MPI-SPIN Usage Notes

Stephen F. Siegel

1. MPI Functions, Types, and Constants Implemented in Mpi-Spin

1.1. General MPI functions.

`MPI_Init`, `MPI_Finalize`, `MPI_Comm_rank`, `MPI_Comm_size`

1.2. MPI nonblocking standard-mode point-to-point functions.

(1) `MPI_Isend`: post a send request
(2) `MPI_Irecv`: post a receive request
(3) `MPI_Wait`: block till request completes, deallocate request object, nullify handle, fill status
(4) `MPI_Test`: return 0 if request not complete, else return 1 and proceed as in `MPI_Wait`
(5) `MPI_Request_free`: mark the request to be freed as soon as complete, nullify handle
(6) `MPI_Request_get_status`: get the status of the request without destroying it
(7) `MPI_Waitany`: block till at least one request in an array completes, then `MPI_Wait` it
(8) `MPI_Testany`: apply `MPI_Wait` to a completed request if there is one, else return 0
(9) `MPI_Waitall`: block till all requests in array complete and apply `MPI_Wait` to all
(10) `MPI_Testall`: if all requests have completed, wait all and return 1, else return 0
(11) `MPI_Waitsome`: block till at least one completes then wait on and return completed subset
(12) `MPI_Testsome`: apply `MPI_Wait` to all completed requests and return completed subset
(13) `MPI_Iprobe`: probe for incoming message matching parameters or say if there isn't one
(14) `MPI_Probe`: block until there is an incoming message matching parameters
(15) `MPI_Cancel`: attempt to cancel a request
(16) `MPI_Test_canceled`: test status to see if request was really canceled
(17) `MPI_Send_init`: initialize a persistent send request
(18) `MPI_Recv_init`: initialize a persistent receive request
(19) `MPI_Start`: start an (inactive) persistent request
(20) `MPI_Startall`: start all persistent requests in an array

1.3. Types added to those already available in Promela.

(1) `MPI_Request`: a handle for a request object
(2) `MPI_Status`: an object with fields for source, tag, count, "cancelation", error
(3) `MPI_Symbolic`: a symbolic arithmetic expression

1.4. Constants.

(1) `MPI_ANY_SOURCE`: used to receive messages from any source
(2) `MPI_ANY_TAG`: used to receive messages with any tag
(3) `MPI_STATUS_IGNORE`: used for status argument when you don't care about status
(4) `MPI_STATUSES_IGNORE`: like above but in place of an array of statuses
(5) `MPI_REQUEST_NULL`: a handle that doesn't refer to any request object
(6) `MPI_BYTE`: an MPI datatype corresponding to Promela's `byte`, C's `unsigned char`
(7) `MPI_SHORT`: an MPI datatype corresponding to Promela/C's `short` (2 bytes)
(8) `MPI_INT`: an MPI datatype corresponding to Promela/C's `int` (4 bytes)
(9) `MPI_POINT`: a virtual datatype that takes up 0 bytes
(10) `MPI_SYMBOLIC`: a new datatype corresponding to `MPI_Symbolic` (4 bytes)

```
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);
  }
}
```

FIGURE 1. Example 2.19 from *MPI—The Complete Reference*, "Multiple-producer, single-consumer code, modified to use test calls"

2. Mpi-Spin Input Language

The input language for Mpi-Spin is an extension of Promela, the input language for Spin. Consider the MPI/C program of Fig. 1. A model of this program for Mpi-Spin is given in Fig. 2. In the model, the data is abstracted away altogether; hence the use of `MPI_POINT` as the MPI datatype. A few other points:

(1) the Promela file #includes `mpi-spin.prom`
(2) an MPI daemon process is inserted into the Promela file *via* the macro `MPI_PROC`
(3) the MPI rank of a process is declared using `MPI_Init`
(4) the method for referencing state variables uses Spin's C code facility; the syntax is awkward but it works

```
#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
```

FIGURE 2. Abstract model of Example 2.19 appropriate for checking freedom from deadlock

All MPI functions, constants, and types listed in §1 are defined using macros that call embedded C code. The syntax and semantics of each is documented in `mpi-spin.prom`. These mirror almost exactly the real MPI syntax. See Figure 3.

