Using MPI-Spin to Model Check MPI Programs with Nonblocking Communication

Stephen F. Siegel*

Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716, USA
siegel@cis.udel.edu
http://www.cis.udel.edu/~siegel

Many of the difficulties involved in constructing correct parallel programs arise from the fact that a parallel program can behave differently when executed twice, even when the same input is used for both executions. There are numerous sources for such *nondeterminism* in MPI programs: the execution steps from the processes may be interleaved differently in time; a receive statement using `MPI_ANY_SOURCE` may select a message from a different source; a call to `MPI_Waitany` may select a different request for completion; a message sent by a call to `MPI_Send` may be buffered in one execution while in the other the sender is blocked until the message can be delivered synchronously. This nondeterminism makes effective testing and debugging of MPI programs extremely difficult and is one of the main reasons for the interest in formal verification methods, such as model checking [2–9].

In theory, model checking techniques can be used to explore all nondeterministic choices in a parallel program and to verify that the program will satisfy user-specified correctness properties on all executions (or output a trace if a property can be violated). In practice, however, there are many barriers to the successful application of model checking to parallel programs, especially to MPI programs. Among these is the fact that most model checking tools operate on a *model* of the program, rather than on the program code itself. The model is typically expressed in a low-level modeling language which has no notion of MPI (or of many other common program constructs).

Promela, the input language for the widely-used model checker Spin [1], does contain a few rudimentary primitives that correspond to basic message-passing operations; these have been exploited in previous work to model basic *blocking* point-to-point and collective operations in simple MPI programs. But these primitives are inadequate for expressing the complex notions that arise in MPI *nonblocking* communication, such as communication requests, posting, testing, canceling, and waiting on requests, persistent requests, and so on.

For this reason, I have developed Mpi-Spin, an extension to Spin that supports many commonly-used MPI functions, constants, and types, including those used for nonblocking point-to-point communication. As an initial experiment, I have applied Mpi-Spin to the 17 examples (2.17–2.33) in [10] dealing with nonblocking communication. I successfully verified many correctness properties of

* Supported by the U.S. National Science Foundation grant CCF-0541035

these examples and in two cases discovered non-trivial faults. The Mpi-Spin source code and the complete input and output for the experiment can be downloaded from `http://www.cis.udel.edu/~siegel`.

Mpi-Spin adds to Promela (*via* macros and Spin's `c_code` facility) primitives that correspond to ones of the same name in MPI, such as `MPI_Isend`, `MPI_Wait`, `MPI_Cancel`, `MPI_Send_init`, `MPI_Request`, `MPI_Status`, `MPI_ANY_SOURCE`, and so on. The syntax matches closely the syntax for the C bindings for MPI, which greatly simplifies the task of translating a C/MPI program into Promela.

By default, Mpi-Spin checks a number of generic properties that one would expect to hold in any correct MPI program. These include (1) the program cannot deadlock, (2) there are never two incomplete requests whose buffers intersect non-trivially, (3) the total number of outstanding requests never exceeds a specified bound, (4) when `MPI_Finalize` is called there are no request objects allocated for and there are no buffered messages destined for the calling process, and (5) the size of an incoming message is never greater than the size of the receive buffer. In addition, Mpi-Spin can check application-specific properties formulated in temporal logic. In the producer-consumer Ex. 2.19, for instance, Mpi-Spin was used to check that every message produced is eventually consumed.

Mpi-Spin includes some primitives that do not correspond to anything in MPI, but are very useful for modeling MPI programs. For example, there is a type `MPI_Symbolic` (together with a number of operations on that type) that can be used to represent floating-point expressions *symbolically*. Previous work [9] showed how symbolic techniques can be used to verify that a parallel program computes the same result as a trusted sequential version of the program *on any input*. This technique was used to attempt to verify the correctness of the Jacobi iteration example of [10] (Ex. 2.17; the sequential version is Ex. 2.12).

Ex. 2.17 is also one of the programs for which Mpi-Spin discovered a fault. The problem occurs when the number n of matrix columns is less than twice the number of processes p. In this case, on at least one process two send requests will be posted using the same buffer—the single column of local matrix B. In all other cases, this program was verified successfully.

Another fault was discovered in Ex. 2.19. In the consumer's busy-wait loop, the statement incrementing `i` is erroneously executed after the call to `MPI_Test` sets `flag` to `true` and before exiting the loop. This causes the consumer to consume from and repost to the wrong producer. After correcting this problem, the program was verified successfully.

Mpi-Spin is under active development, and will soon be extended to support multiple communicators, all collective operations, and non-standard-mode sends.

