^{th} update of the grid interior on a process p, the value of a ghost cell that mirrors an interior cell on process q should equal the value of that interior cell when q was just about to perform its i^{th} update. A third property is the claim that the program behaves *deterministically*, that is, given the same input twice, it will always produce the same result, independent of the choices made by the MPI implementation.

3. THE THEOREMS

In this section we describe our main results concerning models of MPI programs. Each theorem will also be illustrated with an application to the Jacobi example for `niter` = 5 = `nprocs`. The source code and other artifacts for these applications are available at `http://laser.cs.umass.edu/~siegel/projects`.

Throughout this section, $\mathcal{M}$ will denote a model of an MPI program with no wildcard receives, that is, with no use of MPI_ANY_SOURCE or MPI_ANY_TAG, and Proc will denote the set of all processes of $\mathcal{M}$.

3.1 Deadlock

Checking for deadlocks in MPI codes can be tricky because whether or not a program deadlocks can depend upon the synchronization choices made by the MPI implementation. In general, if a program reaches a state from which the only possible actions are sends that cannot be received synchronously, the program can deadlock if the implementation chooses to force all the sends to synchronize. If, on the other hand, the implementation allows those sends to buffer, the program may continue without deadlocking. Clearly it would be of great benefit to know that a program could never deadlock, no matter what choices are made by the MPI implementation.

To make this precise, let Σ be a subset of Proc, and let S be a finite execution prefix. For each process p, let u_p be the state of p after execution of S. We write $|S|$ for the number of transitions in S.

Definition 1. We say that S is *potentially* Σ*-deadlocked* if, for some $p \in \Sigma$, u_p is not the final state, and S has no universally permitted proper extension.

It is not hard to see that this is equivalent to requiring that all of the following hold: (i) there is a $p \in \Sigma$ for which u_p is not the final state, (ii) no u_p is a local-event state, and (iii) if a process is at a receiving or send-receive state, then for the channel c for which there is a receive transition leaving that state, there are no pending messages on c and no process is at a state from which it can execute a send on c.

The potentially deadlocked prefixes are precisely the ones for which some choice by a legal MPI implementation would lead to deadlock (cf. [8]). Since this is precisely the kind of behavior we wish to avoid, we say that $\mathcal{M}$ is Σ*-deadlock-free* if it has no execution prefix of this form. We say that it is *synchronously* Σ*-deadlock-free* if it has no synchronous execution prefix of this form.

The set Σ in these definitions arises from the fact that, for some systems, we may not wish to consider certain potentially deadlocked prefixes as problematic. For example, if one process p represents a server, then often p is designed to never terminate, but instead to always be ready to accept requests from clients. In this case we probably would not want to consider an execution in which every process other than p terminates normally to be a deadlock. For such a system, Σ might be taken to be all processes other than the server.

Our main theorem concerning deadlock reduces the verification of freedom from deadlock to the synchronous case:

THEOREM 1. *Let* $\Sigma \subseteq$ Proc. *Then* $\mathcal{M}$ *is* Σ*-deadlock-free if, and only if,* $\mathcal{M}$ *is synchronously* Σ*-deadlock-free.*

We remark that the hypothesis forbidding wildcard receives in Theorem 1 is necessary. For an example of how the conclusion may fail if the hypothesis does not hold, see [13, §7].

Theorem 1 may impart an enormous advantage to FSV techniques, since the need to represent all possible states of message channels is one of the major sources of state explosion. The fact that we need only consider synchronous executions means that an FSV tool, such as the model checker SPIN [5], does not need to keep track of pending messages in its representation of the global state and can avoid exploring all of the resulting additional global states.

To demonstrate this benefit, we used SPIN, and the methods described in [14], to analyze the model $\mathcal{M}_0$ of §2.5, with `niter` = 5 = `nprocs`. Allowing messages to buffer, SPIN can indeed verify freedom from deadlock (for Σ = Proc), but only after exploring 1.4 million global states. If we restrict to synchronous communication, only 26,686 global states have to be explored. So by making use of Theorem 1, we reduce the size of the verification by a factor of 50 in this example.

There is a stronger version of the freedom from deadlock property, which in essence says that a specific process can never become permanently blocked. To make this precise, we make the following definition:

Definition 2. We say S is *potentially partially* Σ*-deadlocked* (or Σ*-ppd*, for short) if, for some $p \in \Sigma$, u_p is not the final state, and there is no universally permitted proper extension S' of S with $|S'\downarrow_p| > |S\downarrow_p|$.

The idea here is that a program that has followed the path of S may now be in a state in which process p will never be able to progress (though other processes may continue to progress indefinitely). Again, p *may* be able to progress, depending on the choices made by the MPI implementation. If the implementation allows buffering of messages then p may be able to execute, but if the implementation chooses, from this point on, to force all sends to synchronize, then p will become permanently blocked.

We say that $\mathcal{M}$ is *free of partial* Σ*-deadlock* if it has no execution prefix that is Σ-ppd. We say that $\mathcal{M}$ is *synchronously free of partial* Σ*-deadlock* if it has no synchronous execution prefix that is Σ-ppd.

It follows directly from the definitions that if $\mathcal{M}$ is free of partial Σ-deadlock then it is Σ-deadlock-free. In other

words, this new property is stronger than the old. And although the weaker property is probably more familiar, it is often the case that one expects the stronger version to hold for a large subset Σ of the set of processes. In fact, quite often one expects most or all of the processes in an MPI program to terminate normally on every execution, which certainly implies that the program should be free of partial deadlock for that set of processes.

Finally, to verify this stronger property we are also justified in restricting to the synchronous case:

THEOREM 2. *Let $\Sigma \subseteq \mathsf{Proc}$. Then $\mathcal{M}$ is free of partial Σ-deadlock if, and only if, $\mathcal{M}$ is synchronously free of partial Σ-deadlock.*

Verification of freedom from partial deadlock can be significantly more computationally intensive than verification of the weaker deadlock property. For our example, SPIN was able to verify the stronger property after exploring over 9 million global states. By restricting to synchronous communication, this number was reduced to 242,956 global states.

