Verification of MPI-Based Software for Scientific Computation

Stephen F. Siegel and George S. Avrunin

Laboratory for Advanced Software Engineering Research, Dept. of Computer Science, University of Massachusetts, Amherst MA 01003, USA
{siegel,avrunin}@cs.umass.edu
http://laser.cs.umass.edu

Abstract. We explore issues related to the application of finite-state verification techniques to scientific computation software employing the widely-used Message-Passing Interface (MPI). Many of the features of MPI that are important for programmers present significant difficulties for model checking. In this paper, we examine a small parallel program that computes the evolution in time of a discretized function u defined on a 2-dimensional domain and governed by the diffusion equation. Although this example is simple, it makes use of many of the problematic features of MPI. We discuss the modeling of these features and use Spin and INCA to verify several correctness properties for various configurations of this program. Finally, we describe some general theorems that can be used to justify simplifications in finite-state models of MPI programs and that guarantee certain properties must hold for any program using only a particular subset of MPI.

1 Introduction

The advent of relatively cheap clustered computers and infrastructure supporting parallel programs on them has made supercomputers much more accessible and greatly expanded the range of application of scientific computing. Yet developers of parallel scientific software have encountered many problems that are familiar to specialists in verification of concurrent software: programs deadlock; they display inappropriate nondeterministic behavior; bugs are difficult to reproduce, let alone pinpoint and correct. Finite-state verification (FSV) techniques, such as model checking, can offer solutions to many of these problems, at least in principle. But typical scientific parallel programs pose special challenges for FSV techniques. For instance, in the widely used Message-Passing Interface (MPI) [13], the memory available for buffering messages between two processes, and thus the number of messages that can be buffered, can change dynamically and unpredictably during execution.

In this paper, we describe some first attempts at applying FSV techniques to a small, but realistic, example of a parallel scientific program that uses MPI. This program computes the evolution in time on a 2-dimensional domain of a discretized function governed by the differential equation known as the "diffusion" (or "heat") equation. We discuss some of the issues in modeling such programs and present the results of verifying several data-independent properties of the communication skeletons of various configurations of this program, using SPIN [7] and INCA [3,15]. Finally, we describe some theorems that can be used to justify simplifications in models of MPI programs, and that guarantee certain properties must hold for programs using only a particular subset of MPI.

In the next section, we briefly discuss the basic MPI constructs and introduce our example program, Diffusion2d. In section 3, we explain how we modeled this program for SPIN and present the results of our initial attempts to verify properties with SPIN. We then discuss modeling and verification with INCA. In section 5, we present the theorems. In section 6, we briefly describe some related work, and we discuss our conclusions and plans for future work in section 7.

2 The Message-Passing Interface and an Example Program

Most parallel scientific programs rely on *message-passing* for inter-process communication. The basic ideas of this paradigm 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*, 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, a number of high-quality proprietary and open-source MPI implementations have become available on almost any platform, and MPI has become the *de facto* standard for parallel scientific software.

2.1 MPI Basics

An MPI program consists of autonomous processes executing their own code in an MIMD style, although in practice the code executed by the different processes is often identical. The processes communicate by calls to functions implementing the MPI communication primitives. There are more than 140 such functions in the MPI-1 Standard. In this section, we briefly describe a few of the most basic ones, those used in our Diffusion2d example.

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.

The simplest function for sending a message from one process to another is the *standard mode, blocking send* function, with the form

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

where 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.

The simplest function for receiving a message is the *blocking receive* function, with 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. 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. An MPI receive operation will only select a message for reception if the source, tag, and communicator of the message match the corresponding receive parameters appropriately.

The MPI implementation may decide to buffer an outgoing message. On the other hand, the implementation may decide to block the sending process, perhaps until the size of the system buffer becomes sufficiently small, before sending out the message. Finally, the system may decide to block the sending process until the receiving process is at a matching receive, and there is no pending message in the system buffer that also matches that receive. If the implementation chooses this last option, we say that it forces this particular send to be *synchronous*.

The MPI Standard requires that messages be nonovertaking, in the sense that two messages from a single sender to single destination, both matching the same receive, can only be received in the order in which they were sent. But this is the only guarantee concerning the order in which messages are received—if two different processes send messages to the same process, the messages may be received in the order opposite to that in which they were sent.

MPI also provides *non-blocking* send and receive operations in which, for example, a sending process can continue execution even before the system has finished copying the message from the send buffer. 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.

It is common in MPI programs to have processes exchange data, either directly, when two processes send to and receive from each other, or more generally, as when each process in a grid sends to one neighbor and receives from another neighbor. In such cases, each process must execute one send and one receive, but if the program is coded so that each process first sends and then receives, the program will deadlock if, for example, the MPI implementation chooses to synchronize all the sends. The situation occurs frequently enough that MPI provides a special function that executes a send and a receive in one invocation, as if the process forks off two independent threads—one executing the send, the other the receive—and 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.

Finally, MPI provides a function MPI_BARRIER(comm) that is used to force all processes in the communicator comm to synchronize at a certain point.

