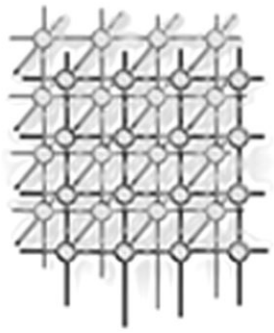
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2002; **14**:911–932 (DOI: 10.1002/cpe.701)

Deadlock detection in MPI programs



Glenn R. Luecke*,†, Yan Zou, James Coyle, Jim Hoekstra and Marina Kraeva

High Performance Computing Group, Iowa State University, Ames, IA 50011-2251, U.S.A.

SUMMARY

The Message-Passing Interface (MPI) is commonly used to write parallel programs for distributed memory parallel computers. MPI-CHECK is a tool developed to aid in the debugging of MPI programs that are written in free or fixed format Fortran 90 and Fortran 77. This paper presents the methods used in MPI-CHECK 2.0 to detect many situations where actual and potential deadlocks occur when using blocking and non-blocking point-to-point routines as well as when using collective routines. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: Message-Passing Interface; deadlock detection; handshake strategy

1. INTRODUCTION

The Message-Passing Interface (MPI) [1,2] is commonly used to write programs for distributed memory parallel computers. Since writing and debugging MPI programs is often difficult and time consuming, MPI-CHECK 1.0 [3] has been developed to help make this process easier and less time consuming by automatically performing argument type checking, bounds checking for message buffers, etc. However, MPI-CHECK 1.0 does not detect situations where possible deadlocks may occur. This paper presents the methods used in MPI-CHECK 2.0 to detect situations where a deadlock may occur when using blocking and some non-blocking point-to-point routines as well when using collective routines.

MPI-CHECK 2.0 detects 'actual' and 'potential' deadlocks in MPI programs. An 'actual' deadlock occurs when a process waits for something to occur that will never occur. For example, an 'actual' deadlock will occur if a process executes the MPI synchronous send, mpi_ssend, and there is no corresponding call to an MPI receive routine. A 'potential' deadlock occurs in those situations where

*Correspondence to: Glenn R. Luecke, High Performance Computing Group, Iowa State University, Ames, IA 50011-2251, U.S.A.
†E-mail: grl@iastate.edu

Copyright © 2002 John Wiley & Sons, Ltd.
Received 22 January 2002
Revised 29 May 2002

the MPI program may have an 'actual' deadlock depending on the MPI implementation. For example, a 'potential' deadlock will occur when a process executes the MPI standard send, mpi_send, if there is no corresponding MPI receive and if the call to mpi_send copies the message to a buffer and execution continues. Notice that if the mpi_send had not copied the message to a buffer, a deadlock would have occurred. Using a decentralized handshaking approach, MPI-CHECK 2.0 detects 'actual' and 'potential' deadlocks when using blocking and some non-blocking point-to-point routines as well as when using collective routines. The software that performs the deadlock detection described in this paper has been integrated into MPI-CHECK and can be obtained by going to [4]. This integrated software package is called MPI-CHECK 2.0.

While MPI-CHECK was being developed, a project named Umpire was being carried out at Lawrence Livermore National Laboratories. Umpire [5] is a tool for detecting MPI errors at run-time. Its deadlock detection function tracks blocking point-to-point and collective MPI communication calls, communicator management routines, completions of non-blocking requests, and detects cycles in dependency graphs prior to program execution. Unlike MPI-CHECK, Umpire uses a central manager to collect the MPI call information and check them with a verification algorithm. The central manager then controls the execution of the MPI program. The manager communicates with all MPI processes via its shared memory buffers. Currently Umpire only runs on shared memory machines.

In this paper, detection of 'actual' and 'potential' deadlock situations involving blocking point-to-point MPI routines is discussed in Section 2. Detection of actual and potential deadlock situations involving non-blocking point-to-point MPI routines is discussed in Section 3. Detection of actual and potential deadlock situations caused by the incorrect use of collective MPI routines is discussed in Section 4. Section 5 contains our conclusions.

2. DEADLOCK DETECTION FOR BLOCKING POINT-TO-POINT MPI ROUTINES

MPI provides both blocking and non-blocking point-to-point communication routines. Recall that when a process executes a blocking point-to-point routine, execution does not continue until it is safe to change the send/receive buffer. This section presents the methods used by MPI-CHECK to detect 'actual' and 'potential' deadlocks for blocking, point-to-point MPI routines.

2.1. Deadlock detection strategy

There are three categories of actual or potential deadlock situations that occur when using blocking, point-to-point MPI routines:

1. a process executes a receive routine and there is no corresponding call to a send routine;
2. a process executes mpi_send, mpi_ssend or mpi_rsend and there is no corresponding call to a receive routine; and
3. a send-receive cycle may exist due to incorrect usage of sends and receives.

It is obvious that the situation described in item 1 above causes an actual deadlock. As was explained in the introduction, the situation described in item 2 will cause an actual deadlock when using the synchronous send, mpi_ssend, and sometimes when using the standard send, mpi_send. The situation in item 2 is a potential deadlock when using mpi_send. According to the MPI standard, a ready mode

Copyright © 2002 John Wiley & Sons, Ltd.

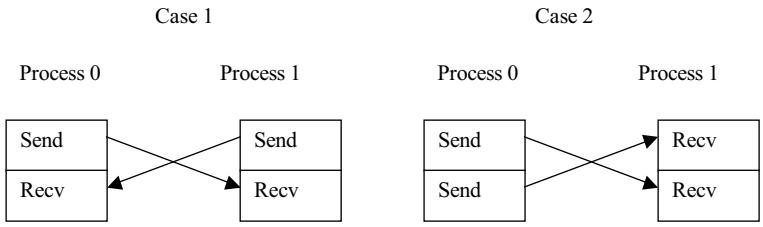


Figure 1. A dependency cycle with two processes.

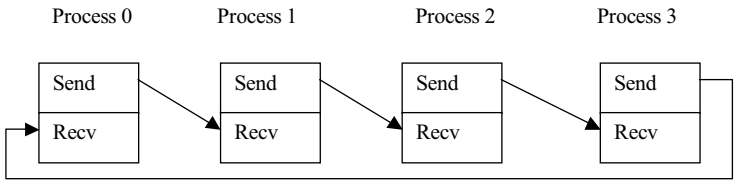


Figure 2. A dependency cycle with four processes.

send operation, mpi_rsend, may be started only if the matching receive has been posted; otherwise, the operation is erroneous and its outcome is undefined. MPI-CHECK does not currently determine if a matching receive has been posted prior to the execution of the call to mpi_rsend, but it does determine if there is a matching receive. If a process calls the buffered send, mpi_bsend, and there is not a matching receive, then this is neither an actual nor potential deadlock situation. Currently, MPI-CHECK does not check for matching receives when mpi_bsend is called. Detailed information about how MPI-CHECK handles mpi_bsend can be found in Section 2.2.