3.2 Barriers

Barriers can facilitate reasoning about the correctness of a program, because they reduce the number of ways events from the different processes can be interleaved. If we were to insert, for example, an `MPI_Barrier` statement in the code of Figure 1 between lines 11 and 12, we would eliminate executions such as (1). For the same reason, barriers can reduce the number of global states, and therefore facilitate finite-state verification. In our verification of freedom from deadlock using the buffering SPIN model of $\mathcal{M}_0$, for example, inserting the barrier reduces the number of global states explored from 1.4 million to 441,010; in the synchronous model, the number is reduced from 26,686 to 12,402. We will see in Theorem 3 below that the fact that the model with the barrier is deadlock-free *implies* that the original model is deadlock-free.

To reason about barriers in the general case, let B be a set of states from the various M_p of our wildcard-free model $\mathcal{M}$. We let $\mathcal{M}^B$ denote the model which is the same as $\mathcal{M}$, except that barriers have been inserted just before every state in B. For technical reasons we assume that B contains no initial state, nor an immediate successor of a send-receive state. The precise construction involves adding a coordinator process to $\mathcal{M}$, and adding a transition labeled by a send to the coordinator (indicating entrance to the barrier) followed by one labeled by a receive from the coordinator (indicating exit from the barrier) just before each state in B. The coordinator process simply receives an entrance message from each process, and then sends an exit message to each process, and repeats. We can prove the following:

THEOREM 3. *Let B be as above and $\Sigma \subseteq \mathsf{Proc}$. Suppose $\mathcal{M}^B$ is Σ-deadlock-free (resp., free of partial Σ-deadlock). Then $\mathcal{M}$ is Σ-deadlock-free (resp., free of partial Σ-deadlock).*

While barriers can actually benefit the analysis of an MPI program, they can also take a significant toll on the program's performance. It may therefore be reasonable to use barriers liberally in the development of an MPI code. Then, after a high level of confidence in the correctness of the program is achieved, through FSV or other techniques, one can begin to remove barriers that can be shown to be unnecessary for the correctness of the program. Theorem 3 can aid in this last step, since one can at least be confident that the removal will not introduce any deadlocks, assuming the program is written in our restricted subset of MPI and contains no wildcard receives.

The next result also concerns a model with barriers. We say that a finite execution prefix terminates inside a barrier, if, at the end of that prefix, every non-coordinator process has entered, but not yet exited, the barrier.

THEOREM 4. *Let B be as above, and $\Sigma \subseteq \mathsf{Proc}$. Suppose $\mathcal{M}^B$ is Σ-deadlock-free. Let S be a finite execution prefix for $\mathcal{M}^B$ that terminates inside a barrier. Then there is a synchronous execution prefix T such that $S{\downarrow_p} \sim T{\downarrow_p}$ for all processes p. In particular, $\mathsf{Pending}_c(S)$ is empty for all channels c.*

While Theorem 4 is stated in terms of barriers, the exact same reasoning applies to any collective function that requires every process to enter the communication before any process leaves it. (The other "barrier-like" functions are MPI_ALLGATHER, MPI_ALLGATHERV, MPI_ALLREDUCE, MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW, and MPI_REDUCE_SCATTER). So, for example, Theorem 4 implies that, whenever execution of the code of Figure 1 is at a point where all processes are "inside" the `MPI_Allreduce` function, there can be no pending messages in the entire system, i.e., every message sent has been received.

Let us see how Theorem 4 can aid in the verification of the ghost cell property for the Jacobi example (in its original form, without the added `MPI_Barrier` statement). To verify that property, it is clear that we need a model that represents the ghost cell values in a more precise way than they are represented in $\mathcal{M}_0$. One way might be to modify $\mathcal{M}_0$ by introducing a local variable `count` into each process. Initially 0, `count` is incremented just after each local update of the grid interior on line 12. Now for each of the MPI send operations, instead of simply sending a 1, we send `count`. In each of the MPI receive operations, the received value is stored in a variable `tmp`. The property of this model we want to check is that, after any process receives into `tmp`, it is always the case that `tmp` equals `count`.

The problem with the approach above is that the variable `count` can increase without bound, since there is nothing in the model which places a bound on the number of times a process can go through the `while` loop. This means that the model described above has an infinite number of global states. To remedy this problem, we can leverage the knowledge that there can be no pending messages when execution is inside the `MPI_Allreduce`. Because this is the case, we know that a message sent by some process p on the n^{th} iteration of the `while` loop can only be received by some process q on its n^{th} iteration of the `while` loop. We may therefore modify our model by resetting `count` to 0 just after the `MPI_Allreduce`, or, equivalently, just using the variable `iter` in place of `count`. This results in a finite-state model, and, after exploring 527,036 global states, SPIN verified that the property indeed holds on all executions of this model.

3.3 Channel Depth

During the course of execution of a prefix T, the number of pending messages in a channel may of course go up and down, and there is not necessarily a correlation between the

number of pending messages at the end of the execution and at the intermediate stages. However, the following theorem shows that in certain circumstances, one can always replace T by an execution prefix S in which the number of pending messages never exceeds the final value:

THEOREM 5. *Suppose $\Sigma \subseteq \mathsf{Proc}$ and $\mathcal{M}$ is free of partial Σ-deadlock. Let T be a finite execution prefix of $\mathcal{M}$ and assume that, after execution of T, no process is at a state that is an immediate successor to a send-receive state. Then there exists an execution prefix S of $\mathcal{M}$ satisfying all of the following:*

1. *$S{\downarrow_p} \sim T{\downarrow_p}$ for all $p \in \Sigma$.*
2. *$S{\downarrow_p} \preceq T{\downarrow_p}$ for all $p \in \mathsf{Proc} \setminus \Sigma$.*
3. *For all channels c for which the receiving process is in Σ, the following holds: if $|\mathsf{Pending}_c(T)| = 0$ then S is c-synchronous, while if $|\mathsf{Pending}_c(T)| > 0$ then for all i, $|\mathsf{Pending}_c(S^i)| \leq |\mathsf{Pending}_c(T)|$.*