2.2 Diffusion2d

Diffusion2d is a parallel program that computes the evolution in time of a discretized function u defined on a 2-dimensional domain and governed by the diffusion equation. It is the "teacher's solution" to a programming project for a course in Parallel Programming (taught by Andrew Siegel) at the University of Chicago. Though quite simple, it has many features common in scientific programs: the physical domain is divided into a "grid" in which one process is responsible for each section; each process also maintains a number of "ghost-cells" that mirror the contents of cells on neighboring processes. The program is written mostly in C, with some Fortran, and consists of 1036 lines spread out over six files. Slightly more than half this total consists of a generic package for dealing with grid structures; this package includes the `exchangeGhostCells` and `grid_write` functions defined below.

Our first task was to create a more compact and abstract representation of the code which captured the essential communication infrastructure. We follow the convention of [6, 17] in using uppercase sans serif type for the general functions, and mixed-case typewriter font for their C bindings. We also abbreviate the parameter lists by including only the send buffer and destination for a send, and the receive buffer and source for a receive, resulting in the following pseudo-code:

```
int nprocsx, nprocsy, nxl, nyl, nprint, nsteps;
int myproc, mycoord0, mycoord1;
int upperNabe, lowerNabe, leftNabe, rightNabe;
int[,] u;
int[] send_buf, recv_buf, buf;

void main() {
  int iter = 0;
  // read nprocsx, nprocsy, nxl, nyl, nprint, nsteps
  MPI_Init();
  myproc = MPI_Comm_Rank();
  mycoord0 = myproc % nprocsx; mycoord1 = myproc / nprocsx;
  upperNabe = mycoord0 + nprocsx*((mycoord1 + 1)%nprocsy);
  lowerNabe = mycoord0 + nprocsx*((mycoord1 + nprocsy - 1)%nprocsy);
  leftNabe = (mycoord0 + nprocsx - 1)%nprocsx + nprocsx*mycoord1;
  rightNabe = (mycoord0 + 1)%nprocsx + nprocsx*mycoord1;
  send_buf = new int[nxl]; recv_buf = new int[nxl]; buf = new int[nxl];
  u = new int[nxl+2,nyl+2];
  setInitialValues();
  exchangeGhostCells();
  MPI_Barrier();
  for (iter = 1; iter <= nsteps; ++iter) {
    update(u);
    exchangeGhostCells();
    if ((iter % nprint) == 0) grid_write();
  }
  MPI_Barrier();
  MPI_Finalize();
}
void exchangeGhostCells() {
  for (int i = 1; i <= nxl; ++i) send_buf[i-1] = u[i,1];
  MPI_Sendrecv(send_buf, lowerNabe, recv_buf, upperNabe);
  for (int i = 1; i <= nxl; ++i) u[i,nyl+1] = recv_buf[i-1];
  for (int i = 1; i <= nxl; ++i) send_buf[i-1] = u[i,nyl];
  MPI_Sendrecv(send_buf, upperNabe, recv_buf, lowerNabe);
  for (int i = 1; i <= nxl; ++i) u[i,0] = recv_buf[i-1];
  for (int j = 1; j <= nyl; ++j) send_buf[j-1] = u[1,j];
  MPI_Sendrecv(send_buf, leftNabe, recv_buf, rightNabe);
  for (int j = 1; j <= nyl; ++j) u[nxl+1,j] = recv_buf[j-1];
  for (int j = 1; j <= nyl; ++j) send_buf[j-1] = u[nxl,j];
  MPI_Sendrecv(send_buf, rightNabe, recv_buf, leftNabe);
  for (int j = 1; j <= nyl; ++j) u[0,j] = recv_buf[j-1];
}
void grid_write() {
  if (myproc != 0) {
    for (int n = 0; n < nprocsy; ++n)
      if (mycoord1 == n)
        for (int j = 1; j <= nyl; ++j)
          for (int m = 0; m < nprocsx; ++m)
            if (mycoord0 == m) MPI_Send(u[1..nxl,j], 0);
  } else {
    for (int n = 0; n < nprocsy; ++n)
      for (int j = 1; j <= nyl; ++j)
        for (int m = 0; m < nprocsx; ++m) {
          int from_proc = m + nprocsx*n;
          if (from_proc != 0) MPI_Recv(buf, from_proc);
          else for (int i = 0; i < nxl; ++i) buf[i] = u[i+1,j];
          disk_write(buf);
        }
  }
  MPI_Barrier();
}
```

Each process executes its own copy of this code, and all of the variables are local to this process. One process may take a different execution path through the code than another by branching on its rank. The total number of processes is specified by a flag to the MPI implementation when the program is executed.

The first six variables in the code are read from a file at the beginning of program execution; they act essentially as the parameters which define the geometry of the grid, the number of loop iterations, and the frequency with which the data are to be written to disk. (Actually, the original source code implicitly requires that $\texttt{nxl} = \texttt{nyl}$, so there are really only 5 parameters.) The processes themselves may be thought of as being arranged in a global $M \times N$ grid, where $M = \texttt{nprocsx}$ and $N = \texttt{nprocsy}$. The variable `myproc` is set to the rank r of this process; it is obtained from the MPI infrastructure via the function MPI_COMM_RANK. The position of this process on the global grid is $(m, n) = (\texttt{mycoord0}, \texttt{mycoord1})$, where $0 \leq m < M$, $0 \leq n < N$, and $r = m + nM$.