Figures 1 and 2 illustrate the incorrect usage of sends and receives when using mpi_ssend for a dependency cycle with two processes and with four processes. Notice that no send can complete until the corresponding receive has been posted. This causes an actual deadlock. If one uses mpi_send in Figures 1 and 2, then either an actual or potential deadlock will occur depending on the implementation of mpi_send and the message size used. If one uses mpi_bsend in Figures 1 and 2 for at least one of the sends, then no deadlock will occur and the usage is correct. MPI-CHECK will detect such cycles when using mpi_ssend, mpi_rsend, and mpi_send.

We next discuss methods that could be used to automatically detect the actual and potential deadlocks discussed above. One possible method would be to have MPI-CHECK automatically replace all mpi_send and mpi_rsend calls in the MPI program with mpi_ssend. When the modified program is executed under the control of a debugger, the debugger will stop at the point of the deadlock. There are several problems with this approach. The first is that there may not be a parallel debugger available on the machine being used. If there were a parallel debugger, then recompiling a large application code for

Copyright © 2002 John Wiley & Sons, Ltd.

a debugger and executing it under the debugger may take an unacceptably long time. For these reasons, this methodology for detecting actual and potential deadlocks was not used in MPI-CHECK.

Another possible methodology for finding actual and potential deadlocks for blocking routines would be to have the MPI program execute under the control of a central manager, similar to what is done in Umpire [5] for shared memory debugging. However, there are difficulties when using a central manager. For example, suppose one were debugging an application using **p** processes and the central manager is executing on one of these processes. If a deadlock were to occur on the process the central manager is executing on, then the central manager cannot function. Notice the central manager will likely significantly delay MPI communication on its process. Thus, one would have to request **p**+1 processes when executing an MPI program with **p** processes. Also notice that using a central manager does not scale well for large numbers of processors. In [5], it was stated that Umpire might be extended to distributed memory parallel machines using the central manager approach. We decided not to use this approach for MPI-CHECK.

MPI-CHECK takes a different approach to the automatic detection of actual and potential deadlocks. The idea is for MPI-CHECK to insert 'handshaking' code prior to each call to a send routine and each call to a receive routine. If the 'handshake' does not occur within a time set by the user (the user can adjust the threshold to accommodate their platform and application), then MPI-CHECK will issue a warning message that a 'handshake' has not occurred within the specified time, give the line number in the file where the problem occurred, and list the MPI routine exactly as it appears in the source listing. Users have the option of having MPI-CHECK stop execution of the MPI program when an actual or potential deadlock is detected or allowing MPI-CHECK to continue program execution after the problem is detected.

The 'handshaking' strategy utilized by MPI-CHECK can be described as follows. Part of the handshaking involves comparing data from the call to the MPI send routine and the call to the MPI receive routine. If MPI-CHECK encounters a call to

mpi_send(buf, count, datatype, dest, tag, comm, ierror)

then the following information is stored in the character*512 variable send_info:

send_info = {filename, start_line, end_line, count, get_rank(comm), datatype, tag},

where start_line and end_line are the beginning and ending line numbers of the call to mpi_send in the file named 'filename'. If the line containing the mpi_send is not continued, then start_line and end_line will be the same.

The 'handshake' for the mpi_send proceeds as follows: the process executing mpi_send sends send_info to process 'dest' using a non-blocking send, mpi_isend, with a (hopefully) unique tag, MPI_CHECK_Tag1 + tag, to avoid possible tag conflicts with other messages. The following three possibilities may occur.

1. The message was never received on process 'dest' and the sending process does not receive a response message within a specified time. In this case, a warning message is issued.
2. The message was received on process 'dest', the information in send_info was consistent with the argument information in the call to mpi_recv, and process 'dest' sends a reply to the sending process stating that everything is okay. The reply is received by calling mpi_irecv.

Copyright © 2002 John Wiley & Sons, Ltd.

3. The message was received on process 'dest', the information in send_info was NOT consistent with the argument information in the call to mpi_recv. In this case, process 'dest' issues a message stating what the inconsistencies are and then sends a reply to the sending process to indicate that the message was received.

The 'handshake' for the mpi_recv proceeds as follows: the process executing mpi_recv waits to receive (by calling mpi_irecv) a message from 'source' with tag, MPI_CHECK_Tag1 + tag, where 'tag' is obtained from the call to mpi_recv. (If 'tag' is mpi_any_tag, then mpi_any_tag is used as the tag for receiving the message.) The following three possibilities may now occur.

1. A message was never received within the specified time. In this case, a warning message is issued.
2. A message was received and the information in send_info was consistent with the argument information in the call to mpi_recv. A reply message is sent to the sending process indicating that everything is okay.
3. The message was received and the information in send_info was NOT consistent with the argument information in the call to mpi_recv. In this case, a warning message is issued and then a reply is sent to the sending process to indicate that the message was received.

If the call to mpi_recv uses mpi_any_source and/or mpi_any_tag, then it might happen that the original call to mpi_recv may receive a message from a different mpi_send than the one from which the handshake was done. To avoid this problem, MPI-CHECK changes the original mpi_recv from

call mpi_recv(buf, count, datatype, source, tag, comm, status, ierror)

to

call mpi_recv(buf, count, datatype, send_rank, send_tag, comm, status, ierror)

where send_rank and send_tag come from the send_info in the handshake. Also notice that in this non-deterministic situation, MPI-CHECK will only detect a deadlock that occurs in the order of actual execution.

Figures 3 and 4 show the code inserted prior to calling each mpi_send, mpi_ssend, and mpi_recv for the case when MPI-CHECK is configured to stop program execution when an actual or potential deadlock problem is found. This instrumentation is accomplished by having MPI-CHECK simply inserting subroutine 'handshake_send' before the mpi_send or mpi_ssend, and by inserting subroutine 'handshake_recv' before the call to mpi_recv. In Figures 3 and 4, the variable MPI_CHECK_TIMEOUT is the waiting time, in minutes, specified by the user. This is specified by using the environmental variable MPI_CHECK_TIMEOUT. For example, issuing the command

setenv MPI_CHECK_TIMEOUT 5

sets the waiting time to five minutes. MPI-CHECK has been designed so that users have the option of having MPI-CHECK stop execution of the MPI program when an actual or potential deadlock is detected or allowing MPI-CHECK to continue program execution after the problem is detected. These options are also specified using the environmental variable ABORT_ON_DEADLOCK. For example, issuing the command