To see how Theorem 5 can be applied to our Jacobi example, let us again consider the question of the correctness of the ghost cells. We saw in §3.2 how to verify this property using a finite-state model that required buffering, but we will now see that there is a way to safely restrict to synchronous executions. The idea is just to delay the check that `tmp` = `iter` until a point where all channels are empty and then apply Theorem 5. Specifically, we add to each process a boolean variable `flag` that is initially 0. After each receive operation, we check to see if `tmp` $\neq$ `iter`, and, if that is the case, we set `flag` to 1. Now we ask SPIN to check that at the point just after the `MPI_Allreduce`, the value of `flag` can never be 1. Because the model is free of partial deadlock, we know that if `flag` is ever set to 1 by a process p, then p will eventually reach the point just after the `MPI_Allreduce` statement. Furthermore, by Theorem 4, we know that this point can be reached by an execution prefix T such that, just after execution of T, all channels will be empty. By Theorem 5, there exists a synchronous execution S such that $S{\downarrow_p} \sim T{\downarrow_p}$ for all p. In particular, for each p, the state of p after execution of S is the same as that resulting from the execution of T, and so if T results in a state for which some `flag` = 1, so will S. Hence if there exists a violation of the property, there exists a synchronous violation, and therefore if we can verify the property for synchronous executions we have verified that it holds in general. Using this synchronous approach, SPIN explored only 35,391 global states, rather than the 527,036 required by the buffering approach described in §3.2.

3.4 Locally Deterministic Models

We now consider a particularly well-behaved class of models of MPI programs, which we call *locally deterministic models*. These are models in which there are not only no wildcard receives, but no non-deterministic local choices in the automaton for any process.

Definition 3. We say that a model $\mathcal{M}$ of an MPI program is *locally deterministic* if it has no wildcard receives, and, for every local-event state u, there is precisely one transition t departing from u.

Notice that the model $\mathcal{M}_0$ of our example program is *not* locally deterministic because of the non-deterministic choice made by the coordinator in choosing between the two possible values for the global error. However, there are locally deterministic models of the program that can be used to reason about the program, and which we now describe.

Suppose, more generally, that we are given any MPI program that uses only the MPI functions enumerated in §2, contains no wildcard receives, and uses no non-deterministic functions (such as a function that returns a random value). Say we are also given an *input vector* $\mathbf{v}$ for that program, i.e., a vector specifying the initial value of every variable in the program. Then we may consider the *full-precision real model* $\mathcal{M}_{\mathbf{R}}(\mathbf{v})$ in which every floating-point variable of the program is treated as a real number, and the initial state of the model is determined by $\mathbf{v}$. Since all of the arithmetic and other functions are deterministic, $\mathcal{M}_{\mathbf{R}}(\mathbf{v})$ is a locally-deterministic model.

We may also consider the *floating-point model* $M_{\mathbf{F}}(\mathbf{v})$, in which the variables and arithmetic are represented exactly as they are on a particular computing platform. Because floating-point arithmetic is only an approximation to real arithmetic, $M_{\mathbf{F}}(\mathbf{v})$ is not necessarily equivalent to $\mathcal{M}_{\mathbf{R}}(\mathbf{v})$. In particular, there are ways in which $M_{\mathbf{F}}(\mathbf{v})$ can fail to be locally deterministic. This may happen if reduction functions such as MPI_ALLREDUCE are used with operations that are not strictly associative and commutative, like floating-point addition. This is the case with our example program. The problem is that the MPI Standard allows the MPI implementation to apply the reduction operation to the terms in any order it likes, so the results returned by two calls to MPI_ALLREDUCE may differ, even if given the same input.

In any case, given a locally deterministic model, the analysis is greatly simplified. For even though there may still exist many possible executions of the model—due to the interleaving and buffering choices allowed by MPI—these executions must all be the same in certain significant ways. In particular, the same messages will always be sent on each channel, in the same order, and the same paths will be followed in each process automaton (except possibly for the order in which the send and receive operations take place within a send-receive call). The precise result is:

THEOREM 6. *Suppose $\mathcal{M}$ is a locally deterministic model of an MPI program. Then there exists an execution prefix S for $\mathcal{M}$ with the following property: if T is any execution prefix of $\mathcal{M}$, then for all $p \in \mathsf{Proc}$, $T{\downarrow_p} \preceq S{\downarrow_p}$.*

Applying Theorem 6 to $M_{\mathbf{R}}(\mathbf{v})$, we conclude that, for a given $\mathbf{v}$, the same values would be computed by any execution of the program, if all arithmetic used in execution were precisely real arithmetic. If the program uses no reduction functions on operations that are not associative and commutative, the same reasoning applies to $M_{\mathbf{F}}(\mathbf{v})$; in this case we can conclude that, for a given $\mathbf{v}$, any two executions of the program on the same platform will return the exact same floating-point results. If the program does contain such reduction functions, the only source for differences between two executions is the error introduced by the failure of the floating-point operations to be associative and commutative.

Questions concerning deadlock are also easily answered for locally deterministic models:

COROLLARY 7. *Let $\mathcal{M}$ be a locally deterministic model of an MPI program, and $\Sigma \subseteq \mathsf{Proc}$. Then*

1. *$\mathcal{M}$ is Σ-deadlock-free if, and only if, there exists a synchronous execution prefix T such that either T is infinite or T is finite and ends with every process in Σ at its final state.*

2. *$\mathcal{M}$ is free of partial Σ-deadlock if, and only if, there exists a synchronous execution prefix T such that for each $p \in \Sigma$, either $T{\downarrow_p}$ is infinite or $T{\downarrow_p}$ is finite and ends with p at its final state.*

Hence one need only examine a single synchronous execution to determine whether or not that model is deadlock-free. There is no need to examine all possible executions.

4. CONCLUSION

We believe that FSV techniques, such as model checking, have considerable potential for finding bugs in, or verifying correctness properties of, parallel programs. In particular, we are concerned with programs that carry out large scale computations using MPI. Such programs are hard to write correctly, and extremely hard to debug. Several features of MPI, such as the unpredictable buffering of messages, make the state spaces of such programs especially large and thus present significant obstacles to the successful application of FSV techniques.

