Verification of Halting Properties for MPI Programs Using Nonblocking Operations[*]

Stephen F. Siegel[1] and George S. Avrunin[2]

[1] Verified Software Laboratory, Department of Computer and Information Sciences,
University of Delaware, Newark, DE 19716, USA, `siegel@cis.udel.edu`,
`http://www.cis.udel.edu/~siegel`
[2] Laboratory for Advanced Software Engineering Research, Department of Computer
Science, University of Massachusetts, Amherst, MA 01003, USA,
`avrunin@cs.umass.edu`, `http://www.math.umass.edu/~avrunin`

Abstract. We show that many important properties of certain MPI programs can be verified by considering only a class of executions in which all communication takes place *synchronously*. In previous work, we showed that similar results hold for MPI programs that use only blocking communication (and avoid certain other constructs, such as `MPI_ANY_SOURCE`); in this paper we show that the same conclusions hold for programs that also use the *nonblocking* functions `MPI_ISEND`, `MPI_IRECV`, and `MPI_WAIT`. These facts can be used to dramatically reduce the number of states explored when using model checking techniques to verify properties such as freedom from deadlock in such programs.

1 Introduction

MPI includes nonblocking communication constructs, such as `MPI_ISEND` and `MPI_IRECV`, that can increase the parallelism of a program by allowing processes to overlap computation and communication. But this same parallelism can make programs hard to understand, test, and debug. In principle, model checking techniques [1] can explore all the possible executions of a parallel program and detect problems such as deadlock even if those problems arise only on rare and unusual execution paths, and these techniques have attracted growing interest from the scientific computing community. One of the main difficulties in the application of model checking, however, is the *state explosion problem*—the number of system states that the model checker must examine typically grows exponentially in the number of processes in the program. In particular, for MPI programs the buffering of messages makes an enormous contribution to this state explosion. In [13], we showed that many properties of a class of MPI programs using only the (standard mode) blocking communication constructs (`MPI_SEND`, `MPI_RECV`, etc.) could be verified by checking only synchronous executions, thereby eliminating the buffering and greatly expanding the range of model checking. In this paper, we extend those results to the nonblocking communication constructs.

In the next section, we explain how we model MPI programs and state a key lemma. In Sec. 3, we use the lemma to show that checking for deadlock can be restricted to synchronous executions, and in fact to an even more restricted type of synchronous execution. We present a few experimental results showing its impact on model checking using the model checker Mpi-Spin [10] in Sec. 4. The last section presents some conclusions and discusses future work.

2 Traces

In what follows, we let P be a *model of an MPI program* in which the only MPI functions used are MPI_INIT, MPI_FINALIZE, MPI_COMM_RANK, MPI_COMM_SIZE, MPI_ISEND, MPI_IRECV, and MPI_WAIT, and which does not use MPI_ANY_SOURCE. We call such a model *permissible*. (We will see below that the list of permitted functions can be expanded significantly.) The definition of such a model can be made mathematically precise in a variety of ways (see, for example, [10]). However, in the discussion here we will use a somewhat informal notion of model in order to emphasize the essential concepts underlying the theorem and its proof.

In essence, P may be thought of as an abstract version of an MPI program consisting of a fixed number of single-threaded processes. Each process has a unique integer *process ID* (pid); the rank of the process in MPI_COMM_WORLD could be used as the pid, for example. A process of P may use MPI_ANY_TAG and may even make nondeterministic choices. Such choices are common in models of MPI programs that have been created by abstracting the data and variables in the original program. Each process maintains a local *state*, which may be thought of an assignment of values to all variables in the process, including "invisible" variables such as the program counter.

An execution of P is considered to progress in a series of discrete atomic steps, each step corresponding to the execution of a statement by one process in P and possibly modifying the state of that process. In general, there are many different ways P can execute, because the steps from the different processes can be interleaved in various ways and a process can make nondeterministic choices. An execution can consist of a finite or (countably) infinite number of steps. Finite executions can stop at any point—it is not required that the processes terminate or become permanently blocked—and are sometimes called *execution prefixes*.