setenv ABORT_ON_DEADLOCK true

Copyright © 2002 John Wiley & Sons, Ltd.

```
! Attempt to send the information in send_info to process of rank ''dest'':
call mpi_isend (send_info, 512, mpi_character, dest, MPI_CHECK_Tag1+tag, &
                comm, req1, ierror)
timer = mpi_wtime() ! start the timer
do while(.not.flag) ! spin wait for MPI_CHECK_TIMEOUT minutes or until req1 is satisfied
   if(mpi_wtime() − timer > MPI_CHECK_TIMEOUT) then
    ! Print warning message when time>MPI_CHECK_TIMEOUT and stop program execution
     call time_out(filename, start_line, end_line, 'mpi_send' MPI_CHECK_TIMEOUT)
   endif
   call mpi_test (req1 flag, status, ierror) ! Test whether the isend has finished.
enddo

! Check for a response from process of rank ''dest'':
call mpi_irecv(response, 1, mpi_integer, dest, MPI_CHECK_Tag2+tag, comm, req2, ierror)
Timer = mpi_wtime() ! start the timer
do while(.not.flag) ! spin wait for MPI_CHECK_TIMEOUT minutes or until req2 is satisfied
   if(mpi_wtime() − timer > MPI_CHECK_TIMEOUT) then
     call time_out(filename, start_line, end_line, 'mpi_send', MPI_CHECK_TIMEOUT)
   endif
   call mpi_test (req2, Flat, status, ierror) ! Test whether the irecv has finished.
enddo

! the original mpi_send, mpi_rsend or mpi_ssend
call mpi_send (buf, count, datatype, dest, tag, comm, ierror)
```

Figure 3. The code inserted prior to calling mpi_send, mpi_rsend or mpi_ssend.

will cause the program to stop execution (an mpi_abort is executed) when an actual or potential deadlock is detected.

To illustrate the meaning of the information provided by MPI-CHECK, suppose one process issues an mpi_send in an attempt to send a message **A** to another process. Suppose further that there is no corresponding mpi_recv. Notice that when **A** is small, mpi_send will copy **A** into a system buffer and execution will continue, i.e. no actual deadlock will occur. However, MPI-CHECK will issue the following warning message in this situation for all positive values of MPI_CHECK_TIMEOUT:

Warning. [File = test_handshake2.f90, Line = 29, Process 0] mpi_send has been waiting for XX minute. There may not be a corresponding mpi_recv or mpi_irecv ready to receive.
call mpi_send (A,n,mpi_real, p-1,1,mpi_comm._world,ierror)

If the ABORT_ON_DEADLOCK is set to .false., then the above message will be issued every MPI_CHECK_TIMEOUT minutes. However, if there really is a corresponding receive but the executing process does not reach the call to this receive until just after the first warning message, then the above message will only be issued once.

Next suppose that one process issues an mpi_recv and there is no corresponding mpi_send. In this case, the process executing the mpi_recv will not be able to continue beyond this call as may happen

Copyright © 2002 John Wiley & Sons, Ltd.

```
! Attempt to receive send_info from the process of rank ''source'':
if (tag == mpi_any_tag) then
   call mpi_irecv (send_info, 512, mpi_character, source, mpi_any_tag, &
                   comm, req1, ierror)
else
   call mpi_irecv (send_info, 512, mpi_character, source, &
                   MPI_CHECK_Tag1+tag, comm., req1, ierror)

timer = mpi_wtime() ! start the timer
do while(.not.flag) ! spin wait for MPI_CHECK_TIMEOUT minutes or until req1 is satisfied
   if(mpi_wtime() − timer > MPI_CHECK_TIMEOUT) then
     ! Print warning message when time>MPI_CHECK_TIMEOUT and stop program execution
      call time_out(filename, start_line, end_line, 'mpi_recv' MPI_CHECK_TIMEOUT)
   endif
   call mpi_test (req1 flag, status, ierror) ! Test whether the irecv has finished
enddo

! Extract information from send_info
read(send_info, *) send_filename, send_startline, send_endline, send_rank, &
                   send_count, send_type, send_tag

! Check count and datatype from send_info with the count and datatype in the mpi_recv, if data is
! not consistent a warning message issued.
...

! Send response to the sender
call mpi_send(response, 1, mpi_integer, send_rank, MPI_CHECK_Tag2+send_tag, &
              comm, ierror)

! the original mpi_recv, except tag and source have been changed if mpi_any_source
! and mpi_any_tag were used in the original call
call mpi_recv (buf, count, datatype, send_rank, send_tag, comm, status, ierror)
```

Figure 4. The code inserted prior to calling mpi_recv.

in the situation mentioned in the above paragraph. When MPI-CHECK encounters the situation of executing a mpi_recv and there is no corresponding mpi_send, then the following message is issued:

Warning. [File = test_handshake6.f90, Line = 33, Process 0] mpi_recv has been waiting for XX minute. There may not be a corresponding MPI send ready to send.
call mpi_recv(C,n,mpi_real,mpi_any_source,1,mpi_comm._world,status,ierror)

Next suppose there is a send/receive dependency cycle for two processes as in Figure 1. If ABORT_ON_DEADLOCK = .true., then the process first executing the mpi_send will issue a warning and program execution will be terminated. If ABORT_ON_DEADLOCK = .false., then for every time period both the send and receive processes will issue a warning. Notice that with ABORT_ON_DEADLOCK = .false., warning messages from both the send and receive processes will continue to be issued for all actual/potential deadlocks until the program is manually aborted by the

Copyright © 2002 John Wiley & Sons, Ltd.

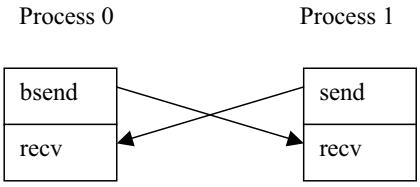


Figure 5. A cycle with mpi_bsend involved.

```
! Attempt to send the information in send_info to process of rank ''dest'':
call mpi_isend (send_info, 512, mpi_character, dest, MPI_CHECK_Tag1+tag, &
                comm, req1, ierror)

! Check for a response from process of rank ''dest'':
call mpi_irecv(response, 1, mpi_integer, dest, MPI_CHECK_Tag2+tag, comm, req2, ierror)

! the original mpi_bsend
call mpi_bsend (buf, count, datatype, dest, tag, comm, ierror)
```

Figure 6. The code inserted prior to calling mpi_bsend.

user. If the time period is set too short and there is no actual or potential deadlock, then these warning messages will stop being issued when execution continues beyond this point.

2.2. Handshake procedure for mpi_bsend

If the buffer space is properly managed and if a process calls the MPI buffered send, mpi_bsend, and if there is no corresponding MPI receive, then this situation is not an 'actual' nor a 'potential' deadlock since there will never be an actual deadlock for all possible valid MPI implementations. However, this situation is clearly incorrect MPI code. MPI-CHECK does not currently detect this incorrect MPI code nor does MPI-CHECK check if buffers are properly managed for those programs that use buffered sends. If a process calls an MPI receive, then this receive may be satisfied by receiving a message from a buffered send. To determine if this is the case, instrumentation needs to be inserted prior to the call to mpi_bsend by MPI-CHECK. Notice that if the same instrumentation were to be used for mpi_bsend as is used for mpi_send, then for the send-receive cycle situation in Figure 5 an actual or potential deadlock would be reported. However, this is not a deadlock. This problem is solved by using the same handshaking procedure for mpi_send except we remove the waiting for the completion of the mpi_isend and mpi_irecv. Figure 6 shows the code inserted prior to the call to mpi_bsend.