In this paper, we have considered programs written using a subset of the MPI communication constructs. This subset includes the standard blocking send and receive operations and the collective operations, such as MPI_BARRIER and MPI_ALLREDUCE, but excludes both MPI_ANY_SOURCE and MPI_ANY_TAG. Although this subset omits many of the MPI functions, it is rich enough to express a large class of parallel algorithms. We have presented several theorems that can be used to substantially reduce the state space that must be considered in finite-state verification of programs written using this subset and have illustrated these results by applying them to a small but realistic example.

While the example we used in this paper was very simple, the theorems have also been applied to significantly more complex codes. One interesting case concerns a component from the FLASH project [2], a sophisticated parallel multiphysics application. The code implements a block redistribution algorithm for an adaptively refined mesh, which requires that blocks of data be redistributed periodically among the processes according to a complex communication pattern. The original redistribution routine, which was known to deadlock in some scenarios, was replaced with a routine written entirely in our restricted subset of MPI. This made it relatively easy, using Corollary 7, to establish freedom from deadlock for the new version.

For our initial applications of FSV techniques to MPI programs, we have created finite-state models for SPIN and other FSV tools by hand. There are three important limitations to this approach. First, it is extremely labor-intensive, requiring the analyst to come to a thorough understanding of the program being analyzed. While merely difficult for small programs, this would be virtually impossible for large, complex ones. Second, it requires the analyst to be an expert in the modeling language being used. Third, in addition to being labor-intensive, manual creation of models of complex systems is error-prone, making the fidelity of such models a special concern. We therefore plan to develop a tool that will take the program code as input and produce a model in the input language of an FSV tool as output, building on recent research in automatic model extraction and construction for FSV (e.g., [4]). We are also exploring other ways to extend the applicability of FSV techniques to larger classes of MPI programs, including sophisticated techniques for automating abstractions like those used in the models of the Jacobi example given in this paper.

The results presented here depend crucially on the avoidance of the wildcard constructs MPI_ANY_SOURCE or MPI_ANY_TAG, which allow a receive operation to collect a message from any of several channels. For programs making use of wildcard receives, other, somewhat weaker, reduction methods are possible [12]. Furthermore, we believe that at least some programs using wildcard receives can be rewritten to avoid them. While this may incur a cost in compactness or understandability, this cost may well be offset by the improvement in analyzability, leading to lower development cost and greater reliability.

Acknowledgments

We thank Andrew Siegel and Rusty Lusk for a great many helpful discussions about MPI and scientific computation.

This research was partially supported by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement number DAAD190110564.

5. REFERENCES

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[2] ASCI/Alliance Center for Astrophysical Thermonuclear Flashes web site. `http://flash.uchicago.edu`.

[3] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, Cambridge, 1999.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, June 2000.

[5] G. J. Holzmann. *The* SPIN *Model Checker.* Addison-Wesley, Boston, 2004.

[6] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[7] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18:717–721, Dec. 1975.

[8] G. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14:911–932, 2002.

[9] Message Passing Interface Forum. MPI: A Message-Passing Interface standard, version 1.1. `http://www.mpi-forum.org/docs/`, 1995.

[10] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. `http://www.mpi-forum.org/docs/`, 1997.

[11] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[12] S. F. Siegel. Efficient verification of halting properties for MPI programs with wildcard receives. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation: 6th International Conference, VMCAI 2005, Paris, January 17–19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 413–429, 2005.

[13] S. F. Siegel and G. S. Avrunin. Modeling MPI programs for verification. Technical Report UM-CS-2004-75, Department of Computer Science, University of Massachusetts, 2004.

[14] S. F. Siegel and G. S. Avrunin. Verification of MPI-based software for scientific computation. In S. Graf and L. Mounier, editors, *Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1–3, 2004, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*, pages 286–303. Springer-Verlag, 2004.

[15] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*. IEEE Computer Society, 2000. Article 51.

APPENDIX

A. PROOFS

Space limitations prevent us from giving complete proofs of the theorems in this paper; they can be found in [13]. In this appendix, however, we establish some of the key technical results needed to prove the theorems stated above, and sketch the proofs of some of those theorems. The proofs of the other theorems have a similar flavor.

The basic idea of the proofs is to show that, given an execution prefix for a model $\mathcal{M}$, we can reorder certain transitions to obtain another execution prefix. These reorderings change the interleaving of transitions from different processes, but do not affect the relative order of transitions in a single process. Thus, for example, given a potentially deadlocked execution for a model without wildcard receives, we can reorder the transitions to obtain a potentially deadlocked execution in which all the communication takes place synchronously. These techniques are related to those developed in the large literature on reduction and atomicity, beginning with [7].

In what follows, if t is a transition in a process automaton from a state u to a state v, we define $\mathsf{src}(t) = u$ and $\mathsf{des}(t) = v$.

Definition 4. If $\pi = (t_1, \ldots, t_n)$ is a finite path through M_p, we let $\mathsf{terminus}(\pi)$ be $\mathsf{des}(t_n)$ if $n \geq 1$ and the start state of M_p otherwise. Let S be a finite execution prefix of $\mathcal{M}$. The *terminal state of* S is the global state $\mathsf{terminus}(S)$ defined by $\mathsf{terminus}(S)_p = \mathsf{terminus}(S{\downarrow_p})$.

Definition 5. Let ρ and σ be paths through M_p for some $p \in \mathsf{Proc}$. We say ρ and σ are *compatible* if there exists a path π through M_p such that $\rho \preceq \pi$ and $\sigma \preceq \pi$. We say that two execution prefixes S and T of $\mathcal{M}$ are *compatible* if $S{\downarrow_p}$ is compatible with $T{\downarrow_p}$ for all $p \in \mathsf{Proc}$.

The following is not hard to verify:

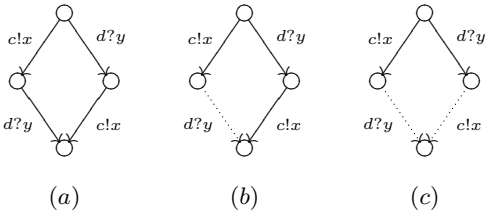

Figure 2: Compatible paths from a send-receive state. In each case, ρ, the path on the left (excluding the dotted arrows), is compatible with σ, the path on the right. In (a), $\rho \sim \sigma$. In (b), $\rho \prec \sigma$. In (c), ρ and σ are non-comparable compatible paths. In all cases, π may be taken to be either of the paths from the top node to the bottom node of the diamond.

