MODELING MPI PROGRAMS FOR VERIFICATION

STEPHEN F. SIEGEL AND GEORGE S. AVRUNIN

Abstract. We investigate the application of finite-state verification techniques to parallel programs that employ the Message Passing Interface (MPI). We develop a formal model sufficient to represent programs that use a particular subset of MPI, and then prove a number of theorems about that model that ameliorate the state explosion problem or that show that certain properties of particular programs must necessarily hold. Most of these theorems require that the programs use neither MPI_ANY_SOURCE nor MPI_ANY_TAG. As an example, we show that for such programs, to verify freedom from deadlock, it suffices to consider only synchronous executions. While our motivation is to prove theorems that make finite-state verification techniques more tractable, the same results could also assist theorem-proving or other formal methods.

1. Introduction

Message Passing is a widely-used paradigm for the communication between processes in a parallel or distributed system. The basic ideas have been around since the late 1960s, and by the early 1990s, several different and incompatible message-passing systems were being used to develop significant applications. The desire for portability and a recognized standard led to the creation of the *Message Passing Interface* (MPI) [7, 16, 17, 22], which defines the precise syntax and semantics for a library of functions for writing message-passing programs in a language such as C or Fortran. Since that time, MPI has become widely adopted, with numerous proprietary and open-source implementations available on almost any platform.

Developers of programs that use MPI encounter many of the problems typically encountered with concurrent programs. Programs deadlock; they behave non-deterministically due to scheduling or buffering choices made by the MPI implementation; bugs are extremely difficult to pin down or even reproduce.

The most widely used method for deadling with these problems is *testing*, i.e., executing the program on a set of inputs and examining the results. While testing is certainly a key part of any software development effort, it has a number of significant drawbacks. First, it is extremely expensive. The best estimates indicate that testing represents at least 50% of the cost of developing typical commercial software, and the percentage is certainly higher for safety- or mission-critical projects where correctness is vital. Second, even with the large amount of effort invested in testing, it is usually infeasible to test more than a tiny fraction of the inputs that a program will encounter in use. Thus, testing can reveal bugs, but it cannot show that the program behaves correctly on the inputs that are not tested. Finally, the behavior

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

Version 1.55 of 2005/04/06.

of concurrent programs, including most MPI programs, typically depends on the order in which events occur in different processes. This order depends in turn on the load on the processors, the latency of the communication network, and other such factors. A concurrent program may thus behave differently on different executions with the same inputs, so getting the correct result on a test execution does not even guarantee that the program will behave correctly on another execution with the same input.

Another approach is to construct a logical theory describing the program and to use automated theorem-proving tools such as ACL2 [9,10] and PVS [18] to attempt to show that the program behaves correctly on all executions. While techniques based on this approach have achieved some notable successes, they typically require a great deal of effort and guidance by experts in mathematical logic and use the automated theorem prover as a "proof-checker" at least as much as a "proof-finder." In practice, such techniques are far too expensive to apply routinely in software development and are reserved for small components of the most safety-critical systems.

The third main approach involves building a finite model that represents all possible executions of the program and using various algorithmic methods for determining whether particular requirements for the program hold in the model. For instance, the occurrence of deadlock during an execution of the program might be represented in the model by the reachability of certain states from the start state, and algorithmic methods for exploring the states of the model could determine whether or not deadlock was possible. These methods, called *finite-state verification* (FSV) techniques, are less powerful but much more fully automated than theorem-proving approaches, and, unlike testing, can give results about all possible executions of the program. (The term "model checking" is sometimes used to refer to this class of methods, although in its original technical meaning it refers to only a subset of them.)

While FSV techniques have been applied to many domains—including various protocols, programming languages, hardware, and other message-passing systems—we have not found in the literature any specific application of them to MPI programs. (See, however, [13] for an attempt to apply FSV to a parallel system that is used in implementing MPI.)

In fact, we will restrict our attention to a small subset of MPI. The advantage of this approach is that one can optimize techniques for the specific domain. Some FSV tools, such as the model checker SPIN [8], have very general constructs for modeling a wide range of synchronous and asynchronous communication mechanisms. It is not too difficult to translate, in a straightforward way, our subset of MPI into Promela (the input language for SPIN). The problem is that, for any realistic MPI program, the state space that SPIN must explore is so large as to render the analysis infeasible. This "state-space explosion" is a familiar problem in the FSV literature, and many different approaches have been explored to combat it.

One of the primary sources of state-explosion is the use of *buffered* message passing. When messages can be buffered, as they can in MPI, the state of the system must include not only the state of each local process, but also the state of the buffer(s) holding the messages. To create a finite-state model of an MPI program, one must impose an upper bound on the number of messages in the buffers, even though the MPI Standard does not. In general it is difficult or impossible to know if the imposition of this bound is safe: if one has verified that a property holds as long as the number of buffered messages never exceeds, say, 4, how does one know that the property is not violated in an execution in which, at some point, there are 5 buffered messages? This question, which in general is very difficult to answer, is an example of the kind of question that motivates our inquiry.

In this paper, we first develop a precise, formal model for programs that use a particular subset of MPI, and then prove a number of theorems about that model that ameliorate the state-explosion problem, or that show that certain properties follow automatically or can be verified in a very simple way. Our strategy is to take advantage of the features of our subset of MPI, and to focus on specific properties or kinds of properties.

Our subset of MPI consists of

MPI_INIT, MPI_FINALIZE, MPI_COMM_SIZE, MPI_COMM_RANK,

together with the four blocking standard-mode point-to-point functions

MPI_SEND, MPI_RECV, MPI_SENDRECV, MPI_SENDRECV_REPLACE,

and the 16 collective functions

MPI_ALLGATHER,	MPI_ALLGATHERV,	MPI_ALLREDUCE,
MPI_ALLTOALL,	MPI_ALLTOALLV,	MPI_ALLTOALLW,
MPI_BARRIER,	MPI_BCAST,	MPI_EXSCAN,
MPI_GATHER,	MPI_GATHERV,	MPI_REDUCE,
MPI_REDUCE_SCATTER,	MPI_SCAN,	MPI_SCATTER,
MPI_SCATTERV.		

The semantics of these functions are discussed in §2. It will become clear in the subsequent sections that much of the complexity of the model derives from including MPI_SENDRECV and MPI_SENDRECV_REPLACE in this list. These functions allow a send and receive operation to happen concurrently; we model this by allowing the two operations to occur in either order, which adds another layer of non-determinism to the model. This requires us to develop a language in which we can talk about two paths being essentially the same if they differ only in the order in which they perform the two events within each send-receive.

In §3 we define precisely our notion of a model of an MPI program. While the model is suitable for representing the entire subset of MPI discussed above, we defer the discussion of how it can be used to represent the collective functions. In essence, our model consists of a state-machine for each process in the system, and these communicate via buffered channels that have fixed sending and receiving processes.

In §4 we give the execution semantics for the model. An execution is represented as an (interleaved) sequence of transitions in the state machines. In §§5–6 we establish some technical results that will allow us to construct new executions from old; these will be used repeatedly in the proofs of the real applications, which begin in §7 with an application to the verification of freedom from deadlock.

If the property of concern is freedom from deadlock, and the program satisfies suitable restrictions, then Theorem 7.4 answers precisely the question on buffer depth mentioned above. Theorem 7.4 states that, if there exists a deadlocking execution of the program, then there must also exist a deadlocking execution in which all communication is synchronous. This means that if we are using SPIN, for example, to verify freedom from deadlock, we may let all the channels in our model have depth 0. This confirms the intuition underlying a common practice among developers of MPI software: that a good way to check a program for deadlocks is to replace all the sends with synchronized sends, execute the resulting program, and see if that deadlocks.

The "suitable restrictions" required by Theorem 7.4 essentially amount to the requirement that the program contain no *wildcard receives*, i.e., that it uses neither MPI_ANY_SOURCE nor MPI_ANY_TAG. In fact, we give a simple counterexample to the conclusion of the theorem with a program that uses a wildcard receive. This program cannot deadlock if all communication is synchronous, but may deadlock if messages are buffered. This demonstrates that considerable complexity is added to the analysis by the use of wildcard receives. In fact, almost all of the results presented here require their absence. We hope, in future work, to develop techniques to handle the additional complexity introduced by wildcard receives.

We also prove a similar result for *partial deadlock*: this is where some subset of the processes becomes permanently blocked, though other processes may continue to execute forever. Again, we show that it suffices to check only synchronous executions.

In §8, we describe one way to obtain a safe upper bound on channel depths for checking certain properties.

In §9, we show in detail how one can represent barriers in our formal model. One of the common problems plaguing developers is the question of whether barriers are necessary at particular points in the code. Unnecessary barriers can take a huge toll on performance. Thus a technique that could tell developers whether or not the removal of a barrier could possibly lead to the violation of some correctness property would be very useful. While we certainly are not able to answer this question in general, we will take a step in that direction in §10. There, we give conditions under which the removal of all barriers from a program cannot possibly introduce deadlocks or partial deadlocks. In §11 we also show that if the program with barriers is deadlock-free, then in any execution all the channel buffers must be empty whenever all the processes are "inside" a barrier.

In §12 we sketch a way to represent the remaining collective functions using our model.

§13 deals with a particularly simple 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 state machine for any process. The only possible states in one of these state machines with multiple departing transitions are those just before a receive (or send-receive) operation, and the different transitions correspond to the various possible values that could be received.

If one begins with an MPI program with no wildcard receives and no non-deterministic functions, and if one fixes a choice of input and execution platform for the program, then one can always build a locally deterministic model of that program. In some cases, it is even possible to build a locally deterministic model of the program that is independent of the choice of input or platform.

In any case, given a locally deterministic model, we show that 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 cannot differ by much. In particular, the same messages will always be sent on each channel, and the same paths will be followed in each state machine (except possibly for the order in which the send and receive operations take place within a send-receive call). This means, in particular, that if we start with a program as described in the paragraph above, then the same values will be computed on any execution of the program. The questions concerning deadlock are also easily answered for locally deterministic models: 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.

§14 illustrates how the theory may be used by applying the theorems to two example MPI programs. A brief discussion of related work is given in §15.

2. Modeling Issues

2.1. **Memory Model.** The MPI Standard [16, §2.6] states that

> [a]n MPI program consists of autonomous processes, executing their own code, in an MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible. This document specifies the behavior of a parallel program assuming that only MPI calls are used for communication. The interaction of an MPI program with other possible means of communication (e.g., shared memory) is not specified.

We will therefore take this as our definition of an MPI program, and just assume throughout that there is no shared memory between any processes in the program. Hence the only way one process can affect another is by invoking one of the MPI functions. We will also assume throughout that each process is single-threaded.

2.2. **Communicators.** Each of the MPI functions described here takes, as one of its arguments, an MPI *communicator*. A communicator specifies a set of processes which may communicate with each other through MPI functions. Given a communicator with n processes, the processes are numbered 0 to $n-1$; this number is referred to as the *rank* of the process in the communicator.

MPI provides a pre-defined communicator, MPI_COMM_WORLD, which represents the set of all processes that exist at system initialization. For simplicity, in this paper we will consider MPI programs that use only this communicator, though in the future we hope to expand our results to deal with arbitrary communicators.

2.3. **Basic Functions.** In each process, the functions MPI_INIT must be called before that process calls any other MPI function. MPI_FINALIZE must be called once by each process, and must be the last MPI function called by that process. Since these functions do not pose any particular analytical difficulties, we will simply ignore them.

The function MPI_COMM_RANK returns the rank of the calling process in the given communicator, and MPI_COMM_SIZE returns the total number of processes in the communicator. In constructing our formal model, we do not so much ignore these two functions as treat them in a different way. In our context, an MPI program consists of a fixed set of processes, so the rank of a process, as well as the total number of processes, may be considered constants that are "known" to each process from the beginning.

2.4. **Send and Receive.** The semantics of the point-to-point communication functions are described in [16, Chapter 3]. We summarize here the facts that we will need.

MPI provides many ways to send a message from one process to another. Perhaps the simplest is the *standard mode, blocking send* function, which has the form

MPI_SEND(buf, count, datatype, dest, tag, comm).

Here buf is the address of the first element in the sequence of data to be sent; count is the number of elements in the sequence, datatype indicates the type of each element in the sequence; dest is the rank of the process to which the data is to be sent; tag is an integer that may be used to distinguish the message; and comm is a handle representing the communicator in which this communication is to take place.

One may think of the message data as being bundled up inside an "envelope." The envelope includes the rank of the sending process (the *source*), the rank of the receiving process (the *destination*), the tag, and the communicator.

Likewise, there are different functions to receive a message that has been sent; the simplest is probably the *blocking receive*, which has the form

MPI_RECV(buf, count, datatype, source, tag, comm, status).

Here, buf is the address for the beginning of the segment of memory into which the incoming message is to be stored. The integer count specifies an upper bound on the number of elements of type datatype to be received. (At runtime, an overflow error will be generated if the length of the message received is longer than count.) The integer source is the rank of the sending process and the integer tag is the tag identifier. However, unlike the case of MPI_SEND, these last two parameters may take the *wildcard* values MPI_ANY_SOURCE and MPI_ANY_TAG, respectively. The parameter comm represents the communicator, while status is an "out" parameter used by the function to return information about the message that has been received.

A receive operation will only select a message for reception if the message envelope *matches* the receive parameters in the following sense: of course, the destination of the message must equal the rank of the process invoking the receive, the source of the message must match the value of source (i.e., either the two integers are equal, or source is MPI_ANY_SOURCE), the tag of the message must match the value of tag (either the two integers are equal, or tag is MPI_ANY_TAG), and the communicator of the message must equal comm. It is up to the user to make sure that the data-type of the message sent matches the data-type specified in the receive. (In fact, the message tag is often used to facilitate this.)

2.5. **Buffering and Blocking.** When a process executes a send operation, the MPI implementation may buffer the outgoing message. As soon as the message is buffered, the send operation completes and the sending process may proceed, even though the receiving process may not have received the message or even initiated a receive operation. Between the time the message is sent, and the time it is received, the message is thought of as existing in a *system buffer*; we say that such a message is *pending*. On the other hand, assuming the receiving process initiates a matching receive, the message might be copied directly from the send buffer into the receive buffer; in this case we say that the communication occurs *synchronously*.

Once a send operation has been initiated, the MPI implementation may choose to block the sending process for an undetermined time. For example, it might block the sender until sufficient space becomes available in the system buffer, or until the communication can take place synchronously. Of course, the MPI Standard places limits on how long an MPI implementation may block the sender. We have already seen that if the implementation chooses to buffer the message, the sender must be allowed to proceed as soon as the message is completely copied into the system buffer. The MPI Standard also states that once both (1) the receiving process is at a matching receive, and (2) there is no pending message in the system buffer that also matches that receive, the implementation must allow the communication to take place synchronously, and the sender (as well as the receiver) to proceed. If the implementation chooses to block the sender until these conditions are met, we say that it *forces the send to synchronize*. The reason for condition (2) is that, if there is already a pending matching message, the implementation might choose to receive that one. (In fact, if both messages originate from the same source, the rules concerning order dictate that the implementation select the message that was sent first.)

A process that has initiated a receive operation must also block until there is either a pending matching message, or another process initiates a matching send that can take place synchronously. As soon as either of these conditions becomes true, the message is copied into the receive buffer, the receive completes, and then the receiving process is allowed to proceed. Unlike the case for sends, there are no choices available to the MPI implementation.

2.6. **Order.** The MPI Standard [16, §3.5] imposes certain restrictions concerning the order in which messages may be received:

> Messages are non-overtaking: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives.

These are the only guarantees made concerning order. So, for example, if two different processes send messages to the same receiver, the messages may be received in the order opposite that in which they were sent.

The order restrictions suggest natural ways to model process communication. For example, for each triple

$$(\mathsf{sender}, \mathsf{receiver}, \mathsf{tag})$$

we may create a message channel that behaves as a queue. (Here we are using our assumption that the only communicator used is MPI_COMM_WORLD.) A send simply places the message into the appropriate queue. The modeling of a receive is a bit more complicated because of the non-determinism introduced by the wildcards. Without wildcards, a receive simply removes a message from the queue specified by the receive parameters. If the receive uses only the wildcard MPI_ANY_SOURCE, then it is free to choose non-deterministically among the queues for the various senders. However, if it uses the wildcard MPI_ANY_TAG, the situation is more complicated, since the MPI Standard prescribes that for a fixed sender, the receive choose the oldest matching message from among those with different tags. Hence if we allow the use of MPI_ANY_TAG, the queues alone may not contain enough information to model the state of the system buffer precisely.

One way to get around this would be to instead create a channel for each pair (sender, receiver). Now a receive must be able to pull out the oldest item in the channel with a matching tag. While this is certainly possible, it does complicate the modeling of the state enormously, since the channels would no longer be FIFO. For this reason, we have chosen—in this paper—to ignore the possibility of MPI_ANY_TAG. In fact, while our model can deal precisely with MPI_ANY_SOURCE, almost all of the results presented here require that one does not use either wildcard.

2.7. **Variations.** The send and receive operations described above are called *blocking* for the following reason: after the return of a call to MPI_SEND, one is guaranteed that all the data have been copied out of the send buffer—either to the system buffer or directly to the receiving process. Therefore it is safe to reuse the memory in the send buffer, as this can no longer affect the message in any way. Likewise, upon the return of an invocation of MPI_RECV, one is guaranteed that the incoming message has been completely copied into the receive buffer. Therefore it is safe to immediately read the message from that buffer. Hence MPI_SEND "blocks" until the system has completed copying the message out of the send buffer, and MPI_RECV "blocks" until the system has completed copying the message into the receive buffer.