2.3. Handshake procedure for mpi_sendrecv and mpi_sendrecv_replace

To avoid actual and potential deadlocks, mpi_sendrecv and mpi_sendrecv_replace routines should be used when exchanging data between processes instead of using mpi_send and mpi_recv, see [1], unless

Copyright © 2002 John Wiley & Sons, Ltd.

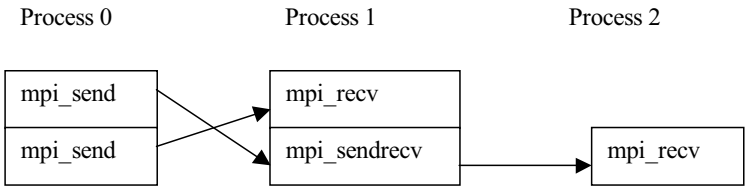


Figure 7. A dependency cycle involving mpi_sendrecv.

non-blocking sends and receives are used. Recall that when a process executes an mpi_sendrecv or mpi_sendrecv_replace, the process sends a message to another process and expects to receive a message from a possibly different process. Thus, actual and/or potential deadlocks may occur because of missing sends and receives. In addition, send-receive cycles may occur when using mpi_sendrecv or mpi_sendrecv_replace and may cause actual or potential deadlocks. This is illustrated in Figure 7.

Since mpi_sendrecv and mpi_sendrecv_replace involve both the sending and receiving of messages, the handshaking procedure used by MPI-CHECK for these routines is a combination of the handshaking used for mpi_send and for mpi_recv. Figure 8 shows the code inserted by MPI-CHECK prior to calling mpi_sendrecv. The code inserted prior to calling mpi_sendrecv_replace is identical.

2.4. Detection strategy considering the MPI_PROBE problem

Actual and potential deadlocks can also be caused by the incorrect usage of mpi_probe. Recall that mpi_probe allows one to poll an incoming message to determine how to receive the message. Since mpi_probe is blocking, if there is no corresponding send, then there will be an actual deadlock on the process executing the mpi_probe. The situation of deadlock detection is complicated by the fact that a single send can satisfy multiple calls to mpi_probe [1, p. 52]. An additional problem may occur when mpi_probe causes a dependency cycle, see Figure 9.

To detect an actual or potential deadlock situation when mpi_probe is used, the handshake strategy introduced before is still applicable but some changes are required. If a corresponding mpi_probe exists before a MPI receive call, the receiver side handshake procedure should be inserted before the probe. If more than one corresponding probes exist before a MPI receive call, the receiver side handshake procedure should be inserted before the first probe. Thus, the problem we need to solve is to identify whether there has been a corresponding probe existing before any mpi_probe or mpi_recv (mpi_sendrecv, mpi_sendrecv_replace) call in a MPI program, and then decide whether a handshake procedure needs to be inserted. The modified handshake strategy for the receiver side is described in Figure 10. Because of space limitations, the 'MPI_RECV' in Figure 10 includes the mpi_recv, mpi_sendrecv, and mpi_sendrecv_replace routines.

To detect these problems, we first insert the same handshaking code before the call to mpi_probe as is used for mpi_recv. However, notice that this will cause the handshaking strategy for mpi_recv and other calls to mpi_probe to not perform as desired. To avoid this problem, MPI-CHECK keeps a list of all calls to mpi_probe with a unique {communicator, tag, source}. In the code inserted prior

Copyright © 2002 John Wiley & Sons, Ltd.

```
! Attempt to send the information in send_info to process of rank ''dest'':
call mpi_isend (send_info, 512, mpi_character, dest, MPI_CHECK_Tag1+sendtag, &
                comm, req1, ierror)
! Check for a response from process of rank ''dest'':
call mpi_irecv(response, 1, mpi_integer, dest, MPI_CHECK_Tag2+tag, comm, req2, ierror)
! Attempt to receive send_infro from the process of rank ''source'':
if (recvtag == mpi_any_tag) then
   call mpi_irecv (recv_info, 512, mpi_character, source, mpi_any_tag, comm, req3, ierror)
else
   call mpi_irecv (recv_info, 512, mpi_character, source, &
                   MPI_CHECK_Tag1+recvtag, comm, req3, ierror)
! Timing until req3 is done. Otherwise timeout.
......

! Extract information from recv_info
read(recv_info, *) send_filename, send_startline, send_endline, send_rank, &
                   send_count, send_type, send_tag
! Check count and datatype from recv_info with the recvcount and recvtype in the mpi_sendrecv,
! if data is not consistent a warning message is issued.
......

! Send to response to the sender
call mpi_isend (response, 1, mpi_integer, send_rank, MPI_CHECK_Tag2+send_tag, &
                comm, req, ierror)
! Timing until req1 is done. Otherwise timeout.
......

! Timing until req2 is done. Otherwise timeout.
......

! the original mpi_sendrece, except recvtag and source have been changed if mpi_any_source
! and mpi_any_tag were used in the original call
call mpi_sendrecv (sendbuf, sendcount, sendtype, dest, sendtag, recvbuf &
recvcount, recvtype, send_rank, send_tag, comm, status, ierror)
```

Figure 8. The code inserted prior to calling mpi_sendrecv.

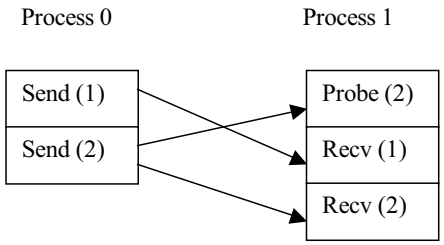


Figure 9. A dependency cycle involving a call to mpi_probe.

Copyright © 2002 John Wiley & Sons, Ltd.