LEMMA 8. *Suppose $\rho = (s_1, s_2, \ldots)$ and $\sigma = (t_1, t_2, \ldots)$ are compatible paths through M_p. If $|\rho| \neq |\sigma|$, then $\rho \prec \sigma$ or $\sigma \prec \rho$. If $|\rho| = |\sigma|$ then either $\rho \sim \sigma$ or ρ is finite, say $|\rho| = n$, and all of the following hold:*

(i) $\mathsf{terminus}(\rho^{n-1}) = \mathsf{terminus}(\sigma^{n-1})$ *and is a send-receive state.*

(ii) $\rho^{n-1} \sim \sigma^{n-1}$.

(iii) One of s_n, t_n is a send, and the other a receive.

(iv) If $\pi = (s_1, \ldots, s_n, \bar{t}_n)$ or $\pi = (t_1, \ldots, t_n, \bar{s}_n)$, where $\bar{s}_n$ and $\bar{t}_n$ are chosen so that $(s_n, \bar{t}_n)$ is the dual path to $(t_n, \bar{s}_n)$, then $\rho \preceq \pi$ and $\sigma \preceq \pi$.

See Figure 2 for an illustration of the different cases of compatibility described in the Lemma. The last case ($|\rho| = |\sigma|$ but $\rho \not\sim \sigma$) describes precisely the non-comparable compatible paths.

The next lemma provides the key inductive step for arguments about compatibility.

LEMMA 9. *Let $\mathcal{M}$ be a model of an MPI program with no wildcard receives. Let $S = (s_1, \ldots, s_n)$ be a finite execution prefix and T an arbitrary execution prefix for $\mathcal{M}$. Suppose S^{n-1} is compatible with T and s_n is a send or receive. Then S is compatible with T.*

We observe that the set of sequences that may be appended to S to create universally permitted extensions depends only on the states $\mathsf{terminus}(S)_p$ and the sequences $\mathsf{Pending}_c(S)$ (and not on the history of how one arrived at those states and queues). From this observation, it follows that if S and S' are two finite execution prefixes such that $S{\downarrow_p} \sim S'{\downarrow_p}$ for all $p \in \mathsf{Proc}$, then there is a 1-1 correspondence between the universally permitted extensions of S and those of S', with the property that if T corresponds to T' under this correspondence, then $T \setminus S = T' \setminus S'$.

Suppose now that we are given a fixed, finite, execution prefix T, and a second execution prefix S that is compatible with T. We will often have a need to extend S, in a universally permitted way, so that it maintains compatibility with T. The following proposition shows that, if we extend S far enough in this way, then we reach a point where any further universally permitted extension *must* be compatible with T. An even stronger statement can be made if T is synchronous. Like most of the results in this paper, we require that there are no wildcard receives. This proposition is a key part of the proofs of the theorems stated above.

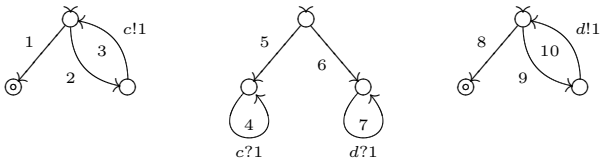

Figure 3: A Model of an MPI Program with 3 Processes. Edges with no label represent local-event transitions. Process 0 (left) chooses to either terminate, or to move to a state from which it will send a message to process 1 on channel c, then return to its start state and repeat. Process 2 (right) does the same for channel d. Process 1 (middle) chooses, once and for all, whether to move to a state from which it will loop forever receiving messages on c, or to do that for d.

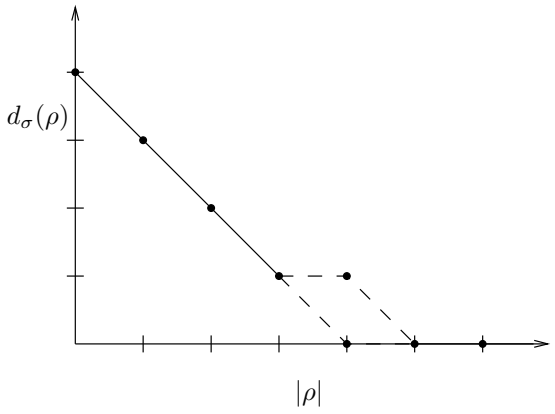

Figure 4: $d_\sigma(\rho)$ as a function of $|\rho|$

PROPOSITION 10. *Let $\mathcal{M}$ be a model of an MPI program with no wildcard receives. Let S and T be compatible finite execution prefixes for $\mathcal{M}$. Then there is a universally permitted finite extension S' of S, with the property that any universally permitted extension of S' is compatible with T. Moreover, if T is synchronous, then S' may be chosen so that $T{\downarrow_p} \preceq S'{\downarrow_p}$ for all $p \in \mathsf{Proc}$.*

Let us look at an example using the model illustrated in Figure 3. We will take

$$T = (2, 3, 2, 3, 1, 9, 10, 9, 10, 8, 6, 7).$$

To summarize T, first process 0 sends two messages on c and terminates, then process 2 sends two messages on d and terminates, then process 1 chooses the d branch and receives one message on d. Suppose that

$$S = (2, 3, 9, 10).$$

Clearly, S is compatible with T, as $S{\downarrow_p} \prec T{\downarrow_p}$ for all p. Now, there are many universally permitted extensions of S that are not compatible with T, for example

$$(2, 3, 9, 10, 5, 4).$$

This cannot be compatible with T since the projection onto process 1 begins with transition 5, while the projection of T onto that process begins with transition 6.