MPI also provides *non-blocking* send and receive operations. A non-blocking send may return immediately—even before the message has been completely copied out of the send buffer. A non-blocking send operation returns a "request handle"; a second function (MPI_WAIT) is invoked on the handle, and this will block until the system has completed copying the message out of the buffer. By decomposing the send operation in this way, one may achieve a greater overlap between computation and communication, by placing computational code between the send and the wait. The non-blocking receive is similar.

In addition, both the blocking and non-blocking sends come in several *modes*. We have described the *standard* mode, but there are also the *synchronous*, *buffered*, and *ready* modes. The synchronous mode forces the send to block until it can execute synchronously with a matching receive. The buffered mode always chooses to buffer the outgoing message and makes use of a user-specified segment of a process' local memory for this purpose. The ready mode can be used as long as a matching receive will be invoked before the send is (else a runtime error is generated). In this paper we are concerned only with the blocking operations and the standard mode send; we hope to extend these results to the more general non-blocking operations and the other send modes in future work.

2.8. **Send-Receive.** One often wishes to have two processes exchange data: process 0 sends a message to process 1 and receives from process 1, while process 1 sends to 0 and receives from 0. More generally, one may wish to have a set of processes exchange data in a cyclical way: process 0 sends to 1 and receives from n, 1 sends to 2 and receives from 0, …, n receives from $n-1$ and sends to 0. In both cases, each process must execute one send and one receive. One must be careful, however: if this is coded so that each process first sends, then receives, the program will deadlock if the MPI implementation chooses to synchronize all the sends. While there are ways to get around this, the situation occurs frequently enough that MPI provides a special function that executes a send and a receive in one invocation, without specifying the order in which the send and receive are to occur. The semantics are defined so that execution of this function is equivalent to having the process fork off two independent threads—one executing the send, the other the receive. The function returns when both have completed.

This function has the form

```
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag,
              recvbuf, recvcount, recvtype, source, recvtag, comm, status)
```

The first 5 parameters are the usual ones for send, the next 5 together with status are the usual ones for receive, and comm specifies the communicator for both the send and receive.

2.9. **Collective Operations.** The collective functions are those that involve all processes in a communicator. The simplest is MPI_BARRIER(comm), which is used to force all processes in the communicator comm to synchronize at a certain point. Upon invocation, we say that a process has *entered* the barrier. That process is then blocked until all the processes (in the communicator) are "inside" the barrier (i.e., have invoked but not yet returned from a call to the barrier function). Once this happens, the function may return, and we say that the process *exits* the barrier.

The other collective functions can be used to manipulate and transfer data in quite complex ways. We give a brief overview here; for details, see [22], [16], and [17].

Some of the collective functions, such as the broadcast function MPI_BCAST, require a root argument. In this case, that is the rank of the process which is broadcasting the message. All processes should invoke MPI_BCAST with the same value for root, but the interpretation is different for the process whose rank is root: for this process, the message from the specified buffer is sent to all non-root processes, while for the non-root processes the message is received and placed in the specified buffer. MPI_GATHER also takes a root, and is in a sense dual to MPI_BCAST: in this case the root process receives data from all the non-root processes and stores ("gathers") the data into distinct sections of the specified buffer.

MPI_SCATTER is used to send a different slice of the data on the root process to each of the non-root processes. MPI_ALLGATHER is like gather, but without a root: each process receives a copy of the data from every other process. MPI_ALLTOALL is another rootless function which combines scattering and gathering by having every process send a different slice of data to every process.

The *reduction* operations take an additional argument which specifies a function, such as "sum", "max", "min," etc., which is to be applied component-wise to the data. MPI_REDUCE applies the reduction function across all processes and stores the result in the buffer of the root process; MPI_ALLREDUCE does not take a root and instead the result is given to all processes. MPI_REDUCE_SCATTER combines reduction and scattering by effectively performing the reduction and then sending a different slice of the result to every process. MPI_SCAN performs a reduction operation but returns to process i the partial result obtained by applying the operation to processes 0 through i (inclusive). MPI_EXSCAN is similar except it returns to process i the result obtained by applying the operation to processes 0 through $i-1$. Finally, there are variations of some of these functions (the names that end in "V" or "W") which provide greater control by allowing the user, for example, to specify different lengths for the data slices that will be sent to the different processes, rather than using a single uniform length.

Unlike the point-to-point functions, there are no "non-blocking" versions of the collective functions. Nor are there different modes for the collective functions. (Or, to put it another way, there is only one mode, which corresponds conceptually to the standard mode for point-to-point functions.) Nor is there a tag argument. In addition, the MPI Standard guarantees that all communication carried out by collective functions be kept entirely distinct from those carried out by point-to-point functions. So there is never a danger, for example, of an MPI_RECV function receiving a message sent out by an MPI_BCAST call.

3. Models

In this section, we describe precisely what we mean by a *model of an MPI program.* We start with the formal mathematical definitions, and then explain (in §3.3) how these capture the semantics of the actual functions in a real MPI program.

3.1. **Context.**

First, we describe what we call a *context.* This is a tuple

$$\mathcal{C} = (\mathsf{Proc}, \mathsf{Chan}, \mathsf{sender}, \mathsf{receiver}, \mathsf{msg}, \mathsf{loc}, \mathsf{com}).$$

We describe each of these components in turn.

First, Proc is a finite set. It represents the set of *processes* in the MPI program.

Second, Chan is also a finite set, representing the set of communication *channels.* Next, sender and $\mathsf{receiver}$ are functions from Chan to Proc. The idea is that each channel is used exclusively to transfer messages from its sender process to its receiver process. (Recall from §2 that, in creating a model from code, we intend to make one channel for each triple $(\mathsf{sender}, \mathsf{receiver}, \mathsf{tag})$. It is because of the tags that we allow the possibility of having more than one channel from one process to another.)

Next, msg is a function that assigns, to each $c \in \mathsf{Chan}$, a (possibly infinite) nonempty set $\mathsf{msg}(c)$, representing the set of messages that can be sent over channel c.

Now loc is a function that assigns, to each $p \in \mathsf{Proc}$, a (possibly infinite) set $\mathsf{loc}(p)$, representing the set of *local events* for process p. We require that

$$p \neq q \Rightarrow \mathsf{loc}(p) \cap \mathsf{loc}(q) = \emptyset.$$

Finally, com is the function that assigns, to each $p \in \mathsf{Proc}$, the set consisting of all triples (c, send, x), where $c \in \mathsf{Chan}$ and $\mathsf{sender}(c) = p$, and $x \in \mathsf{msg}(c)$, together with all triples $(d, \mathsf{receive}, y)$, where $d \in \mathsf{Chan}$ and $\mathsf{receiver}(d) = p$, and $y \in \mathsf{msg}(d)$. This is the set of all *communication events* for p. We abbreviate (c, send, x) as $c!x$ and $(d, \mathsf{receive}, y)$ as $d?y$. The first represents the event in which process p inserts message x into channel c, and the second represents the event in which p removes message y from d.

We also require that $\mathsf{loc}(p) \cap \mathsf{com}(q) = \emptyset$ for all $p, q \in \mathsf{Proc}$, and we define

$$\mathsf{event}(p) = \mathsf{loc}(p) \cup \mathsf{com}(p).$$

That completes the definition of a *context* $\mathcal{C}$.

3.2. **MPI State Machines.** Suppose we are given a context $\mathcal{C}$ as above and a $p \in \mathsf{Proc}$. We will define what we mean by an *MPI state machine for* p *under* $\mathcal{C}$. This is a tuple

$$M = (\mathsf{States}, \mathsf{Trans}, \mathsf{src}, \mathsf{des}, \mathsf{label}, \mathsf{start}, \mathsf{End}).$$

We define the components of M in order.

First, States is a (possibly infinite) set. It represents the set of possible *states* in which process p may be during execution.

Second, Trans is a (possibly infinite) set, representing the set of possible transitions from one state of p to another during execution. Now, src and des are just functions from Trans to States. These are interpreted as giving the *source state* and *destination state*, respectively, for a transition.

Next, label is a function from Trans to $\mathsf{event}(p)$.

Finally, start is an element of States, and End is a (possibly empty) subset of States. The former represents the initial state of process p, and elements of the latter represent the state of p after process termination. We allow the possibility that $\mathsf{start} \in \mathsf{End}$. Furthermore, we require that there is no $t \in \mathsf{Trans}$ with $\mathsf{src}(t) \in \mathsf{End}$.

We are not quite finished with the definition of *MPI state machine*. We also require that for each $u \in \mathsf{States}$, exactly one of the following hold:

(i) The state $u \in \mathsf{End}$. We call this a *final state*.

(ii) The state $u \notin \mathsf{End}$, there is at least one $t \in \mathsf{Trans}$ with $\mathsf{src}(t) = u$, and, for all such t, $\mathsf{label}(t) \in \mathsf{loc}(p)$. In this case we say u is a *local-event state*.

(iii) There is exactly one $t \in \mathsf{Trans}$ such that $\mathsf{src}(t) = u$, and $\mathsf{label}(t)$ is a communication event of the form $c!x$. We say u is a *sending state*.

(iv) There is a nonempty subset R of Chan with $\mathsf{receiver}(d) = p$ for all $d \in R$, such that the restriction of label to the set of transitions departing from u is a 1-1 correspondence onto the set of events of the form $d?y$, where $d \in R$ and $y \in \mathsf{msg}(d)$. We say u is a *receiving state*.

(v) (See Figure 1.) There is a $c \in \mathsf{Chan}$, a nonempty subset R of Chan, an $x \in \mathsf{msg}(c)$, a state u', and states $v(d, y)$, and $v'(d, y)$, for all pairs (d, y) with $d \in R$ and $y \in \mathsf{msg}(d)$, such that the following all hold:
   - We have $\mathsf{sender}(c) = p$ and, for all $d \in R$, $\mathsf{receiver}(d) = p$.
   - The states u, u', and the $v(d, y)$ and $v'(d, y)$ are all distinct.
   - The set of transitions departing from u consists of one transition to u' whose label is $c!x$, and, for each (d, y), one transition labeled $d?y$ to $v(d, y)$. Furthermore, these are the only transitions terminating in u' or $v(d, y)$.
   - For each (d, y), there is precisely one transition departing from $v(d, y)$, it is labeled $c!x$, and it terminates in $v'(d, y)$.
   - For each (d, y), there is a transition from u' to $v'(d, y)$, it is labeled $d?y$, and these make up all the transitions departing from u'.
   - For each (d, y), the only transitions terminating in $v'(d, y)$ are the two already mentioned: one from u' and one from $v(d, y)$.

   We call u a *send-receive state*. Notice that u' is a receiving state, and each $v(d, y)$ is a sending state.

This completes the definition of an *MPI state machine for* p *under* $\mathcal{C}$.

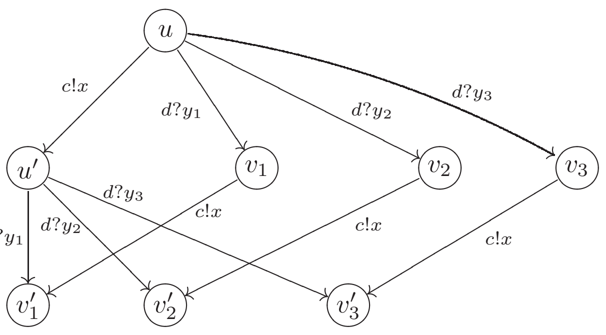


FIGURE 1. A send-receive state u with one receiving channel d.

Finally, a *model of an MPI program* is a pair $\mathcal{M} = (\mathcal{C}, M)$, where $\mathcal{C}$ is a context and M is a function that assigns, to each $p \in \mathsf{Proc}$, an MPI state machine

$$M_p = (\mathsf{States}_p, \mathsf{Trans}_p, \mathsf{src}_p, \mathsf{des}_p, \mathsf{label}_p, \mathsf{start}_p, \mathsf{End}_p)$$

for p under $\mathcal{C}$, such that

$$\mathsf{States}_p \cap \mathsf{States}_q = \emptyset = \mathsf{Trans}_p \cap \mathsf{Trans}_q$$

for $p \neq q$. We will leave off the subscript "p" for the maps src_p, des_p, and label_p, since each transition is in precisely one M_p so there is no need to specify p.

A *global state* of $\mathcal{M}$ is a map U that assigns to each $p \in \mathsf{Proc}$ an element $U_p \in \mathsf{States}_p$.

We say that $\mathcal{M}$ *has no wildcard receives* if, in each M_p, the sets R for the receiving or send-receive states all have cardinality one.

3.3. **Building Models From Code.** Let us briefly describe how our model relates to actual program code. The idea of representing an ordinary sequential program as a finite state machine goes back to the dawn of computer science. The basic idea is that one creates a state for every possible combination of values of all variables, including the program counter, which represents a position in the program code. If there are functions then the call stack must also be included in the state, and if there is a heap then this may also have to be represented. On the other hand, by means of abstraction, one may include less information in the state and therefore reduce the total number of states in the model.

In creating a model from code, one usually wishes to employ abstraction in such a way that the resulting model is still *conservative*. This means that every execution of the actual program is represented by an execution of the model, though there may be additional executions of the model that do not correspond to actual executions of the program. If one proves that a property holds on every execution in a conservative model, then one is guaranteed that the property holds on every execution of the program. On the other hand, if an execution of the model is found that violates the property, there is the possibility that that execution is *spurious*, i.e., it does not correspond to an actual execution of the program.

A state in an MPI state machine M_p represents the local state of process p—the values of its variables and program counter, etc. A local 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 transitions do not play a significant role in this paper. One could, for example, just use a single label for all the local transitions in a process. On the other hand, if one wishes to reason about particular local transitions in a correctness property, one could use different local labels for those transitions so that they could be referenced in the property specification.

There are several reasons why we allow *local non-determinism* in our models, i.e., a local-event state with more than one departing transition. One of these is the result of abstraction. Suppose, for example, that we have chosen to abstract away a variable `x` from the state, i.e., the local state no longer contains a component for the value of `x`. Suppose now that there is a local state s, which represents a point in the code just before a statement of the following form:

```
if (x == 0) then {...} else {...}.
```

If we want our model to be conservative then we must allow two transitions to depart from s—one for each branch in this conditional statement—since this local-event state does not "know" the value of `x`.

Other cases in which we might have multiple local transitions departing from a state include the case where we wish the model to simultaneously represent the same program operating on various inputs, and the case where the program code uses an actual non-deterministic function, such as a function that returns a random value.

All of this is standard, and so we concentrate now on what is particular to the representation of MPI communication in our model.

As we have already pointed out, we create one channel in the model for each triple $(\mathsf{sender}, \mathsf{receiver}, \mathsf{tag})$, where sender and $\mathsf{receiver}$ are any processes for which the former might possibly send a message to the latter, and tag is any tag that might be used in such a communication. The order restrictions imposed by the MPI Standard imply that this channel should behave like a queue: the messages enter and exit it in first-in, first-out order.

The sets $\mathsf{msg}(c)$ are likely candidates for abstraction. A typical MPI program might send blocks of 100 IEEE 754 single precision floating point numbers across c. In this case we could, in theory, use no abstraction at all, and take $\mathsf{msg}(c)$ to be the set of all vectors of length 100 of floating point numbers. We would then have a fully precise representation of the messages being sent along c. While it might be useful to reason theoretically about such a model, the size of the state space would make the model intractable for automated finite-state verification. On the other end of the abstraction spectrum, we could take $\mathsf{msg}(c)$ to contain a single element, say $\mathsf{msg}(c) = \{1\}$. If the property we wish to verify cannot be influenced by the actual data in the messages, this may be a perfectly reasonable abstraction. Of course, there are many choices between these two extremes, and finding appropriate abstractions for the program and property of interest is part of the art of model creation.

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 information 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 value received will of course change the state of this process—the variable or buffer into which the data is stored will take on a new value or values. Hence a transition to a different state must be included for every possible value that could be received. If this is a wildcard receive (because $\mathsf{MPI\_ANY\_SOURCE}$ is used as the source parameter in the code), then we must also allow a different set of transitions for each of the possible senders.

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.

4. Execution Semantics

We will represent 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.

4.1. **Basic Definitions.**

Let X be a set. A sequence $S = (x_1, x_2, \ldots)$ of elements of X may be either infinite or finite. We write $|S|$ for the length of S; we allow $|S| = \infty$ if S is infinite and $|S| = 0$ if S is empty. By the *domain* of S, we mean the set of all integers that are greater than or equal to 1 and less than or equal to $|S|$. For i in the domain of S, define $S|_i = x_i$.

If S and T are sequences, and S is a prefix of T, we write $S \subseteq T$; this allows the possibility that $S = T$. If S is a proper prefix of T, we write $S \subset T$.

We also say T *is an extension of* S if $S \subseteq T$, and T *is a proper extension of* S if $S \subset T$.

If $S \subseteq T$ then we define a sequence $T \setminus S$ as follows: if $S = T$ then we let $T \setminus S$ be the empty sequence. Otherwise, S must be finite, and if $T = (x_1, x_2, \ldots)$, we let

$$T \setminus S = (x_{|S|+1}, x_{|S|+2}, \ldots).$$

If A is a subset of a set B, and S is a sequence of elements of B, then *the projection of* S *onto* A is the sequence that results by deleting from S all elements that are not in A.

If S is any sequence and n is a non-negative integer, then S^n denotes the sequence obtained by truncating S after the n^{th} element. In other words, if $|S| \leq n$, then $S^n = S$, otherwise, S^n consists of the first n elements of S.

We now define the execution semantics of a model of an MPI program. Fix a model $\mathcal{M} = (\mathcal{C}, M)$ as in §3.