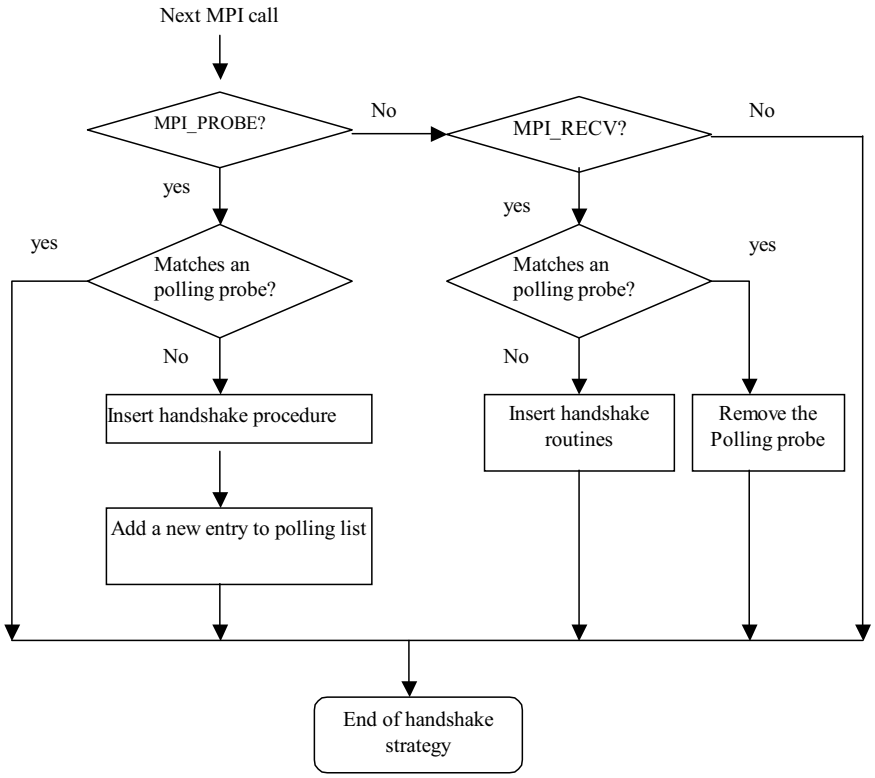


Figure 10. The handshake strategy for the receiver side.

to calling mpi_recv and mpi_probe, checking is done to determine if the {communicator, tag, source} for the mpi_recv or mpi_probe matches an entry in this list. If it matches an entry in the list, then the handshaking is bypassed; otherwise, the handshaking is performed. If the {communicator, tag, source} of mpi_recv matches an entry in this list, then this entry is removed from the list. Figure 10 illustrates this handshake strategy.

There is a non-blocking version of mpi_probe, called mpi_iprobe. No handshaking is required for mpi_iprobe since its usage cannot cause deadlocks.

3. DEADLOCK DETECTION FOR NON-BLOCKING POINT-TO-POINT MPI ROUTINES

MPI also allows the use of non-blocking point-to-point routines, mpi_isend, mpi_issend, mpi_ibsend, mpi_irsend, and mpi_irecv. Non-blocking sends/receives can be matched with blocking receives/sends.

Copyright © 2002 John Wiley & Sons, Ltd.

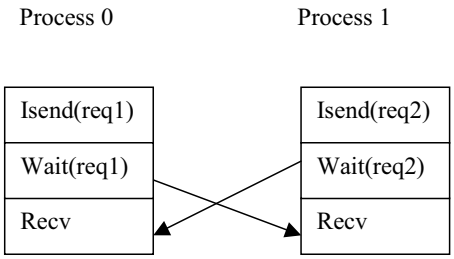


Figure 11. A dependency cycle involving non-blocking calls.

Completion of non-blocking sends/receives is indicated by calls to mpi_wait, mpi_test, mpi_waitany, mpi_testany, mpi_waitall, mpi_testall, mpi_waitsome, and mpi_testsome. Currently, MPI-CHECK only checks for non-blocking sends/receives completed by mpi_wait and mpi_waitall. If other routines are used to indicate completion, MPI-CHECK will not check for completion, and under some circumstances MPI-CHECK may incorrectly report errors. If there are non-blocking send or receive calls without corresponding calls to mpi_wait or mpi_waitall, MPI-CHECK issues a warning message suggesting that the user add matching mpi_wait or mpi_waitall calls. Wildcards in non-blocking receive routine are currently not supported by MPI-CHECK.

As is the case with blocking sends and receives, actual and potential deadlocks may occur when using non-blocking sends and receives. The actual or potential deadlock will occur at the call to mpi_wait or mpi_waitall and not at the call to the non-blocking send or receive. For example, an actual or potential deadlock will occur if there is no matching send or receive. Dependency cycles may also occur with non-blocking routines. Figure 11 shows a dependency cycle when using non-blocking and blocking calls that causes either an actual or potential deadlock.

Figures 12 and 13 show situations where a deadlock will never occur because of the progress statement in Section 3.7.4 in the MPI specification [1]. Figure 14 shows a potential deadlock situation whereas there is no actual or potential deadlock the situation in Figure 15. The situation in Figure 15 is the same as in Figure 14 except both mpi_wait's occur after calling the mpi_irecv's. There is also no actual or potential deadlock for the situation in Figure 16. Notice that this progress statement implies that the calls to mpi_wait in Figures 15 and 16 may be in any order and may be replaced by a single call to mpi_waitall using any order for req1 and req2.

MPI-CHECK detects actual and potential deadlocks involving non-blocking routines using a handshaking strategy that is similar to the handshaking strategy used for blocking routines. When a non-blocking send or receive is encountered, MPI-CHECK inserts code prior to the call that initiates the handshake but does not wait for the handshake to be completed. The code that completes the handshaking is inserted prior to the call to the corresponding mpi_wait (or mpi_waitall). Prior to the call to any of the non-blocking sends/receives, MPI-CHECK inserts the desired code by inserting a call to subroutine 'handshake_isend'/'handshake_irecv'. Prior to the call for all corresponding waits, MPI-CHECK inserts the desired call to subroutine 'handshake_wait'. Figure 17 shows the code inserted

Copyright © 2002 John Wiley & Sons, Ltd.

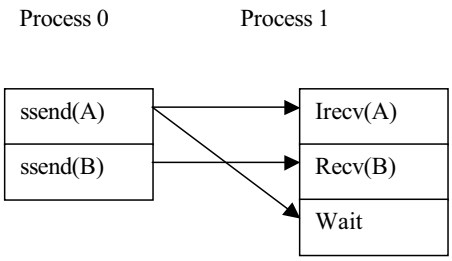


Figure 12. A situation where a deadlock will never occur.

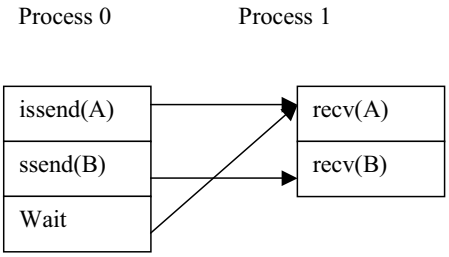


Figure 13. A situation where a deadlock will never occur.

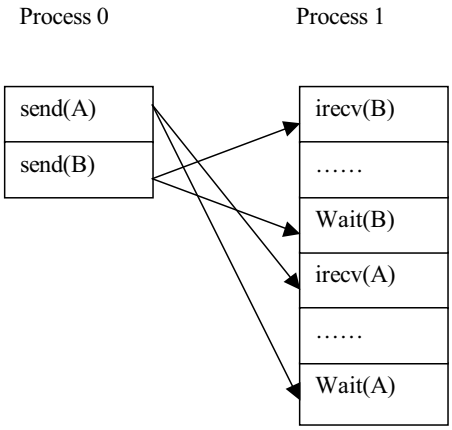


Figure 14. A potential deadlock situation.