References

1. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Boston (2004)
2. Matlin, O.S., Lusk, E., McCune, W.: SPINning parallel systems software. In Bosnacki, D., Leue, S., eds.: Model Checking of Software: 9th International SPIN

Workshop, Grenoble, France, April 11–13, 2002, Proceedings. Volume 2318 of LNCS., Springer-Verlag (2002) 213–220
3. 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. LNCS, Springer (2006)
4. Post, D.E., Votta, L.G.: Computational science demands a new paradigm. Physics Today (2005) 35–41
5. 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 International Conference, VMCAI 2005, Paris, January 17–19, 2005, Proceedings. Volume 3385 of Lecture Notes in Computer Science. (2005) 413–429
6. 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)
7. Siegel, S.F., Avrunin, G.S.: Verification of MPI-based software for scientific computation. In Graf, S., Mounier, L., eds.: Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1–3, 2004, Proceedings. Volume 2989 of Lecture Notes in Computer Science., Springer-Verlag (2004) 286–303
8. 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, June 15–17, 2005, Chicago, Illinois, USA, ACM Press (2005) 95–106
9. 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., Pezzé, M., eds.: ISSTA 2006: Proceedings of the ACM SIGSOFT 2006 International Symposium on Software Testing and Analysis, Portland, ME (2006) 157–168
10. 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)

A Source code for [10, Ex. 2.19]: Multiple-producer, single-consumer

A.1 Original C source code

```
typedef struct {
  char data[MAXSIZE];
  int datasize;
  MPI_Request req;
} Buffer;
Buffer *buffer;
MPI_Status status;
...
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
if (rank != size-1) {  /* producer code */
  buffer = (Buffer *)malloc(sizeof(Buffer));
  while(1) {  /* main loop */
```

```
    produce( buffer->data, &bufffer->datasize);
    MPI_Send(buffer->data, buffer->datasize, MPI_Char,
             size-1, tag, comm);
  }
}
else { /* rank == size-1; consumer code */
  buffer = (Bufffer *)malloc(sizeof(Buffer)*(size-1));
  for (i=0;  i< size-1; i++)
    MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag,
              comm, &buffer[i].req);
  i = 0;
  while(1) { /* main loop */
    for (flag=0; !flag; i= (i+1)%(size-1)) {
      /* busy-wait for completed receive */
      MPI_Test(&(buffer[i].req), &flag, &status);
    }
    MPI_Get_count(&status, MPI_CHAR, &buffer[i].datasize);
    consume(bufer[i].data, buffer[i].datasize);
    MPI_Irecv(bufer[i].data, MAXSIZE, MPI_CHAR, i, tag,
              comm, &buffer[i].req);
  }
}
```

A.2 Promela model of communication skeleton for MPI-SPIN

```
#include "mpi-spin.prom"

#if (NPRODUCERS != NPROCS - 1)
#error "Wrong NPRODUCERS"
#endif
#define TAG 0
#define SEND_BUFF NULL
#define RECV_BUFF NULL
#define COUNT 0

active [NPRODUCERS] proctype producer() {
  MPI_Request req;

  MPI_Init(Pproducer, Pproducer->_pid);
  do
  :: MPI_Isend(Pproducer, SEND_BUFF, COUNT, MPI_POINT,
               NPROCS - 1, TAG, &Pproducer->req);
     MPI_Wait(Pproducer, &Pproducer->req, MPI_STATUS_IGNORE)
  od;
  MPI_Finalize(Pproducer)
}
```

```
active proctype consumer() {
  MPI_Request req[NPRODUCERS];
  byte i = 0;
  bit flag;

  MPI_Init(Pconsumer, Pconsumer->_pid);
  do
  :: i < NPRODUCERS ->
     d_step {
       MPI_Irecv(Pconsumer, RECV_BUFF, COUNT, MPI_POINT,
                 Pconsumer->i, TAG,
                 &Pconsumer->req[Pconsumer->i]);
       i++
     }
  :: else -> i = 0; break
  od;
  do
  :: flag = 0;
     do
     :: !flag ->
        atomic {
          MPI_Test(Pconsumer, &Pconsumer->req[Pconsumer->i],
                   flag, MPI_STATUS_IGNORE);
          i = (i + 1)%NPRODUCERS
        }
     :: else -> break
     od;
     MPI_Irecv(Pconsumer, RECV_BUFF, COUNT, MPI_POINT,
               Pconsumer->i, TAG,
               &Pconsumer->req[Pconsumer->i]);
  od;
  MPI_Finalize(Pconsumer);
}

active MPI_PROC
```