Let us consider however the following universally permitted extension of S:

$$S' = (2, 3, 9, 10, 6, 7, 9, 10, 7, 2, 8).$$

We have $S'{\downarrow_0} \prec T{\downarrow_0}$, $T{\downarrow_1} \prec S'{\downarrow_1}$, and $S'{\downarrow_2} = T{\downarrow_2}$. Clearly, no extension of S' could receive any messages on c, and therefore no universally permitted extension of S' could send any more messages on c. This means that no universally permitted extension of S' can have any additional transitions in process 0. Since in the other two processes, S' has already "covered" T, any universally permitted extension of S' will be compatible with T. So S' is the sort of prefix whose existence is guaranteed by Proposition 10.

The second part of the proposition says that if T were synchronous, then there would exist an S' that "covers" T on every process.

The idea behind the proof of Proposition 10 is to choose an S' that maximizes its "coverage" of T. To make this precise, we introduce the following function. Suppose ρ and σ are compatible paths through some M_p and ρ is finite. We want to measure the part of σ that is not "covered" by ρ. Recall from Lemma 8 that if $|\sigma| > |\rho|$ we must have $\rho \prec \sigma$, while if $|\sigma| = |\rho|$, ρ and σ are equivalent except possibly for the last transition in each. Define

$$d_\sigma(\rho) = \begin{cases} \max(0, |\sigma| - |\rho|) & \text{if } |\sigma| \neq |\rho| \\ 1 & \text{if } |\sigma| = |\rho| \text{ but } \sigma \nsim \rho \\ 0 & \text{if } \sigma \sim \rho. \end{cases}$$

Figure 4 shows the graph of $d_\sigma(\rho)$ as a function of the length of ρ for the case where $|\sigma| = 4$. Note that $d_\sigma(\rho)$ is a non-increasing function of $|\rho|$.

It follows from this that if ρ' is also compatible with σ, and $\rho \preceq \rho'$, then $d_\sigma(\rho) \geq d_\sigma(\rho')$.

PROOF OF PROPOSITION 10. Let S and T be as in the statement of the proposition. For any execution prefix R that is compatible with T, and $p \in \mathsf{Proc}$, define

$$d_p(R) = d_{\sigma(p)}(R{\downarrow_p}),$$

where $\sigma(p) = T{\downarrow_p}$. Define

$$d(R) = \sum_{p \in \mathsf{Proc}} d_p(R).$$

Next, consider the set of all universally permitted finite extensions of S that are compatible with T, and let S' be an element of that set that minimizes the function d. (The set of such extensions is non-empty, as S is a universally permitted extension of itself.) It follows from the previous paragraph that if R is any extension of S' that is compatible with T then $d_p(R) \leq d_p(S')$ for all $p \in \mathsf{Proc}$; so if R is also universally permitted then in fact we must have $d_p(R) = d_p(S')$ for all p.

Let $\tilde{S} = (s_1, s_2, \ldots)$ be a universally permitted extension of S'. We will assume $\tilde{S}$ is not compatible with T and arrive at a contradiction.

Let n be the greatest integer such that $\tilde{S}^n$ is compatible with T. Let $\sigma = \tilde{S}^n{\downarrow_p}$, where s_{n+1} is a transition in M_p. By definition, σ and $T{\downarrow_p}$ are compatible. Moreover, $n \geq |S'|$ since S' is compatible with T. Finally, by Lemma 9, s_{n+1} must be a local-event transition.

Now precisely one of the following must hold: (i) $T{\downarrow_p} \preceq \sigma$, (ii) $\sigma \prec T{\downarrow_p}$, or (iii) $T{\downarrow_p}$ and σ are non-comparable.

However, (i) cannot be the case, for it would imply that $T\downarrow_p \preceq \tilde{S}^{n+1}\downarrow_p$, which in turn implies that T is compatible with $\tilde{S}^{n+1}$, which contradicts the maximality of n.

Nor can (iii) be the case. For then σ and $T\downarrow_p$ would be non-comparable compatible paths, and Lemma 8 would imply that $\mathsf{terminus}(\sigma)$ is either a sending or receiving state. But since s_{n+1} is a local-event transition, $\mathsf{terminus}(\sigma)$ must be a local-event state.

Hence $\sigma \prec T\downarrow_p$, i.e., σ is a proper prefix of a sequence τ that is equivalent to $T\downarrow_p$. Now let t be the $(|\sigma|+1)^{\text{th}}$ transition of τ, so that s_{n+1} and t are distinct local-event transitions departing from the same state.

Consider the sequence $R = (s_1, \ldots, s_n, t)$. Then R is an execution prefix, it is a universally permitted extension of S', and it is compatible with T. However, $d_p(R) \leq d_p(S') - 1$, a contradiction, completing the proof of the first part of Proposition 10.

Now suppose that T is synchronous. By replacing S with S', we may assume that S and T are compatible and that any universally permitted extension R of S is compatible with T and satisfies $d_p(R) = d_p(S)$ for all $p \in \mathsf{Proc}$. Write $S = (s_1, \ldots, s_n)$ and $T = (t_1, t_2, \ldots)$.

We wish to show $T\downarrow_p \preceq S\downarrow_p$ for all p. So suppose this is not the case, and let k be the greatest integer such that $T^k\downarrow_p \preceq S\downarrow_p$ for all p. Now let $t = t_{k+1}$ and let p be the element of Proc for which t is a transition of M_p, and we have $T^{k+1}\downarrow_p \not\preceq S\downarrow_p$. We will arrive at a contradiction by showing there exists a universally permitted extension R of S with $d_p(R) < d_p(S)$.

For each $r \in \mathsf{Proc}$ there is a path

$$\sigma_r = (s_1^r, \ldots, s_{n(r)}^r)$$

through M_r that is equivalent to $S\downarrow_r$ such that for all $r \neq p$,

$$T^k\downarrow_r = T^{k+1}\downarrow_r = (s_1^r, \ldots, s_{m(r)}^r)$$

for some $m(r) \leq n(r)$, and such that

$$T^{k+1}\downarrow_p = (s_1^p, \ldots, s_{m(p)}^p, t).$$

We will consider first the case that $m(p) = n(p)$.