Copyright © 2002 John Wiley & Sons, Ltd.

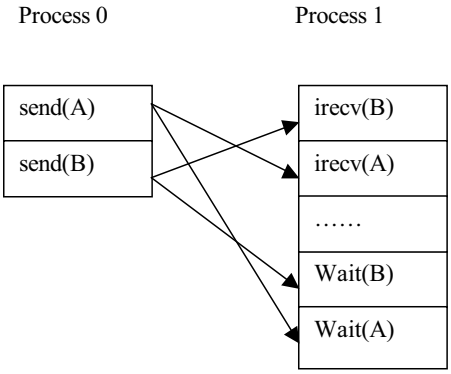


Figure 15. No actual or potential deadlock situation.

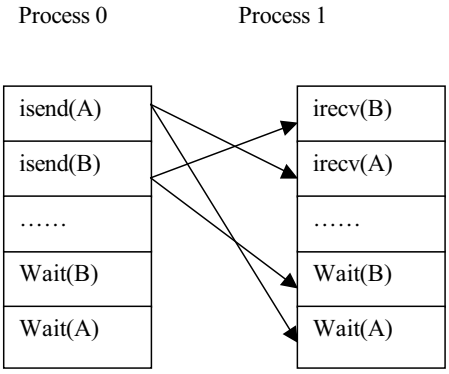


Figure 16. No actual or potential deadlock situation.

prior to calling mpi_issend, mpi_issend, and mpi_irsend as well as the code inserted prior to calling the corresponding mpi_wait. The non-blocking buffered send, mpi_ibsend, is handled in a way similar to what was done for the blocking buffered send. Figure 18 shows the code inserted prior to calling mpi_irecv and its corresponding mpi_wait.

By inserting code prior to the call to mpi_finalize, MPI-CHECK is able to determine if there are any pending calls to non-blocking send or receive routines. This situation only occurs if a call to a non-blocking routine has been made and there is no corresponding call to a mpi_wait. MPI-CHECK keeps track of all non-blocking calls and their corresponding requests by writing this information into the globally accessible integer array, MPI_CHECK_Ireq of size MAX. MAX is set to 512 by default.

Copyright © 2002 John Wiley & Sons, Ltd.

```
! Attempt to send the information in MPI_CHECK_Isendinfo(k) to process of rank ''dest'':
call mpi_isend (MPI_CHECK_Isendinfo(k), 512, mpi_character, dest, &
                        MPI_CHECK_Tag1+tag, comm, MPI_CHECK_Ireq1(k), ierror)
! Attempt to receive a response from process of rank ''dest'':
call mpi_irecv(response, 1, mpi_integer, dest, MPI_CHECK_Tag2+tag, comm, &
                        MPI_CHECK_Ireq2(k), ierror)
! the original non-blocking send
call mpi_isend(buf, count, datatype, dest, tag, comm, req, ierror)
...

! spin wait for MIP_CHECK_TIMEOUT minutes or until MPI_CHECK_Ireq1(k) is satisfied
Timer = MPI_Wtime()
Flat = FALSE
do while(.Not.Flat)
    if(MPI_Wtimer() − Timer > MPI_CHECK_TIMEOUT) then
       call outtime('test.f90', 31, 31, 'mpi_wait', MPI_CHECK_TIMEOUT)
    endif
    call mpi_test(MPI_CHECK_Ireq1(k), Flat, status, ierror)
enddo
! spin wait for MIP_CHECK_TIMEOUT minutes or until MPI_CHECK_Ireq2(k) is satisfied
Timer = MPI_Wtime()
Flat = FALSE
do while(.Not.Flat)
    if(MPI_Wtimer() − Timer > MPI_CHECK_TIMEOUT) then
       call outtime('test.f90', 31, 31, 'mpi_wait', MPI_CHECK_TIMEOUT)
    endif
    call mpi_test(MPI_CHECK_Ireq2(k), Flat, status, ierror)
enddo

! The original mpi_wait call
mpi_wait (req, status, ierror)
```

Figure 17. The code inserted prior to calling mpi_isend/mpi_issend/mpi_irsend and the corresponding mpi_wait.

When a request has been satisfied, it is removed from this array. When there are pending calls to non-blocking routines encountered, MPI-CHECK issues a warning, such as

> *Warning. At least one non-blocking routine has no explicit termination. Please add mpi_wait for them to get more information from MPI-CHECK!*

4. DEADLOCK DETECTION FOR COLLECTIVE MPI ROUTINES

Unlike point-to-point communication, collective routines come in blocking versions only. This section presents the methods used by MPI-CHECK to detect 'actual' and 'potential' deadlocks for MPI collective routines.

Copyright © 2002 John Wiley & Sons, Ltd.

```
! Attempt to receive MPI_CHECK_Isendinfo(k) from process of rank ''src''.
call mpi_irecv (MPI_CHECK_Isendinfo(k), 512, mpi_character, src, &
                MPI_CHECK_Tag1+tag, comm, MPI_CHECK_Ireq(k), ierror)
! Attempt to send response to process of rank ''src''.
call mpi_send(response, 1, mpi_integer, src, MPI_CHECK_Tag2+tag, &
                comm, temp_req, ierror)
! The original mpi_irecv
call mpi_irecv(buf, count, datatype, src, tag, comm, req, ierror)
......

! Timing before the original mpi_wait
Timer = MPI_Wtimer()
! spin wait for MPI_CHECK_TIMEOUT minutes or until MPI_CHECK_Ireq(k) is satisfied
do while(.Not.Flag)
    if (MPI_Wtimer() − Timer > MPI_CHECK_TIMEOUT) then
       call timeout('test.f90', 36, 36, 'mpi_wait', MPI_CHECK_TIMEOUT)
    endif
    call mpi_test(MPI_CHECK_Ireq(k), Flag, status, ierror)
enddo

! Check count and datatype from MPI_CHECK_Isendinfo(k) with the count and datatype
! in the mpi_irecv. If data is not consistent, a warning message is issued.
......

! the original mpi_wait call
call mpi_wait (req, status, ierror)
```

Figure 18. The code inserted prior to calling mpi_irecv and the corresponding mpi_wait.

The following are the categories of actual and potential deadlock situations that can occur when using collective routines.

1. A collective routine is not called by all the processes in the communicator.
2. All processes in a communicator may not call distinct collective routines in the same order.
3. Improper ordering of point-to-point and collective routines.

MPI-CHECK automatically detects all of the above problems.