The next four variables are used to store the ranks of the neighboring processes. The grid "wraps" around in both the x- and y-directions, so every process has an upper, lower, left, and right neighbor. The four neighbors are not necessarily distinct. In fact, if $\texttt{nprocsx} = 1$ then this process will be its own left and right neighbor, and if $\texttt{nprocsy} = 1$ this process will be its own lower and upper neighbor. In these cases some of the MPI calls will actually send messages from a process to itself; this is allowed by MPI.

The variable `u` is used to store the values of the function that is evolving with time. It stores the values of this function for the coordinates that lie within the portion of the grid corresponding to this process. The dimensions of `u` are $(\texttt{nxl} + 2) \times (\texttt{nyl} + 2)$ because the top and bottom rows, and the left and right columns, are used to store the ghost cells—these are the cells that mirror the corresponding cells in the neighboring processes. (The four "corner" positions are not used.) The next three variables are used as temporary buffers for the MPI communication carried out in `exchangeGhostCells` and `grid_write`.

After initializing the variables, the basic structure of execution is relatively simple. First, a ghost-cell exchange is carried out. This updates the ghost cells by sending out the current values of the boundary cells to the appropriate neighbors and in turn receiving the values of their boundary cells, and storing the received values into the ghost-cell positions of `u`.

Next, a barrier is called, and then the main loop begins. In each iteration, first the values of `u[1..nxl,1..nyl]` are updated according to a formula derived from the discretization of the diffusion equation. This is purely a local function—it does not involve any MPI communication. Then a ghost-cell exchange takes place, and, if `iter` is divisible by `nprint`, `grid_write` is called.

The function `grid_write` is somewhat complicated. The goal is to write the cells to disk in the proper order: first, the entire global row 0 must be written, from left to right, then the same for global row 1, etc. To control the order, the data are sent, one local row at a time, from all of the processes of positive rank to the process of rank 0, which then writes to the disk. The nested loops encode a common MPI idiom for dealing with grid structures: `n` runs through global

y-coordinates of the processes; for each `n`, `j` runs through the local rows, and then `m` runs through the global x-coordinates of the processes. For a process of positive rank, action is taken only when $(\mathtt{n}, \mathtt{m})$ matches the global coordinates of this process.

Through discussions with the programmer, we arrived at several correctness properties for Diffusion2d. We describe three of these here: **Deadlock-Free**, **GlobalLockstep**, and **LocalLockstep**. The first says that the program can never deadlock. The second says that, for all $n > 0$, no process will call `update(u)` at step n unless every process has already completed `update(u)` at step $n-1$. The programmer pointed out that though he expected this property to hold, all that was actually required for correctness was the weaker property **LocalLockstep**—this says the same thing, but only for the four neighbors of a process.

In addition to these correctness properties, there were several questions concerning Diffusion2d that arose in our discussions with the programmer. One is **Q1**: *Could the removal of the* `MPI_Barrier` *statements from the code lead to a calculation that would not have occurred with the barriers, or could their removal lead to deadlocks if there were none before?* The ability to answer questions of this sort could be very valuable to scientific programmers: barriers can take a huge toll on the performance of a parallel program, but it is often difficult to reason about them informally. Another is **Q2**: *Are the final values written to disk independent of the interleaving or buffering choices made by the MPI implementation?* The expectation is certainly that this should be so, but it is not obvious, a priori, why that must be.

3 Applying Spin

3.1 Modeling the MPI Communication Primitives

Our first task was to model precisely the MPI communication functions. This task was simplified somewhat by the fact that Diffusion2d uses only one tag, and never makes use of the wildcards MPI_ANY_SOURCE or MPI_ANY_TAG. For these reasons, we could simply use one Spin channel `chan_`i`_`j, for every ordered pair of processes (i, j), to transfer messages from process i to process j.

Of course we must place a bound on the size of the channels. We call this `chan_size`. A priori, this means that our model fails to be conservative, in that the number of pending messages sent from one process to another can never exceed `chan_size` in the model, whereas MPI imposes no such bound. However, we will see that in certain cases we can justify a particular choice for `chan_size`.

We allow for the possibility that `chan_size` $= 0$, i.e., for the case where all communication is forced to take place synchronously. In this case the definitions of send and receive coincide exactly with those of Spin:

```
inline MPI_Send(schan, msg) { schan!msg }   /* chan_size = 0 */
inline MPI_Recv(rchan, rbuf) { rchan?rbuf }
```

A process i wishing to send a message `msg` to process j calls `MPI_Send` with `schan` = `chan_`i`_`j. The receive is used in a similar way, and there `rbuf` is a variable, representing the receive buffer, into which the incoming message will be stored.