We think of each execution of P as leaving behind a *trace* which captures all information regarding that execution. We will treat the trace as a sequence of *events* that occur instantaneously and atomically when the processes execute certain actions. Each event contains complete information describing the change made to the system state. The events fall into the following categories:

1. A *post-send* event ps is generated when an MPI_ISEND statement returns. The information incorporated into this event includes the pid of the sending process, the pid of the destination process, the tag, and some unique identifier for the request object instantiated by the call, which we will call the *request id* (rid).

2. A *post-receive* event pr is generated upon the return of an MPI_IRECV. The event information includes the pid of the process executing the statement, the pid of the source, the tag (or MPI_ANY_TAG), and the rid.
3. An *enter-wait* event ew is generated just before executing a call to MPI_WAIT. The event information includes the pid of the process executing the call and the rid of the request being waited on.
4. An *exit-wait* event xw is generated when a call to MPI_WAIT returns. The event information includes the pid of the process executing the call and the rid of the request. If the request is for a receive operation, the data received is also included.
5. A *term* event term is generated just before a process terminates.
6. A *local* event local is generated for every statement that does not fall into one of the categories above (and hence does not involve communication). The event includes all information necessary to determine the change in local state resulting from execution of the statement.

We need both the ew and xw events since an MPI_WAIT call can block. To simplify the presentation we have not recorded MPI_INIT and MPI_FINALIZE in the trace. We assume those functions are invoked properly by each process.

We say that traces T and T' are *equivalent* if for each process p, the sequence of events from p are identical in the two traces and the state of p after the last event in T is the same as the state after the last event in T'. In particular, the two traces differ only in how events from different processes are interleaved.

Note that, since we have forbidden MPI_ANY_SOURCE, there is a unique way that send and receive operations can be paired by the MPI infrastructure, and this pairing relation is clearly discernible from the trace. An important point about the pairing relation is that it depends only on the order of the posting events within each process, and not in any way on how events from different processes are interleaved. We will say that a ps or pr event in T is *paired in T* if the pr or ps event with which it must be paired also occurs in T.

The rids allow us to determine which posting events correspond to which ew and xw events. Hence every communication involves at most six related entries: the ps and matching pr, the two corresponding ew events, and the two corresponding xw events. Of course, not all six events need occur. It is possible, for example, for a send to post and then complete without a matching receive ever posting, because the message emanating from the send can be buffered. On the other hand, a receive request can never complete if the related send has not posted. In fact, this last restriction is the only barrier to changing the interleaving of events from different processes. This is made precise by the following:

Lemma 1. *Suppose that T is a trace from P, and let e_i and e_{i+1} be consecutive events in T satisfying the following conditions:*

(i) The two events e_i and e_{i+1} do not come from the same process.
(ii) If e_{i+1} is an xw for a receive request, then e_i is not the matching ps.

Then the sequence obtained by transposing e_i and e_{i+1} is a trace of P equivalent to T.

Lemma 1 may be taken as an "axiom" following from the informal semantics described in the MPI Standard [4], or may be proved from a formal model such as the one described in [10].

3 Deadlock

Let T be a finite trace of a model P. A process p is in a *potentially blocked state* at the end of T if p has either terminated or the last event for p is an ew for which the corresponding posting event is not paired in T. The motivation for this terminology is the fact that the Standard permits—but does not require— an MPI implementation to block a (standard mode) send operation until the message can be delivered synchronously, i.e., until the receiving process has posted a matching receive. We say that T ends in a *potentially halted state* if at the end of T every process is potentially blocked. Hence in a potentially halted state it is not possible to progress unless, possibly, send operations are allowed to complete without their matching receives having been posted. To distinguish between the "good" case where all processes have terminated normally and the "bad" one where deadlock may occur, we say T ends in a *potentially deadlocked state* if T ends in a potentially halted state and furthermore at least one process has *not* terminated. We say P is *deadlock-free* if it has no trace ending in a potentially deadlocked state.