Let $S = (t_1, t_2, \ldots)$ be a (finite or infinite) sequence of transitions (i.e., the elements of S are in $\bigcup_{p \in \mathsf{Proc}} \mathsf{Trans}_p$), and let $c \in \mathsf{Chan}$. Let $(c!x_1, c!x_2, \ldots)$ denote the projection of

$$(\mathsf{label}(t_1), \mathsf{label}(t_2), \ldots)$$

onto the set of events that are sends on c. Then define

$$\mathsf{Sent}_c(S) = (x_1, x_2, \ldots).$$

This is the sequence of messages that are sent on c in S. The sequence $\mathsf{Received}_c(S)$ is defined similarly as the sequence of messages that are received on c in S.

We say that S is *c-legal* if for all n in the domain of S,

$$\mathsf{Received}_c(S^n) \subseteq \mathsf{Sent}_c(S^n).$$

This is exactly what it means to say that the channel c behaves like a queue. If S is c-legal, we define

$$\mathsf{Pending}_c(S) = \mathsf{Sent}_c(S) \setminus \mathsf{Received}_c(S).$$

This represents the messages remaining in the queue after the last step of execution in S.

Suppose next that we are given a sequence $\pi = (t_1, t_2, \ldots)$ in Trans_p for some $p \in \mathsf{Proc}$. We say that π is a *path through M_p* if π is empty, or if $\mathsf{src}(t_1) = \mathsf{start}_p$, and $\mathsf{des}(t_i) = \mathsf{src}(t_{i+1})$ for all $i \geq 1$ for which $i + 1$ is in the domain of π.

Given any sequence S of transitions, and $p \in \mathsf{Proc}$, we let $S{\downarrow_p}$ denote the projection of S onto Trans_p. Now we may finally define precisely the notion of *execution prefix*:

Definition 4.1. An *execution prefix* of $\mathcal{M}$ is a sequence S of transitions such that (i) for each $p \in \mathsf{Proc}$, $S{\downarrow_p}$ is a path through M_p, and (ii) for each $c \in \mathsf{Chan}$, S is c-legal.

If $\pi = (t_1, \ldots, t_n)$ is a finite path through M_p, we let

$$\mathsf{terminus}(\pi) = \begin{cases} \mathsf{des}(t_n) & \text{if } n \geq 1 \\ \mathsf{start}_p & \text{otherwise.} \end{cases}$$

Definition 4.2. Let S be a finite execution prefix of $\mathcal{M}$. The *terminal state of S* is the global state $\mathsf{terminus}(S)$ defined by $\mathsf{terminus}(S)_p = \mathsf{terminus}(S{\downarrow_p})$.

4.2. **Synchronous Executions.**

Definition 4.3. Let $S = (t_1, t_2, \ldots)$ be an execution prefix and $c \in \mathsf{Chan}$. We say that S is *c-synchronous* if, for all i in the domain of S for which $\mathsf{label}(t_i)$ is a send, say $c!x$, the following all hold:

- $i + 1$ is in the domain of S.
- $\mathsf{label}(t_{i+1}) = c?x$.
- $\mathsf{sender}(c) \neq \mathsf{receiver}(c)$ or $\mathsf{src}(t_i)$ is a send-receive state.

We say that S is *synchronous* if S is c-synchronous for all $c \in \mathsf{Chan}$.

The idea is the following: we are representing executions of a model of an MPI program as sequences of transitions. For simplicity, our representation does 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. In general, in the latter case, the send and receive *could* have happened synchronously. The exception to this rule is the somewhat pathological case in which a process sends a message to itself and then receives the message. If this is done with an $\mathsf{MPI\_SEND}$ followed by an $\mathsf{MPI\_RECV}$, then that send can never happen synchronously, as it is impossible for the process to be at two positions at once. However, if it is done with an $\mathsf{MPI\_SENDRECV}$, it may happen synchronously since the send and receive are thought of as taking place in two independent processes. This is the reason for the third item in the list above.

4.3. **SRC Equivalence and Associated Synchronous Prefixes.** Let $\mathcal{M}$ be a model of an MPI program, $p \in \mathsf{Proc}$, and $\pi = (t_1, t_2, \ldots)$ a path through M_p.

Suppose that k and $k+1$ are in the domain of π and that $u = \mathsf{src}(t_k)$ is a send-receive state. Then one of t_k, t_{k+1} is labeled by a send and the other by a receive. Assume also that the sending and receiving channels are distinct. (If the sending and receiving channels are the same, then we are in the case where a process is sending a message to itself using the same sending and receiving tag.) We define the *dual path* $\bar{t}_{k+1}, \bar{t}_k$ by

$$\begin{aligned} u &= \mathsf{src}(\bar{t}_{k+1}) \\ \mathsf{des}(\bar{t}_{k+1}) &= \mathsf{src}(\bar{t}_k) \\ \mathsf{label}(\bar{t}_i) &= \mathsf{label}(t_i) \quad (i \in \{k, k+1\}). \end{aligned}$$

This is just the path around the send-receive diamond that performs the same send and receive, but in the opposite order. Now let π' be the sequence obtained from π by replacing the subsequence (t_k, t_{k+1}) with the dual path $(\bar{t}_{k+1}, \bar{t}_k)$. Then π' is also a path, since the two subsequences start at the same state, and end at the same state.

Definition 4.4. Let π and ρ be paths through M_p. If ρ can be obtained from π by applying a (possibly infinite) set of transformations such as the one above, we say that π and ρ are *equivalent up to send-receive commutation*, or *src-equivalent*, for short, and we write $\pi \sim \rho$.

Clearly, src-equivalence is an equivalence relation on the set of paths through M_p. It is also clear that, if $\pi \sim \rho$, then $\mathsf{terminus}(\pi) = \mathsf{terminus}(\rho)$.

Lemma 4.5. *Let S and T be execution prefixes for $\mathcal{M}$. Let $c \in \mathsf{Chan}$, $p = \mathsf{sender}(c)$, and $q = \mathsf{receiver}(c)$. Suppose $S{\downarrow_p} \sim T{\downarrow_p}$ and $S{\downarrow_q} \sim T{\downarrow_q}$. Then $\mathsf{Sent}_c(S) = \mathsf{Sent}_c(T)$, $\mathsf{Received}_c(S) = \mathsf{Received}_c(T)$, and $\mathsf{Pending}_c(S) = \mathsf{Pending}_c(T)$.*

Proof. The definition of src-equivalence does not allow one to transpose a send and receive that use the same channel. Hence for any fixed channel c, the transformations can not change the projections of the event-sequences onto c. $\square$

We now define some relations on the set of paths through M_p.

Definition 4.6. If π and ρ are paths through M_p, we say $\pi \preceq \rho$ if π is a prefix of a sequence that is src-equivalent to ρ. We write $\pi \prec \rho$ if $\pi \preceq \rho$ but $\pi \not\sim \rho$.

The following is easily verified:

Lemma 4.7. *Let π, π', ρ, ρ', and σ be paths through M_p. Then*

(1) *If $\pi \sim \pi'$ and $\rho \sim \rho'$ then $\pi \preceq \rho \iff \pi' \preceq \rho'$.*
(2) *$\pi \preceq \pi$.*
(3) *If $\pi \preceq \rho$ and $\rho \preceq \sigma$ then $\pi \preceq \sigma$.*
(4) *If $\pi \preceq \rho$ and $\rho \preceq \pi$ then $\pi \sim \rho$.*

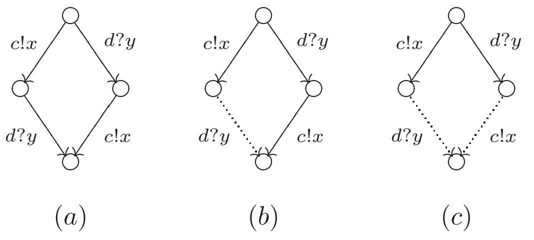


Figure 2. Compatible paths from a send-receive state. In each case, ρ, the path on the left (excluding the dotted arrows), is compatible with σ, the path on the right. In (a), $\rho \sim \sigma$. In (b), $\rho \prec \sigma$. In (c), ρ and σ are non-comparable compatible paths. In all cases, π may be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

In other words, $\preceq$ induces a partial order on the set of src-equivalence classes of paths through M_p.

Definition 4.8. If π and ρ are paths through M_p, we say that π and ρ are *comparable* if $\pi \preceq \rho$ or $\rho \preceq \pi$.

Suppose S is any finite execution prefix of $\mathcal{M}$. Consider the set consisting of all synchronous execution prefixes S' with $S'{\downarrow_p} \preceq S{\downarrow_p}$ for all $p \in \mathsf{Proc}$. This set is finite, and it is not empty, as it contains the empty sequence as a member. Let T be an element of this set of maximal length. We say that T is an *associated synchronous prefix* for S.

5. Compatible Prefixes

Let $\mathcal{M}$ be a model of an MPI program.

Definition 5.1. 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:

Lemma 5.2. *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) $\mathsf{terminus}(\rho^{n-1}) = \mathsf{terminus}(\sigma^{n-1})$ *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.

Lemma 5.3. *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.*

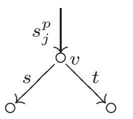
Proof. Let $s = s_n$. Say $s \in \mathsf{Trans}_p$. Clearly $T{\downarrow_r}$ and $S{\downarrow_r}$ are compatible for all $r \neq p$. Write

$$S{\downarrow_p} = (s^p_1, \ldots, s^p_j, s)$$
$$T{\downarrow_p} = (t^p_1, t^p_2, \ldots).$$

Let $\sigma = (s^p_1, \ldots, s^p_j)$.

By hypothesis, σ is compatible with $T{\downarrow_p}$. If $T{\downarrow_p} \preceq \sigma$ then $T{\downarrow_p} \preceq \sigma \preceq S{\downarrow_p}$ and so $T{\downarrow_p}$ and $S{\downarrow_p}$ are compatible and we are done. So there remain two possibilities to consider: (a) $\sigma \prec T{\downarrow_p}$, and (b) σ and $T{\downarrow_p}$ are non-comparable compatible paths.

Consider first case (a). Then there is a proper extension τ of σ that is src-equivalent to $T{\downarrow_p}$. Let $t = \tau|_{j+1}$ and $v = \mathsf{src}(t)$. Then $v = \mathsf{src}(s)$ as well, since if $j = 0$ then both s and t must depart from the start state, while if $j > 0$ then either of these transitions may follow s^p_j. It suffices to show that $S{\downarrow_p}$ and τ are compatible. As this is certainly the case if $s = t$, we will assume $s \neq t$, and we have the following configuration:

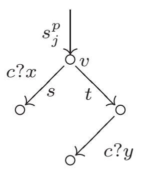


Suppose now that s and t are both receives. Since there are no wildcard receives, s and t must be receives on the same channel c; say $\mathsf{label}(s) = c?x$ and $\mathsf{label}(t) = c?y$, where $c \in \mathsf{Chan}$ and $x, y \in \mathsf{msg}(c)$. Say that s is the i^{th} receive on c in S. Then t is also the i^{th} receive on c in τ. Hence

$$y = \mathsf{Received}_c(\tau)|_i = \mathsf{Received}_c(T)|_i = \mathsf{Sent}_c(T)|_i,$$

Similarly, $x = \mathsf{Sent}_c(S^{n-1})|_i$. However, $S^{n-1}{\downarrow_q}$ and $T{\downarrow_q}$ are compatible, where $q = \mathsf{sender}(c)$. So $\mathsf{Sent}_c(S^{n-1})$ and $\mathsf{Sent}_c(T)$ are prefixes of a common sequence. This means $x = y$, and so $s = t$, a contradiction.

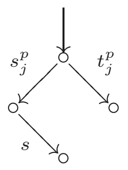
Suppose instead that v is a send-receive state and that s is a receive and t a send. If $|\tau| = j + 1$ then $S{\downarrow_p}$ and τ are non-comparable compatible paths, and we are done. So suppose $|\tau| \geq j + 2$. Then $\mathsf{label}(\tau|_{j+2}) = c?y$ and $\mathsf{label}(s) = c?x$ for some $c \in \mathsf{Chan}$ and $x, y \in \mathsf{msg}(c)$:



If s is the i^{th} receive on c in S then $\tau|_{j+2}$ is the i^{th} receive on c in τ, and, arguing as in the paragraph above, we arrive at $x = y$. This means $S{\downarrow_p} \preceq \tau$ and hence that $S{\downarrow_p}$ and τ are compatible.

Suppose instead that s is a send and t a receive. If $|\tau| = j + 1$ then $S{\downarrow_p}$ and τ are non-comparable compatible paths and we are done. If not, then $\tau|_{j+2}$ is labeled $c!x$ and so $S{\downarrow_p} \preceq \tau$ and again we are done.

We now turn to case (b). By Lemma 5.2, $\mathsf{src}(s_j^p) = \mathsf{src}(t_j^p)$ is a send-receive state and $|T{\downarrow_p}| = j$:



Suppose first that s_j^p is the send and t_j^p the receive. Then we must have $\mathsf{label}(s) = c?x$ and $\mathsf{label}(t_j^p) = c?y$ for some $c \in \mathsf{Chan}$ and $x, y \in \mathsf{msg}(c)$. Arguing as before, we have $x = y$, whence $T{\downarrow_p} \preceq S{\downarrow_p}$. If instead s_j^p is the receive and t_j^p is the send then $\mathsf{label}(s) = \mathsf{label}(t_j^p)$ and $T{\downarrow_p} \preceq S{\downarrow_p}$ in this case as well. In either case, we have shown that $T{\downarrow_p}$ and $S{\downarrow_p}$ are compatible, completing the proof. $\square$

6. Universally Permitted Extensions

Definition 6.1. Let $S = (s_1, \ldots, s_m)$ be a finite execution prefix for $\mathcal{M}$. A finite execution prefix $T = (s_1, \ldots, s_m, \ldots, s_n)$ extending S is said to be *universally permitted* if for all i such that $m < i \leq n$ and s_i is a send, say $\mathsf{label}(s_i) = c!x$, then $\mathsf{Pending}_c(T^{i-1})$ is empty and $\mathsf{label}(s_{i+1}) = c?x$.

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.

Note that S is a universally permitted extension of itself. Notice also that if S is synchronous, then a universally permitted extension of S is the same thing as a synchronous extension of S.

We also 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.

Proposition 6.2. *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

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

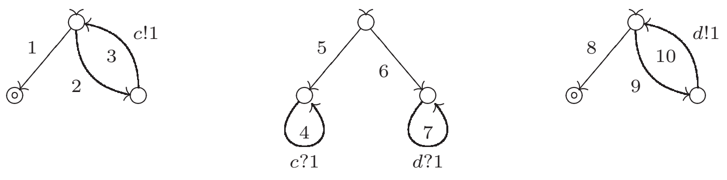


Figure 3. A Model of an MPI Program with 3 Processes. Edges with no label are local. Process 0 (left) chooses to either terminate, or to move to a state from which it will send a message to process 1 on channel c, then return to its start state and repeat. Process 2 (right) does the same for channel 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.

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 local transition 5, while the projection of T onto that process begins with local 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 6.2.

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 6.2 will be 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 5.2 that if $|\sigma| > |\rho|$ we must have $\rho \prec \sigma$, while if $|\sigma| = |\rho|$, ρ and σ are src-equivalent except

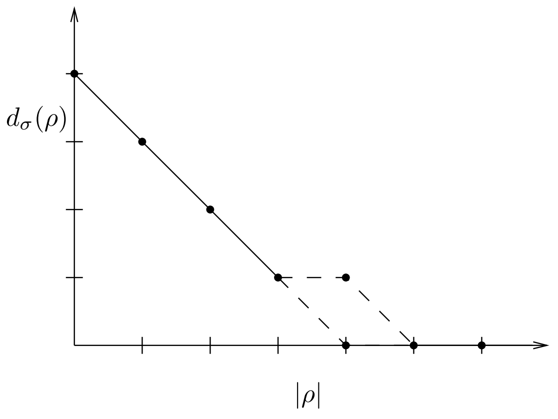


FIGURE 4. $d_\sigma(\rho)$ as a function of $|\rho|$

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 $d_\sigma(\rho) \geq d_\sigma(\rho')$.

We now turn to the proof of Proposition 6.2. 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 $s_{n+1} \in \mathsf{Trans}_p$. By definition, σ and $T{\downarrow_p}$ are compatible. Moreover, $n \geq |S'|$ since S' is compatible with T. Finally, by Lemma 5.3, s_{n+1} must be a local 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 5.2 would imply that $\mathsf{terminus}(\sigma)$ is either a sending or receiving state. But since s_{n+1} is a local 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 src-equivalent to $T{\downarrow_p}$. Now let $t = \tau|_{|\sigma|+1}$, so that s_{n+1} and t are distinct local 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, $d_p(R) \le d_p(S') - 1$, a contradiction, completing the proof of the first part of Proposition 6.2.

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 $t \in \mathsf{Trans}_p$, 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

$$\sigma_r = (s^r_1, \ldots, s^r_{n(r)})$$

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

$$T^k{\downarrow_r} = T^{k+1}{\downarrow_r} = (s^r_1, \ldots, s^r_{m(r)})$$

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

$$T^{k+1}{\downarrow_p} = (s^p_1, \ldots, s^p_{m(p)}, t).$$

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 $q = \mathsf{receiver}(c)$. 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 with $d_p(R) < d_p(S)$.

If t is local, we may take $R = (s_1, \ldots, s_n, t)$.

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 $q = \mathsf{sender}(c)$. Since $T^k{\downarrow_q} \preceq S{\downarrow_q}$, there must be at least i sends on c in S, and $\mathsf{Sent}_c(S)|_i = x$. 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

$$R = (s_1, \ldots, s_n, t),$$

will 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 5.2 implies $n(p) = m(p) + 1$ and one of $s = s^p_{m(p)+1}$, 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 6.2.