If `chan_size` > 0 the situation is a little more complicated. The semantics of Spin channels guarantees that each channel will maintain the messages in FIFO order. However, recall that the MPI Standard allows for the MPI implementation to choose, at any time, to force a particular communication to synchronize. To capture this behavior in our model, we leave the receive procedure as is, but we modify the send procedure by inserting a statement that may block until the channel is empty:

```
inline MPI_Send(schan, msg) {  /* chan_size > 0 */
  schan!msg; if :: 1 -> empty(schan) :: 1 fi }
```

We represent MPI_SENDRECV by allowing the send and receive to happen in either order. Again, we must also allow for the possibility that the send is synchronized. For `chan_size` > 0 we hence arrive at the following:

```
inline MPI_Sendrecv(schan, msg, rchan, rbuf) { /* chan_size > 0 */
  if :: schan!msg -> rchan?rbuf :: rchan?rbuf -> schan!msg fi;
  if :: 1 -> empty(schan) :: 1  fi }
```

The case `chan_size` $= 0$ must be dealt with separately because of the possibility that `schan` = `rchan`, i.e., the case where process i attempts to send/receive a message to/from itself. The MPI Standard guarantees that the send and receive must be able to take place synchronously if there are no pending messages (sent from i to i). This situation requires no special handling in the positive `chan_size` case because, if there are no pending messages, the channel will be empty, and so the message can first be placed in the channel and then received. However, for `chan_size` $= 0$, the procedure defined above would always block, when in fact it should always be able to complete without blocking. Hence we modify the definition in this case so that the message is just placed directly into the receive buffer:

```
inline MPI_Sendrecv(schan, msg, rchan, rbuf) { /* chan_size = 0 */
  if :: (schan == rchan) -> rbuf = msg :: else ->
     if :: schan!msg -> rchan?rbuf :: rchan?rbuf -> schan!msg fi
  fi }
```

For `chan_size` > 0, we will refer to the procedures above as the *complex communication model*. We will also consider the *simple communication model*, in which we just remove the potentially blocking statement from the send and send-receive procedures. The effect of this will be that some statements in the simple model will not block whereas they might when the actual MPI program is executed. However, we will see that in some cases, the simple model will suffice for verifying certain properties, and can also improve the performance of Spin.

Finally, we turn to MPI_BARRIER. There are many standard ways to model a barrier; we chose a simple "coordinator barrier" approach. This entailed introducing an extra process with which an ordinary process interacts when it wishes to enter and exit a barrier. A special synchronous channel is used exclusively for this purpose. A 0 is sent on this channel when the process enters the barrier, and then a 1 is sent when the process attempts to exit the barrier. The barrier function waits to receive one 0 for each original process before accepting any 1s:

```
chan barrier_chan = [0] of {bit};
inline MPI_Barrier() { barrier_chan!0; barrier_chan!1 }
active proctype Barrier() {
end_b: do :: barrier_chan?0; ...; barrier_chan?0;
             barrier_chan?1; ...; barrier_chan?1 od }
```

The label `end_b` is used to indicate to Spin that an execution in which all other processes have terminated and the barrier is at position `end_b` should not be considered deadlocked.

3.2 Modeling Diffusion2d and Properties in Spin

Having dealt with the model of the MPI infrastructure, we turned to the Diffusion2d program itself. The first question was how to represent the floating point data. However, we observed that none of the properties discussed here depend in any way on the actual values of the data. This fact reflects the simplicity of the program we have chosen as our example. Had our program contained, for example, a conditional statement that branched on an expression involving some of the data, we would have had to think carefully about the appropriate abstractions. Indeed, we plan on looking at such an example in future work. But for this example, we could simply abstract away the data altogether, and we sent only the bit 1, representing any message, on the channels. The upshot is that the channels in our model keep track only of the number of pending messages sent from one process to another.

We made a few other simple optimizations (for example, leaving out channels that would never be used). Otherwise, the Promela code for each process looks very much like the pseudo-code for Diffusion2d given above. To make it easy to experiment with different choices of the parameters and other options, we wrote a simple Java program that takes as input those choices, and outputs the corresponding Spin model. The 5 parameters that come from the program itself are `nprocsx`, `nprocsy`, `nxl`, `nsteps`, and `nprint`. The additional choices affecting the model include `chan_size`, the choice between the simple or complex communication model (if `chan_size` > 0), and the choice of whether or not to include the barriers.

We now explain how we expressed the lockstep properties. (Spin already has a built-in capacity to check for deadlock.) Let $\mathsf{far}(i, j)$ be the predicate which is true precisely when process i is at the position just before the call to `update(u)` and the value of `iter` in process i minus the value of `iter` in process j is greater than 1. This can be expressed in SPIN by making the two `iter` variables global, inserting a label (`Calculate`) at the appropriate positions in the code, and adding the following:

```
#define far_i_j (proc_i@Calculate && (iter_i - iter_j > 1))
```

This predicate represents the undesirable behavior; let **Lockstep**(i, j) be the claim that $\mathsf{far}(i, j)$ is always false. We can use SPIN to check this property by including the never claim generated from the LTL formula `<>far_i_j`.