A trace T is *synchronous* if every xw event on a send request is preceded by the related pr. Hence a synchronous trace represents an execution in which no message buffering was required. We say P is *synchronously deadlock-free* if it has no synchronous trace ending in a potentially deadlocked state.

A synchronous trace $T = e_1, e_2, \ldots$ is *greedy* if, for all i, if e_i is an event from process p and e_{i+1} is an event from process $q \neq p$, then p is in a potentially blocked state at the end of $e_1, \ldots, e_i$. In other words, control switches from one process to another in T only when the first process becomes potentially blocked.

Theorem 1. *Let P be a permissible model of an MPI program and assume that no greedy trace of P ends in potential deadlock. Suppose that T is a finite trace from P ending in a potentially halted state. Then there exists a greedy trace T' from P which is equivalent to T.*

Proof. We will produce the greedy trace T' by modifying the interleaving of T. The modification proceeds in m stages numbered $1, \ldots, m$, where m is the number of events in T. Let $T_0 = T$ and let T_i denote the modified trace after stage i ($1 \leq i \leq m$). We will show by induction on i that T_i is equivalent to T and its prefix of length i is greedy. The case $i = 0$ is vacuously true and the case $i = m$ is the desired result.

Suppose $1 \leq i \leq m$ and let the event sequence T_{i-1} just before stage i be $e_1, \ldots, e_m$. By the induction hypothesis, we may assume that the prefix $R = e_1, \ldots, e_{i-1}$ is greedy. Stage i proceeds as follows. Let p be the process of e_{i-1}. (If $i = 1$ we may let p be any process.) Assume p satisfies the following criterion: there is an event from p in the suffix $S = e_i, \ldots, e_m$, and if the first such event

e is an xw then the posting event associated with e is paired in R. Then, by Lemma 1, the sequence T_i obtained by commuting e to the left until it is in position i is a trace equivalent to T_{i-1}, which the induction hypothesis tells us is equivalent to T. Moreover, the prefix of T_i of length i is greedy. So if p satisfies the criterion, we have completed the inductive step.

Now if p fails to satisfy the criterion, then it must be potentially blocked. For suppose that there are no events from p in S. Since T ends in a potentially halted state, that means p must either have terminated in R or ended with an ew for which one of the two related posts does not occur in R. In this case, we may choose any process q satisfying the criterion and proceed as before, and the resulting prefix of length i is still guaranteed to be greedy, and we have again completed the inductive step.

We are left with the possibility that no process p satisfies the criterion. In that case, every process is potentially blocked at the end of R, yet not every process has terminated, since there remain events in S. This means R is a greedy trace ending in potential deadlock, contradicting the hypothesis that no greedy trace ends in potential deadlock. □

Corollary 1. *Let P be a permissible model of an MPI program. Then P is deadlock-free if, and only if, P is synchronously deadlock-free.*

Proof. If P is deadlock-free then no trace ends in potential deadlock, so certainly no synchronous trace does. So suppose P is synchronously deadlock-free. Since any greedy trace is synchronous, no greedy trace of P can end in potential deadlock. Now if P had a trace T ending in potential deadlock, then by Theorem 1, P would have to have a greedy trace ending in potential deadlock, a contradiction. So P has no trace ending in potential deadlock, i.e., P is deadlock-free. □

While we have expressed Theorem 1 using a very limited subset of MPI, it is clear that a number of other functions can be safely added to that subset. For example, MPI_SEND is functionally equivalent to an MPI_ISEND followed immediately by an MPI_WAIT, and so can be included. The same goes for MPI_-RECV. MPI_SENDRECV may be replaced by the post of the send, followed by the post of the receive, followed by the two waits. MPI_SENDRECV_REPLACE may be expressed in a similar way, using a temporary buffer to handle the receive. All of the collective functions can be modeled using these basic point-to-point functions (see [11]) and so can be included as well. The functions MPI_-WAITALL, MPI_SEND_INIT, MPI_RECV_INIT, MPI_START, MPI_STARTALL, MPI_REQUEST_GET_STATUS, and MPI_REQUEST_FREE are also permissible.