Corollary 6.3. *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 6.2 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. □

7. Deadlock

The main result of this section is Theorem 7.4, which implies that, under certain hypotheses on the MPI program, to verify that the program is deadlock-free it suffices to consider only synchronous executions. We also describe a stronger property, which in essence says that no subset of the processes in the program can ever deadlock, and prove an analogous theorem (Theorem 7.7) for that property.

7.1. Definitions.

Let $\mathcal{M} = (\mathcal{C}, M)$ be a model of an MPI program, $\Sigma \subseteq \mathsf{Proc}$, and let S be a finite execution prefix.

Definition 7.1. We say that S is *potentially Σ-deadlocked* if $\mathsf{terminus}(S)_p \notin \mathsf{End}_p$ for some $p \in \Sigma$ and S has no universally permitted proper extension.

It is not hard to see that S is potentially Σ-deadlocked if, and only if, all of the following hold:

(i) For some $p \in \Sigma$, $\mathsf{terminus}(S)_p \notin \mathsf{End}_p$.

(ii) For all $p \in \mathsf{Proc}$, $\mathsf{terminus}(S)_p$ is not a local-event state.

(iii) For each $p \in \mathsf{Proc}$ for which $\mathsf{terminus}(S)_p$ is a receiving or a send-receive state: for all $c \in \mathsf{Chan}$ for which there is a transition departing from $\mathsf{terminus}(S)_p$ labeled by a receive on c: $\mathsf{Pending}_c(S)$ is empty, and, letting $q = \mathsf{sender}(c)$, no transition departing from $\mathsf{terminus}(S)_q$ is labeled by a send on channel c.

We use the word "potentially" because it is not necessarily the case that a program that has followed such a path will deadlock. It is only a possibility—whether or not an actual deadlock occurs depends on the buffering choices made by the MPI implementation at the point just after the end of the prefix. For example, if the MPI implementation decides (for whatever reason) to force all sends to synchronize at this point, then the program will deadlock. On the other hand, if the implementation decides to buffer one or more sends, the program may not deadlock. Hence the potentially deadlocked prefixes are precisely the ones for which some choice by a legal MPI implementation would lead to deadlock. Since our motivation is to write programs that will perform correctly under any legal MPI implementation, and independently of the choices made by that implementation, we most likely want to write programs that have no potentially deadlocked execution prefixes. This observation motivates the following definition:

Definition 7.2. We say that $\mathcal{M}$ is *Σ-deadlock-free* if it has no potentially Σ-deadlocked execution prefix. We say that $\mathcal{M}$ is *synchronously Σ-deadlock-free* if it has no potentially Σ-deadlocked synchronous execution prefix.

We will also have occasion to talk about execution prefixes that must necessarily result in deadlock, though the concept will not be as important to us as the one above.

Definition 7.3. We say that S is *absolutely Σ-deadlocked* if $\mathsf{terminus}(S)_p \notin \mathsf{End}_p$ for some $p \in \Sigma$ and there is no proper extension of S to an execution prefix.

It is not hard to see that S is absolutely Σ-deadlocked if, and only if, statements (i)–(iii) above and

(iv) For all $p \in \mathsf{Proc}$, U_p is not a sending or send-receive state.

all hold. We use the word "absolutely" here because a program that has followed such a path *must* deadlock at this point—no matter the choices made by the MPI implementation. For, at the end of the prefix, no process is at a point where it can send a message or perform a local operation, and those processes ready to perform a receive have no pending messages that they can receive.

We remark that both definitions of deadlock (potential and absolute) depend only on knowing the src-equivalence class of $S\downarrow_p$ for each $p \in \mathsf{Proc}$. For we have already observed, in §5, that the universally permitted extensions of an execution prefix depend only on that information. It is clear that the set of arbitrary extensions also depends only on that information.

The role of 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.

However, in many applications $\Sigma = \mathsf{Proc}$, and, to simplify the terminology, in these cases we will feel free to leave out the "Σ" in all of the definitions above (as well as in the definitions concerning partial deadlock in §7.4 below).

7.2. **The Deadlock Theorem.** Our main result concerning deadlock is the following:

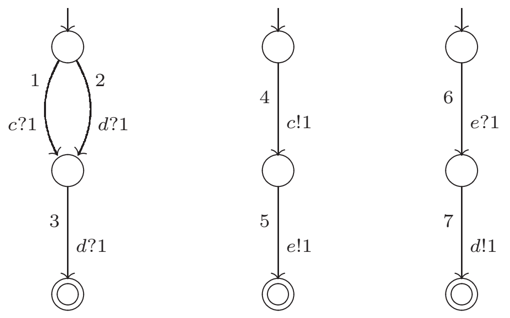


FIGURE 5. Counterexample to Deadlock Theorem with wildcard receive

Theorem 7.4. *Let $\mathcal{M}$ be a model of an MPI program with no wildcard receives. Let Σ be a subset of* Proc. *Then $\mathcal{M}$ is Σ-deadlock-free if, and only if, $\mathcal{M}$ is synchronously Σ-deadlock-free.*

Proof. 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 $\mathsf{terminus}(T)_p \notin \mathsf{End}_p$ for some $p \in \Sigma$. We must show there exists a universally permitted proper extension T' of T.

By Corollary 6.3, there is a synchronous finite execution prefix S with the property that any synchronous extension of S is compatible with T.

By hypothesis, either $S{\downarrow_p} \in \mathsf{End}_p$ 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 6.2 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. □

7.3. **Counterexample with Wildcard Receive.** Theorem 7.4 fails if we allow $\mathcal{M}$ to have wildcard receives. Consider the example with three processes illustrated in Figure 5. It is not hard to see that the synchronous execution prefixes for this model are all prefixes of the sequence $(4, 1, 5, 6, 7, 3)$, and none of these is potentially deadlocked. However, the non-synchronous execution prefix $(4, 5, 6, 7, 2)$ is potentially deadlocked.

7.4. **Partial Deadlock.** Let $\mathcal{M} = (\mathcal{C}, M)$ be a model of an MPI program, and let S be a finite execution prefix. Let Σ be a subset of Proc.

Definition 7.5. We say S is *potentially partially* Σ*-deadlocked* (or Σ*-ppd*, for short) if for some $p \in \Sigma$, $\mathsf{terminus}(S)_p \notin \mathsf{End}_p$ 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.

Definition 7.6. 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.

We will prove the following analogue of Theorem 7.4 for partial deadlock:

Theorem 7.7. *Let $\mathcal{M}$ be a model of an MPI program with no wildcard receives. Let Σ be a subset of* Proc. *Then $\mathcal{M}$ is free of partial Σ-deadlock if, and only if, $\mathcal{M}$ is synchronously free of partial Σ-deadlock.*

The proof of Theorem 7.7 will require the following, which will also come in useful elsewhere:

Lemma 7.8. *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 $\mathsf{terminus}(T) \notin \mathsf{End}_p$.*

Proof. By Corollary 6.3, 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 $\mathsf{terminus}(S)_p \in \mathsf{End}_p$, or there exists a synchronous proper extension S' of S satisfying $|S{\downarrow_p}| < |S'{\downarrow_p}|$. Replace S with S' and repeat, until $\mathsf{terminus}(S)_p \in \mathsf{End}_p$ 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 $\mathsf{terminus}(S)_p \in \mathsf{End}_p$ then by Lemma 5.2, $S{\downarrow_p}$ 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, $\mathsf{terminus}(T)_p \in \mathsf{End}_p$. So both (ii) and (iii) hold.

If $\mathsf{terminus}(S)_p \notin \mathsf{End}_p$ then by construction, $|S{\downarrow_p}| > |T{\downarrow_p}|$. Again by Lemma 5.2, 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 7.7. 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 $\mathsf{terminus}(T)_p \notin \mathsf{End}_p$. We must show there exists a universally permitted proper extension T' of T with $|T{\downarrow_p}| < |T'{\downarrow_p}|$.

By Lemma 7.8, there is a finite synchronous execution prefix S that is compatible with T and satisfies $|S{\downarrow_p}| > |T{\downarrow_p}|$. Now apply Proposition 6.2 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. □

8. Application to Channel Depths

As we have seen, one of the important questions facing the analyst of an MPI program is the upper bound to be placed on the depth of the communication channels. If a property has been verified under the assumption that the size of the message buffers never exceeds, say, 4, how do we know that a violation will not be found with 5?

The results on deadlock show that, in certain circumstances, we are justified in assuming all communication is synchronous. For a model checker such as Spin, that means we may use channels of depth 0, which may greatly reduce the size of the state space that Spin will explore. In this section we attempt to gain some control on the channel depths for other kinds of properties. The following result could be applicable to a property that is an assertion on what types of states are reachable.

Theorem 8.1. *Let $\mathcal{M}$ be a model of an MPI program with no wildcard receives, and $\Sigma \subseteq \mathsf{Proc}$. Suppose $\mathcal{M}$ is free of partial Σ-deadlock. Let T be a finite execution prefix of $\mathcal{M}$ such that, for all $p \in \mathsf{Proc}$, $\mathsf{terminus}(T)_p$ is not 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 $c \in \mathsf{Chan}$ for which $\mathsf{receiver}(c) \in \Sigma$, if $|\,\mathsf{Pending}_c(T)| = 0$ then S is c-synchronous, while if $|\,\mathsf{Pending}_c(T)| > 0$ then*

$$|\,\mathsf{Pending}_c(S^i)| \leq |\,\mathsf{Pending}_c(T)|$$

for all i in the domain of S.

Proof. By Lemma 7.8, 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 5.2 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 Trans_p but that occur after the $m(p)^{\text{th}}$ transition in Trans_p. Hence the resulting sequence S will have exactly $m(p)$ transitions in Trans_p for each $p \in \mathsf{Proc}$. We will show that S has the properties listed in the statement of the Theorem.

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 S is c-legal. 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 is clearly c-legal. Moreover, if $s' = r'$ then S is c-synchronous, while if not then

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

for all i in the domain of S.

Now if $\mathsf{receiver}(c) \in \Sigma$, 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. □

We outline here one way Theorem 8.1 could prove useful. Suppose that R is a set of global states, and one wishes to verify the property $\mathcal{P}$ that says that no state in R is reachable. One could prove $\mathcal{P}$ in a series of steps. In the first step, one should show that the model is free of partial deadlock (perhaps using Theorem 7.7

to reduce to the synchronous case). Next, one must somehow show that, if there exists a violation to $\mathcal{P}$, then there exists a violating execution prefix T in which, at the end of execution of T, all the queues have size no greater than some fixed integer d. Finally, one may verify $\mathcal{P}$ while bounding the channel depths by d. Theorem 8.1 justifies this last step, for, if there were a violation to $\mathcal{P}$, then by Theorem 8.1 there would have to be one for which the channel depths never exceed d at any stage. See §14.1 for an example of an argument of this form.

9. Barriers

In this section, we will describe how the statement

$$\mathsf{MPI\_BARRIER(MPI\_COMM\_WORLD)}$$

in program code can be represented in our model. We will present this by starting with a model of the program without barriers, then identifying those states that correspond to a position in code just after a barrier statement, and then showing how to modify the state machines to incorporate the barrier before those states.

Let $\mathcal{M} = (\mathcal{C}, M)$ be a model of an MPI program. Suppose B is a set of states in $\mathcal{M}$, and we wish to insert "barriers" before each of these states. We assume that B does not contain any start_p. Also, we assume that B contains no immediate successors of send-receive states (These are the u' and v states of part (v) in the definition of MPI State Machine, §3.2.) We exclude such states because we do not allow a barrier statement before the process begins, nor "inside" an MPI_SENDRECV statement. We call such a set B a *barrier-acceptable* state set for $\mathcal{M}$.

Given $\mathcal{M}$ and B as above, we now describe how to create a new model $\mathcal{M}^B = (\mathcal{C}^B, M^B)$ which represents the model $\mathcal{M}$ with barriers added just before the states in B. Write

$$\mathcal{C}^B = (\mathsf{Proc}^B, \mathsf{Chan}^B, \mathsf{sender}^B, \mathsf{receiver}^B, \mathsf{msg}^B, \mathsf{loc}^B, \mathsf{com}^B).$$

We define $\mathsf{Proc}^B = \mathsf{Proc} \cup \{\beta\}$, where β is some object not in Proc. We call β the *barrier process*. The precise definition of $\mathcal{C}^B$ is given in Figure 6(a), but the basic idea is as follows: for all $p \in \mathsf{Proc}$, we add two new channels ϵ_p and ξ_p to Chan, to form Chan^B. Channel ϵ_p sends a bit from p to β signifying that p is ready to enter the barrier, and ξ_p sends a bit from β to p telling p it may exit the barrier.

Now we modify each state machine M_p to produce the state machine M_p^B. The precise definition of M_p^B is given in Figure 6(b), and Figure 7 gives a graphical representation of the transformation. The idea is simply to add, for each state $v \in B$, two new states $b_1(v), b_2(v)$, and two new transitions $t_1(v), t_2(v)$. The functions src^B and des^B are defined so that $t_1(v)$ leads from $b_1(v)$ to $b_2(v)$, and $t_2(v)$ leads from $b_2(v)$ to v, and so that any transition in M_p that terminates in v is redirected to terminate in $b_1(v)$ in M_p^B. Transition $t_1(v)$ is labeled by $\epsilon_p!1$ and $t_2(v)$ is labeled by $\xi_p?1$. The state $b_2(v)$ will be referred to as a *barrier state*. One can check that M_p^B does indeed satisfy the definition of MPI state machine; notice that this requires our assumption that B not contain any immediate successors of send-receive states.

Now we describe the state machine M_β^B for the barrier process, which is depicted in Figure 8. We let $M_\beta = M_\beta^B$ to simplify the notation. First, choose a total order for Proc, say $\mathsf{Proc} = \{p_1, \ldots, p_N\}$, and let $\epsilon_i = \epsilon_p$ and $\xi_i = \xi_p$, where $p = p_i$. Now M_β has states u_i and v_i, for $1 \leq i \leq N$. The start state is u_1. For each i, there is

$$(a)\qquad \begin{aligned}
\mathsf{Proc}^B &= \mathsf{Proc} \cup \{\beta\} \\
\mathsf{Chan}^B &= \mathsf{Chan} \cup \bigcup_{p \in \mathsf{Proc}} \{\epsilon_p, \xi_p\} \\
\mathsf{sender}^B(c) &= \begin{cases} p & \text{if } c = \epsilon_p \text{ for some } p \in \mathsf{Proc} \\ \beta & \text{if } c = \xi_p \text{ for some } p \in \mathsf{Proc} \\ \mathsf{sender}(c) & \text{otherwise} \end{cases} \\
\mathsf{receiver}^B(c) &= \begin{cases} \beta & \text{if } c = \epsilon_p \text{ for some } p \in \mathsf{Proc} \\ p & \text{if } c = \xi_p \text{ for some } p \in \mathsf{Proc} \\ \mathsf{receiver}(c) & \text{otherwise} \end{cases} \\
\mathsf{msg}^B(c) &= \begin{cases} \{1\} & \text{if } c = \epsilon_p \text{ or } c = \xi_p \text{ for some } p \in \mathsf{Proc} \\ \mathsf{msg}(c) & \text{otherwise} \end{cases} \\
\mathsf{loc}^B(p) &= \begin{cases} \emptyset & \text{if } c = \beta \\ \mathsf{loc}(p) & \text{otherwise} \end{cases} \\
\mathsf{com}^B(p) &= \begin{cases} \{\epsilon_p?1, \xi_p!1\} & \text{if } c = \beta \\ \mathsf{com}(p) \cup \{\epsilon_p!1, \xi_p?1\} & \text{otherwise} \end{cases}
\end{aligned}$$

$$(b)\qquad \begin{aligned}
\mathsf{States}_p^B &= \mathsf{States}_p \cup \bigcup_{v \in B} \{b_1(v), b_2(v)\} \\
\mathsf{Trans}_p^B &= \mathsf{Trans}_p \cup \bigcup_{v \in B} \{t_1(v), t_2(v)\} \\
\mathsf{src}_p^B(t) &= \begin{cases} b_1(v) & \text{if } t = t_1(v) \text{ for some } v \in B \\ b_2(v) & \text{if } t = t_2(v) \text{ for some } v \in B \\ \mathsf{src}_p(t) & \text{otherwise} \end{cases} \\
\mathsf{des}_p^B(t) &= \begin{cases} b_2(v) & \text{if } t = t_1(v) \text{ for some } v \in B \\ v & \text{if } t = t_2(v) \text{ for some } v \in B \\ b_1(v) & \text{if } \mathsf{des}(t) = v \text{ for some } v \in B \\ \mathsf{des}_p(t) & \text{otherwise} \end{cases} \\
\mathsf{label}_p^B(t) &= \begin{cases} \epsilon_p!1 & \text{if } t = t_1(v) \text{ for some } v \in B \\ \xi_p?1 & \text{if } t = t_2(v) \text{ for some } v \in B \\ \mathsf{label}_p(t) & \text{otherwise} \end{cases} \\
\mathsf{start}_p^B &= \mathsf{start}_p \\
\mathsf{End}_p^B &= \mathsf{End}_p
\end{aligned}$$

FIGURE 6. (a) The relationship between the context $\mathcal{C}$ and the context with barriers $\mathcal{C}^B$, and (b) the relationship between the MPI state machine M_p and the state machine after adding barriers M_p^B.