Now, **GlobalLockstep** is equivalent to the conjunction of the **Lockstep**(i, j) over all (i, j) for which $i \neq j$. To check this property, we could either verify the **Lockstep**(i, j) individually, or we could define p to be the disjunction of the $\mathsf{far}(i, j)$ and check the single never claim generated from `<>`p. The first approach may allow SPIN to scale further, though the second is probably more efficient for small configurations. Similar comments apply to **LocalLockstep**, which is the conjunction over all (i, j) for which i and j are neighbors and $i \neq j$.

There is another approach which is more specific but could help the model checker further by reducing the number of states that need to be explored. Let $\mathsf{calc}(i, n)$ be the predicate which is true precisely when process i is at the position just before the call to `update(u)` and the value of `iter` in process i is n:

```
#define calc_i_n (proc_i@Calculate && (iter_i == n))
```

For $n \geq 1$, let **Lockstep**$(i, j; n)$ be the claim that, on any execution, $\mathsf{calc}(i, n)$ must be preceded by $\mathsf{calc}(j, n - 1)$. Hence **Lockstep**(i, j) is equivalent to the conjunction, over all n, of the **Lockstep**$(i, j; n)$. To check this property with SPIN, we use the never claim generated from the LTL formula `(!`q`)U`p, where $p = \mathsf{calc}(i, n)$ and $q = \mathsf{calc}(j, n - 1)$. The potential advantage arises from the fact that the search can be truncated as soon as q becomes true when p is false.

3.3 Verification with SPIN

We instantiated a large number of models for various choices of the parameters and model options, and checked various versions of the properties. We used Spin version 4.0.0 of 1 January 2003, running on a Linux box with a 2.2 GHz Xeon processor and 4 GB of memory. In all cases we used the SPIN options `-DSAFETY` (as ours are all safety properties), `-DNOFAIR` (as no fairness assumptions were needed), `-DCOLLAPSE` (for better compression), and `-DMEMLIM=3000` (to utilize all 3 GB of memory available to a single process). To simplify the discussion that follows, we will use the term "$n \times m$ configuration" to refer to the configuration with `nprocsx` $= n$, `nprocsy` $= m$, and, unless explicitly stated otherwise, `nxl` $=$ 1 and `nprint` $=$ `nsteps` $=$ 2. (All inputs and results are available at `http://laser.cs.umass.edu/~siegel/projects`.)

We were able to verify **DeadlockFree** and **LocalLockstep** in all cases where SPIN did not run out of memory. But in verifying **GlobalLockstep**, a surprising thing occurred: for certain configurations, SPIN found a counterexample. In general, the existence of a counterexample required `nprocsx` or `nprocsy` to be at least 4, `chan_size` ≥ 1, and `nsteps` ≥ 2. In such circumstances, it is possible for one process to begin its calculation of u^2 (the value of u at time step 2) before another has begun its calculation of u^1.

To see how this can happen, we outline the trace produced by Spin for the 4×1 configuration. We may ignore the vertical exchanges of ghost cells as these just involve a process communicating with itself. Say all processes have just exited the first barrier and are about to enter the `for` loop. At this point `iter` is 0 in each process. Now process 3 may proceed to call `exchangeGhostCells` and send a message to process 2 as part of its first `Sendrecv`. Process 2 may also proceed to this point, receive this message, and send a message to process 1. At this point, process 2 may proceed to its second `Sendrecv` statement and send a message to process 3. Now process 1 may proceed to send a message to process 0, receive the message from process 2, proceed to its second `Sendrecv` and send a message to process 2. Process 2 can then receive that message, completing its participation in the `exchangeGhostCells` routine, return to the top of the loop, and then begin its calculation of u^2. Process 0 has still not entered the loop. Notice that at this point there will be two pending messages: one sent from process 2 to process 3, the other sent from process 1 to process 0.

As remarked earlier, the correctness of Diffusion2d does not depend on this property. However, the violation was a surprise to the programmer, who thought the synchronization enforced by the sends and receives would force all the processes to stay "close". In fact, once we have understood this counterexample, it is not hard to see that in a strip of $2n$ processes, processes 0 and n can become n time steps apart, if `chan_size` ≥ 1. It appears, however, that for `chan_size` $= 0$, **GlobalLockstep** holds.

We did run out of memory for relatively small configurations—for example, 4×4 for **DeadlockFree**, even with `chan_size` $= 0$. We were able to verify the 4×3 case with `chan_size` $= 0$ and with barriers. In this case there were 3.8×10^6 states stored, Spin used 153 MB of memory, and it took slightly over 3 minutes to execute `spin -a` and compile and execute the analyzer. Without barriers, the performance was 1.5×10^6 states, 60 MB, and just over 1 minute.

For positive `chan_size`, the choice of simple or complex communication model made a big difference in the size of the state space. Consider, for example, the 4×2 configuration without barriers and with `chan_size` $= 1$. Using the complex model, there were 2.8×10^7 states stored in verifying **DeadlockFree**; with the simple model this number was reduced to 9.6×10^4.