It is instructive to see how the proof fails if P uses MPI_ANY_SOURCE. Consider the code in Fig. 1(a). In the sole synchronous trace of this program, every process terminates normally. If buffering is allowed, process 1 may send both messages, then process 2 may execute to completion, then the receive in process 0 may get paired with the message from process 2, and then process 0 will deadlock at the second receive. In attempting to apply the algorithm from the proof to convert this sequence into a synchronous one, one gets to a point where there is no process satisfying the criterion, but nevertheless the synchronous

```
if (rank==0) { MPI_Recv(...,MPI_ANY_SOURCE,...); MPI_Recv(...,2,...); }
else if (rank==1) { MPI_Send(...,0,...);  MPI_Send(...,2,...); }
else if (rank==2) { MPI_Recv(...,1,...);  MPI_Send(...,0,...); }
```
(a) Counterexample using MPI_ANY_SOURCE and 3 processes

```
if (rank==0) {
  MPI_Isend(...,1,tag0,...,&req[0]); MPI_Isend(...,1,tag1,...,&req[1]);
  MPI_Waitany(2,req,&i,...);
  if (i==0) { MPI_Recv(...,1,tag2,...); MPI_Wait(&req[1],...); }
    else { MPI_Wait(&req[0],...); }
} else if (rank==1) {
  MPI_Recv(...,0,tag0,...); MPI_Send(...,0,tag2,...);
  MPI_Recv(...,0,tag1,...);
}
```
(b) Counterexample using MPI_WAITANY and 2 processes

```
assert (rank == 0 || rank == 1);
MPI_Isend(..., 1 - rank, tag1, &req, ...);
MPI_Test(&req, &flag, ...); MPI_Barrier(...);
if (flag) { MPI_Recv(..., 1 - rank, tag2, ...); }
  else { MPI_Recv(..., 1 - rank, tag1, ...); MPI_Wait(&req, ...); }
```
(c) Counterexample using MPI_TEST and 2 processes

Fig. 1. Counterexamples to Corollary 1 for models using "impermissible" primitives

prefix is not potentially deadlocked. The reason is that the wildcard receive can be paired with a *different* send in order to escape from the deadlock. The proof of Theorem 1 depends on the fact that the way sends and receives are paired cannot be changed.

The functions MPI_TEST and MPI_WAITANY are impermissible. Counterexamples using these are given in Fig. 1(b,c).

4   Application

Theorem 1 has implications for model checking to verify properties such as deadlock-freedom: It is sufficient to use the model checker to explore all synchronous (or just all greedy) executions. If no potentially deadlocked state is found, then P is deadlock-free.

But Theorem 1 applies to more than just deadlock-freedom. Suppose we have established deadlock freedom for P (as above) and we wish to prove some property about the state of P at termination—for example, that the values computed by P are correct. If we can use the model checker to establish this correctness for all greedy traces, then Theorem 1 implies correctness must hold on any execution. For if there were some execution leading to an incorrect result then Theorem 1 says there must exist an equivalent greedy execution. Since the final values of all variables are the same in equivalent executions, the greedy

one will also be incorrect. Hence if the model checker can show that all greedy executions lead to a correct result, we can conclude that all executions do.

Mpi-Spin is an extension to the model checker Spin [2] that adds a number of features corresponding closely to the MPI primitives, making it much easier to model MPI programs for verification. Mpi-Spin can be used to verify properties such as deadlock-freedom and the correctness of the numerical results produced by P. The latter is accomplished by providing Mpi-Spin with both a sequential version of the program, assumed to be correct, and the parallel version and then using Mpi-Spin to show that the parallel and sequential versions must produce the same output on any input. The technique uses symbolic execution to model the numerical operations in the programs and is described in detail in [14]. Mpi-Spin can check properties of programs employing the nonblocking operations but uses a general checking algorithm that does not take advantage of the reductions made possible with Theorem 1. One can force Mpi-Spin to search only synchronous traces, however, by invoking it with the option `-buf=0`. If in addition one encloses the body of each process in an `atomic` block, then only greedy traces will be explored.