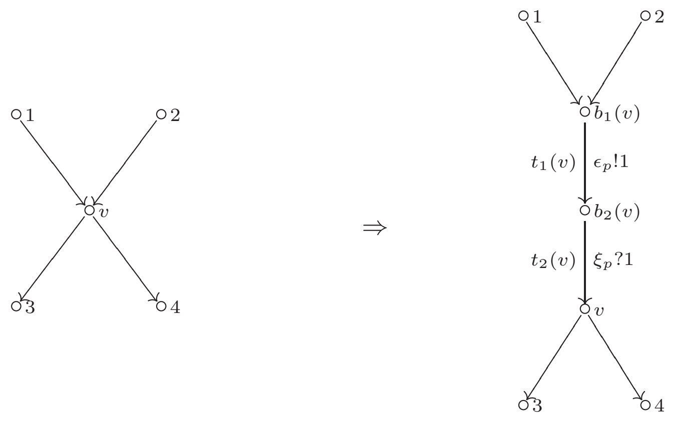


FIGURE 7. Insertion of barrier before state v.

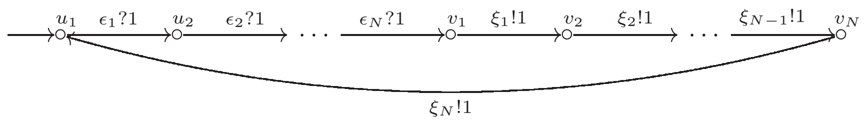


FIGURE 8. The barrier process M_β.

a transition s_i, departing from u_i, with $\mathsf{label}(s_i) = \epsilon_i?1$. For $i < N$ this terminates in u_{i+1}, while s_N terminates in v_1. For each i there is also a transition t_i departing from v_i, labeled $\xi_i!1$. For $i < N$ this terminates in v_{i+1}, while t_N terminates in u_1. Finally, let $\mathsf{End}_\beta = \emptyset$; the barrier process never terminates.