Figure 19 illustrates the situation in item 1, where two processes execute mpi_gather while the third process does not. Notice that this may be an actual or potential deadlock depending on which processor is the root processor and depending on whether or not there is synchronization after the call to mpi_gather. Figure 20 also illustrates the situation of item 1, where two processes execute mpi_allgather while the third executes mpi_gather.

Figures 21 and 22 illustrate the situation in item 2, incorrectly ordered collective operations. The MPI standard requires that 'a correct, portable program must invoke collective communications so that deadlock will not occur, whether collective communications are synchronizing or not [1]'.

Copyright © 2002 John Wiley & Sons, Ltd.

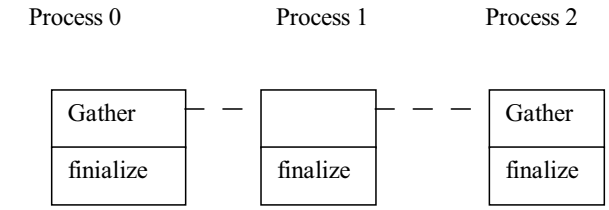


Figure 19. A collective operation with a missing call.

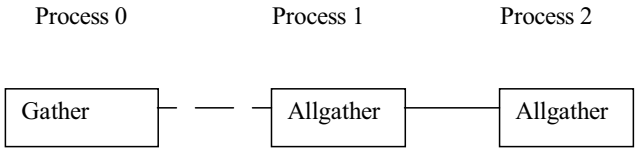


Figure 20. Mismatched collective operations.

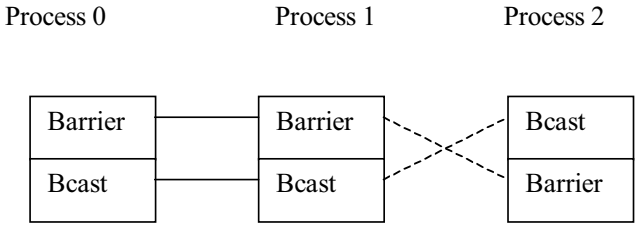


Figure 21. Incorrectly ordered collective operations.

Figure 23 illustrates the improper ordering of point-to-point and collective routines. If the root processor for the mpi_bcast is process 2, then there will be an actual deadlock. If the root processor for mpi_bcast is process 0 or 1, then there may or may not be an actual deadlock, depending on whether or not mpi_bcast is synchronizing. In all of these cases, the MPI code is incorrect, since the MPI standard [1] requires that 'the relative order of execution of collective operations and point-to-point operations should be such, that even if the collective operations and the point-to-point operations are synchronizing no deadlock will occur'.

MPI-CHECK detects the above errors by using a handshake strategy similar to what is done for the point-to-point routines. For the collective routines, **p** processes are involved rather than the two

Copyright © 2002 John Wiley & Sons, Ltd.

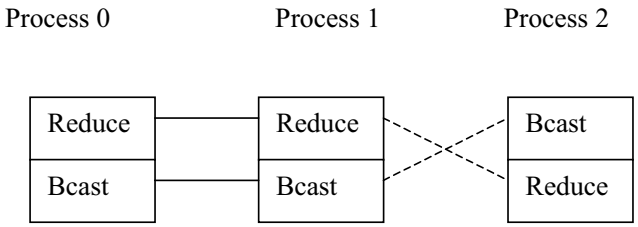


Figure 22. Incorrectly ordered collective operations.

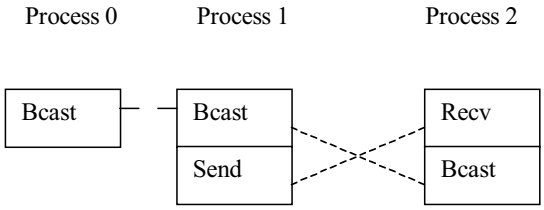


Figure 23. Interleaved collective and send-receive operations.

processes used for the point-to-point routines. The strategy is to use process 0 to collect information from all other processes. To do handshaking for the collective routines, all non-zero processes send the following information to process 0 in the send_info array

send_info = {file_name, start_line, end_line, get_rank(comm), call_name},

where start_line and end_line are the beginning and ending line numbers of the call to the collective routine in the file named 'file_name', and 'call_name' is the name of that collective routine. The process then sends send_info to the process of rank 0 in that communicator, with a unique tag, MPI_CHECK_Tag3, to avoid possible tag conflicts with other messages. The following three possibilities may occur.

1. The message was never received on process 0 and the sending process does not receive a response message within a specified time. In this case, a warning message is issued.
2. The message was received on process 0, the call_name in send_info was not the same one as the collective call on process 0. In this case, process 0 issues a message stating what the inconsistencies are and the program execution is stopped no matter what mode of execution is being used.
3. The message was received on process 0, the call_name in send_info was the same as the collective call on process 0, and process 0 sends a reply to the sending process stating that everything is

Copyright © 2002 John Wiley & Sons, Ltd.

```
! Attempt to send the information in send_info to process of rank 0:
call mpi_isend(send_info, 512, mpi_character, 0, MPI_CHECK_Tag3, comm, req1, ierror)
Flag = False
Timer = MPI_Wtime()
do while(.Not.flag)
    if(MPI_Wtime() − Timer > MPI_CHECK_TIMEOUT) then
      call outtime('test.f90', 12, 12, 'mpi_gather', MPI_CHECK_TIMEOUT)
    endif
    call mpi_test(req1, Flag, status, ierror)
enddo

! Check for a response from process of rank 0:
call mpi_irecv(response, 1, mpi_integer, 0, MPI_CHECK_Tag4, comm, req2, ierror)
Flag = False
Timer = MPI_Wtime()
do while(.Not.Flag)
    if(MPI_Wtime() − Timer > MPI_CHECK_TIMEOUT) then
      call outtime('test.f90', 12, 12, 'mpi_gather', MPI_CHECK_TIMEOUT)
    endif
    call mpi_test(req2, Flag, status, ierror)
enddo

! the original collective call
call mpi_gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)
```

Figure 24. The code inserted prior to a collective routine (on process with rank > 0).

okay. The reply is received by calling mpi_irecv with tag MPI_CHECK_Tag4. Program execution is continued.

The above describes the handshake procedure for the processes with non-zero rank. We now describe the handshake procedure for process zero. Processes zero issues **p**-1 mpi_irecv calls with tag MPI_CHECK_Tag3, attempting to obtain **p**-1 copies of send_info sent by non-zero processes. The following three possibilities may now occur.

1. **p**-1 copies of send_info were received and in each copy the call_name was consistent with the call_name on process 0. In this case, a reply message is sent to the each of the **p**-1 sending process indicating that the information is consistent.
2. At least one of the **p**-1 of send_info is never received within the specified time. In this case, a warning message is issued.
3. One of the **p**-1 copies of send_info was received, but the call_name in send_info was inconsistent with the call_name on process 0. In this case, a warning message is issued stating an unmatched collective call situation, and the program execution is stopped.