To illustrate the impact of the optimizations, we consider Example 2.17, "Use of nonblocking communications in Jacobi computation" from [15]. The program distributes a matrix by columns, maintaining appropriate ghost cell columns. At each iteration, a process first computes the new values for its left- and right-most columns, then posts two send and two receive requests to update the ghost cells, then computes the new values for its interior columns, and finally waits on the four requests before moving to the next iteration.

We considered two properties: deadlock-freedom and the functional equivalence of this parallel program to the simple sequential version (Example 2.12). We scaled n, the number of processes, and set the global matrix size to $3n \times 3n$ and the number of loop iterations to 3. For each n, we used Mpi-Spin to verify each property in three different ways: (1) allowing buffering of messages, (2) allowing only synchronous communication, and (3) allowing only greedy executions. The number of states for each search can be seen in Fig. 2, where each curve shows exponential growth as expected. The synchronous optimization certainly helped but restricting to greedy executions made a dramatic improvement and allowed the verification to scale much further. In fact, our theorem shows that we could look only at the greedy executions in which the processes execute (if not blocked) in some fixed order such as round-robin. This would presumably allow a much greater reduction in the number of states, but we do not see a straightforward way to implement that in Spin.

5 Conclusion

To the best of our knowledge, Matlin, Lusk, and McCune [3] were the first to apply model checking techniques to an MPI problem, using Spin to investigate a component of the MPI implementation MPICH. In [12,13] we showed how Spin could be used to verify properties of simple MPI-based scientific programs; we
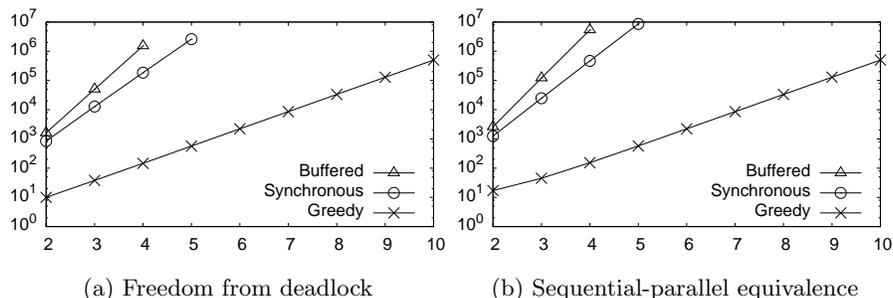
(a) Freedom from deadlock　　　(b) Sequential-parallel equivalence

Fig. 2. The number of states (y-axis) vs. number of processes (x-axis) for verifying two properties of the Jacobi iteration program

also presented theorems for countering state-explosion for MPI programs that used only blocking functions and no wildcard receives. These reduction results were generalized to deal with wildcard receives in [9]. Pervez, Gopalakrishnan, Kirby, Thakur, and Gropp [8] used SPIN to verify programs that use MPI's "one-sided" operations and found a subtle bug in one such program. The problem of verifying the numerical computations carried out by MPI programs was tackled by Siegel, Mironova, Avrunin, and Clarke in [14], which introduced the symbolic method for establishing the equivalence of sequential and parallel versions of a program. This method was incorporated into the MPI-SPIN tool, which was introduced in [10], along with a technique for modeling nonblocking functions.

Other very recent results appear to be closely connected to this work. In [6], Palmer, Gopalakrishnan, and Kirby introduce a model for a somewhat different subset of MPI and show how dynamic partial order methods allow checking properties by considering only a subset of the possible traces. Pervez, Gopalakrishnan, Kirby, Palmer, Thakur, and Gropp [7] have developed a model checking technique that works directly on source code, bypassing the model construction step. Using the modeling language TLA+, Palmer, Delisi, Gopalakrishnan, and Kirby [5] have developed a formal description of a large portion of the MPI Standard, and have integrated this into model checking tools. In future work, we will explore the connections between these approaches and ours.