Using MPI-SPIN is much like using SPIN, except in place of `spin -a` one uses `ms` (see Fig. 4). In place of `cc` one uses `mscc`. These two scripts provide a convenient way to invoke SPIN and the C compiler from the command line with the right parameters for MPI-SPIN. With `ms`, the user must specify the number of MPI processes as well as upper bounds on (1) the total number of outstanding requests, and (2) the total number of buffered messages. If the request bound is exceeded during

```
/* MPI_Irecv: Post a receive request.
 *
 *   proc:     the letter "P" followed by the proctype name, e.g.
 *             Pproducer
 *   buffer:   pointer to the very beginning of array that contains the
 *             recv buffer.  A C expression of type void*.
 *   count:    upper bound on number of elements of type datatype
 *             to be received.   C expression of type uchar.
 *   datatype: one of datatype ID's.  C expression of type uchar.
 *             Predefined include MPI_POINT, MPI_BYTE, MPI_INT.
 *   source:   the rank of the sending process, or MPI_ANY_SOURCE.
 *             C expression of type uchar.
 *   tag:      the tag of the incoming message or MPI_ANY_TAG.  C expr
 *             of type uchar.
 *   request:  pointer to the request handle variable.  A C expression
 *             of type uchar*.
 */
#define MPI_Irecv(proc, buffer, count, datatype, source, tag,       \
                  request)                                          \
    c_code{                                                         \
      MPI_irecv(RANK(proc), buffer, count, datatype,                \
                source, tag, request, MPI_DEFAULT_TESTABLE,         \
                MPI_DEFAULT_CANCELABLE);                            \
    }
```

FIGURE 3. Excerpt from `mpi-spin.prom` defining `MPI_Recv`

the search, an error is reported. The buffer bound is treated differently: the MPI daemon's *upload* transition is disabled if the upload would cause this bound to be exceeded.

Implementation detail: the communication record values are maintained in a separate hashtable. Each is assigned a unique integer ID; this is all SPIN knows about. (Just as we deal with symbolic expressions.) So the only state we add to the Promela model is an integer array whose length is the sum of the bounds described above; the integers refer to the ID numbers of communication record values.

The `ms` option -dl causes a model to be contructed in which, for every send posted, SPIN will make a nondeterministic choice between forcing the send to be delivered synchronously and allowing it to be buffered. This is necessary for a conservative verification of properties such as freedom from deadlock, but entails a (large) cost in terms of the number of states.

3. EXAMPLES

I applied MPI-SPIN to the 17 examples in *MPI—The Complete Reference: Volume 1: the MPI Core* from the section of that book dealing with the (standard-mode) nonblocking point-to-point functions. These are examples 2.17 through 2.33. I was able to verify a range of generic and application-specific properties of these. The generic properties which are checked automatically by MPI-SPIN are:

(1) there are never two incomplete requests whose buffers intersect non-trivially
(2) the total number of outstanding requests never exceeds the specified bound
(3) `MPI_Init` is called before any other MPI function
(4) `MPI_Finalize` and `MPI_Init` are called only once (each)

```
ms: generate mpi-spin verifier source code from Promela file
Usage: ms [options] <filename>
Options:
-np=<NPROCS>
   number of MPI processes (required)
-buf=<BUFFER>
   bound on number of buffered message (required)
-req=<REQUESTS>
   bound on number of outstanding requests (required)
-symhash=<SYMBOLIC_HASHTABLE_LENGTH>
   hashtable length for symbolic values
-symmax=<SYMBOLIC_MAX_VALUES>
   bound on number of symbolic values
-crmax=<CR_MAX_VAL>
   bound on number of communication record values
-crhash=<CR_HASHTABLE_LENGTH>
   hashtable length for communication record values
-notest
   the model contains no MPI_Test*, MPI_*any*, nor MPI_*some*
-nocancel
   the model contains no MPI_Cancel
-noprobe
   the model contains no MPI_Iprobe nor MPI_Probe
-dl
   nondeterministically choose to synch/buffer sends for full
   deadlock detection
<SPIN options>
  any valid option for spin -a
```

FIGURE 4. Usage notes for executable script `ms`

(5) upon calling `MPI_Finalize`:
   - there are no request objects allocated for the calling process
   - there are no buffered messages destined for the calling process

(6) no incoming message ever exceeds in size the size of the receive buffer
(7) freedom from deadlock (though `-dl` is needed for full check)

Two nontrivial bugs were also found: these occurred in Examples 2.17 and 2.19 and are described below.

Example 2.17: *Use of nonblocking communications in Jacobi computation.* There is a fault in the code that is revealed by certain configurations. The problem occurs when on at least one process, $m = 1$, which will happen whenever $n < 2p$. Then two sends are posted from the same buffer (the single column of local matrix B on at least one process). For all configurations we examined outside of this problematic region, we are able to verify that the sequential and parallel versions are Herbrand equivalent, using the symbolic execution method. (The sequential version is given as Example 2.12 earlier in the book.)

Example 2.18: *Multiple producer, single-consumer code using nonblocking communication.* We are able to verify four properties:

p_0: freedom from deadlock and standard assertions,

p_1: every message produced is eventually consumed,
p_2: no producer becomes permanently blocked,
p_3: for a fixed producer, messages are consumed in the order produced.