An example will illustrate how an MPI_BARRIER call in the code is represented in our model. Consider an MPI program written in C and consisting of 3 processes, with the following code occurring in process 1:

```
MPI_Recv(buf, 1, MPI_INT, 2, 0, MPI_COMM_WORLD, &stat);
MPI_Barrier(MPI_COMM_WORLD);
MPI_Send(buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

Hence process 1 first receives an integer from process 2, then participates in a barrier, and then sends that integer to process 0. Let us say that in our model we will limit the possible values of this integer to the set $\{1, 2, 3\}$. Let c be the channel used for sending messages from process 2 to process 1 with tag 0, and let d be the channel used for sending messages from process 1 to process 0. Then the portion of the state machine corresponding to this code appears in Figure 9.

The translation process described above does not explain how to translate code with two or more consecutive barrier statements. However, we may always first modify this code by inserting "null" statements between the consecutive barriers. These null statements could be represented by local transitions in the state machine, and then the process already described will work fine.

Definition 9.1. We say that a global state U of $\mathcal{M}^B$ is *inside a barrier* if U_β is the state v_1 of M_β.

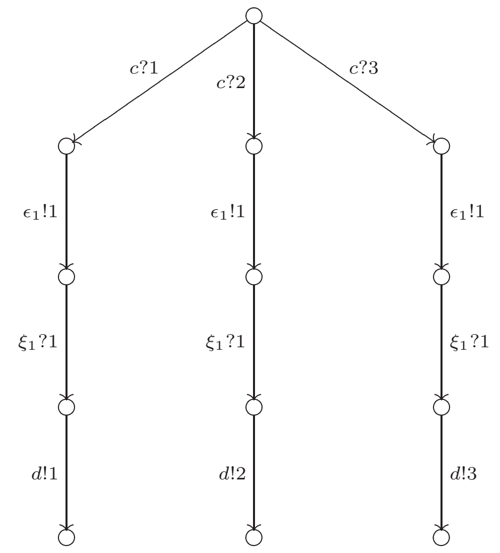


FIGURE 9. A process that receives a message, participates in a barrier, and then sends a message.

Although the definition above does not make any mention of the states of the non-barrier processes, we will see, in the following proposition, that, in any execution, these must all be at barrier states whenever M_β is at the state v_1.

Proposition 9.2. *Let S be a finite execution prefix for $\mathcal{M}^B$ terminating inside a barrier. Then for all $p \in \mathsf{Proc}$, $\mathsf{terminus}(S)_p$ is a barrier state. Moreover, there exists $k \geq 1$ such that for all $p \in \mathsf{Proc}$,*

$$|\,\mathsf{Received}_{\epsilon_p}(S)| = |\,\mathsf{Sent}_{\epsilon_p}(S)| = k,$$

and

$$|\,\mathsf{Received}_{\xi_p}(S)| = |\,\mathsf{Sent}_{\xi_p}(S)| = k - 1.$$

In particular, $\mathsf{Pending}_{\epsilon_p}(S)$ and $\mathsf{Pending}_{\xi_p}(S)$ are empty for all $p \in \mathsf{Proc}$.

Proof. For $p \in \mathsf{Proc}$, let $l(p)$ be the number of messages received on ϵ_p in S, and $\bar{l}(p)$ the number of messages sent on ϵ_p in S. Let $m(p)$ and $\bar{m}(p)$ be the analogous numbers for ξ_p.

Let $k = l(p_1)$. By examining M_β (see Figure 8), it is clear that $l(p) = k$ and $\bar{m}(p) = k-1$ for all $p \in \mathsf{Proc}$. Considering the construction of M_p^B, we see that, for all p, $\bar{l}(p) \leq m(p) + 1$, and equality holds if, and only if, $\mathsf{terminus}(S)_p$ is a barrier state. Hence

$$k = l(p) \leq \bar{l}(p) \leq m(p) + 1 \leq \bar{m}(p) + 1 = k - 1 + 1 = k. \tag{1}$$

So we must have equality throughout in equation (1), which completes the proof. □

Proposition 9.3. *Let $S = (s_1, \ldots, s_n)$ be a finite execution prefix for $\mathcal{M}^B$. Suppose for some $p \in \mathsf{Proc}$, and some $n' < n$, $s_{n'}$ is labeled $\epsilon_p!1$, s_n is labeled $\xi_p?1$, and for* *all k, $n' < k < n$, $s_k \notin \mathsf{Trans}_p^B$. Then for some k, $n' \leq k < n$, S^k terminates inside a barrier.*

Propositions 9.2 and 9.3, taken together, may be interpreted as a justification that our representation of barriers corresponds to the semantics given in the MPI Standard. For the Standard states that "MPI_BARRIER blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call" ([16, §4.3]). Proposition 9.3 shows that in any execution in which a process enters and exits a barrier, at some point in between the program must have been "inside a barrier," while Proposition 9.2 established that being "inside a barrier" corresponds to every process being at a barrier state, i.e., having entered but not yet exited a barrier.

Proof of Proposition 9.3. Let l be the number of messages received on ϵ_p in S, $\bar{l}$ the number sent on ϵ_p, and m and $\bar{m}$ the analogous numbers for ξ_p. Define $l', \bar{l}', m', \bar{m}'$ to be the analogous numbers for the sequence $S^{n'-1}$.

We know that $l \leq \bar{l}$ and $m \leq \bar{m}$, since the number of messages received on a channel can never exceed the number sent. We know that $\bar{l} = m$ because it is clear from the construction of M_p^B that the number of $\epsilon_p!1$ events must equal the number of $\xi_p?1$ events in any path that does not terminate in a barrier state. We also know that $\bar{m} \leq l$ since in the cyclic path through M_β the transition labeled $\epsilon_p?1$ precedes the one labeled $\xi_p!1$. Putting these together yields

$$l \leq \bar{l} = m \leq \bar{m} \leq l,$$

which implies

$$l = \bar{l} = m = \bar{m}. \tag{2}$$

Similar reasoning yields

$$l' = \bar{l}' = m' = \bar{m}'. \tag{3}$$

Now, in the sequence $S' = (s_{n'}, \ldots, s_n)$ there occur only two events in M_p^B; one of these is $\epsilon_p!1$ and the other $\xi_p?1$. Hence

$$\bar{l} = \bar{l}' + 1$$
$$m = m' + 1.$$

Putting these together with equations (2) and (3), we have

$$l = \bar{l} = \bar{l}' + 1 = l' + 1$$
$$\bar{m} = m = m' + 1 = \bar{m}' + 1.$$

This implies that in S' there is exactly one occurrence of $\epsilon_p?1$ and one occurrence of $\xi_p!1$. Moreover, since $l' = \bar{m}'$, $\mathsf{terminus}(S^{n'-1})_p$ lies on the path joining v_p' to u_p, where v_p' is the immediate successor to v_p in M_β (see Figure 8). This means that in S', the $\epsilon_p?1$ must occur before the $\xi_p!1$. But any path in M_β for which this is the case must pass through v_1. $\square$

10. Removal of Barriers

A common question that arises with MPI programs is "when is it safe to remove a barrier?" In this section, we show that, under appropriate hypotheses, at least the removal of barriers can not introduce deadlocks into the program.

Theorem 10.1. *Let $\mathcal{M}$ be a model of an MPI program with no wildcard receives, and let B be a barrier-acceptable state set for $\mathcal{M}$, and Σ a subset of* 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).*

To prove the Theorem, we first define a new model $\mathcal{N}$. This is obtained by modifying $\mathcal{M}^B$ in a fairly simple way which we will now describe. We will only change the values of the sender and receiver functions for certain channels, and we will change the label function for certain transitions, but everything else, including the state set, the transition set, and the src and des functions, will be exactly the same in the two models.

First, we change the direction of each channel ξ_p, so that in the new model, this channel carries messages from p to β, just like ϵ_p. Second, we change the labeling function so that any transition that was labeled by $\xi_p?1$ in $\mathcal{M}^B$ is labeled by $\xi_p!1$ in $\mathcal{N}$, and similarly, any transition that was labeled by $\xi_p!1$ in $\mathcal{M}^B$ is labeled by $\xi_p?1$ in $\mathcal{N}$.

Hence in $\mathcal{N}$, the original processes send messages to the barrier process to both enter and exit the barrier. Now, for arbitrary executions, this arrangement will not necessarily function as a barrier, since the messages may all be buffered. However, for synchronous executions, it will function as a barrier in exactly the same way that this worked in $\mathcal{M}^B$. In our proof, $\mathcal{N}$ will act as a bridge between $\mathcal{M}$ and $\mathcal{M}^B$.

There is a 1-1 correspondence ψ from the set of synchronous execution prefixes of $\mathcal{N}$ to the set of those of $\mathcal{M}^B$, defined as follows: suppose $S = (t_1, t_2, \ldots)$ is a synchronous execution prefix of $\mathcal{N}$. Suppose for some i, t_i and t_{i+1} are labeled by $\xi_p!1$ and $\xi_p?1$, respectively. By replacing each such subsequence (t_i, t_{i+1}) with (t_{i+1}, t_i), we obtain a synchronous execution prefix $\psi(S)$ for $\mathcal{M}^B$. The definition of the inverse of ψ is clear. It is also clear that

$$S \subseteq T \iff \psi(S) \subseteq \psi(T)$$

and that $|S\!\downarrow_p| = |\psi(S)\!\downarrow_p|$ for all $p \in \mathsf{Proc}^B$. It follows from these observations that S is potentially Σ-deadlocked (resp., Σ-ppd) if, and only if, $\psi(S)$ is. We may conclude that $\mathcal{N}$ is Σ-deadlock-free (resp., free of partial Σ-deadlock) if, and only if, $\mathcal{M}^B$ is, as Theorems 7.4 and 7.7 reduce these questions to the synchronous case.

We now turn to the relationship between $\mathcal{M}$ and $\mathcal{N}$. We define a map θ from the set of all finite execution prefixes of $\mathcal{M}$ to the set of those of $\mathcal{N}$, as follows: suppose $S = (t_1, \ldots, t_n)$ is a finite execution prefix for $\mathcal{M}$. For each i for which $\mathsf{des}(t_i) \in B$, insert, immediately after t_i, the appropriate transitions labeled $\epsilon_p!1$ and $\xi_p!1$. This guarantees that we get paths through each state machine in $\mathcal{N}$, and the resulting sequence is still c-legal for each channel c since we only inserted send transitions. Now, at the end of this prefix, there may be messages in the queues for the ϵ_p and ξ_p, so we append the longest possible sequence of transitions from Trans_β so that the resulting sequence is still an execution prefix (i.e., we just let the barrier process run until it reaches a state in which the channel for the unique outgoing transition has an empty message queue). This results in an execution prefix $\theta(S)$ for $\mathcal{N}$.

Observe that, by construction, $\mathsf{terminus}(\theta(S))_p = \mathsf{terminus}(S)_p$ for all $p \in \mathsf{Proc}$. In particular, $\mathsf{terminus}(\theta(S))_p$ does not have an outgoing barrier transition (i.e., a transition labeled by a communication event involving an ϵ_p or a ξ_p).

We next define a map ϕ in the opposite direction—from the set of all finite execution prefixes of $\mathcal{N}$ to the set of those of $\mathcal{M}$—by simply deleting all the barrier transitions. It is clear that $\phi \circ \theta$ is the identity, and that, if $S \subseteq T$ are prefixes for $\mathcal{N}$, then $\phi(S) \subseteq \phi(T)$. Furthermore, if T is a universally permitted extension of S then $\phi(T)$ is a universally permitted extension of $\phi(S)$.

Now suppose $\mathcal{N}$ is Σ-deadlock-free, and we wish to show the same of $\mathcal{M}$. Suppose S is a finite execution prefix of $\mathcal{M}$ such that $\mathsf{terminus}(S)_p \notin \mathsf{End}_p$ for some $p \in \Sigma$. Then $\mathsf{terminus}(\theta(S))_p \notin \mathsf{End}_p$ as well. Hence there exists a universally permitted proper extension T of $\theta(S)$. Moreover, the first transition t in $T \setminus \theta(S)$ must lie in Trans_q for some $q \in \mathsf{Proc}$, since, by the construction of $\theta(S)$, the barrier process is blocked at the end of $\theta(S)$. As we have seen, t cannot be a barrier transition. It follows that

$$\phi(T) \supset \phi(\theta(S)) = S,$$

so $\phi(T)$ is a universally permitted proper extension of S. Hence $\mathcal{M}$ is Σ-deadlock-free.

Suppose instead that $\mathcal{N}$ is free of partial Σ-deadlock, and we wish to show the same of $\mathcal{M}$. In this case we may choose T as above but with the additional requirement that $|T\downarrow_p| > |\theta(S)\downarrow_p|$. Again, it must be the case that the first transition t of $T\downarrow_p \setminus \theta(S)\downarrow_p$ is not a barrier transition. Hence

$$|\phi(T)\downarrow_p| > |\phi(\theta(S))\downarrow_p| = |S\downarrow_p|,$$

and this shows that $\mathcal{M}$ is free of partial Σ-deadlock, completing the proof of Theorem 10.1.

11. Emptiness of Channels Inside Barriers

We have already seen, in Proposition 9.2, that inside a barrier, the barrier channels ξ_p and ϵ_p must be empty. Now we will show that, under suitable restrictions on the model, all channels must be empty inside a barrier.

Theorem 11.1. *Let $\mathcal{M}$ be a model of an MPI program with no wildcard receives, and B a barrier-acceptable state set for $\mathcal{M}$. Let Σ be a nonempty subset of 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. Let T be an associated synchronous prefix for S. Then $S\downarrow_p \sim T\downarrow_p$ for all $p \in \mathsf{Proc}^B$. In particular, $\mathsf{Pending}_c(S)$ is empty for all $c \in \mathsf{Chan}^B$.*

Proof. By Proposition 9.2, there is an integer k such that for each $p \in \mathsf{Proc}$, there are exactly k occurrences of $\epsilon_p?1$ and k occurrences of $\epsilon_p!1$ in S, and $k-1$ occurrences of $\xi_p?1$ and $\xi_p!1$.

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

$$\pi_r = (s_1^r, \ldots, s_{n(r)}^r) \sim S\downarrow_r$$

through M_r^B and an integer $m(r)$ in the domain of π_r such that

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

Clearly we cannot have $\mathsf{terminus}(T)_r \in \mathsf{End}_r$ for any r, since no $\mathsf{terminus}(S)_r$ is an end state. Nor can $\mathsf{terminus}(T)_r$ be a local-event state, since then we would have $n(r) > m(r) + 1$ and $s_{m(r)+1}^r$ is a local transition, and we could append this transition to T.

Suppose $m(\beta) = n(\beta)$. Then Propositions 9.2 and 9.3 imply $m(r) = n(r)$ for all $r \in \mathsf{Proc}$, so $T\downarrow_r \sim S\downarrow_r$ for all r, as required.

So let us assume that $m(\beta) < n(\beta)$. We will arrive at a contradiction.

If T were potentially deadlocked then we would have $\mathsf{terminus}(T)_r \in \mathsf{End}_r$ for some r, since Σ is non-empty. As this is not the case, T cannot be potentially deadlocked. So there must exist $p, q \in \mathsf{Proc}^B$, and $c \in \mathsf{Chan}^B$, such that $\mathsf{terminus}(T)_p$ has an outgoing transition t labeled $c!x$ and $\mathsf{terminus}(T)_q$ has an outgoing transition t' labeled $c?x$. Clearly $m(p) < n(p)$, since π_p terminates at a receiving state. We claim that $m(q) < n(q)$ as well. For if not, then $m(q) = n(q)$, $c = \xi_q$ and $p = \beta$. So β is at the state with an outgoing transition labeled $\xi_p!1$. Moreover, since $m(q) = n(q)$, there are $k - 1$ occurrences of $\xi_q?1$ in $T{\downarrow_q}$ and therefore $k - 1$ occurrences of $\xi_p!1$ in $T{\downarrow_\beta}$. This implies $m(\beta) \geq n(\beta)$, a contradiction.

Hence either $s^p_{m(p)+1}$ is labeled $c!x$, or $\mathsf{terminus}(T)_p$ is a send-receive state and $s^p_{m(p)+2}$ is labeled $c!x$. Similarly, either $s^q_{m(q)+1}$ is labeled $c?y$, or $\mathsf{terminus}(T)_q$ is a send-receive state and $s^q_{m(q)+2}$ is labeled $c?y$ for some $y \in \mathsf{msg}(c)$. (We know that the receiving channel must be c, since there are no wildcard receives.) Since $\mathsf{Received}_c(S)$ is a prefix of $\mathsf{Sent}_c(S)$, and T is synchronous, we must have $x = y$. Hence if we append t and then t' to T, the result is a synchronous execution prefix T' which is longer than T and satisfies $T' \preceq S$, a contradiction. $\square$

12. Other Collective Functions

The collective functions are described in Chapter 4 of the MPI Standard. We have already dealt with MPI_BARRIER in the previous sections. We now sketch a way to express all of the collective functions in our formalism, generalizing the technique we used for MPI_BARRIER. We must take care to do so in such a way that the model allows for all possible behaviors that the Standard allows for. We would also like to not introduce any new non-determinism.

What we describe is not necessarily the most efficient way to do this (from an implementation or model-checking point of view), but it suffices to show that it is possible, so that the theorems of this paper may be applied to programs which use these functions.

One of the key points made by the Standard is that collective functions may or may not be implemented in such a way that they impose a barrier:

> Collective routine calls can (but are not required to) return as soon as their participation in the collective communication is complete. The completion of a call indicates that the caller is now free to access locations in the communication buffer. It does not indicate that other processes in the group have completed or even started the operation (unless otherwise indicated in the description of the operation). Thus, a collective communication call may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier function. [16, §4.1]

In some cases, it is logically necessary that they impose a barrier (e.g., MPI_BARRIER or MPI_ALLREDUCE), but in others it is not (e.g., MPI_BCAST). The model must allow for the possibility that the functions in the latter class may or may not impose a barrier.

Another point the Standard makes is that, given a communicator, all processes in that communicator must call the same collective functions with the same parameters (e.g., same value of root, same reduction function, etc.). Furthermore they should do so in the same order. It is considered an error, for example, if one process invokes first a reduce and then a scatter function, while another process, using the same communicator, calls a scatter and then a reduce.

One fact that simplifies our work is that the Standard guarantees that the messages sent in collective communications are kept entirely separate from those used in point-to-point communications. This suggests that we use different channels for the collective and point-to-point communications.

We now assume our MPI program contains n processes and a single communicator MPI_COMM_WORLD. We will use a single coordinator process C for handling all collective communication in the communicator; it is a generalization of the barrier process described in §9. We will also take care to use no wildcard receives in constructing C. This will make the construction a bit more cumbersome than one that uses wildcard receives, but has the important advantage that it allows us to construct a *locally deterministic* model, a concept explored in §13 below.

We now add a set of *collective channels* to the system. This set consists of one channel $c_{p,q}$ for each ordered pair

$$(p, q) \in (\{C\} \cup \mathsf{Proc}) \times (\{C\} \cup \mathsf{Proc}).$$

We let $\mathsf{sender}(c_{p,q}) = p$ and $\mathsf{receiver}(c_{p,q}) = q$. We do not specify $\mathsf{msg}(c_{p,q})$ as in general this will depend on the particular abstractions used in constructing the model. All of these channels will be used exclusively for collective communication.

As the first step in invoking a collective function, a process will send a message to C (using the collective channel) containing the name of the function and all relevant data and arguments; we call this message the *initial message.* What happens next depends on the semantics of the particular function.

The coordinator receives the initial messages in rank order (i.e., first from the process of rank 0, then 1, etc., up to $n-1$). It immediately goes to a state indicating an error if it receives an initial message that does not match the previous ones—e.g., if the name of the collective function or the rank of the root differs from those of the previous processes. Assuming no such error occurs, what the coordinator does next depends on the particular function, and is described for each case below. When it has finished, it returns to its initial state, ready to coordinate the next collective communication.

We divide the functions into four categories, depending on the communication pattern the functions impose.

The first category includes the "barrier-like" functions

MPI_BARRIER,	MPI_ALLGATHER,	MPI_ALLGATHERV,
MPI_ALLREDUCE,	MPI_ALLTOALL,	MPI_ALLTOALLV,
MPI_ALLTOALLW,	MPI_REDUCE_SCATTER.	

Each of these requires, either implicitly or explicitly, that all processes invoke the function before any of them can return from it. They do not involve a root. For all of these functions, each process, after sending its initial message to C, receives a message from C. For MPI_BARRIER, each process just receives a single bit "1" and does nothing with it. For the others, the set of possible messages will depend upon the particular abstractions used to construct the model. The transition taken in receiving the message from C represents the change in state of the receive buffer; the exact representation will depend on the precise way this part of the state is represented in the model.

After receiving the initial messages, C computes whatever is required for the particular function and sends back the appropriate messages to each of the processes, in rank order.

The second category includes the "gather-like" functions MPI_GATHER, MPI_-GATHERV, and MPI_REDUCE. For these, a root is specified. The root, after sending its initial message to C, then receives a message from each non-root process, in rank order. The root updates its local state (the receive buffer) as these messages come in, according to the semantics of the particular function and the abstractions employed. After the last message is received, the root sends a 1 bit to C.

A non-root process sends its initial message to C, then sends its relevant data to the root on the collective channel, and finally sends a 1 bit to C.

C receives all the initial messages in rank order, and then receives all the 1 bits in rank order. That is all C does—it does not compute anything or send any messages out.

The purpose of sending the 1 bits to C at the end is to allow the possibility that this communication forces a barrier to occur. For, if the sending of a 1 from process i to C is forced to synchronize, then process i will not be able to leave the communication until every process has entered it. On the other hand, all three sends performed by a non-root process may be buffered, in which case that process can enter and leave the communication before any other process has entered it. Hence the full range of possibilities allowed by the Standard is accounted for in the model.

The third category consists of the "broadcast-like" functions MPI_BCAST, MPI_-SCATTER, and MPI_SCATTERV. The situation is dual to that of the gather-like functions. After sending its initial message to C, the root then sends a message to each non-root process in rank order. The root then sends a 1 bit to C.

A non-root process sends its initial message to C, then receives a message from the root, after which it immediately updates its local state (the receive buffer) accordingly. Finally, it sends a 1 bit to C.

C will receive all the initial messages in rank order, and then receive all the 1 bits in rank order. That is all C does. Again, this allows the possibility of creating a barrier. On the other hand, if all three sends performed by the root buffer, the root may enter and exit the communication before any other process has entered. Also, if the two sends performed by a non-root process buffer, then that process will be able to proceed as soon as the message from the root is sent, regardless of the state of the other non-root processes. Again, the full range of possibilities is accounted for.

The fourth and final category includes the "scan-like" functions MPI_SCAN and MPI_EXSCAN. Each process, after sending its initial message, receives a result from C. After receiving this result, the process updates its local state and sends a 1 bit to C.

C loops from 0 to $n-1$ receiving the initial messages. Immediately after receiving the initial message from process i, C computes the partial sum (or whatever function is specified for the scan) and passes the partial result back to process i (before moving on to receiving the initial message from process $i+1$). After all this is done, C receives a 1 bit from each process. This allows the possibility that process i can leave the communication as soon as all processes of lower rank have entered it—or it might act as a barrier, if the 1 bit communications are synchronized.

13. Local Determinism

13.1. **Locally Deterministic Models.** In this section, we will explore a particularly simple class of models, called *locally deterministic* models. We will show that many of the common questions concerning execution have simple answers for models in this class.

Definition 13.1. 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 with $\mathsf{src}(t) = u$.

Note that there may still be states in a locally deterministic model with more than one outgoing transition, namely, the receiving states, which have one transition for each possible message that could be received, and the send-receive states, which have an outgoing send transition as well as one outgoing receive transition for each message. All other states, however, will have at most one outgoing transition.

Theorem 13.2. *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 $p \in \mathsf{Proc}$, $T\downarrow_p \preceq S\downarrow_p$. In particular, any two execution prefixes for $\mathcal{M}$ are compatible.*

One consequence of Theorem 13.2 is that it is very easy to check many properties of a locally deterministic model. For, given such a model, we may "execute" it by constructing an execution prefix, one step at a time, making any buffering and interleaving choices we like. Let us assume for now that this terminates with every process in a final state. Then we know that any other execution prefix will follow the same paths through each state machine (except possibly taking dual paths from the send-receive states), though of course the way the events from the different processes are interleaved may differ. So if we wish to prove, for example, that certain local states are not reachable on any execution, we have only to check that this holds on this one execution.

We now turn to the proof of Theorem 13.2. The execution prefix S will be constructed using the following inductive procedure. Start by choosing any total order on Proc, say $\mathsf{Proc} = \{p_1, \ldots, p_N\}$. To begin, set $S = ()$, the empty sequence, and let $p = p_1$.

We define a sequence S' as follows. If there is no $s \in \mathsf{Trans}_p$ such that the sequence obtained by appending s to S is an execution prefix, let $S' = S$. Otherwise, we pick one such s as follows. Let U be the terminal state of S. If U_p is a local-event state, by hypothesis there is only one outgoing transition, so s is uniquely defined. The same is true if U_p is a sending state. If U_p is a receiving state, then there is precisely one channel c such that there is one outgoing transition, labeled $c?x$, for each $x \in \mathsf{msg}(c)$. Only one of these can possibly be appended to S to yield a new execution prefix—this is the one corresponding to the element x that is the first element in the message channel queue for c after the last step of S. Finally, if U_p is a send-receive state, let s be the transition departing from U_p labeled by a send. Now let S' be the sequence obtained by appending s to S.

Now let $S = S'$, $p = p_2$, and repeat the paragraph above. Continue in this way, and after p_N cycle back to p_1. Continue cycling around the processes in this way. If one ever passes through N processes in a row without adding a transition to S, then there can be no further extension of S (either because $U_p \in \mathsf{End}_p$ for all p, or because S is absolutely deadlocked) and the process stops with S finite. Otherwise, this yields a well-defined infinite execution prefix S.

We now make two observations concerning the execution prefix $S = (s_1, s_2, \ldots)$ defined above.

First, it cannot be the case that for some $p \in \mathsf{Proc}$, $S{\downarrow_p}$ is finite and terminates at a local-event, sending, or send-receive state. For, in any of these cases, the first time p is reached in the cyclic schedule after reaching this state, the local or send transition would be available to append to S.

Second, suppose for some $p \in \mathsf{Proc}$, $S{\downarrow_p}$ is finite and terminates at a receiving state u. Let n be the least integer such that $s_i \notin \mathsf{Trans}_p$ for all $i > n$. Let c be the receiving channel for the receive transitions departing from u. Then $\mathsf{Pending}_c(S^i)$ is empty for all $i \geq n$. For, if at some point the queue became non-empty, then the next time after that point when p is reached in the cyclic schedule, one of the receive transitions would be available to append to S.

We now show that S satisfies the conclusion of Theorem 13.2. Suppose there is an execution prefix $T = (t_1, t_2, \ldots)$ such that for some $p \in \mathsf{Proc}$, $T{\downarrow_p} \not\precsim S{\downarrow_p}$. It is not hard to see that there must exist $n > 0$ such that

$$T^n{\downarrow_p} \not\precsim S{\downarrow_p} \tag{4}$$

for some p. Choose n so that it is the least integer with this property, and replace T with T^n. Clearly (4) holds if p is the element of Proc for which $t_n \in \mathsf{Trans}_p$.

To summarize, we now have the following situation: there exist a finite execution prefix $T = (t_1, \ldots, t_n)$, a $p \in \mathsf{Proc}$, and for each $r \in \mathsf{Proc}$, a path

$$\pi_r = (s_1^r, s_2^r, \ldots)$$

through M_r and a non-negative integer $m(r)$ in the domain of π_r such that:

$$\pi_r \sim S{\downarrow_r} \text{ for all } r \in \mathsf{Proc} \tag{5}$$

$$T^{n-1}{\downarrow_r} = T{\downarrow_r} = (s_1^r, \ldots, s_{m(r)}^r) \text{ for all } r \in \mathsf{Proc} \setminus \{p\} \tag{6}$$

$$T^{n-1}{\downarrow_p} = (s_1^p, \ldots, s_{m(p)}^p) \tag{7}$$

$$T{\downarrow_p} \not\precsim \pi_p. \tag{8}$$

We will obtain a contradiction.

Let $n(r)$ denote the length of π_r, allowing the possibility that $n(r) = \infty$. Let $u = \mathsf{src}(t_n)$.

Suppose u is a local-event or a sending state. Then there is a unique transition departing from u. As we have seen, it is not possible that $n(p)$ is finite and $\mathsf{terminus}(\pi_p) = \mathsf{terminus}(S{\downarrow_p}) = u$. So we must have $n(p) > m(p)$ and $s^p_{m(p)+1} = t_n$. But this means $T{\downarrow_p}$ is a prefix of π_p, contradicting (8).

Suppose u is a send-receive state and that t_n is a send, say $\mathsf{label}(t_n) = c!x$. Since it is not possible that π_p terminates at a send-receive state, we must have $n(p) > m(p)$, and, considering (8), $s^p_{m(p)+1}$ is a receive. However, this means that $\mathsf{des}(s^p_{m(p)+1})$ is a sending state, so $n(p) > m(p) + 1$ and $s^p_{m(p)+2}$ is a send transition labeled $c!x$. Now let π'_p be the sequence obtained from π by replacing $s^p_{m(p)+1}, s^p_{m(p)+2}$ with the dual path. Then $\pi'_p \sim \pi_p$ and $T{\downarrow_p}$ is a prefix of π'_p, contradicting (8).

Suppose now that t_n is a receive, say $\mathsf{label}(t_n) = c?x$. We claim that for some $i \in \{1, 2\}$, $n(p) \geq m(p) + i$ and $s^p_{m(p)+i}$ is a receive on c. For if not, then $S{\downarrow_p}$ is finite and

$$\mathsf{Received}_c(\pi_p) \subset \mathsf{Received}_c(T),$$

whence

$$\begin{aligned}\mathsf{Received}_c(S) = \mathsf{Received}_c(S{\downarrow_p}) &= \mathsf{Received}_c(\pi_p)\\ &\subset \mathsf{Received}_c(T)\\ &\subseteq \mathsf{Sent}_c(T)\\ &= \mathsf{Sent}_c(T{\downarrow_q})\\ &\subseteq \mathsf{Sent}_c(\pi_q)\\ &= \mathsf{Sent}_c(S{\downarrow_q})\\ &= \mathsf{Sent}_c(S).\end{aligned}$$

In other words, $\mathsf{Received}_c(S)$ is a proper prefix of $\mathsf{Sent}_c(S)$. Moreover,

$$\mathsf{terminus}(S{\downarrow_p}) = \mathsf{terminus}(\pi_p)$$

is a receiving state on channel c. We have seen this is not possible, so our claim must hold.

Now let $q = \mathsf{sender}(c)$ and let k be the length of $\mathsf{Received}_c(T)$. Then

$$\begin{aligned}x = \mathsf{Received}_c(T)|_k = \mathsf{Sent}_c(T)|_k = \mathsf{Sent}_c(\pi_q)|_k &= \mathsf{Sent}_c(S{\downarrow_q})|_k\\ &= \mathsf{Sent}_c(S)|_k = \mathsf{Received}(S)|_k.\end{aligned}$$

Hence $\mathsf{label}(s^p_{m(p)+i}) = c?x$.

Now, if $i = 1$, then $T{\downarrow_p}$ is a prefix of π_p, while if $i = 2$, $T{\downarrow_p}$ is a prefix of a sequence $\pi'_p \sim \pi_p$. In either case, we have $T{\downarrow_p} \preceq \pi_p$, a contradiction, completing the proof of Theorem 13.2.

Corollary 13.3. *Suppose $\mathcal{M}$ is a locally deterministic model of an MPI program, and $\Sigma \subseteq \mathsf{Proc}$. Then $\mathcal{M}$ 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 $\mathsf{terminus}(T)_p \in \mathsf{End}_p$ for all $p \in \Sigma$.*

Proof. If $\mathcal{M}$ is Σ-deadlock-free then for any synchronous execution prefix T, either $\mathsf{terminus}(T)_p \in \mathsf{End}_p$ for all $p \in \Sigma$ or T has a proper synchronous extension. So we may construct the required T inductively by beginning with the empty sequence and applying this fact repeatedly.

Now suppose such a prefix T exists and we wish to show that $\mathcal{M}$ is Σ-deadlock-free. By Theorem 7.4, it suffices to show that $\mathcal{M}$ has no synchronous execution prefix S that is potentially Σ-deadlocked. So suppose S is a finite execution prefix with $\mathsf{terminus}(S)_p \notin \mathsf{End}_p$ for some $p \in \Sigma$. We must show that S has a proper synchronous extension.

If T is infinite, then there is an $n > |S|$ such that T^n is synchronous. Thus, for some $r \in \mathsf{Proc}$,

$$|T^n{\downarrow_r}| > |S{\downarrow_r}|.$$

By Theorem 13.2, S and T^n are compatible. So by Proposition 6.2, there exists a synchronous extension S' of S with the property that $T^n{\downarrow_q} \preceq S'{\downarrow_q}$ for all $q \in \mathsf{Proc}$. Hence

$$|S{\downarrow_r}| < |T^n{\downarrow_r}| \leq |S'{\downarrow_r}|,$$

which shows that S' is a proper extension of S, as required.

If T is finite and $\mathsf{terminus}(T)_r \in \mathsf{End}_r$ for all $r \in \Sigma$, then we may apply Proposition 6.2 directly to S and T to conclude there is a finite synchronous extension S' of S with $T{\downarrow_q} \preceq S'{\downarrow_q}$ for all $q \in \mathsf{Proc}$. This implies, in particular, that $\mathsf{terminus}(S')_p \in \mathsf{End}_p$, which shows that S' must be a proper extension of S. □

The practical consequence of Corollary 13.3 is that it is very easy to check freedom from deadlock for a locally deterministic model. For given such a model, we may simply "execute" the model synchronously by constructing a synchronous execution prefix one step at a time, making any interleaving choices we like (as long as a send is always immediately followed by its matching receive). This procedure will either (1) terminate with every process at a final state, (2) terminate at a potentially deadlocked state, or (3) not terminate. If (2) is the case, then we have obviously found a deadlock. It is a consequence of Corollary 13.3 that if (1) or (3) is the case, then the model is in fact deadlock-free.

There is an analogous result for partial deadlock:

Corollary 13.4. *Suppose $\mathcal{M}$ is a locally deterministic model of an MPI program, and $\Sigma \subseteq \mathsf{Proc}$. Then $\mathcal{M}$ 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* $\mathsf{terminus}(T{\downarrow_p}) \in \mathsf{End}_p$.

Proof. If $\mathcal{M}$ is free of partial Σ-deadlock then for any synchronous execution prefix T and $p \in \Sigma$, either $\mathsf{terminus}(T)_p \in \mathsf{End}_p$ or T has a synchronous extension T' with $|T'{\downarrow_p}| > |T{\downarrow_p}|$. So we may construct the required T inductively by beginning with the empty sequence and then cycling repeatedly through all elements of Σ, applying this fact.

Now suppose such a prefix T exists and we wish to show that $\mathcal{M}$ is free of partial Σ-deadlock. By Theorem 7.7, it suffices to show that $\mathcal{M}$ has no synchronous execution prefix S that is Σ-ppd. So suppose S is a finite execution prefix, $p \in \Sigma$, and $\mathsf{terminus}(S)_p \notin \mathsf{End}_p$. We must show that S has a synchronous extension S' with $|S{\downarrow_p}| < |S'{\downarrow_p}|$.

If $T{\downarrow_p}$ is infinite, then there is an $n > 0$ such that T^n is synchronous and

$$|T^n{\downarrow_p}| > |S{\downarrow_p}|.$$

By Theorem 13.2, S and T^n are compatible. So by Proposition 6.2, there exists a synchronous extension S' of S with the property that $T^n{\downarrow_q} \preceq S'{\downarrow_q}$ for all $q \in \mathsf{Proc}$. Hence

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

as required.

If $T{\downarrow_p}$ is finite and $\mathsf{terminus}(T{\downarrow_p}) \in \mathsf{End}_p$, then for some $n > 0$, T^n is synchronous and $\mathsf{terminus}(T^n{\downarrow_p}) \in \mathsf{End}_p$. By Proposition 6.2, there is a finite synchronous extension S' of S with $T^n{\downarrow_q} \preceq S'{\downarrow_q}$ for all $q \in \mathsf{Proc}$. This implies, in particular, that $\mathsf{terminus}(S')_p \in \mathsf{End}_p$, which shows that $|S'{\downarrow_p}| > |S{\downarrow_p}|$, as required. □

13.2. **Locally Deterministic Programs.** Keep in mind that "locally deterministic," as used above, is a property of a model of a program, and not of the program itself. It is certainly possible to have two models (even conservative ones) of the same program for which one of those models is locally deterministic and the other is not. This might depend, for example, on the abstractions each model employs.

Let us describe a case where we can always create a locally deterministic model, and see what conclusions we can draw from this. Suppose we are given an actual

MPI program that uses only the MPI functions listed in §1. Assume that the program uses neither MPI_ANY_SOURCE nor MPI_ANY_TAG. And finally, assume it uses no non-deterministic functions.

This last condition requires clarification. For in one sense, any numerical operation, such as addition of floating point numbers, may be non-deterministic, since these operations may differ from platform to platform. However, it is almost always the case that, for a given platform, the numerical operations are deterministic functions, in the sense that, given the same inputs repeatedly, they will always return the same output. (The platform may also specify other information not included in the program itself, such as the number of bits used to represent a floating point number.) So we will allow numerical operations in our program; what we want to prohibit is, for example, a function that returns a random value, or any other function whose behavior is not a function solely of the program state.

By a slight abuse of terminology, we will call such an MPI program a *locally deterministic* program. Now when we use the term *locally deterministic* it will be important to specify if one is talking about a program, or a model of that program. For it is certainly possible for a locally deterministic program to have a model that is not locally deterministic, particularly if that model attempts to capture the behavior of the program on all possible inputs, or if the model makes generous use of abstractions.

However, given (1) a locally deterministic MPI program, (2) a fixed platform, and (3) the input to the program, we may always consider the "full precision" model. This is the model in which there is a state for every possible combination of values for all variables (including the program counter, the call stack, etc.) in each process. The collective functions are modeled using the procedure described in §12 above. Our assumptions ensure that this model will be locally deterministic, and so the results of §13.1 apply to it. Moreover, an execution of the program essentially corresponds to our notion of an execution prefix of this model.

Say, for example, we execute the program once on the given platform and with the given input and that every process of the program terminates normally. Then we may use Theorem 13.2 to conclude that, if run again on that platform and with the same input, each process will always perform the same computation (assuming the program does not deadlock). The execution steps may be interleaved differently on different executions, but for a fixed process, the sequence of events and values computed will be exactly the same, and the messages sent and received will be the same, on every execution.

A check for deadlock could be performed using an MPI implementation with a "synchronous mode," i.e., an option to execute an MPI program while forcing all communication to take place synchronously. Executing a locally deterministic program using this mode would essentially be equivalent to constructing a synchronous execution of the program's "full precision" model. Hence if we execute the program synchronously on particular input and platform, and it either terminates normally or runs forever, then we can conclude the program will never deadlock when run on that platform, with that input. In fact, for programs which use only the standard mode blocking send and receive operations, one can accomplish the same thing by replacing all the sends with the synchronous mode send—a common practice among developers of MPI software. Unfortunately, MPI does not provide for synchronous versions of the collective operations, or for the send-receive functions, so one really would need an MPI implementation with a synchronous execution mode in order to perform this test generally.

Of course, it is possible that when given other inputs the program may deadlock. This may happen, for example, if the program branches on a condition that is a function of one of the inputs. However, it is sometimes the case that we can use the methods of §13.1 to prove that a program is deadlock-free on any input. We may be able to do this by abstracting away the values of the inputs when building the (still conservative) model. If there are no branches that rely on the input values then with suitable abstractions the model we build may be locally deterministic. In this case we need only check that one synchronous execution of this model is deadlock-free, and then we have shown that any execution of the program, with any inputs, will not deadlock.

The concept of "platform" may be broadly construed. For example, one platform might consist of a particular C compiler with a particular operating system and machine architecture. However, we may also consider theoretical platforms: for example, the platform in which the values of all floating point variables are considered to be actual real numbers, integer variables actual integers, and so on. It may not be unreasonable to require that the program behave correctly on this kind of platform. Another example would be a platform in which all computations are performed symbolically. In this scenario, all input and initial values may be represented by symbols, and then an operation such as addition of two values `x` and `y` would return a tree structure of the form `(+ x y)`. Such a representation would allow one to reason about very specific aspects of the calculations performed by the program. In some cases it may be possible to build locally deterministic models using this platform.

Of course, it is still possible that the MPI program might behave differently on two different platforms, due to, say, differences in the numerical operations. The same could be true of even a sequential program. The results of this section simply do not address this sort of non-determinism. We have addressed instead the source of non-determinism that arises from the choices available in buffering messages, and in interleaving the execution steps of the various processes. What we have shown is that, if we restrict to a certain subset of MPI, this source of non-determinism can have no bearing on the eventual outcome of execution of that program.

14. Examples

In this section, we illustrate how the theorems of the previous sections can be applied to two simple MPI programs, called Ring and Noexit.

14.1. **Ring.**

The source code for Ring is shown in Figure 10. This program computes the minimum and maximum of a set of values, where there is one value on each process (cf. [1, Figure 7.13]). The input is a single integer for each process. Data flow around the processes in a cyclical order. Process 0 begins by sending out its value as the current best estimate of the maximum and minimum. Process 1 receives these and updates them by comparing each to its value, and then passes along the revised estimates to process 2. This continues until the two estimates come back to process 0, at which point they are guaranteed to be the actual maximum and minimum. The two values must be sent around the ring a second time so that every process obtains them. The program may be thought of as an implementation of