In this paper we have generalized some of the earlier reduction theorems to the case of nonblocking functions, and demonstrated how these results can be used to improve the performance of MPI-SPIN. In particular, we have shown that many important properties of MPI programs that use the (standard mode) non-blocking operations can be verified by checking only a special class of executions in which no messages are buffered by the MPI infrastructure. A small example shows that this can provide a very substantial reduction in the resources required for verification, allowing model checking of significantly larger MPI programs. Our results do not hold for programs that make use of MPI_ANY_SOURCE, MPI_WAITANY, MPI_WAITSOME, MPI_TEST, MPI_TESTANY, or MPI_TEST-

SOME. We plan to investigate the generalization of the Urgent algorithm of [9] to handle these constructs.

References

1. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
2. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley, Boston (2004)
3. Matlin, O.S., Lusk, E., McCune, W.: SPINning parallel systems software. In Bosnacki, D., Leue, S., eds.: Model Checking of Software: 9th Intl. SPIN Workshop. Volume 2318 of LNCS., Springer-Verlag (2002) 213–220
4. Message Passing Interface Forum: MPI: A Message-Passing Interface standard, version 1.1. `http://www.mpi-forum.org/docs/` (1995)
5. Palmer, R., Delisi, M., Gopalakrishnan, G., Kirby, R.M.: An approach to formalization and analysis of message passing libraries. In: Proceedings of the 12th Intl. Workshop on Formal Methods for Industrial Critical Systems (FMICS). LNCS, Springer (2007) To appear.
6. Palmer, R., Gopalakrishnan, G., Kirby, R.M.: Semantics driven partial-order reduction of MPI-based parallel programs. In: Parallel and Distributed Systems: Testing and Debugging (PADTAD V), London (2007) To appear.
7. Pervez, S., Gopalakrishnan, G., Kirby, R.M., Palmer, R., Thakur, R., Gropp, W.: Practical model checking method for verifying correctness of MPI programs. In: Proceedings of the 14th European PVM/MPI Users' Group Meeting. LNCS, Springer (2007) This conference.
8. Pervez, S., Gopalakrishnan, G., Kirby, R.M., Thakur, R., Gropp, W.: Formal verification of programs that use MPI one-sided communication. In: Proceedings of the 13th European PVM/MPI Users' Group Meeting. Volume 4192 of LNCS. (2006) 30–39
9. Siegel, S.F.: Efficient verification of halting properties for MPI programs with wildcard receives. In Cousot, R., ed.: Verification, Model Checking, and Abstract Interpretation: 6th Intl. Conference (VMCAI 2005). Volume 3385 of LNCS. (2005) 413–429
10. Siegel, S.F.: Model checking nonblocking MPI programs. In Cook, B., Podelski, A., eds.: Verification, Model Checking, and Abstract Interpretation: 8th Intl. Conference (VMCAI 2007). Volume 4349 of LNCS. (2007) 44–58
11. Siegel, S.F., Avrunin, G.S.: Modeling MPI programs for verification. Technical Report UM-CS-2004-75, Department of Computer Science, University of Massachusetts (2004)
12. Siegel, S.F., Avrunin, G.S.: Verification of MPI-based software for scientific computation. In Graf, S., Mounier, L., eds.: Model Checking Software: 11th Intl. SPIN Workshop. Volume 2989 of LNCS., Springer-Verlag (2004) 286–303
13. Siegel, S.F., Avrunin, G.S.: Modeling wildcard-free MPI programs for verification. In: Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05), ACM Press (2005) 95–106
14. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Using model checking with symbolic execution to verify parallel numerical programs. In Pollock, L.L., Pezzé, M., eds.: Proceedings of the ACM SIGSOFT Intl. Symposium on Software Testing and Analysis (ISSTA 2006), ACM (2006) 157–168
15. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI—The Complete Reference, Volume 1: The MPI Core. Second edn. MIT Press (1998)