Figure 24 and 25 show the code inserted prior to calling each collective MPI routine on the process with rank = 0 and processes with rank > 0, respectively.

Copyright © 2002 John Wiley & Sons, Ltd.

```
! Attempt to receive p − 1 pieces of sendinfo from process 1, 2, . . . , p − 1:
do i = 1, get_comm_size(comm) − 1
        call mpi_irecv(handshake_info, 512, mpi_character, mpi_any_source, &
                       MPI_CHECK_Tag3, comm, req, ierror)
        Timer = MPI_Wtime()
        do while(.Not.Flat)
          if(MPI_Wtime − Timer > MPI_CHECK_TIMEOUT) then
            call outtime('test.f90', 12, 12, 'mpi_gather', MPI_CHECK_TIMEOUT)
          endif
          call mpi_test(req, Flag, status, ierror)
        enddo
        ! Extract information from send_info
        read(handshake_info, *) come_filename, come_startline, come_endline, come_rank, &
                                come_funcname
        ! unmatched collective call check
        if(come_funcname.NE. 'MPI_GATHER') then
          call mismatch('test.f90', 12, 12,'MPI_GATHER', 0, come_filename, come_startline, &
                        come_endline, come_funcname, come_rank)
        endif
        ! Send response to the sender!
        call mpi_send(response, 1, mpi_integer, come_rank, MPI_CHECK_Tag4, comm, ierror)
enddo

! the original collective call
call mpi_gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, &
                comm, &ierror)
```

Figure 25. The code inserted prior to a collective routine (on process with rank = 0).

For the deadlock situation described in Figure 19, MPI-CHECK will issue the following message:

Warning. [File = test_collective3.f90, Line = 28, Process 0] mpi_gather has been waiting for XX minutes. There may not be corresponding mpi_gather on other processes ready.
call mpi_gather(A,n,mpi_real,B,n,mpi_real,root,mpi_comm_world ,ierror)

For the deadlock situation described in Figures 20–22, MPI-CHECK will issue the following message:

Warning. [File = test_collective3.f90, Line = 28] This collective routine on process 0 cannot match the corresponding routine (the second one listed below) on process 1.
call mpi_gather(A,n,mpi_real,B,n,mpi_real,root,mpi_comm._world & ,ierror)
call mpi_allgather(A,n,mpi_real,B,n,mpi_real,mpi_comm._world,ierror)

For the deadlock situation described in Figure 23, MPI-CHECK will issue following message:

Warning. [File = test_collective.f90, Line = 29, Process 0] mpi_bcast has been waiting for XX minutes. There may not be corresponding routine on other processes ready.
call mpi_bcast(A1, 1, mpi_real8, 0, mpi_comm_world, ierror)

Copyright © 2002 John Wiley & Sons, Ltd.

Warning. [File = test_collective.f90, Line = 34, Process 2] mpi_recv has been waiting for XX minutes. There may not be corresponding MPI send ready to send.
call mpi_recv(A, n, mpi_real8, 0, 1, mpi_comm_world, status, ierror)

Note that the deadlock detection methods for collective MPI routines described in this section only apply to communication within a single group of processes (intra-communication) and not to disjoint groups of processes (inter-communication). The deadlock detection methods MPI-CHECK uses for detecting actual and potential deadlock caused by point-to-point MPI routines described in Sections 2 and 3 are applicable to both intra- and inter-communicator domains. Currently MPI-CHECK does not support the detection of actual and potential deadlocks for inter-communicator domains.

MPI 1.0 only allows intracommunicators to be used for collective routines. MPI 2.0 allows intercommunicators to be used for most of the collective routines, see [2]. For MPI-CHECK to detect actual and potential deadlocks for intercommunicator domains for collective routines, the following would need to be done. The MPI routine mpi_comm_test_inter(comm, flag) could be used to determine if the communicator is an intra or intercommunicator. To detect actual and potential deadlocks for intercommunicators, MPI-CHECK would have to be changed to accommodate the following differences.

1. Process zero in both the local and remote group will collect information from all the non-zero processes in the remote group.
2. The handshaking will then take place between each process zero and the non-zero processes in the remote group.

5. USING MPI-CHECK

MIP-CHECK has been designed to be easy to use. If one normally compiles a main program and two subroutines as

f90 –o a.out –O3 main.f90 sub1.f90 sub2.f90 –lmpi

then one would compile as follows when using MPI-CHECK

mpicheck f90 –o a.out –O3 main.f90 sub1.f90 sub2.f90 –lmpi

On the compile step, MPI-CHECK requires that source files appear rather than object files since MPI-CHECK needs instrument the source files. If MPI-CHECK does not find any static errors, then MPI-CHECK's dynamic checking is accomplished exactly as one would do without MPI-CHECK, e.g.

mpirun –np 16 a.out

To illustrate the overhead of using MPI-CHECK, the class A NAS LU Parallel Benchmark [6] was run using 4 MPI processes on an SGI Origin 2000. The compile time of the original program was 39 seconds when using the –O3 compiler option. When one first compiles with MPI-CHECK many support routines are generated and compiled so the first compile will take longer than subsequent compiles. The first compile took 214 seconds and all subsequent compiles took 106 seconds. The original code took 192 seconds to execute and the code instrumented with MPI-CHECK took 207

Copyright © 2002 John Wiley & Sons, Ltd.

seconds, about 8% longer. About 5 Mbytes of additional disk space were required by MPI-CHECK due primarily to storing the support routines.

MPI-CHECK is being used to help debug MPI application codes at Iowa State University. Many times MPI-CHECK has automatically found problems in MPI codes that would have been extremely difficult to find without this tool.

6. CONCLUSIONS

MPI-CHECK 1.0 [3] is a tool developed to aid in the debugging of MPI programs that are written in free or fixed format Fortran 90 and Fortran 77, but does not contain any deadlock detection methods. This paper presents methods for the automatic detection of many, but not all, actual and potential deadlocks in MPI programs. These methods have been implemented in MPI-CHECK 2.0 for MPI programs written in free or fixed format Fortran 90 and Fortran 77. The methods presented in this paper may also be applied to MPI programs written in C or C++.

REFERENCES

1. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, 1995. http://www.mpi-forum.org.
2. Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, 1997. http://www.mpi-forum.org.
3. Luecke G, Chen H, Coyle J, Hoekstra J, Kraeva M, Zou Y. MPI-CHECK: A tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience* 2002. To appear.
4. MPI-CHECK. http://www.hpc.iastate.edu/MPI-CHECK.htm.
5. Vetter JS, Supinski BR. Dynamic software testing of MPI applications with Umpire. *Proceedings of SC2000*, Dallas, November 2000.
6. NAS Parallel Benchmarks. http://webserv.gsfc.nasa.gov/neumann/hive_comp/bmarks/node5.html.

Copyright © 2002 John Wiley & Sons, Ltd.