```
   int main(int argc,char *argv[]) {
     int nprocs, myrank, value, extrema[2];
     MPI_Status status;

     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
0:   read_input(&value);
     if (myrank == 0) {
       extrema[0] = extrema[1] = value;
1:     MPI_Send(extrema, 2, MPI_INT, 1, 9, MPI_COMM_WORLD);
2:     MPI_Recv(extrema, 2, MPI_INT, nprocs-1, 9, MPI_COMM_WORLD, &status);
3:     MPI_Send(extrema, 2, MPI_INT, 1, 9, MPI_COMM_WORLD);
     } else {
4:     MPI_Recv(extrema, 2, MPI_INT, myrank-1, 9, MPI_COMM_WORLD, &status);
5:     if (value < extrema[0]) extrema[0] = value;
       if (value > extrema[1]) extrema[1] = value;
6:     MPI_Send(extrema, 2, MPI_INT, (myrank+1)%nprocs, 9, MPI_COMM_WORLD);
7:     MPI_Recv(extrema, 2, MPI_INT, myrank-1, 9, MPI_COMM_WORLD, &status);
       if (myrank < nprocs - 1)
8:       MPI_Send(extrema, 2, MPI_INT, myrank+1, 9, MPI_COMM_WORLD);
     }
     MPI_Finalize();
9: }
```

FIGURE 10. Ring: calculates the minimum and maximum of the values over all processes. Input: one integer for each process. At termination, `extrema[0]` should be the minimum and `extrema[1]` the maximum.

MPI_ALLREDUCE (with two reduction operations computed simultaneously) using only the send and receive primitives.

We have labeled key control points of the program using the integers in the left margin of Figure 10. A label represents the point of control just before the execution of the statement that follows the label. We will use these labels as state components in two different models of Ring.

The first model is defined in the following way. Let n be a positive integer, representing the number of processes. We let $\mathcal{M}_0(\text{Ring}, n)$ be the model depicted in Figure 11. In this model, the values of the variables are not represented at all; a state simply corresponds to a position in the code. The messages sent in the program are all represented by the single bit 1 in the model. The local transitions have been left unlabeled since they are not relevant to any of the properties we wish to check here. (An intelligent model-checker would remove those transitions, as an optimization, but we leave them in for clarity.)

The second model is constructed as follows. Suppose that, in addition to n, we are also given a non-negative integer m, and we assume that the input values all lie between 0 and m (inclusive). We can then design a more precise but still finite-state model of Ring in which the values of the variables are represented in the state. We call this model $\mathcal{M}_1(\text{Ring}, n, m)$; it is depicted in Figure 12. The state

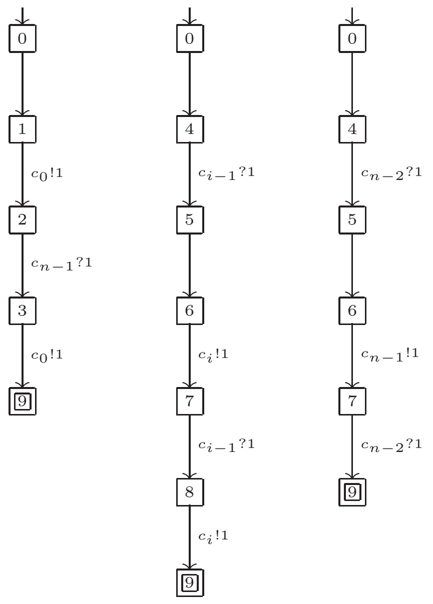


FIGURE 11. Model $\mathcal{M}_0(\text{Ring}, n)$: process 0 (left), process i ($1 \leq i \leq n-2$, middle), and process $n-1$ (right). Channel c_i sends from process i to $(i+1)\%n$.

vector consists of four components: the position label, and the values of `value`, `extrema[0]`, and `extrema[1]`.

Which model one should use depends on the property one wishes to verify. In general, it is preferable to use the most abstract conservative model for which the property of interest holds for all executions of that model. Suppose, for example, we want to show that Ring is deadlock-free. For this, the more abstract model suffices, because as we will see this model is deadlock-free (and even free of partial deadlock). In essence, the reason the abstract model suffices in this case is that the values of the data have no effect on the flow of control of the program, nor on the sequence of MPI functions invoked, and hence cannot have any bearing on whether the program deadlocks.

Suppose, on the other hand we wish to show that Ring always arrives at the correct result. That is, when a process is about to terminate, the value of `extrema[0]` is the minimum of the values, over all processes, of `value`, and that `extrema[1]` is the maximum. In this case, the abstract model clearly will not suffice, but the larger model will.

To show that $\mathcal{M}_0(\text{Ring}, n)$ is free of partial deadlock, we may apply Theorem 7.7 to reduce to synchronous communication. However, in this case we can do even better, since the model is locally deterministic. By Corollary 13.4, it suffices to produce a single synchronous execution prefix T such that for each $p \in \Sigma$, $T{\downarrow_p}$ is finite and $\mathsf{terminus}(T{\downarrow_p}) \in \mathsf{End}_p$. But there clearly are such sequences; in all of these the sequence of communication labels is as follows:

$$c_0!1, c_0?1, c_1!1, \ldots, c_{n-2}?1, c_{n-1}!1, c_{n-1}?1, c_0!1, c_0?1, c_1!1, \ldots, c_{n-2}?1.$$

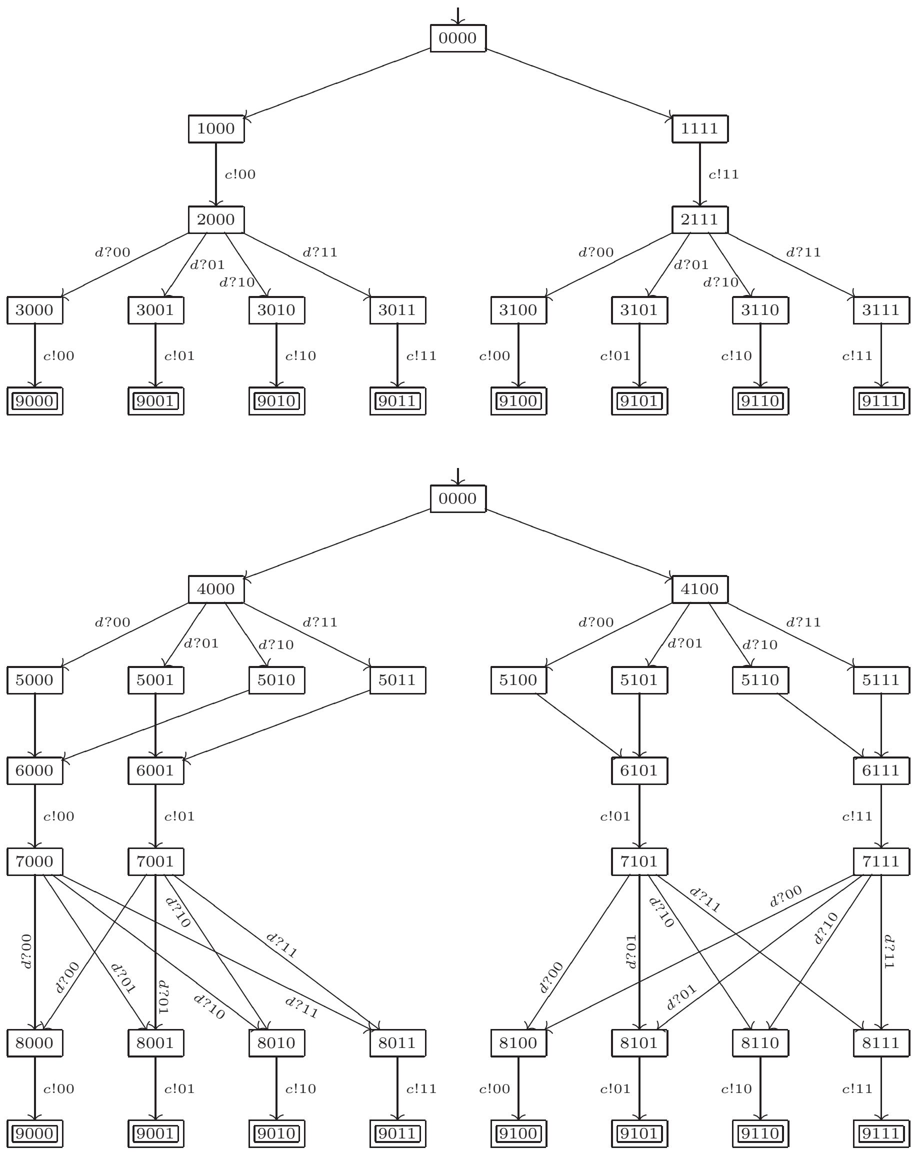


FIGURE 12. Model $\mathcal{M}_1(\text{Ring}, n, 1)$. The components of the state are position, `value`, `extrema[0]`, `extrema[1]`. For process 0 (top), $c = c_0$ and $d = c_{n-1}$. For process i ($1 \leq i \leq n-2$, bottom), $c = c_i$, $d = c_{i-1}$. Process $n-1$ is not shown, but is like process i with the last transition removed.

The local transitions may be interleaved in various ways, but do not present any difficulties.

To show that Ring always arrives at the correct result takes a bit more work. The property can be made precise as follows. For $\mathcal{M}_1(\text{Ring}, n, m)$, we wish to show that there is no execution prefix T such that (1) $\mathsf{terminus}(T{\downarrow_p})$ is an end-state for all p, and (2) for some p, the value of `extrema[0]` at $\mathsf{terminus}(T{\downarrow_p})$ is not the minimum, over all q of the value of `value` at $\mathsf{terminus}(T{\downarrow_q})$. (A similar statement holds for the maximum). Notice that the model in this case is not locally deterministic, because of the branches on the various inputs.

Now, if (1) holds, then it must be the case that there are no pending messages (i.e., messages sent but not received) after the last step in T. That is because the total number of sends equals the total number of receives: there are 2 sends and 1 receive in process 0, 1 send and 2 receives in process $n-1$, and 2 sends and 2 receives in the remaining processes. Since we already know that the model is free of partial deadlocks, we may apply Theorem 8.1 to conclude that, if there exists such a prefix T, then there must also exist a synchronous prefix S terminating in the same state. In other words, we have reduced the verification of this property to the synchronous case.

Finally, for given input, consider the full model for that particular input. Observe that this model is locally deterministic. The same result must be reached on any two executions. In other words, the result is a function solely of the input, and does not depend in any way on buffering or interleaving choices made by the MPI implementation. Hence if we test the program once on a given input and it returns the correct result, we are guaranteed that it will return the correct result every time it is executed on that input.

A final observation involving barriers may be unrealistic but illustrates the meaning of Theorem 11.1. Suppose we modify the source code for Ring by inserting barrier statements immediately after the first sends in both branches of the `if` statement (i.e., after lines 1 and 6). We wish to determine whether this modification could introduce a deadlock. If we test the program, it may perform as expected, without deadlocking, every time. The sequence of events in a successful execution might look something like the following: process 0 executes its first send and then enters the barrier; process 1 receives the message from 0, updates `extrema`, sends to process 2, and then enters the barrier; ...; process $n-1$ receives, updates, sends to 0, and enters the barrier. At this point all processes are "inside the barrier" and so they all may exit the barrier. Then process 0 receives the message from $n-1$, and execution continues normally until termination. However, observe that in this sequence, when all processes are inside the barrier there is a pending message—sent from process $n-1$ to process 0. By Theorem 11.1, the existence of such a prefix implies that the model must contain a partial deadlock. In this case it is easy to see the deadlock: the prefix in question would have resulted in deadlock if the send from process $n-1$ to 0 had been forced to synchronize.

14.2. **Noexit.** The source code for Noexit is shown in Figure 13. This program uses a "coordinator" process to coordinate a simultaneous exit from a loop. The input is a sequence of integers at each process of positive rank. Each of these processes repeatedly reads an integer and then sends it to process 0. Process 0 then returns the integer 0 if all of the integers read were 0, else it returns 1. If the value returned by process 0 is 0, all processes break out of their loops and terminate.

As in the case of Ring, we may make an abstract model $\mathcal{M}_0(\text{Noexit}, n)$ of this program which does not keep track of the values of any of the variables; this is depicted on the left side of Figure 14. However, unlike the case for Ring, this model will not suffice to verify freedom from deadlock. That is because the model actually does contain a potentially-deadlocked prefix, but that prefix does not correspond