As we suspected, verification of **Lockstep**$(0, 1; 2)$ required fewer states than for the stronger **Lockstep**$(0, 1)$. For example, verification in the 3×3 configuration with `chan_size` $= 1$, no barriers, and the simple communication model, required storing $89,838$ states for the former property, and $400,482$ states for the latter. This allowed us to scale **Lockstep**$(0, 1; 2)$ as far as the 4×4 configuration (2.5×10^7 states, 1655 MB, 29 minutes). Spin was able to find the counterexample to **Lockstep**$(0, 2; 2)$ in quite large configurations—at least 7×7 (beyond this point, Spin complains that there are too many channel types).

The lockstep properties provide an example for which the simple communication model is conservative. For, if we ignore what goes on within a call to an MPI function, the set of all execution prefixes is the same whether one uses the simple or complex model; the difference lies in the fact that some of those finite prefixes may be considered deadlocked in the complex model, but not in the simple one. Since a lockstep property does not depend on this distinction, it will hold under the simple model if, and only if, it holds under the complex one.

All of the results were the same with and without the barriers. At the very least, this provided some evidence that the barriers might not be necessary.

4 Applying INCA

INCA takes as input a description of a concurrent program in the S-Expression Design Language (SEDL) together with a property expressed in the INCA query language [14]. (The query actually describes a violation of the property, much like the never claims in Spin.) From these it produces an Integer Linear Programming (ILP) problem which can be analyzed by standard linear programming tools such as CPLEX. If the ILP problem has no solution, the property is guaranteed to hold on all executions of the program. On the other hand, a solution to the ILP problem may or may not correspond to an actual counterexample to the property; if not, then there are ways to augment the ILP problem to increase the precision of the model. A constrained search using the values in a solution to the ILP problem as the counts of events in an execution can be used to determine whether a solution corresponds to a counterexample.

SEDL resembles a subset of Ada in a Lisp-like syntax. One defines tasks, which have their own local variables, execute concurrently, and communicate via rendezvous. SEDL does not explicitly provide for buffered communication, nor for shared variables (which is essentially what the Spin channels are). For these reasons, for our initial experiments with INCA we restricted ourselves to the case `chan_size` = 0.

As with Spin, our first task was to model the MPI primitives. To do this, each task j declares entries `chan_`i`_`j. Process i sends a message to process j by calling that entry; process j receives from i by issuing an accept on that entry. The MPI_SENDRECV is modeled using the SEDL `select` statement, which is like the Spin `if` statement. For example, an MPI_SENDRECV issued in process 0, sending to process 6 and receiving from 3, would be represented as follows:

```
(select (when t (call proc_6 chan_0_6) (accept chan_3_0))
        (when t (accept chan_3_0)  (call proc_6 chan_0_6)))
```

This even works in the case where a process send-receives itself as INCA allows a task to call one of its own entries. The barrier was modeled exactly as with Spin, and again, we abstracted away the floating point data to produce a model much like the one used for Spin.

There is a standard INCA deadlock query, and the queries for **Lockstep**(i, j) and **Lockstep**$(i, j; n)$ are just like the corresponding never claims in Spin. There is no easy way to represent the conjunction of two properties as a single property in INCA (this is because one cannot easily represent the disjunction of two ILP problems as a single ILP problem) and so, unlike the case for Spin, we could not check all parts of **GlobalLockstep** or **LocalLockstep** all at once.

We instantiated models for the same configurations that we had used for Spin, and used INCA version 3.5 and CPLEX Optimizer 8.1.0 on the same hardware. For INCA, the limiting factor is more often time than memory: the ILP problems generated by INCA can often be solved by CPLEX very quickly, but sometimes CPLEX will run for hours without reaching a conclusion; in these cases we say we "ran out of time."

We were able to verify all of the properties whenever we did not run out of time. (Recall that the violation to **GlobalLockstep** requires a positive value for `chan_size`.) For **DeadlockFree**, the choice of whether or not to include the barriers made a big difference in how far we could scale. With the barriers we ran out of time on the 3×3 grid. Without them we could scale as far as 8×8, and the time required (which includes the time to run INCA and CPLEX) grew roughly exponentially in the number of processes: 33 seconds for the 5×5 grid, 5 minutes for 6×6, 54 minutes for 7×7, $2,577$ minutes for 8×8.

We were able to verify **Lockstep**$(0, 1; 2)$ and **Lockstep**$(0, 2; 2)$ for very large grids as well (at least 12×12), with and without barriers. For even the largest of these, the analysis time is under one minute.

4.1 Using INCA for Buffered Communication

In order to incorporate buffers into our INCA model, we created a separate channel task chan_i_j for each pair of processes. Since we are only keeping track of the number of messages in a channel, each channel task contains one integer variable (`len`) which is incremented or decremented as messages are deposited or removed. The sending task calls an entry `send` in the channel task to deposit a message; the receive task calls an entry `receive` to pick up a message.

As with Spin we had two channel models: the simple one, in which a send blocks only if the channel is full, and the complex one, which might block the sender until the message can be received. For the simple model, the definition of `chan_0_1` appears as follows:

```
(task chan_0_1
 ((entry send) (entry receive) (variable len chan_range 0))
 ((loop (select
   (when (< len chan_size) (accept send) (assign len (+ len 1)))
   (when (> len 0) (accept receive) (assign len (- len 1)))))))
```

Now an MPI_SEND from process 0 to 1 becomes simply `(call chan_0_1 send)`, while the corresponding receive is `(call chan_0_1 receive)`. The send-receive statement is modeled using the `select` construct as before.