Suppose t is a send, say $\mathsf{label}(t) = c!x$. Then $\mathsf{label}(t_{k+2}) = c?x$, as T is synchronous. Moreover, $\mathsf{Pending}_c(T^k)$ is empty. Let q be the receiving process of c. If $p = q$ then $\mathsf{src}(t)$ is a send-receive state with the same sending and receiving channel, but let us assume for now that $p \neq q$. Let $u = \mathsf{src}(t_{k+2})$, so that $u = \mathsf{terminus}(T^k)_q$. Say that t is the i^{th} send on c in T^{k+1}. Then there are $i-1$ sends on c in S, and therefore no more than $i-1$ receives on c in S. This implies $m(q) = n(q)$: if not, there would be at least i receives on c in S. Hence $\mathsf{Pending}_c(S^n)$ is empty. Now, whether or not $p = q$, let $R = (s_1, \ldots, s_n, t, t_{k+2})$. Then R is a universally permitted extension of S satisfying $d_p(R) < d_p(S)$.

If t is a local-event transition, we may take

$$R = (s_1, \ldots, s_n, t). \tag{2}$$

Suppose t is a receive, say $\mathsf{label}(t) = c?x$, and say t is the i^{th} receive on c in T. Then t_k must be the matching send, i.e., t_k must be the i^{th} send on c in T and $\mathsf{label}(t_k) = c!x$. Let q be the process that sends on c. Since $T^k\downarrow_q \preceq S\downarrow_q$, there must be at least i sends on c in S, and the i^{th} element of $\mathsf{Sent}_c(S)$ is x. As there are $i-1$ receives on c in $S\downarrow_p$, we may conclude that $\mathsf{Pending}_c(S^n)$ begins with x. So taking R as in (2) will again suffice.

Now we turn to the case where $m(p) < n(p)$. Since $T^{k+1}\downarrow_p$ and $S\downarrow_p$ are compatible and neither $T^{k+1}\downarrow_p \preceq S\downarrow_p$ nor $S\downarrow_p \preceq T^{k+1}\downarrow_p$, Lemma 8 implies $n(p) = m(p)+1$ and one of $s = s_{m(p)+1}^p$, t is a send, and the other, a receive.

Suppose s is the send and t the receive. Then there is a receive transition $\bar{t}$ with $\mathsf{label}(\bar{t}) = \mathsf{label}(t)$ and $\mathsf{src}(\bar{t}) = \mathsf{des}(s)$. Arguing as in the case in which $n(p) = m(p)$, we see that the extension $R = (s_1, \ldots, s_n, \bar{t})$ is universally permitted, and satisfies $d_p(R) < d_p(S)$.

If, on the other hand, s is the receive and t the send, then t_{k+2} must be the receive matching t. Let $\bar{t}$ be the transition departing from $\mathsf{des}(s)$ (so $\mathsf{label}(\bar{t}) = \mathsf{label}(t)$). Arguing just as in the $m(p) = n(p)$ case we see that we may take

$$R = (s_1, \ldots, s_n, \bar{t}, t_{k+2}),$$

completing the proof of Proposition 10. □

Corollary 11. *Let $\mathcal{M}$ be a model of an MPI program with no wildcard receives, and let T be a finite execution prefix for $\mathcal{M}$. Then there exists a finite synchronous execution prefix S for $\mathcal{M}$, with the property that any synchronous extension of S is compatible with T.*

Proof. Apply Proposition 10 to the empty sequence and T, and recall that for a synchronous prefix, an extension is universally permitted if, and only if, it is synchronous. □

Using the Proposition, it is fairly easy to prove Theorem 1.

Proof of Theorem 1. If $\mathcal{M}$ is Σ-deadlock-free then, by definition, it has no execution prefix that is potentially Σ-deadlocked. So it suffices to prove the opposite direction.

So suppose $\mathcal{M}$ is synchronously Σ-deadlock-free, and that T is a finite execution prefix with $\mathsf{terminus}(T)_p$ not the unique final state of M_p for some $p \in \Sigma$. We must show there exists a universally permitted proper extension T' of T.

By Corollary 11, there is a synchronous finite execution prefix S with the property that any synchronous extension of S is compatible with T.

By hypothesis, either $\mathsf{terminus}(S\downarrow_p)$ is the final state of M_p for all $p \in \Sigma$ or there exists a synchronous proper extension S' of S. If the former is the case then we must have $|S\downarrow_p| > |T\downarrow_p|$ for some $p \in \Sigma$, by compatibility. If the latter is the case, then replace S with S' and repeat this process, until $|S\downarrow_r| > |T\downarrow_r|$ for some $r \in \mathsf{Proc}$; this must eventually be the case as the length of S is increased by at least 1 in each step.

Hence there is a finite synchronous execution prefix S, compatible with T, and an $r \in \mathsf{Proc}$ for which $|S\downarrow_r| > |T\downarrow_r|$. Now apply Proposition 10 to conclude there exists a finite, universally permitted extension T' of T with the property that $S\downarrow_p \preceq T'\downarrow_p$ for all $p \in \mathsf{Proc}$. We have

$$|T\downarrow_r| < |S\downarrow_r| \leq |T'\downarrow_r|,$$

so T' must be a proper extension of T. □

The proof of Theorem 2 uses the following lemma.

Lemma 12. *Let $\mathcal{M}$ be a model of an MPI program with no wildcard receives and $\Sigma \subseteq \mathsf{Proc}$. Assume $\mathcal{M}$ is synchronously free of partial Σ-deadlock. Then given any finite execution prefix T for $\mathcal{M}$, there exists a finite synchronous execution prefix S satisfying all of the following:*

(i) S *is compatible with* T.

(ii) $T{\downarrow_p} \preceq S{\downarrow_p}$ *for all* $p \in \Sigma$.

(iii) $T{\downarrow_p} \prec S{\downarrow_p}$ *if* $p \in \Sigma$ *and* $\mathsf{terminus}(T)_p$ *is not the final state of* M_p.

PROOF. By Corollary 11, there is a synchronous finite execution prefix S with the property that any synchronous extension of S is compatible with T. Fix $p \in \Sigma$.

By hypothesis, either $\mathsf{terminus}(S)_p$ is the final state of M_p, or there exists a synchronous proper extension S' of S satisfying $|S{\downarrow_p}| < |S'{\downarrow_p}|$. Replace S with S' and repeat, until $\mathsf{terminus}(S)_p$ is the final state of M_p or $|S{\downarrow_p}| > |T{\downarrow_p}|$. At least one of those two conditions must become true after a finite number of iterations, since in each iteration $|S{\downarrow_p}|$ is increased by at least 1.