```
   int main(int argc,char *argv[]) {
     int nprocs, myrank, i, sbuf[1], rbuf[1];
     MPI_Status status;

     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
     if (myrank == 0) {
       while (1) {
         sbuf[0] = 0;
         for (i = 1; i < nprocs; i++) {
0:         MPI_Recv(rbuf, 1, MPI_INT, i, 9, MPI_COMM_WORLD, &status);
           if (rbuf[0] > 0) sbuf[0] = 1;
         }
         for (i = 1; i < nprocs; i++)
1:         MPI_Send(sbuf, 1, MPI_INT, i, 9, MPI_COMM_WORLD);
2:       if (sbuf[0] == 0) break;
       }
     } else {
       while (1) {
         rbuf[0] = 1;
3:       read_input(&(sbuf[0]));
4:       MPI_Send(sbuf, 1, MPI_INT, 0, 9, MPI_COMM_WORLD);
5:       MPI_Recv(rbuf, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, &status);
6:       if (rbuf[0] == 0) break;
       }
     }
     MPI_Finalize();
9: }
```

FIGURE 13. Noexit: each process of positive rank reads a sequence of integers; the program terminates when all read 0.

to an actual execution of the program. In this case, the flow of control depends in a significant way on the values of variables, and a model that abstracts away those values altogether does not contain enough information to prove the property of interest.

On the other hand, it is not necessary to represent the variables in their full precision. Instead, we may use a model like $\mathcal{M}_1(\text{Noexit}, n)$, depicted on the right side of Figure 14. In this model, we have abstracted the variables `sbuf[0]` in the processes of positive rank: the symbol `Z` represents a 0 value for that variable while the symbol `N` represents any non-zero value.

Theorems 7.4 and 7.7 allow us to assume communication is synchronous while verifying freedom from (partial) deadlock. If we use an automated model checker, this assumption should result in a significant savings, in terms of the number of states explored (and therefore the time and memory consumed). One reason for this is that the number of distinct execution prefixes is much greater for the buffered case. For example, if buffering is allowed, all of the processes of positive rank may send their messages to process 0 before process 0 receives any of them. On the other hand, the sends and receives could alternate, or any interleaving in between these two extremes could take place. Another reason that a savings should occur is that

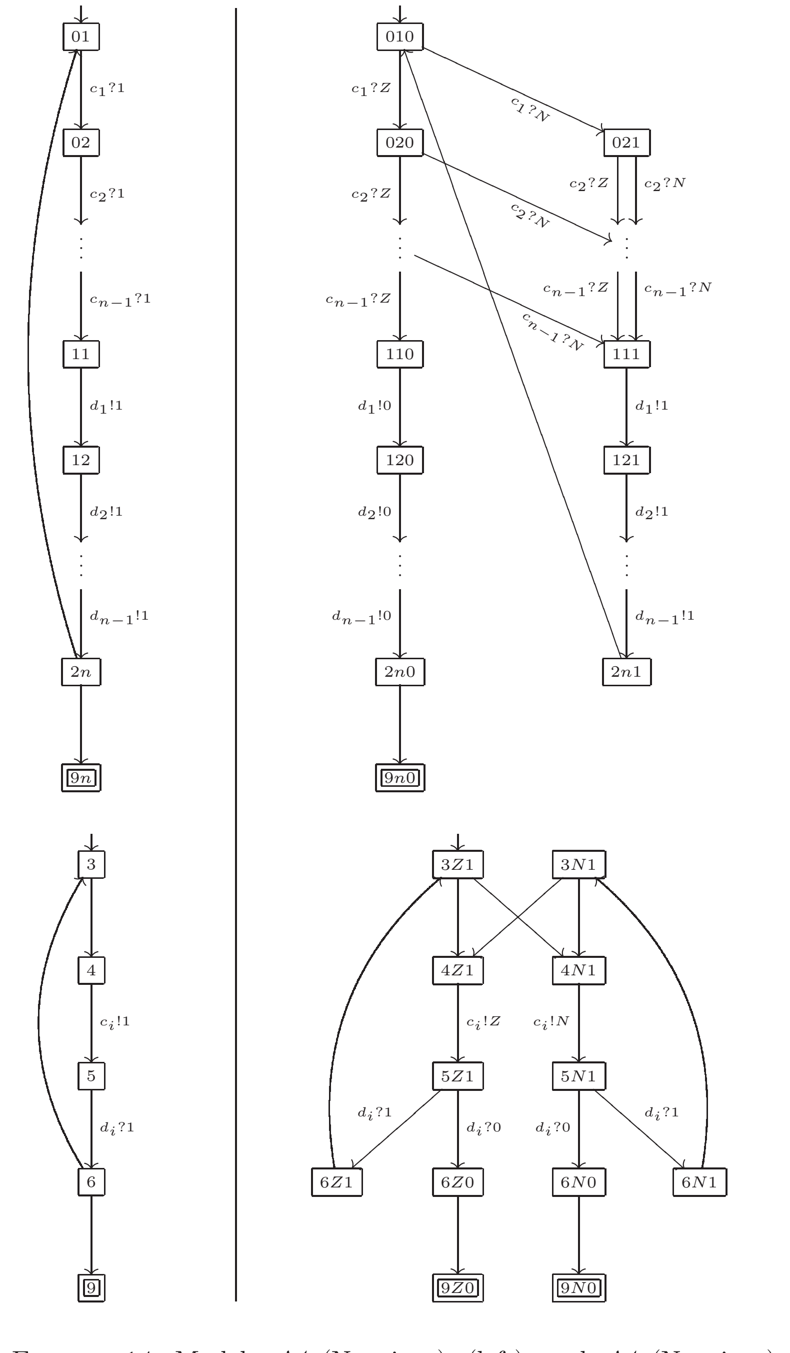


FIGURE 14. Models $\mathcal{M}_0(\text{Noexit}, n)$ (left) and $\mathcal{M}_1(\text{Noexit}, n)$ (right). Channel d_i sends from process 0 (top) to process i (bottom, $1 \leq i \leq n-1$); c_i from i to 0. The first component of the state vector is position. The additional components are `i` (upper left); `i`, `sbuf[0]` (upper right); and `sbuf[0]`, `rbuf[0]` (lower right).

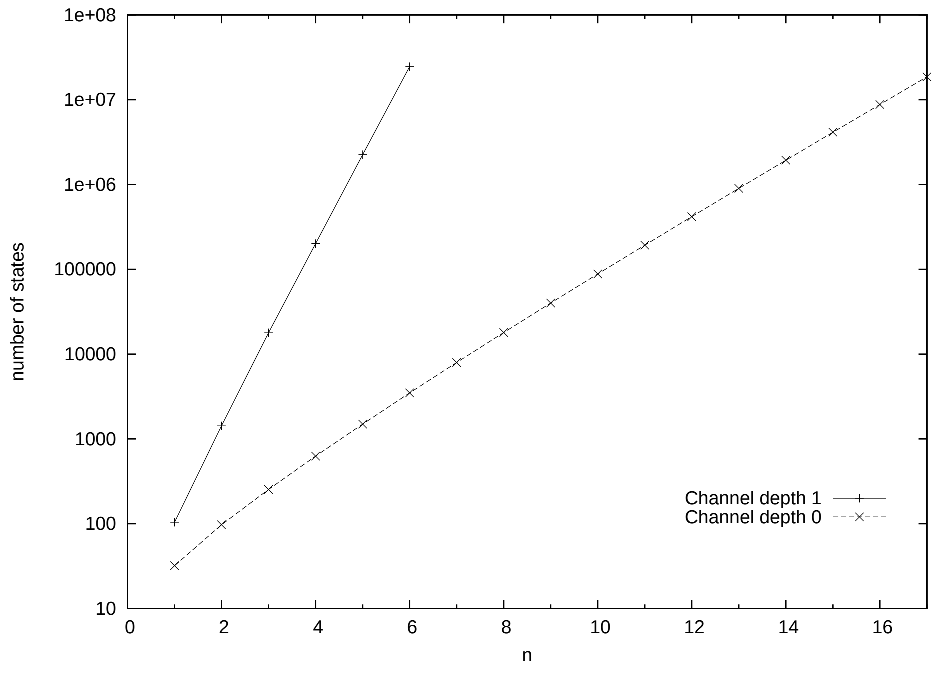


FIGURE 15. SPIN verification of freedom from deadlock for $\mathcal{M}_1(\text{Noexit}, n)$.

in the buffered case the model checker must choose, non-deterministically, whether or not to buffer each send. For synchronous communication there is obviously no choice to be made.

In this case, the savings can be easily investigated with a simple experiment using the model checker SPIN. We created two SPIN models of the code in Figure 13. In one we used channels of depth 1, and the `MPI_Send` statements were translated by first sending the message and then making a non-deterministic choice between proceeding without blocking or waiting until the message was received. In the other we used channels of depth 0, i.e., synchronous communication. For both, we used the same data abstraction that we used in constructing $\mathcal{M}_1(\text{Noexit}, n)$. We then used SPIN to verify freedom from deadlock for both models, steadily incrementing n, and recorded the number of states explored by SPIN in each case. We proceeded in this way until we exhausted 2 GB of RAM. For all runs, we also used the SPIN options `-DSAFETY`, an optimization that can be used for safety properties, and `-DCOLLAPSE`, which provides stronger compression than the default.

The results are shown in Figure 15. We have used a logarithmic scale for the y-axis, and since the graphs of both functions appear to be almost straight lines, both are closely approximated by exponential functions. But the base of the exponential function is clearly much lower for the channel depth 0 case, leading to an enormous and increasing savings over the depth 1 case, and therefore allowing the analysis to scale much further before running out of memory.

15. Related Work

Finite-state verification techniques have been applied to various message-passing systems almost from the beginning and Spin, of course, provides built-in support for a number of message-passing features. Various models and logics for describing message-passing systems (e.g., [2, 15]) are an active area of research. But only a few investigators have looked specifically at the MPI communication mechanisms. Georgelin et al. [6] have described some of the MPI primitives in LOTOS, and have used simulation and some model checking to study the LOTOS descriptions. Matlin et al. [13] used Spin to verify part of the MPI infrastructure.

Our theorems about MPI programs depend on results about the equivalence of different interleavings of events in the execution of a concurrent program. Our results are related to the large literature on reduction and atomicity (e.g., [3,5,11]) and traces [14]. Most of the work on reduction and atomicity has been concerned with reducing sequences of statements in a single process, although Cohen and Lamport [3] consider statements from different processes and describe, for example, a producer and consumer connected by a FIFO channel in which their results allow them to assume that messages are consumed as soon as they are produced. We intend to explore the extent to which their methods and results for TLA can be carried over to the MPI setting.

Manohar and Martin [12] introduce a notion of *slack elasticity* for a variant of CSP. Essentially, a system is slack elastic if increasing the number of messages that can be buffered in its communication channels does not change the behavior of the system. Their goal is to obtain information about pipelining for hardware design and the nondeterminism and communication constructs in their formalism are somewhat different from ours. The theorems they prove, however, are similar in many respects to ones we describe for MPI programs.

16. Conclusions

Future Work? Real Applications needing model extractors? Etc.? Generalize to wildcard receives?

References

[1] Andrews, G.R.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley, Reading, Massachusetts (2000)

[2] Bollig, B., Leucker, M.: Modelling, specifying, and verifying message passing systems. In Bettini, C., Montanari, A., eds.: Proceedings of the Symposium on Temporal Representation and Reasoning (TIME'01), IEEE Computer Society Press (2001) 240–248

[3] Cohen, E., Lamport, L.: Reduction in TLA. In Sangiorgi, D., de Simone, R., eds.: CONCUR '98. Volume 1466 of LNCS., Nice, Springer-Verlag (1998) 317–331

[4] Corbett, J.C., Avrunin, G.S.: Using integer programming to verify general safety and liveness properties. Formal Methods in System Design **6** (1995) 97–123

[5] Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In Cytron, R., Gupta, R., eds.: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, ACM Press (2003) 338–349

[6] Georgelin, P., Pierre, L., Nguyen, T.: A formal specification of the MPI primitives and communication mechanisms. Technical Report 1999-337, LIM (1999)

[7] Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI—The Complete Reference: Volume 2, the MPI Extensions. MIT Press, Cambridge, MA (1998)

[8] Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Boston (2004)

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

[10] M. Kaufmann and J. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Trans. Softw. Eng.*, 23(4):203–213, 1997.

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

[12] Manohar, R., Martin, A.J.: Slack elasticity in concurrent computing. In Jeuring, J., ed.: Proceedings of the Fourth International Conference on the Mathematics of Program Construction. Volume 1422 of LNCS., Marstrand, Sweden, Springer-Verlag (1998) 272–285

[13] Matlin, O.S., Lusk, E., McCune, W.: SPINning parallel systems software. In Bonaki, D., Leue, S., eds.: Model Checking of Software: 9th International SPIN Workshop. Volume 2318 of LNCS., Grenoble, Springer-Verlag (2002) 213–220

[14] Mazurkiewicz, A.: Trace theory. In Brauer, W., Reisig, W., Rozenberg, G., eds.: Petri Nets: Applications and Relationships to Other Models of Concurrency. Volume 255 of LNCS., Berlin, Springer-Verlag (1987) 279–324

[15] Meenakshi, B., Ramanujam, R.: Reasoning about message passing in finite state environments. In Montanari, U., Rolim, J.D.P., Welzl, E., eds.: Automata, Languages and Programming, 27th International Colloquium, ICALP 2000. Volume 1853 of LNCS., Geneva, Springer-Verlag (2000) 487–498

[16] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard (version 1.1). Technical report, 1995. `http://www.mpi-forum.org`.

[17] Message-Passing Interface Standard 2.0. `http://www.mpi-forum.org/docs/` (1997)

[18] 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.

[19] Siegel, S.F.: The INCA query language. Technical Report UM-CS-2002-18, Department of Computer Science, University of Massachusetts (2002)

[20] Siegel, S.F., Avrunin, G.S.: Improving the precision of INCA by eliminating solutions with spurious cycles. IEEE Transactions on Software Engineering **28** (2002) 115–128

[21] Siegel, S.F., Avrunin, G.S.: Analysis of MPI programs. Technical Report UM-CS-2003-036, Department of Computer Science, University of Massachusetts (2003)

[22] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI—The Complete Reference: Volume 1, The MPI Core. 2 ed. MIT Press, Cambridge, MA (1998)

Department of Computer Science, University of Massachusetts, Amherst, MA 01003
E-mail address: `siegel@cs.umass.edu`

Department of Computer Science, University of Massachusetts, Amherst, MA 01003
E-mail address: `avrunin@cs.umass.edu`