For the complex channel model, we modified the channel task by requiring the sender to make two calls: the first to an entry `send` as before, the second to an entry `complete`. The latter has the possibility of blocking until the channel becomes empty, i.e., until the message has been received. The definition of MPI_SENDRECV must also be modified as follows:

```
(select
  (when t (call chan_0_6 send)
    (select
      (when t (call chan_0_6 complete) (call chan_3_0 receive))
      (when t (call chan_3_0 receive) (call chan_0_6 complete))))
  (when t (call chan_3_0 receive)
    (call chan_0_6 send) (call chan_0_6 complete)))
```

In our first attempts applying this approach to the lockstep properties, we obtained many spurious solutions with disconnected cycles in the channel tasks. However, we were able to take advantage of the special structure of the channel tasks to create a very efficient form of the cycle-elimination procedure described in [15], and this provided enough precision for a conclusive result in all cases.

Using the simple communication model with `chan_size` $= 1$, we were able to verify $\mathbf{Lockstep}(0, 1; 2)$ and find the counterexamples to $\mathbf{Lockstep}(0, 2; 2)$, each in configurations up to size 12×12. The times for the 12×12 grid were 2 and 43 minutes, respectively. We were even able to use INCA to find an execution prefix, for a 12×12 grid, in which two processes become 6 time steps apart.

5 Theoretical Results

Our experience analyzing Diffusion2d led us to begin a more general investigation of the properties of MPI programs. Here we give a brief summary of our inquiry; the details and proofs appear in [16]. To simplify matters, we focused on programs that use only the subset of MPI that occurs in Diffusion2d. While these functions represent only a small subset of the MPI library, they are among the most fundamental and commonly used MPI functions, and many interesting and complex parallel programs can be written using only them. Furthermore, we expect that the techniques we have developed to deal with this subset can be extended to a much larger portion of MPI, including the collective functions such as MPI_BROADCAST and MPI_GATHER.

In order to reason about such programs, we defined a precise notion of a model $\mathcal{M}$ of an MPI program. In essence $\mathcal{M}$ consists of an automaton for each process, and a set of channels (each with a fixed sending and receiving process). The transitions may be labeled by either local events, or by communication events. The latter have the form $c!a$ and $c?a$, where a is a constant. Each state is either a *terminal state* (a state with no outgoing transitions, representing process termination), a *local-event state* (all transitions departing from that state are local), a *sending state* (there is only one departing transition and it is labeled by a send event), a *receiving state* (all the departing transitions are labeled by receive events), or a *send-receive state*—a state from which first a send can happen and then a receive, or first a receive then the send.

An *execution prefix* of $\mathcal{M}$ is a sequence of transitions from the various automata such that (i) the projection onto each automaton is a path starting from the initial state, and (ii) for each channel, the sequence of sends and receives on the channel obey FIFO ordering. A *synchronous* prefix is one in which every send is immediately followed by its matching receive. All of the theorems here also require that $\mathcal{M}$ have no wildcard receives—this means that for any state s, there exists a channel c such that every receive transition departing from s has a label of the form $c?a$ for some a. This corresponds to a program that uses neither MPI_ANY_TAG nor MPI_ANY_SOURCE.

An execution prefix results in a *potential deadlock* if (i) at least one process is not at a terminal state, (ii) no process is at a local state, and (iii) if a process p is at a receiving or send-receive state, then for all channels c for which there is a receive transition leaving that state, there are no pending messages on c and no process is at a state from which it can execute a send on c. We say "potential" 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 this prefix: if the implementation decides to force all sends to synchronize at this point, the program will deadlock; if it 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 this is precisely the kind of behavior we wish to avoid, we say that $\mathcal{M}$ is *deadlock-free* if it has no execution prefix of this form. We say that it is *synchronously deadlock-free* if it has no synchronous execution prefix of this form. We can prove the following:

Theorem 1. *Let $\mathcal{M}$ be a model of an MPI program with no wildcard receives. Then $\mathcal{M}$ is deadlock-free if, and only if, $\mathcal{M}$ is synchronously deadlock-free.*

The consequence is that, for such a model, it suffices to check deadlock-free with `chan_size` $= 0$.

The next theorem concerns the question of barriers. It states that if a model is deadlock-free, it must remain deadlock-free after removing all barrier statements. Barriers can be represented in our formalism by adding a coordinator process as we did with our SPIN and INCA models. If $\mathcal{M}$ is a model and B is an appropriate set of states from the automata in $\mathcal{M}$, we let $\mathcal{M}^B$ denote the model with the new barrier process and with barriers added just after the states in B. We have

Theorem 2. *Assume $\mathcal{M}$ has no wildcard receives. If $\mathcal{M}^B$ is deadlock-free then $\mathcal{M}$ is deadlock-free.*

We say that $\mathcal{M}$ is *locally deterministic* if it has no wildcard receives and every local-event state has exactly one outgoing transition.