Example 2.19: *Multiple producer, single-consumer code, modified to use test calls.* There is a bug. The code

```
i = 0;
while (1) { /* main loop */
  for (flag=0; !flag; i=(i+1)%(size-1)) {
    MPI_Test(&(buffer[i].req), &flag, &status);
  }
  consume_and_post(i);
}
```

(where *consume_and_post(i)* stands in for code that consumes from `buffer[i]` and then posts another receive request to process i) is incorrect: the statement `i=(i+1)%(size-1))` is erroneously executed after the call to `MPI_Test` sets `flag` to true but before exiting the loop and calling *consume_and_post(i)*. The correct code should look something like the following:

```
i = 0;
while (1) { /* main loop */
  flag = 0;
  while (1) {
    MPI_Test(&(buffer[i].req), &flag, &status);
    if (flag) break;
    i=(i+1)%(size-1);
  }
  consume_and_post(i);
  i=(i+1)%(size-1);
}
```

The fault was revealed in attempting to verify p_0 in a small configuration. The failure reported by Mpi-Spin was an attempt to exceed the specified bound on the number of outstanding requests. The configuration consisted of 3 processes, no buffering, and an upper bound of 4 outstanding requests. After correcting the bug, we verified all 4 properties over a range of configurations.

Example 2.20: *An example using MPI_REQUEST_FREE.* We verified freedom from deadlock and standard assertions.

Example 2.21: *Message ordering for nonblocking operations.* Verified freedom from deadlock, standard assertions, and that the messages were always matched with the correct receives.

Example 2.22: *Order of completion for nonblocking operations.* Freedom from deadlock, standard assertions, and that in either process, the two requests could complete in either order.

Example 2.23: *An illustration of progress semantics.* Freedom from deadlock, standard assertions, and the correct values are received by both receives.

Example 2.24: *An illustration of buffering for nonblocking messages.* Freedom from deadlock, standard assertions, and the correct values are received by both receives.

Example 2.25: *Out of order communication with nonblocking messages.* The code in the book clearly does not correspond to the discussion given in the text. I took the code from the first edition instead. Freedom from deadlock, standard assertions, and the correct values are received by both receives.

Example 2.26: *Producer-consumer code using waitany.* This is the third version of the multiple-producer single-consumer example to be considered in the book. It replaces the busy-wait test loop in Example 2.19 with a single call to `MPI_Waitany`. As pointed out in the book, this has the disadvantage that it allows "starvation"—the consumer can repeatedly consume messages from one process while ignoring messages sent by all other procesess. This is reflected in the analysis:

- p_0: still holds
- p_1: fails. SPIN finds a counterexample which ends up in an infinite loop in which producer 1 has sent a message and the request has completed, but the consumer repeatedly chooses the message from process 0 in the `MPI_Waitany`.
- p_2: also fails, even with progress assumptions and weak fairness. SPIN produces a counterexample in which producer 1 sends a message, this is matched by the receive posted by the consumer, completes, producer 1 posts the second send, this one isn't matched (because the `MPI_Waitany` in the consumer repeatedly returns the producer 0 match), and it doesn't complete (the MPI infrastructure has decided to block this completion until a matching receive is posted), so the producer blocks at the wait. Meanwhile the consumer repeatedly selects the match for producer 0.
- p_3: still holds.

Example 2.27: *Main loop of Jacobi computation using waitall.* Exact same results as for Example 2.17.

Example 2.28: *A client-server code where starvation is prevented.* (Note *client-server* should be *producer-consumer*.) Fourth version of the multiple-producer single-consumer example. It replaces the `MPI_Waitany` of Example 2.26 with an `MPI_Waitsome`. This is to correct the starvation problem. All 4 properties hold.

Example 2.29: *Use a blocking probe to wait for an incoming message.* There's an `i` used for two different variables but otherwise the program appears to be correct. We verified freedom from deadlock and the standard assertions and that the values are received into `i` and `x`.

Example 2.30: *A similar program to the previous example, but with a problem.* An incorrect use of probe, and, sure enough, SPIN catches the bug.

Example 2.31: *Code using MPI_CANCEL.* I had to add a line to wait for the completion of the request in process 1 after posting the second receive. Without that, you can terminate with an unfreed request object still floating around. Otherwise everything seems to be OK.

Example 2.32: *Jacobi computation, using persistent requests.* Everything works very well (modulo the usual error).

Example 2.33: *Starvation-free producer-consumer code.* This is the 5th and final version of the multiple-producer, single-consumer system. Like the third version, Example 2.26, this uses an `MPI_Waitany` at each loop iteration to select and process one completed receive request. However, unlike that version, starvation is prevented by a judicious use of `MPI_REQUEST_NULL`.