Now we repeat the paragraph above for each $p \in \Sigma$. The result is a finite synchronous prefix S that is compatible with T. Again, let $p \in \Sigma$.

If $\mathsf{terminus}(S)_p$ is the final state of M_p, then by Lemma 8, $S{\downarrow_p}$ and $T{\downarrow_p}$ must be comparable. Since there are no transitions departing from final states, we must have $T{\downarrow_p} \preceq S{\downarrow_p}$, with $T{\downarrow_p} \sim S{\downarrow_p}$ if, and only if, $\mathsf{terminus}(T)_p$ is the final state. So both (ii) and (iii) hold.

If $\mathsf{terminus}(S)_p$ is not the final state, then by construction, $|S{\downarrow_p}| > |T{\downarrow_p}|$. Again by Lemma 8, S and T must be comparable, whence $T{\downarrow_p} \prec S{\downarrow_p}$, and so (ii) and (iii) hold in this case as well. □

PROOF OF THEOREM 2. If $\mathcal{M}$ is free of partial Σ-deadlock then, by definition, it has no execution prefix that is Σ-ppd. So it suffices to prove the opposite direction.

So suppose T is a finite execution prefix, $p \in \Sigma$, and $\mathsf{terminus}(T)_p$ is not the final state. We must show there exists a universally permitted proper extension T' of T with $|T{\downarrow_p}| < |T'{\downarrow_p}|$.

By Lemma 12, there is a finite synchronous execution prefix S that is compatible with T and satisfies $|S{\downarrow_p}| > |T{\downarrow_p}|$. Now apply Proposition 10 to conclude there exists a finite, universally permitted extension T' of T with the property that $S{\downarrow_r} \preceq T'{\downarrow_r}$ for all $r \in \mathsf{Proc}$. We have

$$|T{\downarrow_p}| < |S{\downarrow_p}| \leq |T'{\downarrow_p}|,$$

which completes the proof. □

Theorems 1 and 2 show that, in checking for deadlock or partial deadlock, we are justified in assuming all communication is synchronous. With a model checker such as SPIN, this means that we may use channels of depth 0. We now sketch the proof of Theorem 5, which gives us some control on the channel depths for other types of properties.

PROOF OF THEOREM 5. By Lemma 12, there exists a finite synchronous execution prefix $\tilde{S}$ that is compatible with T and satisfies $T{\downarrow_p} \preceq \tilde{S}{\downarrow_p}$ for all $p \in \Sigma$. Moreover, for any $p \in \mathsf{Proc}$, since $\mathsf{terminus}(T)_p$ is not an immediate successor to a send-receive state, Lemma 8 implies that $T{\downarrow_p} \preceq \tilde{S}{\downarrow_p}$ or $\tilde{S}{\downarrow_p} \preceq T{\downarrow_p}$.

We construct the sequence S as follows. We will begin by letting S be a copy of $\tilde{S}$, and we will then delete certain transitions from S. Specifically, for each $p \in \mathsf{Proc}$, let

$$m(p) = \min\{|\tilde{S}{\downarrow_p}|, |T{\downarrow_p}|\},$$

and then delete from S all the transitions that are in M_p but that occur after the $m(p)^{\text{th}}$ transition in M_p. Hence the resulting sequence S will have exactly $m(p)$ transitions in M_p for each $p \in \mathsf{Proc}$. We will show that S has the properties listed in the statement of Theorem 5.

First we must show that S is indeed an execution prefix. It is clear that $S{\downarrow_p}$ is a path through M_p for each p, and that, if $p \in \Sigma$, $S{\downarrow_p} \sim T{\downarrow_p}$. Now fix a $c \in \mathsf{Chan}$ and we must show that $\mathsf{Received}_c(S^n)$ is a prefix of $\mathsf{Sent}_c(S^n)$ for all n. To do this we argue as follows: let

$$\begin{aligned} r &= |\,\mathsf{Received}_c(T)| \\ s &= |\,\mathsf{Sent}_c(T)| \\ m &= |\,\mathsf{Received}_c(\tilde{S})| = |\,\mathsf{Sent}_c(\tilde{S})|. \end{aligned}$$

Now, if we project the sequence of labels of elements of $\tilde{S}$ onto the set of events involving c, the result is a sequence of the form

$$\tilde{C} = (c!x_1, c?x_1, c!x_2, c?x_2, \ldots, c!x_m, c?x_m),$$

as $\tilde{S}$ is synchronous. Now let

$$\begin{aligned} r' &= \min\{r, m\} \\ s' &= \min\{s, m\}. \end{aligned}$$

If we project the sequence of labels of elements of S onto the set of events involving c, the result is the sequence C obtained from $\tilde{C}$ by deleting all the receive events after the r'-th such event, and deleting all the send events after the s'-th such event. But since $r \leq s$, we have $r' \leq s'$. This means that

$$C = (c!x_1, c?x_1, \ldots, c!x_{r'}, c?x_{r'}, c!x_{r'+1}, \ldots, c!x_{s'}),$$

i.e., C begins with r' send-receive pairs, followed by a sequence of $s' - r'$ sends, which clearly satisfies the condition that the messages received at each point are a subset of the messages sent. Moreover, if $s' = r'$ then each send on c is immediately followed by a corresponding receive, while if not then

$$|\,\mathsf{Pending}_c(S^i)| \leq s' - r'$$

for all i in the domain of S.

Now if the process that receives on c belongs to Σ, then $r' = r$, whence

$$s' - r' \leq s - r = |\,\mathsf{Pending}_c(T)|.$$

So if $|\,\mathsf{Pending}_c(T)| = 0$ then $s' = r'$ and, as we have seen, this implies that S is c-synchronous. If $|\,\mathsf{Pending}_c(T)| > 0$, then for all i in the domain of S we have

$$|\,\mathsf{Pending}_c(S^i)| \leq s' - r' \leq |\,\mathsf{Pending}_c(T)|,$$

as claimed. □