Theorem 3. *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 processes p, the projection of T onto p is a subsequence of the projection of S onto p, up to possible reordering of the send and receive parts of send-receive statements.*

Corollary 1. *Suppose $\mathcal{M}$ is a locally deterministic model of an MPI program. Then M is deadlock-free if, and only if, there exists a synchronous execution prefix T such that either T is infinite or T is finite and ends with each process in a terminal state.*

Now suppose we are given values of the parameters for Diffusion2d, and also a choice of platform for which the numeric operations are deterministic functions. Then the "full-precision" model $\mathcal{M}$ of the program is locally deterministic. Therefore Theorem 3 provides an affirmative answer to **Q2**. As for **Q1**, Theorem 2 shows that if Diffusion2d was deadlock-free with the barriers, it will be deadlock-free after the barriers are removed; it must necessarily result in the same computation by Theorem 3. Hence the barriers really are unnecessary for the correctness of the program. Moreover, to check that Diffusion2d (with the given parameters) is deadlock-free, Corollary 1 shows that it suffices to check *a single execution* of $\mathcal{M}$ and observe that it terminates normally. So verifying **DeadlockFree** does not really require any model checking at all (though the lockstep properties are a different matter).

6 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., [1, 12]) are an active area of research. But only a few investigators have looked specifically at the MPI communication mechanisms. Georgelin et al. [5] 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. [10] 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., [2,4,8]) and traces [11]. Most of the work on reduction and atomicity has been concerned with reducing sequences of statements in a single process, although Cohen and Lamport [2] 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 do not yet fully understand, however, the extent to which our results in the MPI setting correspond to their results for TLA.

Manohar and Martin [9] 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.

7 Conclusions

MPI-based software for scientific computation represents an important and growing domain of concurrent software, with all the standard problems introduced by concurrency. Although FSV techniques can offer solutions to many of these problems, various aspects of the MPI framework, such as dynamic changes in message buffer size, and the scale of typical MPI programs present substantial obstacles to the successful application of FSV methods. In this paper, we have applied FSV techniques to a small, but realistic, example of a parallel scientific program using MPI in order to check some properties of interest to the programmer. We have shown how to model some of the problematic MPI communication constructs, and we have described some theoretical results that simplify the verification process.

For small configurations of the MPI program, both SPIN and INCA could verify the two properties that hold and found counterexamples for the one that did not. The programmer was surprised that **GlobalLockstep** was violated, but the violation can occur only when at least one dimension of the grid is greater than three and messages are buffered, making the system big enough to be hard to reason about informally (although certainly tiny by MPI standards). We do not attach much significance to the fact that INCA could do larger configurations than SPIN. Although we regard ourselves as reasonably skilled SPIN users, we are certainly more expert in applying INCA. It may well be that there are more efficient ways to model the MPI constructs in Promela, or that different settings would significantly improve the performance of SPIN. In any case, it would be foolish to generalize very far on the basis of analysis of a single program.

Although MPI programs like the examples described here exhibit significant symmetry, we do not expect substantial gains from applying standard symmetry methods in model checking. The problem is that these methods cannot reduce the state space by a factor of more than the order of the symmetry group, and the growth of the symmetry group does not keep pace with the growth of the state space as the systems are scaled up. Compositional methods, on the other hand, might very well allow for collapsing the system to some minimal configuration, depending on the property being checked, and we hope to investigate this approach in the future.

Our theoretical results help simplify the verification and suggest that other results tailored to the MPI domain may increase the range of MPI programs to which FSV techniques may be effectively applied. We plan to extend our investigation to the more general cases of programs making use of wildcard receives, non-blocking sends and receives, and the MPI collective operations. It will also be important to find appropriate abstractions for programs where the values of floating point data affect the flow of control.

We thank Andrew Siegel for supplying the Diffusion2d example and for many discussions of the problems of developing MPI-based scientific software. We are also grateful to Ewing Lusk for clarifying some of the more complicated parts of the MPI Standard and for encouraging this work. This research was partially supported by the U. S. Army Research Laboratory and the U. S. Army Research Office under agreement number DAAD190110564.

References

1. 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
2. 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
3. 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
4. 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
5. Georgelin, P., Pierre, L., Nguyen, T.: A formal specification of the MPI primitives and communication mechanisms. Technical Report 1999-337, LIM (1999)
6. 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)
7. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Boston (2004)
8. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Communications of the ACM **18** (1975) 717–721
9. 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
10. 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
11. 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
12. 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
13. Message-Passing Interface Standard 2.0. `http://www.mpi-forum.org/docs` (1997)
14. Siegel, S.F.: The INCA query language. Technical Report UM-CS-2002-18, Department of Computer Science, University of Massachusetts (2002)
15. 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
16. Siegel, S.F., Avrunin, G.S.: Analysis of MPI programs. Technical Report UM-CS-2003-036, Department of Computer Science, University of Massachusetts (2003)
17. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI—The Complete Reference: Volume 1, The MPI Core. 2 edn. MIT Press, Cambridge, MA (1998)